ConQuest: Real-Time Analysis of Congestion in Packet Queues

Paper # 138, 12 pages + references

Abstract

Short-lived traffic surges can cause periods of unexpectedly high queue utilization. As a consequence, network operators are forced to deploy more expensive switches with larger, mostly under utilized packet buffers, or risk packet loss and delay. Instead, we argue that switches should detect congestion as it forms, and take corrective action on individual flows before the situation gets worse. This requires identifying the specific culprit flows responsible for the buildup, and marking, dropping, or rerouting the associated packets automatically in the data plane. However, collecting fine-grained statistics about queue occupancy in real time is challenging, even with emerging programmable data planes. We present ConQuest, a system that identifies the flows causing queue buildup within the data plane. Our evaluations show that ConQuest accurately targets the responsible flows at the sub-millisecond level, achieving Precision and Recall of over 90% using about 2KB of switch memory. Additionally, we propose and implement a novel framework for analyzing queuing in legacy switches using link tapping and an off-path programmable switch running ConQuest.

1 Introduction

Queue utilization in network switches is a major concern for network operators. As shown in Figure 1, on a link with relatively stable utilization, short periods of high queue utilization can arise. *Transient congestion*, which typically lasts hundreds of microseconds to tens of milliseconds, can be caused by legitimate or adversarial traffic bursts. This short-lived congestion often leads to higher queuing delay, requiring longer buffers or otherwise risking potential packet loss.

Understanding the cause of transient congestion requires real-time visibility into the switch's queuing dynamics. Unfortunately, most existing network devices report traffic statistics at the timescale of minutes or at best seconds. They often only report the total queue length, providing little visibility of what is happening *inside* the queue. Therefore, network operators cannot find the root cause for this congestion in real time, let alone alleviate it. Unlike data-center network fabrics, which can perform congestion control by coordinating between hosts in the network, transit network operators cannot prevent bursty workloads from entering their networks. To maintain high performance, transit network operators are forced to run links at lower utilization or maintain sufficiently large queuing buffers, which are both wasteful.

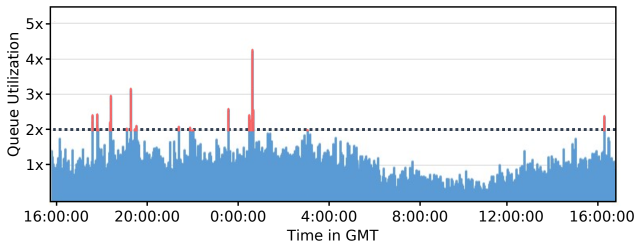

Figure 1: Queue utilization over a 24-hour period in a carrier network switch, highlighting transient congestion events with more than double the average queue length.

Our ultimate goal is to prevent the high queuing delay and potential packet loss caused by transient congestion. We achieve this by identifying the individual *culprit* flows responsible for the queue build-up, and taking immediate action against the associated packets. Possible actions may include *marking* or *dropping* packets, sending feedback upstream to *affect future packets* of this flow, or *reporting* to a collector for further analysis. Subsequently, network operators would not need to maintain large buffers to withstand the presence of bursty flows. They could run their network using cheaper commodity switches with smaller buffers, while sustaining high performance. To accomplish this goal, our solution needs to operate:

- **In real time:** Switches need to act quickly on packets of the culprit flows to alleviate congestion as it forms.
- **In the data plane:** A rapid response relies on measuring and acting on congestion directly in the data plane.
- **At a fine granularity:** To target the responsible traffic,

the switch should measure the queue at the flow level.

- **With high accuracy:** Accuracy is important to avoid missing culprit flows or penalizing innocent traffic.

A new generation of programmable switches offers greater visibility into queue dynamics as well as more flexible packet processing. Moreover, they can perform customized actions on each packet, which is crucial for detecting and alleviating congestion in the data plane. Yet, performing sophisticated analysis completely within the data plane is still challenging. To operate at line rate, these switches process packets in a hardware pipeline with a limited number of stages and a limited number of operations per stage. Furthermore, the memory for storing state across packets is limited, and each memory unit is bound to a particular stage in the pipeline. We discuss these constraints and an overview of the measurement problem we solve in further detail in Section 2.

