In-Network Applications: Beyond Single Switch Pipelines

Xin Zhe Khooi†, Levente Csikor†, Jialin Li†, Dinil Mon Divakaran‡
†National University of Singapore, ‡Trustwave

2021 IEEE 7th International Conference on Network Softwarization (NetSoft) | 978-1-6654-0522-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/NetSoft51509.2021.9492665

***Abstract*—The emergence of commodity programmable switches have spawned a series of innovations in the network data plane. By making the traditionally stateless network architectures to be stateful, we can realize a diverse set of applications, e.g., networking monitoring, load-balancing, firewalls, *entirely in the data plane*. On the other hand, many existing in-network applications assume that the underlying switch is single-pipelined, however, in reality, commodity programmable switches are designed with multiple pipelines in mind. While this approach enables high scalability, it has introduced a serious disadvantage: maintaining states across the pipelines is non-trivial. For instance, without involving the control plane it is infeasible to keep track of a request and its response in different pipelines, thereby rendering many in-network proposals impractical.**

In this paper, we highlight this fundamental limitation that holds back the practical widespread adoption of stateful applications in today's multi-pipeline switches. By scrutinizing recent in-network approaches, we identify that majority of them cannot operate as they are proposed on multi-pipeline switches. After raising awareness of this inevitable consequence, we discuss a set of possible workarounds for in-network applications to overcome this issue on multi-pipeline switches.

***Index Terms*—stateful applications, programmable switches, multi-pipeline, software-defined networks**

I. Introduction

The emergence of commodity programmable switches have resulted in the "Cambrian Explosion" in the network data plane. This has induced a paradigm shift [1] in designing versatile networking applications ranging from network monitoring [2], [3], traffic engineering and load-balancing [4], [5] to high-precision congestion control [6], anomaly detection [7] and DDoS defense [8], [9]. Additionally, switch programmability has opened the door to perform bump-in-the-wire computation for another diverse set of use cases, e.g., distributed coordination [10], [11], distributed machine learning aggregation [12], and in-network caching [13], [14].

These innovations have led to new network managing methodologies [15], [16], while making it possible to offload core functionalities of end-host applications (e.g., replicated storage [17]) to the network to achieve orders-of-magnitude performance gains [18]. Notably, a key enabler of application-level logic offloading is the availability of stateful components such as registers, meters and counters in commodity programmable switches that are exposed to programmers. However, these resources do not come in abundance. Designing in-network applications along with common switching functionalities (e.g., Layer-2/3 switching and routing) and deploying within one hardware appliance is a challenging task [19].

978-1-6654-0522-5/21/$31.00 ©2021 IEEE

Commodity programmable switches commonly consist of more than one packet processing pipeline in order to scale up the available backplane capacity, ranging from one pipeline with 16 ports to four pipelines with 64 ports [20]. This performance gain, however, presents a major challenge to the design of in-network applications. For better scalability, pipelines in a switch are independent from each other and have their own dedicated hardware resources (e.g., SRAM). A direct consequence is the absence of coherent memory among the pipelines, meaning state updates in one pipeline are not visible in the others unless explicitly synchronized. However, network traffic processed by a switch usually traverses across pipelines (i.e., from different ingress port to different egress ports) based on the predefined switching and routing policies. As a result, application developers need to explicitly deal with stateful memory that are inherently distributed across pipelines.

The design and implementation of many existing in-network applications, e.g., [4], [8], [14], have been largely based on the assumption that network traffic traverses the same pipeline exclusively, thereby sweeping the issues under the carpet. Unfortunately, such over-simplified assumption can easily lead to erroneously application behaviors when deployed on today's multi-pipeline programmable switches. We believe this not entirely the fault of the developers – the single-pipeline assumption can be attributed to existing domain specific languages for programming the data plane, e.g., P4 [21], NPL [22]. In particular, they are designed to be architecture agnostic: the language specification has no formally defined memory models nor semantics of reading/updating memory when states are distributed across pipelines. Hence, when deploying applications in real networks, the data plane program developers have to manually define and enforce the expected behavior based on the hardware target they are working with. Given the promising gains of in-network applications, the aforementioned issue undeniably presents a major setback.

While there exists some potential (see later in §V) workarounds, such as recirculating each packet (§V-B) to every pipeline in order to synchronize the states, they complicate the design and implementation of applications, and negatively impacts the overall performance of the switch. A more severe implication is that strong-consistency guarantees, such as linearizability, are not possible under current architectures, ruling out several in-network applications proposed to improve distributed systems [11], [23].

The main purpose of this paper is to raise awareness of this fundamental constraint in designing stateful applications on today's commodity programmable switches. First, we give a brief overview of the architectural designs of programmable switches (§II); next, we discuss an example to comprehensively illustrate the above-mentioned limitation (§III). In §IV, we analyze recent in-network application proposals under multi-pipeline assumptions and discuss their inefficiencies. Lastly, in §V, we elaborate on some possible workarounds to make any in-network application feasible to run on multi-pipeline switches, with the caveat that all of them come with non-negligible impacts on the overall network performance.

While this important side-effect has been superficially covered in [18], [24], in this paper, we delve deeper into the topic. We hope to attract further discussion from the community and advocate future in-network application and programming language designs to consider deployment strategies on multi-pipeline switches, and potentially, devise a network-wide, distributed stateful primitive for multi-pipeline switches.

