PACKETSCOPE: MONITORING THE PACKET LIFECYCLE INSIDE A SWITCH

Ross Anthony Poggi Teixeira

A MASTER'S THESIS

PRESENTED TO THE FACULTY

OF PRINCETON UNIVERSITY

IN CANDIDACY FOR THE DEGREE

OF MASTER OF SCIENCE IN ENGINEERING

RECOMMENDED FOR ACCEPTANCE BY

THE DEPARTMENT OF COMPUTER SCIENCE

Adviser: Jennifer Rexford

June 2019

ABSTRACT

As modern switches become increasingly more powerful, flexible, and programmable, network operators have an ever greater need to monitor their behavior. Many existing systems provide the ability to observe and analyze traffic that *arrives* at switches, but do not provide visibility into the experience of packets *within* the switch. To fill this gap, we present PacketScope, a network telemetry system that lets us peek inside network switches to ask a suite of useful queries about how switches modify, drop, delay, and forward packets. PacketScope gives network operators an intuitive and powerful Spark-like dataflow language to express these queries. To minimize the overhead of PacketScope on switch metadata, our compiler uses a “tag little, compute early” strategy that tags packets with metadata as they move through the switch pipeline, and computes query results as early as possible to free up pipeline resources for later processing. PacketScope also combines information from the ingress and egress pipelines to answer aggregate queries about packets dropped due to a full queue.

ACKNOWLEDGEMENTS

A huge thanks to my advisor Jennifer Rexford for your persistent and invaluable guidance at every step and detail of this work. Thanks to Arpit Gupta and Rob Harrison for pioneering the Sonata system which this work is based on, and for your valuable support in understanding P4 and the Sonata codebase and drafting the contents of this work. Thanks to David Walker for your feedback and suggestions. Thanks also to Noa Zilberman for her guidance on the conference submission of this work. You all made this work possible!

TABLE OF CONTENTS

PacketScope: Monitoring the Packet Lifecycle Inside a Switch

Ross Teixeira

1 INTRODUCTION

Network monitoring is crucial to ensuring high availability and performance in modern networks. At the core of these networks lies a set of switches, which are responsible for delivering data packets and enforcing network policies such as load balancing, access control, and attack detection. Switches themselves are complex devices consisting of multiple pipelines for processing packets, diverse memory for storing different kinds of state, multiple queues for buffering and forwarding packets, and complex logic for implementing network policies. However, today's network operators have limited visibility into the data planes of these switches.

When application performance issues arise, the network and, more specifically, the switches are often to blame [7, 8]. For example, a switch might have incorrect forwarding rules that cause packets to never reach their intended destination. If a switch's buffers become congested, flows will start to experience latency, which can severely impact latency-sensitive applications such as video streaming or gaming. Finally, attacks on network devices can get past switches that are not filtering packets properly; network attacks like a DDoS can completely fill switch buffers and significantly disrupt the network.

A new generation of switches [5] with a protocol independent switch architecture (PISA) allow network operators to write custom packet-processing code in languages like P4 [4]. These programmable switches give network operators much greater flexibility over packet processing, which can lead to more efficient network design and greater insight into network performance. However, this advancement in switch design comes with the risk of introducing bugs in a switch's processing, either due to programmer error or compilers that contain bugs. As the adoption of programmable switches rises, it becomes increasingly important to monitor the processing that occurs *within* the switches to ensure that they are free from incorrect programs, compiler bugs, and hardware errors.

In addition to these new error conditions unique to programmable switches, network operators must still monitor their networks for traditional network events such as congestion, failure, or cyberattack. For all of these events, examining how the switch internally processes individual packets significantly aids in their detection. For example, to detect incorrect forwarding behavior such as a black hole, one could query for the counts of packets being forwarded out each output port to observe whether any ports are not sending out any packets. To determine when a switch is experiencing congestion, a network operator could ask about the size of the queue when a packet enters or leaves the queue. To detect attacks that target an end host's software, such as an SSH exploit, network operators can ask whether packets which should have been dropped by an access control list (ACL) rule (such as inbound SSH connections) were instead forwarded out.