Several recent works leverage programmable switches to analyze queuing dynamics and perform data aggregation. Solutions such as SpeedLight [19], BurstRadar [12], and *FLow [17] report congestion statistics by recording traffic with fine-grained timestamps in the data plane, and perform detailed analysis of the data in control-plane software or at remote hosts. In contrast, our goal is to perform per-packet corrective action *within* the data plane, rather than relying on offline analysis. In this paper, we present ConQuest (Congested Queue Tracking), a scalable system to detect and diagnose congestion in the data plane of programmable switches. ConQuest can run in programmable switches deployed in an operational network, or "on the side" to monitor legacy network devices that offer limited visibility into congestion. We make the following contributions:

Efficient switch-level data structure. We introduce a data structure which maintains multiple "snapshots" of the queue occupants over time. As each packet arrives, ConQuest updates one snapshot and queries multiple past snapshots, to determine the contribution of each flow to the congestion, directly in the data plane. The use of multiple snapshots allows ConQuest to achieve the desired accuracy (§ 3).

Monitoring legacy network devices. Most legacy network devices only report queue length at the timescale of multiple seconds and do not provide flow-level queue statistics. We propose an off-path monitoring technique that taps multiple links of a legacy switch, and feeds the data into a version of ConQuest extended to match the ingress and egress observations of the same packet (§ 4).

ConQuest hardware prototype. We implemented ConQuest on a Barefoot Tofino programmable switch using the P4 language [7]. We verified the accuracy of the prototype on a testbed, where ConQuest analyzed the queuing dynamics of a Cisco CRS switch with multiple 10 Gbps links (§ 5).

Simulations using real packet traces. We further evaluated ConQuest using multi-factor simulation experiments

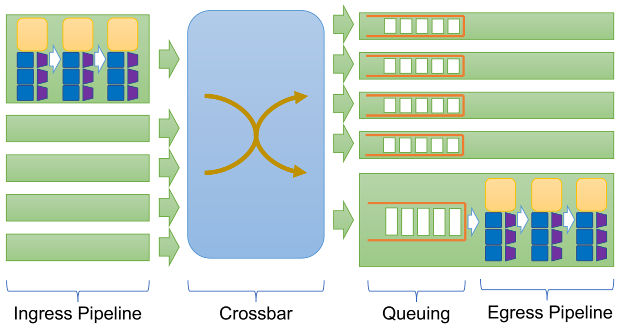

Figure 2: Queuing in a PISA programmable switch typically happens at a per-port basis before the egress pipeline processing. Thus, queuing analysis programs in a PISA switch can reside in the egress pipeline.

with real-world packet traces (§ 6). ConQuest can achieve high accuracy (over 90% Precision and Recall) using a modest amount of register memory (less than $2KB$).

Then, we discuss possible control actions to alleviate congestion (§ 7) and extend ConQuest from FIFO queues to arbitrary queuing disciplines (§ 8). Finally, we summarize the related work (§ 9) and our conclusions (§ 10).

The initial, workshop version of this paper presented only a simulation-based evaluation, whereas this paper provides an implementation and evaluation of a prototype on a hardware switch. Furthermore, the workshop paper focused only on PISA switches, whereas this paper also shows how to use a PISA switch to monitor the packet queues of a legacy device. Additionally, while the workshop paper worked solely for a FIFO queue, in this paper we have extended our algorithm to support general queuing disciplines.

2 Analyzing Congestion in PISA Switches

To analyze congestion, we developed a mechanism to identify the flows contributing to queue buildup in real time. In this section we first provide a description of the queuing and packet-processing model of Protocol Independent Switch Architecture (PISA) programmable switches [4, 7], as well as the challenges imposed by this architecture. We then introduce the measurement problem that ConQuest solves.

2.1 Working Within the PISA Architecture

To analyze queuing dynamics directly in the data plane, our system must support line-rate packet processing. To better explain the limitations this imposes, we give an overview of the PISA switch architecture and describe its constraints.