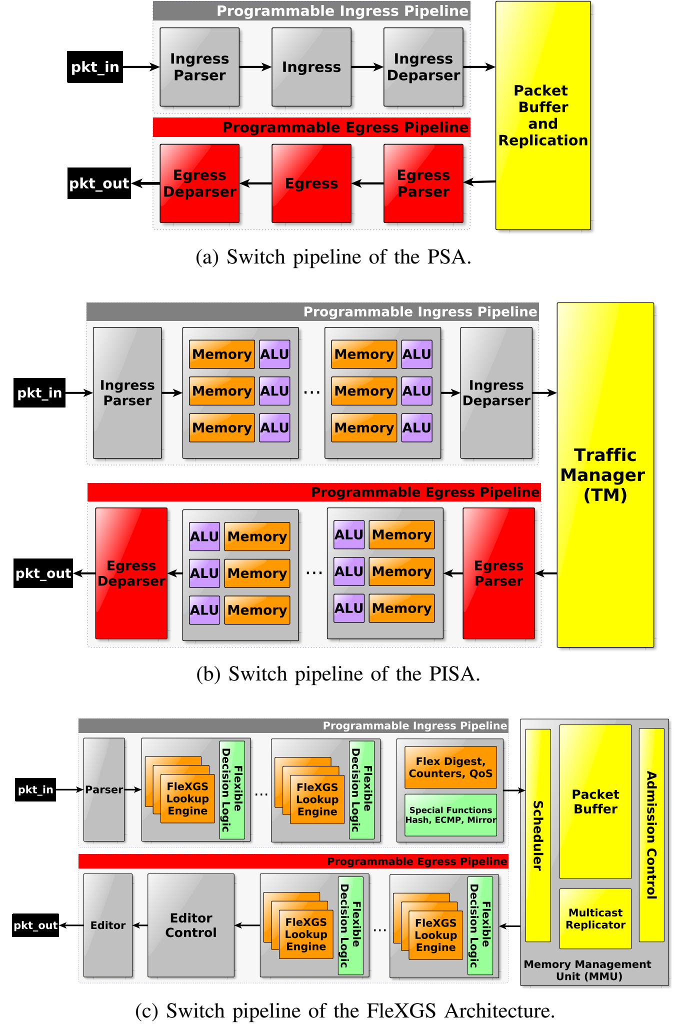
II. Background

A. Programmable Commodity Switches

The Portable Switch Architecture (PSA [25]) defined by the P4 Architecture Working Group illustrates the typical capabilities of network switches that process and forward packets across multiple switch ports. The pipeline of a Portable Switch usually contains three major blocks (see Fig. 1a), namely ingress, packet replication engine, and egress. Both ingress and egress have their own parser and deparser, as well as match-action units. Each pipeline is mapped to a set of switch ports.

Similarities to the PSA can be drawn from the switching silicons used in today's programmable commodity switches such as Broadcom's Trident-series ASICs (Trident 3 [26], Trident 4 [27]), and Intel's Tofino-series ASICs (Tofino [20], [28], Tofino 2 [29]). The Tofino ASICs are based on the Protocol-Independent Switch Architecture (PISA) – a derivative from the Reconfigurable Match-action Table (RMT [30]) architecture. Pipelines in PISA (see Fig. 1b) consist of ingress and egress blocks with multiple match-action stages, and traffic manager between the ingress and egress. On the other hand, the Trident ASICs are based on the FleXGS architecture [31] (see Fig. 1c), which also comprises of a programmable ingress and egress pipelines with multiple stages of lookup engines, and an additional memory management unit similar to the traffic manager in the PISA. One noticeable difference is that, packet parsing and deparsing is only done once in the FleXGS archictecture. Stateful packet processing is possible with per stage SRAM (e.g., registers) and FlexCounters on, PISA and FleXGS, respectively.

To maintain line-rate forwarding, generally, both architectures share similar characteristics in terms of the *(i)* limited number of available stages per pipeline (e.g., 12 on Tofino [32], and 5 on Trident 3 [26]), *(ii)* fixed memory per stages, *(iii)* fixed number of parallel memory accesses, and *(iv)* support for set of simple ALU operations only.



(c) Switch pipeline of the FleXGS Architecture.

Fig. 1: Overview of the different switch pipeline architectures.

B. Multiple Switch Pipelines

Bounded by the law of physics, the packet processing rate is bounded by the clock rate of the switching ASIC (e.g., 1 GHz). To improve the packet processing capacity, switching silicon designers follow the "scale-out" approach, i.e., in contrast to "scaling-up" the hardware resources (e.g., increasing clock speed) within the pipeline, they rather increase the number of *parallel pipelines*. Each pipeline is designed and configured to manage a given set of physical ports without any overlaps [25], [33], [34][1]. Furthermore, the parallel pipelines do not even share any common resources (e.g., SRAM, TCAM) except being connected to the switch backplane, i.e., memory management unit in FleXGS, or the Traffic Manager in PISA (cf. Fig. 1). Accordingly, when an application is deployed on a multi-pipeline switch, it has to be deployed on each pipeline individually. However, as packets may be routed and switched from one pipeline to another via the backplane, any

[1]Furthermore, to cater for different demands of packet processing capacity and number of switch ports, in practice, network device manufacturers also introduce multiple line-cards (with switching ASICs) in parallel [35], [36].

state (e.g., counters, registers) maintained within a pipeline is neither synchronized, shared nor accessible from other pipelines. Consequently, inter-dependent states can happen to be scattered across multiple pipelines due to nature of routing and switching, i.e., when the ingress and egress ports are at different pipelines. Applications relying on such states,e.g., even as simple application as TCP connection tracking (see details in §III), are facing a huge challenge in practical deployments. Recently, some attempts have been made on synchronizing states distributed among the pipelines, however, they have been shown to be futile [33].

While having multiple parallel pipelines have been practiced for years [37] in the switching architecture design, the notion of having multiple pipelines have never been explicitly exposed in traditional fixed function ASICs. SRAM and TCAM entries in the data plane have always been modified only by the control plane. Hence, the control plane keeps all the states synchronized regardless of the number of parallel pipelines. With the advent of programmable switching ASICs, however, the newfound ability to modify states entirely in the data plane (e.g., SRAM – registers) without the need of any control plane interaction has opened up various opportunities, yet, it presents a completely new challenge in designing stateful data plane applications efficiently.