Recent telemetry systems for programmable switches support queries written in a dataflow programming model [6], a powerful and familiar language for expressing computation on a stream of incoming packets [9, 15]. However, these systems only operate on the packets as they *enter* the switch, and they ignore packet processing done by the switch itself. They are useful for detecting network-level attacks, but their limited expressiveness does not allow operators to analyze a switch's internal processing.

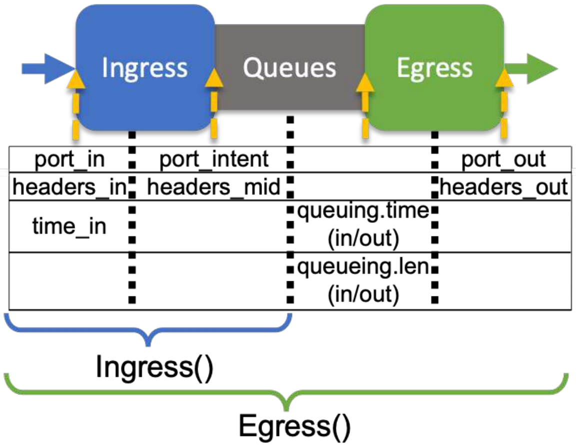


Figure 1: PacketScope's PISA pipeline model and locations where the values of query fields become known.

In this paper, we present PacketScope, a network telemetry system capable of answering queries about internal switch processing using recent advances in programmable switches. PacketScope provides dataflow constructs that allow network operators to reason about how a switch modifies, drops, and delays. In particular, we:

- Enable monitoring of packets at both the *ingress* and *egress* pipelines, and for collecting aggregate statistics about packets *lost* due to queue occupancy;
- Compile queries to PISA switches by tagging each packet with relevant metadata about its journey through the switch and computing statistics as early in packet processing as possible to minimize overhead;
- Overcome limitations in switch programmability to monitor queuing loss by introducing a hybrid switch-controller solution that joins and synchronizes traffic counts from the ingress and egress pipelines; and
- Develop an initial prototype as an extension to the Sonata codebase [1] and evaluate an independent loss query implementation.

In Section 2, we describe the specific language extensions that enable monitoring queries over both the ingress and egress pipelines of a switch. In Section 3, we describe how we compile those extensions to PISA hardware and in Section 4, we evaluate our initial prototype. Finally, we discuss future and related work in Section 5 and Section 6, respectively, and conclude in Section 7.

2 QUERYING THE FULL PACKET LIFE CYCLE

In this section, we first explain the declarative, dataflow programming model for network telemetry queries. We then describe the extensions to the packet tuple abstraction that PacketScope introduces to represent the packet's experience within the switch. We demonstrate example queries for monitoring modifications, ACL drops, and queuing loss. Finally, we introduce a `loss` operator for reasoning about queuing loss.

2.1 Telemetry as Dataflow Queries

Several PISA-based telemetry systems let network operators express declarative queries that treat each packet as a tuple. Two such systems, Sonata [9] and Marple [15], provide an abstraction similar to the dataflow programming paradigm used by Apache Spark [20]. These languages operate on streams of incoming tuple data, where each tuple represents a packet header vector. Queries can then apply `map`, `filter`, `groupby`, and `reduce` operations to evaluate expressions, filter, and aggregate data, respectively, on the set of incoming tuples.

While dataflow programming is a generally expressive model for common network telemetry queries, existing solutions have limitations that prevent them from analyzing a packet's experience *within* the switch. Sonata's tuples represent packets as they appear on arrival at the switch, and thus queries cannot reason about header modifications, queuing delay, or packet loss. Marple tuples include information about queuing, but cannot track modifications or dropped/lost packets without monitoring at multiple switches or sending copies of all packets to a central controller. With PacketScope, we can avoid this network overhead by extending Sonata's tuple abstraction to account for the packet's entire journey through the switch, not just the ingress portion.

2.2 Extending the Tuple Abstraction

To support queries about the packet life cycle, we expand the tuple abstraction to include fields that capture the packet's experience at various stages of processing. Figure 1 shows the full set of fields and their locations. When the packet first enters the switch and is parsed, the switch gets information about `headers_in` (the packet's initial header fields), `port_in` (the port the packet arrived at), and `time_in` (the timestamp when the packet arrived). The packet then moves through ingress processing, where its headers may be modified, and it will either have its intended forwarding behavior set, or be marked for drop because it matched an ACL rule. After ingress processing finishes, `headers_mid` defines the packet's headers after any ingress modifications, while `port_intent` refers to the packet's intended forwarding behavior, which will either be (i) its output port, (ii) a special value for behavior like mirroring or multicasting, or (iii) a `-1` value that indicates the packet is intended to be dropped.