2.1.1 PISA Switch Architecture

Figure 2 illustrates several key components of a PISA switch. To achieve low latency and high throughput, PISA switches process packets in a pipeline with a limited number of stages. A PISA switch is composed of at least one ingress pipeline, a crossbar, and at least one egress pipeline. Each egress port is served by one or more queues.

Packet processing pipeline. Each pipeline stage consists of a match-action unit and a limited amount of stateful register memory, and supports a limited number of concurrent operations. A packet traversing the pipeline may access two kinds of state: (i) metadata that a packet carries from one stage to the next and (ii) the register memory arrays associated with each stage. At any given time, different pipeline stages process different packets simultaneously. Therefore, any register memory array can only be accessed from a particular pipeline stage; otherwise, concurrent writes from different stages by different packets could lead to hazards.

Queuing discipline. Upon entering the switch, a packet first goes through ingress pipeline processing to determine its egress port. Subsequently, the packet enters a queue associated with the selected egress port. Upon exiting the queue, the packet undergoes egress processing before leaving the switch. For ease of illustration, we consider the case where each egress port is associated with a single FIFO (first-in-first-out) queue. We discuss how ConQuest works with more general queuing models in Section 8.

2.1.2 Challenges of Running on PISA Switches

Per-stage memory access. A packet can only access each register memory array once as it goes through the packet-processing pipeline. Although it is possible to recirculate a packet, any recirculated packet must compete with incoming packets for resources, possibly reducing throughput.

Limited pipeline stages. Practical switches have a limited number of pipeline stages to maintain low forwarding latency. Thus, we need to parallelize computation as much as possible to fit sophisticated algorithms in the data plane.

No controller assistance. Since ConQuest works on the timescale of individual packets, any action by a software controller would lag behind. We cannot use the switch controller (onboard CPU) to help maintain the data structure or perform garbage collection. ConQuest should perform all data-structure housekeeping within the data plane.

2.2 ConQuest Congestion Diagnosis Problem

ConQuest identifies the flows causing the buildup in the queue, who are to "blame" for the congestion. Detecting and acting on culprit flows in real time would *seem* to require maintaining a *precise* count of how many packets each flow has in the queue, and support queries on *all* of these counters, *all* of the time. Unfortunately, it is difficult (if not impossible) to design a data structure in the data plane that meets these requirements. Specifically, the memory access requirement prevents us from updating the same data structure in both the ingress and egress pipelines. Otherwise, we could simply maintain some flow size counter data structure from both ends of the queue, adding to it when a packet enters and subtracting from it when the packet departs.

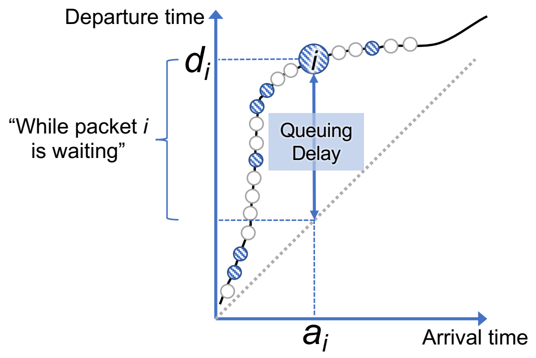

Figure 3: Scatter plot of packet departure vs. arrival time. For packet i arriving at a_i and departing at d_i, how many of packets of flow f_i departed "while packet i is waiting"?

Fortunately, to catch the culprit flows, it is sufficient to maintain approximate flow sizes and only query the size of each packet's own flow.

Querying a packet's own flow size. To take corrective action for the flows causing congestion, as the burst is forming, we need only identify the contribution of the *current* packet's flow to the queue buildup. Thus, we do not need to obtain a list of all culprit flows. Instead, from each packet's perspective, we just need to query the size of that packet's *own* flow within the queue, to see if the packet belongs to a culprit flow. This greatly simplifies the requirement for the data structure we maintain. More precisely, for each packet, we ask: *when I entered the queue, what fraction of the packets in the queue belonged to my flow?*

Accuracy when and where it matters. Furthermore, to alleviate congestion, ConQuest only needs to focus on the *heavy* flows, and does not need to report precise counts for all the small flows. As long as we can clearly distinguish the culprit flows from the harmless small flows, it is sufficient to maintain an approximate count of the flow sizes in the queue. This allows us to use approximation techniques and adapt to the constraints imposed by programmable switches.