Next, we further illustrate and elaborate on this problem through a simple example.

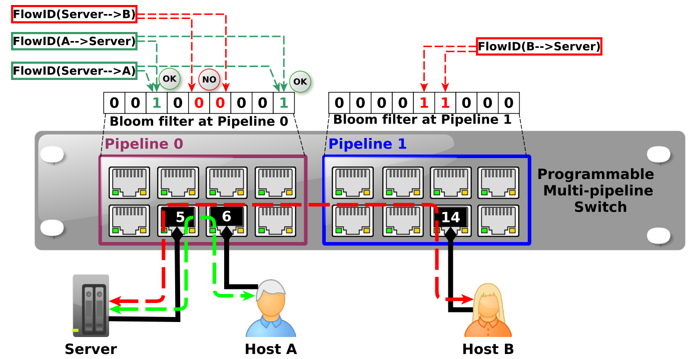
III. Example: TCP Connection Tracking

To demonstrate the problem of implementing stateful applications in multi-pipeline switches, we take the canonical example of stateful packet processing – TCP connection tracking (`conntrack`) for stateful firewalls [38]. We choose this application due to its simplicity, and being sufficiently comprehensive to depict the limitation of stateful packet processing in multi-pipeline switches.

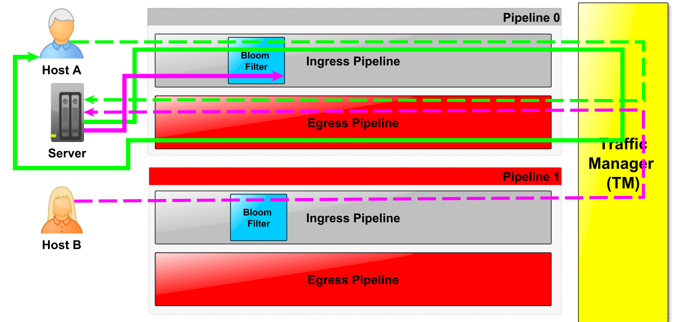
According to network security best practices, TCP `conntrack` only allows TCP connections to be established if they were initiated from the host network *per se*. Put differently, an outgoing TCP `SYN` packet (from the host network to the external networks) must be observed first before allowing the returning TCP `SYN-ACK`; then, the connection can be established accordingly. On the contrary, connections initiated from the external networks are dropped.

To reap the line-rate packet processing benefits of programmable switching ASICs, i.e., operating entirely in the data plane, TCP `conntrack` is typically realized using probabilistic data structures, such as bloom filters [39], to efficiently utilize the precious SRAM (commonly available as registers) in the pipeline stages. Bloom filters [39] are well-known space-efficient data structures to answer set-membership queries. In the case of TCP `conntrack`, whenever there is an outgoing TCP `SYN`, the flow ID[2] will be registered with the bloom filter. Subsequently, for the returning/incoming `SYN-ACK`s, the flow ID[3], will then be checked against the bloom filter to verify that the connection was in fact initiated from the internal network before allowing `SYN-ACK` and the subsequent traffic to go through the firewall.

[2]Here, we refer to the five-tuple comprising of the source and destination IP addresses and ports, and the IP protocol. Note, depending on the goal of the application, flow IDs can be defined differently.



(a) Simplified view on how pipelines are mapped to physical switch ports.



(b) Illustration on how network traffic traverses the pipelines.

Fig. 2: The figures show a multi-pipeline switch where different hosts are connected to different pipelines. Note: The dashed lines in **(b)** refer to traffic originating from the `Hosts`, while the solid lines depict the traffic originating from the `Server`.

In the physical nature of routing and switching, packets do not enter and exit through the same physical interface. Moreover, in the case of a multi-pipeline switch, they might not even enter and exit through the same pipeline. To visualize this, we depict two possible scenarios in Fig. 2. Fig. 2a shows the high level view on how the switch ports are typically mapped to individual pipelines, while Fig. 2b illustrates how traffic traverses the pipelines.

In Fig. 2, the `Server`, `Host A` and `Host B` are connected to a common programmable multi-pipeline switch. Physical connectivity-wise, the `Server` and `Host A` are connected to port 5 and 6 of pipeline 0; while `Host B` is connected to port 14 in pipeline 1. The `Server` is not allowed to initiate connections to the hosts, and can only respond to client requests.

First, we discuss the case where both outgoing and incoming traffic are local within the same pipeline. For `Host A`, it is

[3]More precisely, the flow ID in this case is "reversed" to be in line with flow ID of the original `SYN` packet.

located within the same pipeline with the `Server`. `Host A` attempts to establish a TCP connection with the `Server` by sending a TCP `SYN`. The TCP `SYN`'s flow ID (denoted by FlowID(A–>Server)) is then recorded with the bloom filter maintained in the memory, specifically, at ingress pipeline 0. To reach the server, the client's traffic is then forwarded to port 5. Later, the returning TCP `SYN-ACK` arrives via port 5. Its "reversed" flow ID (marked by FlowID(Server–>A)) is looked up in the corresponding bloom filter (in ingress pipeline 0) to verify that the connection was initiated from `Host A`. If the flow ID exists, the TCP `SYN-ACK` and the subsequent returning TCP traffic for this flow is forwarded to `Host A` through port 6.