Note, we define that a packet has "finished" processing in a pipeline when (i) there is no more processing to do, or (ii) the packet is marked for drop. Thus, for dropped packets, `headers_mid` represents the packet's headers at the time it was marked for drop.

If the packet is not marked for drop, it then attempts to enter the queue. For now we assume the queue has space, so the packet enters the queue, and eventually the packet is dequeued and enters egress processing. At this point, the switch knows the packet's experience in the queue: `queuing.time_in/out` and `queuing.len_in/out`, the times the packet entered and exited the queue and the size of the queue at those times. The packet then enters egress processing, where it may undergo more modifications or be marked for drop. After the egress pipeline finishes processing, the `header_out` values are known as well as `port_out`, the port the packet is sent out. Similar to ingress processing, if the packet is marked for drop during egress processing, `port_out = -1` and `headers_out` represents the packet's headers at the time it was marked.

2.3 Querying Both Ingress and Egress Tuples

Each query in PacketScope must begin by defining a stream of tuples for that query to operate on. To motivate the choice of tuple streams provided by PacketScope, we first consider two alternative approaches. We could provide a single tuple that contains all fields, similar to prior work, but a single tuple does not allow queries to specify whether they observe packets at the ingress or egress pipeline, which is important when dealing with queuing loss. We could provide four tuple types: start/end of ingress and start/end of egress, but the start and end of each pipeline are redundant because packets that start a pipeline always reach the end of that pipeline.

Thus, PacketScope provides two tuple streams that queries can operate on: `ingress()` and `egress()`. Queries on the `ingress()` stream operate on all packets that are seen by the switch, and their tuples contain:

```
(headers_in, headers_mid,
port_in, port_intent,
time_in)
```

Queries on the `egress()` stream operate on all packets that reach egress processing, and their tuples contain:

```
(headers_in, headers_mid, headers_out,
port_in, port_intent, port_out,
time_in, queuing.time_[in/out],
queuing.len_[in/out])
```

Note that while each tuple type defines a single pipeline, the queries might be *compiled* to either the start or end of that pipeline, depending on the fields used in each query and the resource constraints of the switch, as discussed in §4.1. With these streams of tuples, PacketScope's language enables four types of queries about the life of packets in the switch: (i) packet modifications, (ii) access control list drops, (iii) queuing delay, and (iv) queuing loss. Table 1 showcases example queries targeting each.

2.4 Aggregate Queries over Queuing Loss

We now handle the special case of queuing loss. Consider the strawman approach of introducing a third tuple type for "lost" packets, that produces a tuple for each packet lost due to a full queue. Unlike ACL drops, the switch does not provide a programmable hook for analyzing a packet when it attempts to enter a full queue, as this occurs outside the programmable pipelines. This means that to detect queuing loss, we must somehow observe each packet that appears at ingress processing, but never reaches egress processing even after accounting for possible queuing delay.

An expensive option would be to forward each packet at ingress and egress to a central controller for analysis. Alternatively, the switch could keep state about each packet seen at ingress and at egress, and later a central controller could compare the per-packet

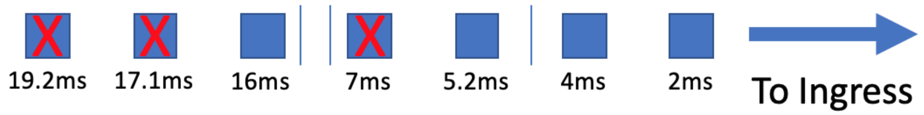


Figure 2: Packets assigned to epochs based on arrival time where `epoch_ms` == 5ms, where the vertical lines represent epoch boundaries. A red 'X' denotes a dropped packet.