Problem definition. A packet i of flow f_i enters the queue at time a_i and departs at time d_i, experiencing a queuing delay of $(d_i - a_i)$, as shown in Figure 3. For each egress port, the packet-processing pipeline witnesses packets leaving the queue as a stream of (a_i, d_i, f_i) tuples. At the egress pipeline of the switch, a_i and d_i are provided in the queuing metadata, and f_i is extracted from packet headers. Congestion arises when the queuing delay of one or more packets exceeds a threshold τ. When congestion occurs, our system aims to identify the culprit flows as those consuming at least an α fraction of the congested queue.

We also denote $\{x \mid a_i \leq d_x \leq d_i\}$ as the packets that departed the queue while packet i was waiting in the queue. Note that for a FIFO queue, this definition is equivalent to *what packets were in the queue when packet i entered*. For each packet i, if $(d_i - a_i) \geq \tau$, ConQuest tries to answer the following query: Does the fraction of packets in $\{x \mid a_i \leq d_x \leq d_i\}$ that satisfy $f_x = f_i$ exceed the culprit threshold α? If the answer is yes, f_i is considered a culprit flow and the data plane should take a corrective action (e.g., mark, drop, report, or reroute), possibly weighted by the fraction of traffic, on packet i.

In Figure 3, all packets belonging to f_i are shaded blue. For now we use $\alpha = 20\%$. When packet i arrives, the queue has four packets from flow f_i in the $\{x \mid a_i \leq d_x \leq d_i\}$ range (satisfying $f_x = f_i$), including i itself. This roughly accounts for 40% of all the packets that were served by the queue while i was waiting (greater than α), thus flow f_i contributes to queue buildup significantly and we call it a culprit flow. After detecting the culprit flow, the switch can avoid further queue buildup by marking, dropping, or rerouting packet i or subsequent packets from flow f_i.

3 Efficient Switch-Level Data Structure

To overcome the challenges of running in the data plane, ConQuest uses a snapshot-based data structure to record traffic and answer queries. In this section, we first present the rationale behind the snapshot-based data structure design. We then explain how we use the data structure to answer real-time flow-size queries and how we can clean and reuse snapshots in the data structure without controller mediation.

For simplicity, all definitions and discussions in this section assume all packets have unit size. However, it is straightforward to extend each definition to consider packet length.

3.1 Traffic Snapshots for Bulk Deletion

Given a stream of packets with (a_i, d_i, f_i) tuples, ConQuest determines how many packets in the queue (i.e., $\{x \mid a_i \leq d_x \leq d_i\}$) belong to the same flow as packet i (i.e., $f_x = f_i$). During congestion, if these packets compose a fraction of the queue that exceeds α, we call flow f_i a culprit flow.

To determine if a packet is a part of a culprit flow, ConQuest needs to maintain a data structure for past packet departures. Whenever a packet departs the queue, ConQuest queries packets in the *past* based on the time range $[a_i, d_i]$, and also inserts this packet and its departure timestamp (i, d_i) into the data structure to support *future* queries.

When implementing the above data structure on PISA switches, ConQuest has to take into account a set of constraints imposed by the architecture, as described in Section 2.1. We observe that one of the primary difficulties for maintaining such a data structure is to accurately *delete* expired packets, whose departure timestamp had become too small to be included in any future packet's query. Given the