On the other hand, for the case of `Host B`, it is situated in a different pipeline than the `Server` is connected to. Whenever `Host B` initiates a TCP connection with the `Server`, the TCP `SYN` packet (denoted by FlowID(B–>Server)) will be logged in pipeline 1 instead of pipeline 0. Therefore, the returning `TCP SYN/ACK (marked by FlowID(Server->B))` from the `Server` will be dropped as there are no relevant states regarding that particular flow in the bloom filter of pipeline 0.

Drawing from the observations in Fig. 2, we can infer the following:

CONSEQUENCE: Stateful applications with objectives based on preceding traffic states cannot work when the corresponding traffic traverse across disjoint pipelines.

IV. Analysis of Recent Applications

We look at recent in-network proposals for commodity programmable switches and investigate whether they can work as intended on multi-pipeline switches. According to the applications' relevance, we divide recent approaches into two key domains, i.e., network monitoring and in-network compute & acceleration, then we discuss the key works in each domain.

A. Networking

1) Network Monitoring: Data plane programmability and the access to stateful resources (e.g., registers, meters, counters) in the data plane open up possibilities to perform fine-grained per-packet monitoring in contrast to traditional coarse-grained sampling-based approaches (e.g., NetFlow [40]). Proposals such as PRECISION [2] or dSketch [19] introduce the ability to track heavy hitters entirely in the data plane, while FCM-Sketch [3] and BeauCoup [41] enable cardinality estimations (e.g., detecting super-spreaders) in the data plane. On the other hand, BurstRadar [42] and ConQuest [43] present approaches to perform fine-grained queue measurements for micro-burst detection in the network data plane.

In general, network monitoring applications keep track of packet-level statistics such as the number of packets and bytes. This action does not rely on any preceding states. Even if the traffic of interest appears to be scattered across different pipelines, the control plane would eventually gather and aggregate statistics for further decision making due to their commutative nature in this context. Hence, in-network traffic monitoring applications operate as intended on multi-pipeline switches.

In addition, In-band Network Telemetry (INT), known as the "killer" application of programmable switches, introduces the ability to encode telemetry information (e.g., queue occupancy) along the network path and has provided unprecedented visibility [44], [45] for network operators. Typical use cases of INT include high-precision congestion control [6], path tracing and network troubleshooting. Programmable commodity switches along the network path play the role of appending INT headers containing information such as queue occupancy. Then, at the penultimate hop, the INT headers are popped and exported for further analysis. Traffic headed towards different egresses do not affect the information being appended to the INT headers as they (e.g., queue occupancy, current link utilization) are always available and processed at the egress pipeline. Accordingly, multi-pipeline switches do not pose any problem for implementing in-band network telemetry either.

2) Load Balancing: Traditional equal-cost multipath (ECMP) based routing strategies are static and do not adapt to link utilization changes. Hence, dynamic in-network load-balancing approaches such as HULA [4], MP-HULA [46], DASH [5], and Contra [47] present real-time feedback mechanisms to better adapt with the ever-changing network loads.

In a nutshell, these approaches depend on periodic probes to inform adjacent switches about the real-time status of the switches such as link utilization and queue occupancy. Information carried by the probes are then stored in register arrays by adjacent switches in their ingress pipelines, which will then be queried to determine which particular link the packet should be forwarded through. However, if the egress ports/links are located in a different pipeline in contrast to the ingress pipeline, the current pipeline would not receive any probes containing the corresponding link information. Hence, the relevant states are absent for the ingress pipeline for performing load-balancing for egress links that are connected to other pipelines.

This presents an issue similar to what has been discussed in §III. As load-balancing network traffic relies on preceding states from the probes that may not be present in the same pipeline (specifically, at the ingress pipeline), under multi-pipeline scenarios, in-network load-balancing approaches break.

3) DDoS Defense: DIDA [8] and Poseidon [9] present the use of commodity programmable switches to perform line-rate defense against volumetric distributed denial-of-service attacks. Take the classic volumetric DNS amplification attack as an example: to identify such an attack, the network data plane has to keep track of both the DNS requests and responses. Whenever the number of responses becomes significantly greater than the number of requests, it can be deduced that the particular host is under attack and immediate reactive measures (e.g., blocking or redirecting the malicious traffic to scrubbers) must be put in place in the data plane.

Identically, the need to keep track of the requests and responses fall within the consequence outlined in §III. If the

requests and its corresponding responses arrive at different pipelines, the data plane itself is not able to deduce whether the network is indeed under an attack or not[4], unless the control plane is involved (§V-C).

B. In-network Compute & Acceleration

1) Caching: Distributed storage system workloads commonly exhibit high skew: a small percentage of popular objects receive disproportionately more traffic than the others. High skew in object accesses presents a major challenge to the efficiency of the storage system, as it naturally leads to uneven load distribution among storage servers, and may cause servers to overload if they hold highly popular objects. Recently, there have been proposals [13], [48] that aim to address this issue by caching popular objects directly in programmable switch data plane. Switch ASICs provide sufficient processing capacity to handle requests to the most popular objects in the system regardless of skew, effectively reducing load on the back-end storage servers and more importantly, making the load more even across the servers.

To ensure consistent caching results, queries for an object must be routed to the same switch pipeline that stores the cached copy of the object. NetCache [13] addresses this challenge by storing object caches in the egress pipeline. It leverages the fact that storage requests are destined to the storage server that holds the object (and the mapping is assumed to be stable), so an object, regardless of which ingress port the packet arrives at, is always mapped to the same egress port (and therefore the same egress pipeline). In-network caching, therefore, can function properly on multi-pipeline switches.

2) Distributed Coordination: Many distributed systems require explicit coordination to provide consistency guarantees: replicated state machines use coordination to ensure the system to behave as a single, correct machine; distributed transactional systems employ concurrency control and distributed commitment to enforce serializable execution of transactions; distributed storage systems use coordination to provide coherence among the cached and replicated object copies. However, excessive coordination among servers introduces high overhead, adding latency penalties and limiting the system's throughput and scalability. A recent line of work attempts to reduce or even eliminate distributed coordination by co-designing distributed systems with programmable switches [11], [14], [17], [23], [49], [50].