Type	Query

Mods	`.egress()`
	`.filter(ipv4.srcIP_in != ipv4.srcIP_out)`
	`.map((ipv4.srcIP_in) => 1)`
	`.reduce(keys=(ipv4.srcIP_in), func=sum)`

ACL drop	`.ingress()`
	`.filter(port_intent == -1)`
	`.map((ipv4.srcIP_in) => 1)`
	`.reduce(keys=(ipv4.srcIP_in), func=sum)`

Delay	`.egress()`
	`.filter(queue.len_out > Th)`
	`.map((ipv4.srcIP_in) => 1)`
	`.reduce(keys=(ipv4.srcIP_in), func=sum)`

Loss	`.ingress()`
	`.filter(tcp.dstPort_in == 80)`
	`.lost([ipv4.srcIP_in], 20ms)`

Table 1: Example PacketScope queries.

state. It would be too expensive to store state for every packet the switch sees, but it *is* possible to keep aggregate counts—for example, packet counts grouped by IP prefixes—in both pipelines.

PacketScope provides a special operator for tracking packets lost to the queue:

```
.lost(groupby_fields, epoch_ms)
```

which computes counts of lost packets grouped by the specified fields. We place two restrictions on using `.lost()` in a query:

- Queries with `lost()` can only operate on `ingress` tuples, as by definition, lost packets never enter egress processing.
- The aggregate operator `.reduce` is not allowed before `.lost()`, but simple operators (`.map`, `.filter`) are allowed.

Next, we define the time windows that `.lost()` counts are aggregated over. A simple strawman would be to read the counts at ingress and egress in absolute time increments, but due to queuing delay, the counts at ingress and egress at any instant would differ by the number of packets currently in the queue. Instead, we use the *arrival* time of packets as the epoch boundary. For example,

```
.lost([ipv4.srcIP_in], 20ms)
```

would report how many packets experience queuing loss that arrive in 20ms windows. Tuples returned by `.lost()` contain `(epoch#, count, groupby_fields)`. Figure 2 shows an example of epoch timings. Note this means that the switch needs to store counts for the previous and next epoch at any time, as discussed later in §3.3.

3 COMPILING PACKET LIFE CYCLE QUERIES

In this section, we describe the PacketScope compiler, and how we overcome two challenges of compiling packet life cycle queries to a switch: (i) where to place state and computation and (ii) how to handle queuing loss.

3.1 The PacketScope Compiler

The PacketScope compiler takes as input (i) a set of queries that the network operator writes and (ii) P4 code for the switch forwarding logic. The compiler distributes the query operators to be executed at different locations in the pipeline, depending on (i) the tuple type used, (ii) the order of operators, and (iii) available switch resources. For example, a `.filter(port_in == 2)` operator could be applied at the start of ingress processing, even if applied to `egress` tuples. The compiler then integrates the portion of the queries that can be executed at the switch with the forwarding logic to produce a single P4 program, which is loaded onto the switch.

3.2 Tag Little, Compute Early

The first challenge in compiling PacketScope queries is deciding where to place the query logic in the pipeline. For example, take the "Modification" query in Table 1, which operates on a stream of `egress()` tuples and filters for packets whose source IP was modified during switch processing. The query must be processed at the egress pipeline by definition, but it also needs access to the packet's `ipv4.srcIP_in` when the packet arrived at the switch, before any modifications. To solve this, we *tag* the packet with metadata that includes its initial source IP when it arrives at the switch. When the packet reaches the end of egress processing, the switch compares its current source IP to the tagged metadata to complete the `filter` operation.

However, excessive packet tagging uses up valuable state in the Packet Header Vector (PHV), where headers and other metadata are stored, that could be used by other queries or other switch functionality. Thus, we want to minimize the additional state added to the PHV, and "free" that state when it is no longer needed. Since PHV state is pre-allocated by the P4 compiler, for our purposes, "freeing" state in the PHV means allowing future operators or switch processing to reuse the space allocated to prior operators/processing when that state is no longer needed.