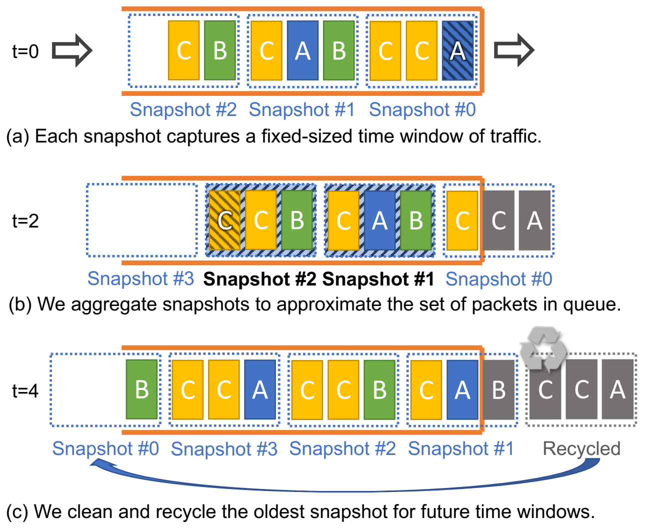
t=0
C
B
C
A
B
C
C
A
Snapshot #2
Snapshot #1
Snapshot #0
(a) Each snapshot captures a fixed-sized time window of traffic.
t=2
C
C
B
C
A
B
C
C
A
Snapshot #3
Snapshot #2
Snapshot #1
Snapshot #0
(b) We aggregate snapshots to approximate the set of packets in queue.
t=4
B
C
C
A
C
C
B
C
A
B
C
C
A
Snapshot #0
Snapshot #3
Snapshot #2
Snapshot #1
Recycled
(c) We clean and recycle the oldest snapshot for future time windows.

Figure 4: Traffic time-window snapshots on a FIFO queue.

strict processing time bound, it is challenging to precisely track each packet's expiration for deletion. For this reason we decided to use *time-window snapshots* that allow us to implicitly delete expired packets in bulk.

The *time-window snapshots* structure works as follows: we split the departing traffic into small time windows, each lasting a fixed interval T. The packets are grouped by their departure timestamp. For example in Figure 4(a), if we choose time window $T = 3$, the rightmost packet (shaded blue) has a departure timestamp $d_i = 0$, thus it goes to time window $\lfloor 0/T \rfloor = 0$. The other blue packet from flow A departs later at time 4 and has a departure timestamp $d_j = 4$ (as shown in Figure 4(c)), thus falling in window $\lfloor 4/T \rfloor = 1$.

During each time window, we count the total number of packets for each flow. Afterwards, we query this snapshot to obtain the sizes of different flows on behalf of other packets.

Using time window snapshots, we can now implicitly delete old packets in bulk from the system by no longer querying the oldest snapshot. We can forget about them, or better yet, *recycle* and reuse them (illustrated in Figure 4(c)). We discuss recycling snapshots in more detail in Section 3.3.

Since we wish to analyze congestion and take action directly in the data plane, the snapshots also need to be queried within the data plane, using primitive operations that PISA supports. If the number of flows is limited and known beforehand, we may assign simple per-flow counters. For a transit network with flow identifiers not known beforehand, we can utilize sketches, or approximate data structures, as we mostly care about the *heavy* flows in the queue. Any sketch that supports inserting/incrementing counts and querying flow sizes with reasonable accuracy can achieve our purpose; in our implementation, we use the Count-Min Sketch (CMS) data structure [8] due to its ease of implementation in the data plane. The CMS is an approximate data structure which can estimate flow sizes with a possible overestimation error due to hash collisions. The error bound is determined by the selected size of the structure.

There are two kinds of approximation errors that should be considered when we analyze the queue in this approach. When we slice traffic into time windows, the query for a particular time range $[a_i, d_i]$ can have some rounding error. Meanwhile, the use of sketches to maintain the counters also incurs some approximation error. We further analyze the effect of both types of error in the evaluation in Section 6.

3.2 Aggregating Over Multiple Snapshots

Ideally, to decide if packet i is a culprit we could look backward in time and compute f_i's flow size within the departure time range $[a_i, d_i]$. ConQuest approximates this range by looking at several recent time windows: from window $\lceil \frac{a_i}{T} \rceil$ to window $\lfloor \frac{d_i}{T} \rfloor$. Thus, if we aggregate the flow size of f_i reported by the corresponding snapshots, we know approximately how many packets from f_i departed during $[a_i, d_i]$.