NOPaxos [23] and Eris [11] propose new multicast primitives with strong ordering properties to facilitate state machine replication and distributed transactional protocols. By providing ordered message delivery guarantees, these new network primitives eliminate distributed coordination in application-level protocols while still enforce linearizability (for state machine replication) and strict serializability (for distributed transactions). To efficiently realize ordered multicast, both systems implement in-network sequencing mechanisms: groups of sequence numbers are maintained on a single programmable switch; each multicast packet going through the switch is stamped with the appropriate sequence numbers. Unfortunately, the in-network sequencing approach does not work on multi-pipeline switches. Sequence numbers cannot be stored in the ingress pipelines as multicast packets may arrive at any ingress port; neither can they be maintained in the egress pipelines since a multicast packet may be replicated to multiple egress ports, while the primitive requires all copies of the packet contain the same sequence numbers.

Instead of co-designing network primitives with distributed protocols running on servers, NetChain proposes to implement a replicated key-value store completely in the switch data plane. It uses chain replication across multiple switches to provide strong consistency and fault tolerance while allowing client queries to finish in sub-RTT latency. Similar to the NetCache design, NetChain maintains the key-value store in the egress pipelines. Queries and updates to a key are consistently routed to the same egress pipeline, making multi-pipeline switches to function correctly. The same approach is applied to NetLock [49], a centralized lock manager that processes lock requests in the switch data plane. In NetLock, the lock tables are also maintained in the egress pipelines which connect to the corresponding lock servers, ensuring consistent lock request processing.

Many distributed storage systems replicate popular objects on multiple storage servers to improve load balancing of the system. However, it is a challenging task to maintain coherence among the multiple copies, and these systems either introduce expensive coordination protocols, or forgo consistency altogether. Pegasus [14] is a new distributed storage architecture that addresses this trade-off by implementing a coherence directory in the switch data plane. Leveraging the fact that the switch serves as a central point of the system, Pegasus routes queries to servers with the most up-to-date copy to ensure linearizability, and uses server replies to update the coherence directory, avoiding expensive invalidation traffic. Unfortunately, Pegasus' approach fails to work on a multi-pipeline switch. In Pegasus, requests and replies for a particular object need to access the same coherence directory entry on the switch. However, the ingress and the egress pipelines processing a request may be completely disjoint from those processing the corresponding reply. Implementing the coherence directory in neither ingress nor egress would offer a consistent view of the coherence directory.

3) Distributed Machine Learning Training: Nodes involved in distributed machine learning training requires significant communication overhead with the central parameter server in order to exchange and aggregate the training weights. The sheer volume of data needed to be exchanged continuously present a bottleneck to distributed machine learning performance.

To address this, the authors at [12] propose SwitchML, an approach to accelerate distributed machine learning by

[4]In fact, in such case, the pipeline architecture-agnostic DDoS detection application would falsely report many legitimate DNS responses as attacks since there are no counts for the corresponding requests at the same pipeline.

performing in-network aggregation using programmable commodity switches. As machine learning models are large in size (in magnitudes of hundreds of Megabytes) and cannot fit within the memory of the switching ASICs (only a few Megabytes are available), therefore, aggregation of model weights are done in per-packet streaming fashion. Upon receiving all the updates from each worker, the switch outputs the results with the aggregated weights for each worker via multicast.

It is not hard to observe that all the workers must be connected to the same pipeline to perform aggregation on all the weight update packets received, since the aggregated weights are maintained in the SRAM within the pipeline. This presents a constraint for SwitchML to operate on multi-pipeline switches, limiting its scalability. In §V-A2, we discuss the approach suggested by the authors to overcome this issue and scale beyond multiple pipelines and racks.

V. Possible Workarounds

In this section, we outline and discuss potential workarounds that can be applied to existing approaches in multi-pipeline switch scenarios. For clarity, we use the TCP `conntrack` example discussed in (§III) to reason about these workarounds.

A. Topology-specific Optimizations

1) Wiring Nodes to the Same Pipeline: A straightforward trick is to design the network topology carefully by taking the ASIC's pipeline-to-port mapping into consideration [18]. For instance, in the case of Fig. 2, as long as `Host A` and `Host B` are connected to the same pipeline as the `Server`, the TCP `conntrack` application works as expected. This ensures that traffic will never "crossover" to other pipelines and all states are maintained at the same place. This effectively treats multi-pipeline switches as multiple single-pipeline switch instances.

However, this approach is bounded by the number of available ports (typically 16 [18]) mapped to a particular pipeline. Consider hyper-scale data center settings where programmable commodity switches are favored for deploying custom solutions, the number of servers per rack can easily exceed the number of ports available per pipeline on the top of the rack switch. This complicates the applications that are designed to co-exist with the top of the rack switch, such as caching [13] or coherence directories [14] which were designed to treat the whole rack of servers under a single entity and function as the common caching layer.

2) Hierarchical Composition: To scale across multiple racks, [12] suggests hierarchical composition of multiple switches, e.g., by combining functionalities on the top of the rack switch and the aggregation switches. For instance, a switch having four pipelines is treated as four disjoint individual switches. For all four "switches", they are individually connected to a few common upstream aggregation switches. Instead of responding to the distributed training workers immediately after computing the aggregates within the respective pipelines, the aggregated weights are forwarded to the upstream switch to perform further aggregation of weights for all the pipelines. Alternatively, the responsibility of performing weight aggregation can also be assigned to the top-level switches only.