To demonstrate our "tag little, compute early" strategy, consider the following two queries. The first query counts the number of packets whose destination IPs are modified during *ingress* processing, by their destination IP at the start of ingress processing:

```
.ingress()
.filter(ipv4.dstIP_in != ipv4.dstIP_mid)
.reduce(keys=(ipv4.dstIP_in), func=sum)
.filter(count > T)
```

The second query counts packets on which the switch sets an Early Congestion Notification (ECN) bit during *egress* processing, by their destination IP at the start of egress processing:

```
.egress()
.filter(ipv4.ecn_mid != 2 && ipv4.ecn_out == 2)
.reduce(keys=(ipv4.dstIP_mid), func=sum)
.filter(count > T)
```

The first query must store each packet's initial destination IP in the PHV, using 32 bits. The second query must store the packet's destination IP at the start of egress processing, as it must wait until the end of egress processing to know whether the ECN bit is eventually set. Naively, storing `ipv4.dstIP_mid` would use another 32 bits; however, because the first query's initial IP is no longer needed, the second query can reuse that space in the PHV.

In this case, the fields are the same size, but the strategy also works when the PHV space being reused is larger than the fields that demand it. For example, PHV space used for an IP address can be reused by multiple other fields, such as a TTL value and a source port. In this case, the compiler must keep track of the bit indices within the original PHV space used by each new field.

Together, our "tag little" and "compute early" strategies reduce the PHV overhead imposed by queries. The PacketScope compiler tags packets with relevant query fields in their PHV when they become available, and executes operators as early as possible so their PHV space can be reused.

3.3 Monitoring Queuing Loss in Epochs

Our second major challenge is monitoring queuing loss. As discussed in §2.4, the switch may not provide a programmable hook into packets that are dropped due to a full queue. This makes it difficult for PacketScope to track individual packets which experience queuing loss. Fortunately, it *is* feasible to track aggregate *counts* of packets lost due to queuing. Our solution stores these counts in registers on both the ingress and egress pipelines for a central controller to retrieve and compare later.

To compile `.lost` queries, we take inspiration from Sonata's ability to join the results of two queries together. The query on queuing loss in Table 1 can be expressed as shown below:

```
ingress()
.map((ipv4.srcIP_in) => count=1)
.reduce(sum)
.join((egress()
      .map((ipv4.srcIP_in) => count=1)
      .reduce(sum)),
       func='diff',
       window='arrival', epoch_ms=5ms)
```

The queries track packet counts per source IP at the ingress and egress pipelines, respectively, and `.join` computes their difference.

In order to handle epochs, PacketScope tags each packet with an epoch number when it arrives at the switch, computed by $\lfloor \frac{time\_in}{epoch\_ms} \rfloor$. For certain epochs (powers of 2), this is easy to compute in the data plane using a bitshift. At each pipeline, the switch then stores `(epoch#, groupby_fields) -> (epoch#, count)` in a d-stage hash table for each IP it observes, and updates the count if it has already been initialized. Our model assumes FIFO processing with no reordering in the queuing phase (e.g., due to priority queuing), so that when the first packet from epoch `x` is dequeued, no other packets from epoch $x - 1$ will be seen by egress processing. Thus, when the egress pipeline first observes a packet from a new epoch, it can alert the central controller to pull results from the switch. In case no packets arrive during an epoch, the central controller can pull results from the switch after no additional packets from the previous could be waiting in the queue: the start time of the current epoch + the max ingress processing + queuing delay. With our FIFO assumption and knowledge of the target architecture's clock speed and queue capacity, we can compute the max ingress processing + queuing delay in advance.

To expire old data, we use the fact that we store the epoch number of each register entry in addition to its contents. If the insertion for a new packet collides with an entry that has an epoch number less than the previous epoch, we assume that the central controller had sufficient time to query the switch for those counts and the data can be overwritten.