As a concrete example, in Figure 4(b), the leftmost packet (shaded yellow) from flow C arrived at $a_i = 2$. Later, it departs the queue at $d_i = 8$ (not shown in the figure). The packets of interest are those that departed in the time range $[2, 8]$, i.e., the seven packets shown *inside* the queue in Figure 4(b); out of these packets, there are four packets from flow C (yellow packets). Snapshot #1 recorded one packet for flow C, while Snapshot #2 recorded two packets. By aggregating the two shaded snapshots, #1 and #2, we can get an approximate value 3, i.e., there are around three packets from flow C that departed between time $[2, 8]$.

Aggregating snapshots would cause at most T rounding error at both ends of the range. When the queuing delay $(d_i - a_i)$ is much larger than T, i.e., when the queue is congested and therefore we are interested in measuring, we have smaller relative error.

Besides simple summation, we may also aggregate snapshots differently to compute other metrics in the data plane. This creates more applications for snapshots beyond analyzing congestion. For example, we can detect rapid changes in flow throughput in the data plane, by computing the difference between the flow sizes reported by the two most recent snapshots. This technique would help network operators locate flows which rapidly ramp-up without obeying congestion control. Note that currently ConQuest uses a fraction of queuing delay as the time window, while different applications may find a different time window more suitable.

3.3 Cleaning and Reusing Expired Snapshot

For a FIFO queue in the switch, the maximum queuing delay $(d_i - a_i)$ is bounded by the maximum queuing buffer size Q (bytes) divided by line rate R (bytes/second). Subsequently, we know each snapshot would only be queried for the next

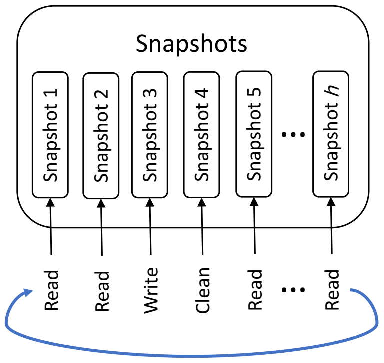

Figure 5: Round-Robin between h Snapshots. In any given time window, ConQuest writes into one snapshot, reads many, and cleans one snapshot for the next time window.

$\lceil \frac{Q}{R \times T} \rceil$ time windows. As a concrete example, a queue with $R = 10Gbps$ drain rate and maximum $Q = 5MB$ buffer size has $Q/R = 4ms$; if we use $T = 1ms$ time window, only the most recent four time windows will be queried.

Therefore, we only need to maintain a constant number of recent snapshots. Figure 4(c) illustrates that when the packets recorded in a snapshot are no longer useful, we can recycle the snapshot for recording future traffic.

We can maintain $h > \lceil \frac{Q}{R \times T} \rceil$ snapshots in total, and use them in a Round-Robin fashion. As illustrated in Figure 5, the roles of snapshots rotate every T seconds, synchronized with the progress of the time window.

Every packet leaving the queue can be added to the snapshot that is currently in a *Write* phase, according to the current time window (modulo h). The other most recent $h-2$ snapshots are in *Read* phase and can be queried. The oldest snapshot is no longer useful, and is in *Clean* phase. The data structure is cleared and prepared for writing in the next time window.

To illustrate the idea further, let us assign h variable indices to indicate which snapshot to read, write, or clean: I^w is the write index, I^c is the clean index and $I^r_1, \cdots, I^r_{h-2}$ are the read indices. Within a snapshot window, for each packet i of flow f_i that departs the queue, the following is performed:

1. In snapshot I^w we increment the count of flow f_i by 1.
2. To extract the estimated flow size in the queue for f_i, we first read the $n = \lfloor \frac{d_i}{T} \rfloor - \lceil \frac{a_i}{T} \rceil$ most recent snapshots based on the arrival time a_i and departure time d_i of the packet i. Subsequently, we sum the estimated flow sizes reported by $I^r_1, \cdots, I^r_n$.
3. Data structure in snapshot I^c is being cleared for future use.