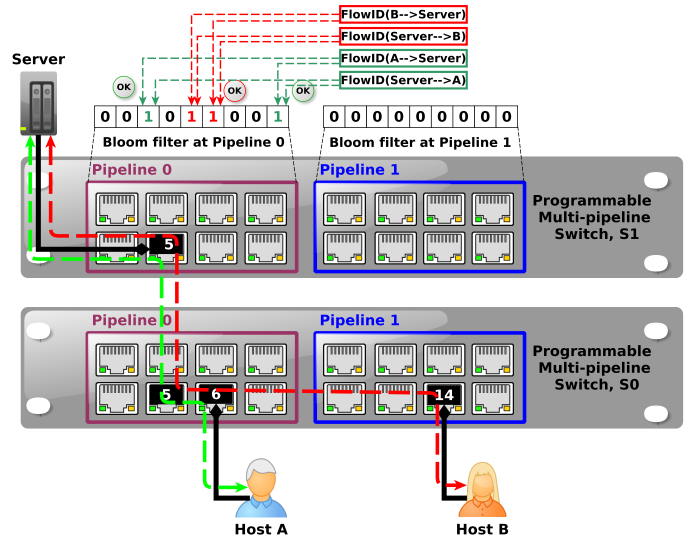


Fig. 3: Hierarchical composition with two switches.

Following this line of thought, we adapt the TCP `conntrack` example with hierarchical composition as shown in Fig. 3, with a two-level hierarchy consisting of two switches, `S0` and `S1`, with `S1` being the upstream switch connected to the `Server`. For this case, pipeline 0 in `S1` keeps track of all the TCP `SYN`s forwarded from the downstream switch, `S0`. While hierarchy composition presents a plausible method, if there are multiple downstream switches connected to `S1` thru pipeline 1, then the same problem about states distributed across pipelines (discussed in §III) resurfaces.

Besides, this approach also incurs additional routing and control complexities to the deployment. To maintain high availability and fault tolerance, downstream switches are typically wired to more than one upstream switches. Also, network traffic are usually load-balanced. Hence, to ensure that hierarchical composition works, it has to be ensured that a particular traffic always reaches exits and enters the same upstream switch. Otherwise, this approach would not be feasible.

B. Packet Recirculation

1) Redirecting Traffic Across Pipelines: Another possible solution would be to redirect traffic to the pipeline, which contains the relevant states, as suggested in [24], [25]. Take the TCP `conntrack` example in Fig. 2, specifically for `Host B`'s case, the workaround would be to recirculate (or in other words, re-inject) all the returning traffic from pipeline 0 to pipeline 1. This way, the returning traffic from the `Server` can then be inspected by the bloom filter in pipeline 1 which contains the preceding connection state, and then forwarded to `Host B`.

This method presents two down-sides. First, the data plane program would have to handle additional routing complexity

in order to correctly recirculate the corresponding traffic to their intended pipelines containing the preceding states (i.e., pipeline 0 needs to contain table entries for hosts that are located in pipeline 1 to know which traffic needs to be recirculated). Second, as the recirculated traffic are being processed more than once by the ASIC, this ultimately penalizes the available backplane capacity.

Depending on the application, the amount of traffic that needs to be recirculated varies. For instance, in TCP `conntrack`, all subsequent traffic from the external network must be redirected in order to be inspected, and therefore, effectively halving the available packet processing capacity. On the other hand, for an amplification attack defense mechanisms [8], [9] as discussed in §IV-A3, only a small subset of packets are affected, i.e., only the requests and responses need to be redirected.

2) Replicating States Across Pipelines: Instead of redirecting traffic to the pipeline for which the states are, one can choose to replicate the states by mirroring (and recirculating) the corresponding packets to the other pipelines to update the states. Again, referring to the example in Fig. 2, whenever the `Host B`'s TCP `SYN` packet is received, a copy of the TCP `SYN` packet is made, tagged with a custom header used for identification, then forward to the pipeline 1 for the state to be recorded. Subsequent returning traffic can then be inspected within pipeline 0 locally instead of needing to be recirculate it to pipeline 1 as discussed in §V-B1.

Similar to §V-B1, this technique involves additional routing complexities in the data plane program. In addition, this technique is only applicable for applications which rely on data structures for which operations are commutative, i.e., bloom filters, sketches. The case does not hold for hash tables as hash collisions result in destructive overwriting of states.

However, for applications demanding strong-consistency guarantees [11], [14], [23], this technique is not a viable option. As the replicated packets may arrive in the other pipelines in different order and timings, linearizability of operations cannot be guaranteed.

C. "Return" of the Control Plane

Apart from performing all the operations in the data plane only, relying on the control plane, still, presents itself as a viable option. For TCP `conntrack` example, whenever a TCP `SYN` packet is seen, it is reported to the control plane. As a response to that, the control plane updates the bloom filters for all pipelines[5] and thus maintaining synchronized states across the ASIC.

Undeniably, the involvement of the control plane inevitably introduces additional latency; as noted in [15], it could be up to the scale of several seconds. While this potentially goes against the aim of reducing overall control latency via entirely in-network approaches, we argue that this is an inevitable trade-off and sacrifice that has to be made, while keeping the communication overhead between the control plane bounded.

[5]Alternatively, the control plane can install forwarding rules in the match-action tables of the switch instead of modifying the stateful data structures.

With that in mind, the control plane should mainly play a passive role, and only reacts whenever there is a need to do so [16]. As another example, consider the amplification attack defense mechanism discussed in §IV-A3, the switch keeps track of the requests and responses, optimizations can be done in which the switch maintains the counts of the requests and responses in their respective pipelines. The control plane is only notified in the event of the tracked counts in the data plane exceed a certain threshold.