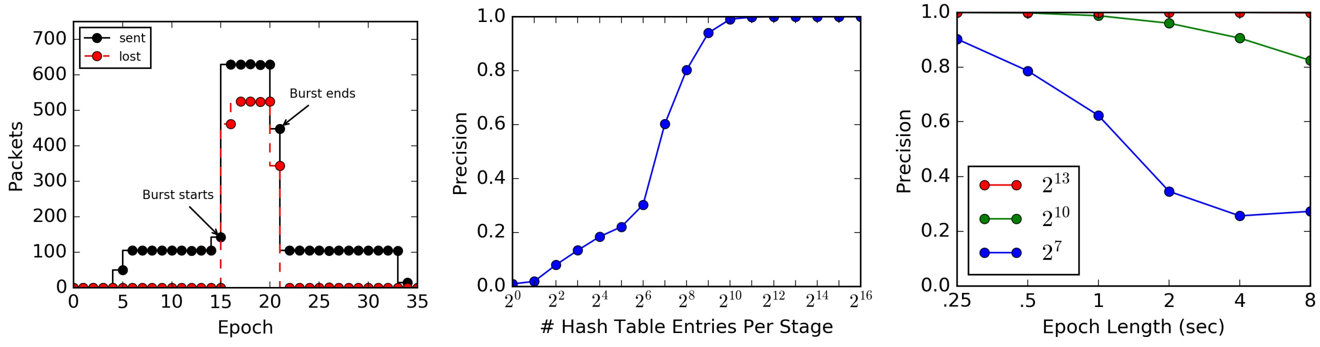
Finally, we handle the case of collisions in the d-stage hash table by adding an additional stage to both the ingress and egress pipelines. If a packet encounters a collision in all d stages of the hash table, it adds a count in the $d+1$ stage to a register according to its epoch in each pipeline. Instead of hashing into the register according to the `groupby_fields` specified by the query, counts in the $d+1$ stage are indexed by the packet's epoch number. This allows PacketScope to count the total packet losses on the switch even when the switch's available memory is insufficient to store all of the counters without collision when indexed by the `groupby_fields`.

4 EVALUATION

We present an initial prototype for PacketScope by extending the Sonata streaming network telemetry system [1], including the query language, compiler, and emitter[9]. So far, we have modified or added approximately 820 lines to the 16, 000 lines in the Sonata codebase. This addition extends Sonata to support both `ingress()` and `egress()` tuple streams and our "tag little, compute early" strategy of compiling query functionality into a P4 packet processing pipeline. We also include in our prototype a manually compiled loss query for evaluation.

4.1 Packet Loss Query

We implement the packet loss query in Table 1, which counts lost HTTP packets by source IP. It follows the Sonata model, and is written in 950 lines of P4 code and 240 lines of Python for the emitter. We used a four-stage hash table implementation with a fifth stage for resolving collisions per epoch on ingress and egress. For our first experiment, we have 2^7 rows per stage and an epoch size of 1 second. We ran this query on the BMv2 software switch configured with a queue length of 100 packets and a dequeuing rate of 100 packets/sec. The switch then processed a synthetic packet trace designed to fill the queue, without drops, for a 30-second



(a) Loss counts reported from a bursty flow.

(b) Precision with epoch length fixed at 1 second.

(c) Precision of epoch lengths with different # of hash table entries per stagef.

Figure 3: PacketScope prototype evaluation.

period. We then injected an additional 500 packet/sec burst for a 6-second period. Figure 3a shows that our query effectively detected the lost packets that resulted from an overwhelmed queue.

4.2 Query Precision

We then evaluate the *precision* of our loss query. The *precision* of a loss query is defined as $\frac{\text{\# groupby losses}}{\text{\# total losses}}$, the fraction of unique *groupby* keys (in this case, source IPs) seen by the switch whose loss counts are maintained and reported at the end of each epoch, compared to the total number of lost packets in that epoch. Precision decreases when the hash tables fill, causing packets that experience hash collisions to only be counted in the *total* number of lost packets stored in the *d+1* stage for that epoch. We then configured the BMv2 switch with a 40 packet buffer and dequeuing rate of 1000 packets/sec, and replayed a CAIDA trace [2] through the switch for 60 seconds at ~1700 packets/sec. Figures 3b and 3c show the precision of our query as we vary (i) the number of rows in each stage of the hash table and (ii) the duration of an epoch, respectively. As expected, our precision increases as the number of rows in each stage of the hash table increases (fewer collisions) and decreases as the epoch duration increases (more collisions).

Because we are evaluating PacketScope on the slower BMv2 software switch, we focus on a smaller packet rate (thus, a small number of flows) and larger epoch lengths. When compiling to a hardware switch, this reverses: a much faster switch can support much shorter epochs, and this would help achieve similar precision with a much larger number of flows. In addition, loss queries that group traffic by coarser keys (e.g., IP prefix, rather than IP address) would reduce the memory requirements for maintaining state.