Implementing the snapshot round-robin is straightforward in PISA switches. First, the PISA switch tags a_i and d_i to each packet as a part of the queuing metadata, and extract packet header fields to determine flow ID f_i. We can divide d_i by the time window T to find the current time window $\lfloor \frac{d_i}{T} \rfloor$. Subsequently, we compute its modulo over h to find I^w, the current snapshot to write into. We can also derive other indexes for read and clean, based on $I^c = I^w + 1 \mod h$ and $I^r_i = I^w - i \mod h$. We also decide how many old snapshots to read from by dividing a_i over T. Due to the limited arithmetic operations supported by the PISA switch, implementing division and rounding is not straightforward. We implement these operations in the PISA switch by using bitwise right shift, and by erasing the most significant bits. Note that these operations requires that both T and h be integer powers of two.

In our implementation of ConQuest, each snapshot is represented as several hash-indexed arrays to form a Count-Min Sketch. To read from or write to a snapshot, a packet first finds the indices based on flow ID f_i, then read or increment the value at those indices. Since snapshots rotate quickly, we cannot rely on the switch controller to clean out register memory arrays. To clean and recycle a snapshot in the data plane, we maintain a cyclic pointer as the index to be cleaned in the array. The pointer is incremented once per packet; when it grows above the largest index in the array it overflows back to zero. This way, the incoming packets can clean the oldest snapshot cyclically, by writing zeroes one index at a time to every index in the array. We store the pointer itself in a single-slot register memory array in an earlier stage.

The operation on each snapshot takes several hardware pipeline stages. If we arrange all snapshots sequentially, we can easily run out of hardware pipeline stages. Fortunately, the operations performed are independent across snapshots and can be done in parallel. We can place multiple hash-indexed arrays in the same pipeline stage and execute their actions simultaneously, essentially "stacking" multiple snapshots to save the number of pipeline stages needed to implement ConQuest. The height of stacking depends on the number of parallel operations we can perform per stage on the particular hardware target.

4 Monitoring Legacy Network Devices

Existing networks are not going to replace legacy switches with programmable switches overnight. Still, advanced fine-grained monitoring techniques are necessary today. We propose a novel framework to analyze and diagnose queuing in legacy, non-programmable switches, in a non-intrusive manner, by tapping the existing ingress and egress links and using the programmable switch to process the tapped traffic. This allows us to gain visibility into a legacy device's queuing dynamics using temporary tapping, without having to replace the device with a programmable one.

4.1 Monitoring Queues on Legacy Devices

Legacy network devices are not designed for precise queuing analysis. They often support polling the queue length statistics, but only at a coarse-grained time interval. Furthermore, they provide no information about which flows are occupying the queue. By monitoring and analyzing packets going through the legacy devices, we can have a much closer look at the underlying congestion. Furthermore, a more detailed understanding of typical queuing dynamics in the network can help operators make informed decisions regarding whether/when/where to deploy programmable commodity switches. Meanwhile, tapping links is quite easy in a carrier network, as the equipment for tapping is already in place.

Figure 6 illustrates the setup for using a PISA programmable switch to monitor queuing in a legacy switch. We tap a subset of the legacy switch's ingress and egress ports and mirror their traffic to the PISA switch. The PISA switch records the arrival timestamps (a_i) and departure timestamps (d_i) for packets going through the legacy switch, based on their time of appearance on the tapped link.

Our goal is to recover a stream of (a_i, d_i, f_i) tuples for the queue in the legacy device, and subsequently run ConQuest on the programmable switch to perform the same queuing analysis. We can then detect transient congestion and report culprit flows. However, we will not be able to act on individual packets in this non-intrusive tapping setup.

There are several practical considerations for this tapping setup. First, we should ensure that all tapped links have equal tapping latency. A higher latency would not affect our analysis, as it creates equal offset for a_i and d_i. However, an inconsistency in latency would cause persistent error in estimated queuing delay. Also, in a multi-pipeline programmable switch, all tapped links need to enter at ports associated with the same ingress packet-processing pipeline, so the arriving and departing packets can have access to the same register memory array. Finally, note that the total tapped throughput is not limited by the line rate or maximum switching throughput of the programmable switch, as we only require programmable *processing*—there is no need to forward or queue the tapped packets. Therefore, we also move ConQuest to the ingress pipeline of the programmable switch under this setup, to avoid being unnecessarily limited by the throughput of the egress links.