VI. Concluding Remarks

In this paper, we discuss the consequences of the multi-pipeline property of today's commodity programmable switching ASICs. By using the ubiquitous TCP `conntrack` as an example, we illustrate how and why stateful applications can easily misbehave on multi-pipeline switches. Then, we highlight that most in-network applications when deployed on multi-pipeline switches, cannot function as desired; therefore, limiting their practicality in actual deployment. Lastly, we present several potential workarounds that can be applied to some of the affected in-network applications; we also discuss the inevitable trade-offs they would impose on the overall network performance.

While the prospects of having in-network applications to operate entirely in the data plane presents promising gains, the issues highlighted in this paper inevitably presents a setback. To the best of our knowledge, there exists *no* universal solution to realize stateful applications entirely in multi-pipeline data planes. To that end, we raise the following questions:

Does an algorithm that synchronize pipeline states entirely in the data plane exists? Aside from the workarounds outlined, are there any more efficient ways to synchronize data plane states entirely in the data plane? If possible, what is the consistency guarantee that such approach can provide?

What should be the proper language constructs and semantics for multi-pipeline architectures? Current domain specific languages, such as P4 [21] and NPL [22], are designed to be target independent and are pipeline-agnostic. Should an extended language with pipeline awareness be developed? If so, what are the necessary constructs and proper semantics for stateful packet processing?

Can alternative hardware designs that provide coherent memory regions across pipelines be possible? The issue with multi pipeline switches stem from their disjoint nature. If there exists shared memory across pipelines, developing stateful applications that operate across multiple pipelines would be trivial. Is it expected that upcoming switch architectures (e.g., dRMT [51]) will address this issue? Is it feasible to develop an alternative hardware design that can provide some coherent and shared memory across pipelines?

Acknowledgement

This work is supported by the National Research Foundation, Prime Minister's Office, Singapore under its Corporate Laboratory@University Scheme, National University of Singapore, and Singapore Telecommunications Ltd.