5 DISCUSSION AND FUTURE WORK

Integration of queries with user P4 program: PacketScope monitors a switch's packet processing, so naturally it must integrate its own queries with the user's existing P4 code. There are three key ways in which this occurs. First, the execution flow of a P4 program is defined in a `control` code block, and PacketScope must augment this block to insert its query processing before/after existing processing. Second, the existing P4 code contains a custom parser that defines the packet's headers; by extracting this parser, PacketScope can allow queries to use any custom headers defined by the user. Finally, we have described the switch as "marking" a packet for drop, but in reality, when a packet matches an ACL rule, a `drop()` action is called that may immediately terminate processing and drop the packet. To account for such targets, PacketScope modifies the user program to override the `drop()` action to set a "mark" bit in the PHV, which is then read implicitly when a query checks if `port_intent/out == -1`.

Query compilation with switch resource constraints: PISA switches are limited in the number of stages per pipeline, the number of instructions that can be executed per stage, the size of the PHV, and the amount of register memory available in each stage [5, 9]. Each of these constraints reduces the number of query operators that can be executed on the switch, and fitting these constraints becomes increasingly difficult when running multiple queries simultaneously while integrating with existing switch processing. Sonata uses these constraints as input to an integer linear program (ILP) [9], and solves it to find an optimal partitioning of queries into the data plane of the switch that minimizes communication with the central controller. PacketScope can also use this ILP formulation, except the ILP must now include constraints on the ingress and egress pipelines separately, and we would solve for the optimal partitioning of queries based on their respective pipelines, metadata requirements, and the division of the existing switch processing among the ingress and egress pipelines.

Queries with multiple pipelines: Our current switch model assumes FIFO processing with a single ingress and egress pipeline shared among all ports. However, switches often contain multiple distinct pipelines that each process packets for a subset of the ports on the switch. For example, consider a query fragment that generates a traffic matrix of in-out port pairs:

```
.egress()
.map((port_in, port_out) => 1)
.reduce(keys=(port_in, port_out), func=sum)
.filter(count > T)
```

In a single-pipeline switch, each in-out port pair has a single count at the egress pipeline, and it is easy to detect when this count exceeds a threshold. But with multiple pipelines, detecting when the count exceeds the threshold becomes a distributed heavy-hitter problem, in which multiple counts may individually fall below the threshold, but whose sum exceeds it. This problem has attracted research in a network-wide setting [11], and applying these techniques *within* a switch is an interesting future direction.

Network-wide queries: Much of PacketScope's query language can be extended to support network-wide queries that abstract the network as "one big switch." In this abstraction, the in-out ports of the abstract switch are the edge switches of the network, and the big-switch's processing accounts for all of the switches within the network. "Tagging" a packet means adding headers to the packet as it traverses the network, rather than stripping the tags before the packet leaves the switch. This design could be combined with other network telemetry systems like Path Queries [16] to reason about the paths packets take through the network.

6 RELATED WORK

Dataflow for Telemetry Queries: Several recent systems express network telemetry queries in a dataflow language. Both Sonata [9] and Marple [15] partition queries between a switch data plane and a central processor, but they only observe packets when they *arrive* at the switch. PacketScope enables queries about processing that occurs within the switch by exposing both ingress *and* egress packet streams, which were not available in either of the prior solutions.

While we have contributed a portion of PacketScope as an extension to the Sonata codebase, the techniques we propose that enable queries about processing *within* the switch are novel and not limited to implementation in Sonata. Since the prototype is designed as an extension of the Sonata codebase, PacketScope shares all of the features and limitations of Sonata, including the need to recompile whenever the set of input queries changes. In addition, PacketScope extends Sonata's query language to enable queries about the egress packet stream and packet loss.

Active measurement: Some tools use active probing to monitor and detect issues in the network [8, 18]. However, active measurement systems only track synthetic probes, whereas we want to monitor the modifications and drop/loss behavior that real network traffic experiences within the switches themselves.

Passive measurement: Some passive systems work by forwarding copies of all packets to a central server (or set of servers) [10, 13], which introduces too muh overhead. Everflow [21] and dShark [19] require operators to filter a limited set of IPs to monitor, whereas PacketScope can support all traffic at line rate. Other systems require control of end hosts [3, 12, 17], while PacketScope does not.

Handling loss: Some systems deal specifically with detecting lost packets, but they rely on coordination between multiple network members. LossRadar [14] requires monitoring at multiple switches in order to detect lost packets, and does not distinguish between link failures and queuing losses, while we focus on detecting queuing losses. 007 [3] also does not make this distinction, and, in addition, it assumes that the operator has control of the end hosts in the network (such as a datacenter) and requires end hosts to participate in order to track losses, while PacketScope can track losses at a switch independent of the surrounding network.

7 CONCLUSION

PacketScope fills an essential gap in network telemetry systems by peeking *inside* modern programmable switches. It offers rich insight into the life cycle of packets inside a switch, where they could experience packet modifications, ACL drops, and queuing delay and loss. PacketScope compiles and integrates dataflow telemetry queries with existing switch processing and employs a "tag little, compute early" compilation strategy to minimize query overhead. We also allow a switch to track the properties of packets lost to full queues by monitoring at both the ingress and egress pipelines.

REFERENCES

[1] 2018. Sonata Repository. https://github.com/Sonata-Princeton. (2018).

[2] 2019. The CAIDA UCSD Anonymized Internet Traces 2019 Dataset. http://www.caida.org/data/passive/passive_dataset.xml. (2019).

[3] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 2018. 007: Democratically Finding the Cause of Packet Drops. In *USENIX Conference on Networked Systems Design and Implementation*. 419–435.

[4] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-independent Packet Processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (July 2014), 87–95.

[5] 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*. 99–110.

[6] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113.

[7] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. 2016. Why Does the Cloud Stop Computing?: Lessons from Hundreds of Service Outages. In *ACM Symposium on Cloud Computing*. 1–16.

[8] Chuanxiong Guo, Hua Chen, Zhi-Wei Lin, Varugis Kurien, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, and Bin Pang. 2015. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *ACM SIGCOMM*, Vol. 45. ACM, 139–152.

[9] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: Query-driven Streaming Network Telemetry. In *ACM SIGCOMM*. 357–371.

[10] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. 2014. NetSight: I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks. In *USENIX Conference on Networked Systems Design and Implementation*. 71–85.

[11] Rob Harrison, Qizhe Cai, Arpit Gupta, and Jennifer Rexford. 2018. Network-Wide Heavy Hitter Detection with Commodity Switches. In *Symposium on SDN Research*. ACM, 8:1–8:7.

[12] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. 2019. Confluo: Distributed Monitoring and Diagnosis Stack for High-speed Networks. In *USENIX Networked Systems Design and Implementation*.

[13] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. FlowRadar: A Better NetFlow for Data Centers. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. Santa Clara, CA, 311–324.

[14] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. LossRadar: Fast Detection of Lost Packets in Data Center Networks. In *International on Conference on Emerging Networking EXperiments and Technologies*. 481–495.

[15] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. 2017. Language-directed hardware design for network performance monitoring. In *ACM SIGCOMM*. ACM, 85–98.

[16] Srinivas Narayana, Mina Tahmasbi, Jennifer Rexford, and David Walker. 2016. Compiling Path Queries. In *USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, 207–222.

[17] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. 2018. Distributed Network Monitoring and Debugging with SwitchPointer. In *USENIX Symposium on Networked Systems Design and Implementation*. Renton, WA, 453–456.

[18] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. 2019. NetBouncer: Active Device and Link Failure Localization in Data Center Networks. In *USENIX Networked Systems Design and Implementation*. Boston, MA, 599–614.

[19] Da Yu, Yibo Zhu, Behnaz Arzani, Rodrigo Fonseca, Tianrong Zhang, Karl Deng, and Lihua Yuan. 2019. dShark: A General, Easy to Program and Scalable Framework for Analyzing In-network Packet Traces. In *USENIX NSDI*.

[20] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *USENIX Networked Systems Design and Implementation*.

[21] Yibo Zhu, Ben Y. Zhao, Haitao Zheng, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, and Ming Zhang. 2015. Packet-Level Telemetry in Large Datacenter Networks (Everflow). In *ACM SIGCOMM*, Vol. 45. 479–491.