REFERENCES

[1] P. G. Kannan and M. C. Chan, "On Programmable Networking Evolution," *CSI Transactions on ICT*, vol. 8, no. 1, pp. 69–76, Mar 2020.
[2] R. Ben Basat, X. Chen, G. Einziger, and O. Rottenstreich, "Designing Heavy-Hitter Detection Algorithms for Programmable Switches," *IEEE/ACM ToN*, vol. 28, no. 3, 2020.
[3] C. H. Song, P. G. Kannan, B. K. H. Low, and M. C. Chan, "FCM-Sketch: Generic Network Measurements with Data Plane Support," in *ACM CoNEXT*, 2020.
[4] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "HULA: Scalable Load Balancing Using Programmable Data Planes," in *ACM SOSR*, 2016.
[5] K.-F. Hsu, P. Tammana, R. Beckett, A. Chen, J. Rexford, and D. Walker, "Adaptive Weighted Traffic Splitting in Programmable Data Planes," in *ACM SOSR*, 2020.
[6] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh, and M. Yu, "HPCC: High Precision Congestion Control," in *ACM SIGCOMM*, 2019.
[7] Y. Zhou, C. Sun, H. H. Liu, R. Miao, S. Bai, B. Li, Z. Zheng, L. Zhu, Z. Shen, Y. Xi, P. Zhang, D. Cai, M. Zhang, and M. Xu, "Flow Event Telemetry on Programmable Data Plane," in *ACM SIGCOMM*, 2020.
[8] X. Z. Khooi, L. Csikor, D. M. Divakaran, and M. S. Kang, "DIDA: Distributed In-Network Defense Architecture Against Amplified Reflection DDoS Attacks," in *IEEE NetSoft*, 2020.
[9] M. Zhang, G. Li, S. Wang, C. Liu, A. Chen, H. Hu, G. Gu, Q. Li, M. Xu, and J. Wu, "Poseidon: Mitigating Volumetric DDoS Attacks with Programmable Switches," in *NDSS*, 2020.
[10] H. T. Dang, M. Canini, F. Pedone, and R. Soulé, "Paxos made switch-y," *ACM SIGCOMM CCR*, vol. 46, no. 2, May 2016.
[11] J. Li, E. Michael, and D. R. K. Ports, "Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control," in *ACM SOSP*, 2017.
[12] A. Sapio, M. Canini, H. Chen-Yu, J. Nelson, P. Kalnis, K. Changhoon, A. Krishnamurthy, M. Moshref, D. Ports, and P. Richtarik, "Scaling Distributed Machine Learning with In-Network Aggregation," in *USENIX NSDI*, 2021.
[13] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "NetCache: Balancing Key-Value Stores with Fast In-Network Caching," in *ACM SOSP*, 2017.
[14] J. Li, J. Nelson, E. Michael, X. Jin, and D. R. K. Ports, "Pegasus: Tolerating Skewed Workloads in Distributed Storage with In-Network Coherence Directories," in *USENIX OSDI*, 2020.
[15] S. Wang, C. Sun, Z. Meng, M. Wang, J. Cao, M. Xu, J. Bi, Q. Huang, M. Moshref, T. Yang, H. Hu, and G. Zhang, "Martini: Bridging the Gap between Network Measurement and Control Using Switching ASICs," in *IEEE ICNP*, 2020.
[16] J. Kučera, D. A. Popescu, H. Wang, A. Moore, J. Kořenek, and G. Antichi, "Enabling Event-Triggered Data Plane Monitoring," in *ACM SOSR*, 2020.
[17] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica, "NetChain: Scale-Free Sub-RTT Coordination," in *USENIX NSDI*, 2018.
[18] D. R. K. Ports and J. Nelson, "When Should The Network Be The Computer?" in *ACM HotOS*, 2019.
[19] X. Z. Khooi, L. Csikor, J. Li, M. S. Kang, and D. M. Divakaran, "Revisiting Heavy-Hitter Detection on Commodity Programmable Switches," in *IEEE NetSoft*, 2021.
[20] Intel, "$P4_{16}$ Intel Tofino Native Architecture - Public Version," Application Note, https://bit.ly/3mtpzJz, Mar 2021 [Accessed: Apr 2021].
[21] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming Protocol-Independent Packet Processors," *ACM SIGCOMM CCR*, vol. 44, no. 3, p. 87–95, Jul. 2014.
[22] NPLang.org, "NPL Specifications," https://bit.ly/3cROvaE, [Accessed: Mar 2021].
[23] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports, "Just Say No to Paxos Overhead: Replacing Consensus with Network Ordering," in *USENIX OSDI*, 2016.
[24] N. Gebara, A. Lerner, M. Yang, M. Yu, P. Costa, and M. Ghobadi, "Challenging the Stateless Quo of Programmable Switches," in *ACM HotNets*, 2020.
[25] The P4 Architecture Working Group, "P4_16 Portable Switch Architecture (PSA)," https://bit.ly/3sTDN9e, Mar 2018.
[26] Broadcom, "Trident3-X7 / BCM56870 Series," https://bit.ly/3wvZ4b9, [Accessed: Feb 2021].
[27] Broadcom, "Trident4 / BCM56880 Series," https://bit.ly/2R3cP0E, [Accessed: Feb 2021].
[28] Intel, "Intel® Tofino™," https://bit.ly/3sY7beD, [Accessed: Feb 2021].
[29] Intel, "Intel® Tofino™ 2," https://bit.ly/3oer5hX, [Accessed: Feb 2021].
[30] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN," *ACM SIGCOMM CCR*, vol. 43, no. 4, p. 99–110, Aug. 2013.
[31] A. Arcilla and T. Palmer, "Broadcom Trident 3 Platform Performance Analysis," Technical validation, https://bit.ly/2PxCAGe, May 2021.
[32] "A Deeper Dive Into Barefoot Networks Technology." https://bit.ly/3iKB5i3, 2017.
[33] "[PSA] Document effect of multiple 'pipelines' on Register extern?" https://github.com/p4lang/p4-spec/issues/353, [Accessed: Feb 2021].
[34] B. Wheeler, "Tomahawk 4 Switch First TO 25.6Tbps," Microprocessor Report - Newsletter, https://bit.ly/3dFePnn, Dec. 2019.
[35] N. Farrington, E. Rubow, and A. Vahdat, "Data Center Switch Architecture in the Age of Merchant Silicon," in *IEEE HOTI*, 2009.
[36] Cisco, "Cisco Silicon One Product Family," White paper, https://bit.ly/39OI9Xx, 2021.
[37] S. Iyer and N. W. McKeown, "Analysis of the Parallel Packet Switch Architecture," *IEEE/ACM ToN*, vol. 11, no. 2, 2003.
[38] p4.org, "Implementing A Basic Stateful Firewall," https://github.com/p4lang/tutorials/tree/master/exercises/firewall, [Accessed: Mar 2021].
[39] B. H. Bloom, "Space/Time Trade-Offs in Hash Coding with Allowable Errors," *Commun. ACM*, vol. 13, no. 7, Jul. 1970.
[40] R. Sommer and A. Feldmann, "NetFlow: Information Loss or Win?" in *ACM IMW*, 2002.
[41] X. Chen, S. Landau-Feibish, M. Braverman, and J. Rexford, "BeauCoup: Answering Many Network Traffic Queries, One Memory Update at a Time," in *ACM SIGCOMM*, 2020.
[42] R. Joshi, T. Qu, M. C. Chan, B. Leong, and B. T. Loo, "BurstRadar: Practical Real-Time Microburst Monitoring for Datacenter Networks," in *ACM APSys*, 2018.
[43] X. Chen, S. L. Feibish, Y. Koral, J. Rexford, O. Rottenstreich, S. A. Monetti, and T.-Y. Wang, "Fine-Grained Queue Measurement in the Data Plane," in *ACM CoNEXT*, 2019.
[44] P4.org Applications Working Group, "In-band Network Telemetry (INT) Dataplane Specification Version 2.1," May 2020.
[45] R. Ben Basat, S. Ramanathan, Y. Li, G. Antichi, M. Yu, and M. Mitzenmacher, "PINT: Probabilistic In-Band Network Telemetry," in *ACM SIGCOMM*, 2020.
[46] C. H. Benet, A. J. Kassler, T. Benson, and G. Pongracz, "MP-HULA: Multipath Transport Aware Load Balancing Using Programmable Data Planes," in *ACM SIGCOMM NetCompute*, 2018.
[47] K.-F. Hsu, R. Beckett, A. Chen, J. Rexford, and D. Walker, "Contra: A Programmable System for Performance-aware Routing ," in *USENIX NSDI*, 2020.
[48] Z. Liu, Z. Bai, Z. Liu, X. Li, C. Kim, V. Braverman, X. Jin, and I. Stoica, "DistCache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching," in *USENIX FAST*, 2019.
[49] Z. Yu, Y. Zhang, V. Braverman, M. Chowdhury, and X. Jin, "NetLock: Fast, Centralized Lock Management Using Programmable Switches," in *ACM SIGCOMM*, 2020.
[50] H. Zhu, Z. Bai, J. Li, E. Michael, D. R. K. Ports, I. Stoica, and X. Jin, "Harmonia: Near-Linear Scalability for Replicated Storage with in-Network Conflict Detection," *VLDB Endow.*, vol. 13, no. 3, p. 376–389, Nov. 2019.
[51] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, A. Orda, and T. Edsall, "dRMT: Disaggregated Programmable Switching," in *ACM SIGCOMM*, 2017.