4.2 Matching Ingress/Egress Packets

A major implementation challenges is matching packets between tapped ingress and egress links. Once packets are matched, we may derive queuing latency for the monitored egress port in the legacy switch. For multiple monitored egress ports, we run a copy of ConQuest in the programmable switch for each egress port in the legacy switch.

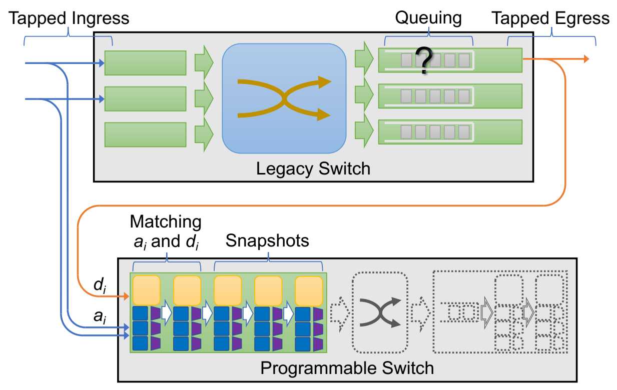

Figure 6: Using a programmable switch to analyze queuing in a legacy switch, by tapping both ingress and egress links.

Matching packets. We would like to recover the (a_i, d_i, f_i) tuple from the tapped traffic. Getting d_i is relatively straightforward, as we can witness all packets departing from the queue at the tapped egress port. We can parse packet header to get flow ID f_i.

If the same packet comes from a tapped ingress link, we also know a_i for this packet. The programmable switch needs to match the two appearances and pair a_i with d_i.

When the programmable switch first sees packet i from the tapped ingress link, it uses a packet identifier ID_i as the hash key to store the (ID_i, a_i) pair in a hash-indexed register memory array. Later, when it sees i again from the tapped egress link, it tries to retrieve the entry for packet i to get a_i and erase the entry. We need to generate a packet identifier ID_i in order to recognize the two appearances of the same packet, and distinguish it from other packets.

For IPv4 packets, we can examine the IPID field. For TCP packets, we also can observe the sequence/ack number to distinguish individual packets within the same flow. Uniquely identifying UDP or IPv6 packets is more challenging and we omit the implementation details here. In our prototype, we simply take the usually unique header fields (sequence numbers, checksums, etc.) and hash all of them using CRC32 to form a 32-bit packet identifier ID_i.

Hash collisions. Although a collision for the 32-bit packet ID is possible, it is unlikely to happen given the timescale ConQuest is working on—ConQuest sees much less than 2^{32} packets during an entire cycle of snapshot rotation, which should be roughly equivalent to the maximum queuing delay in a switch. Two different packets appearing on tapped links on this timescale are unlikely to have the same packet identifier.

Meanwhile, as we store arrival timestamps in a hash-indexed register memory array, hash collision can cause failures in finding a_i when multiple packets are hashed into the same entry in the array. We must maintain a sufficiently large array to reduce hash collisions to an adequately low level. In practice, we need the size of the array to be as large as the maximum number of packets in the monitored queue. In our implementation we use 65,536 entries, about the same magnitude as the maximum number of packets in queue.

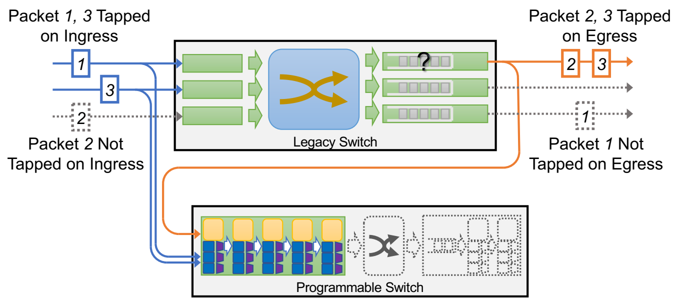
Packet *1*, *3* Tapped on Ingress
1
3
2
Packet *2* Not Tapped on Ingress
?
Legacy Switch
Packet *2*, *3* Tapped on Egress
2
3
1
Packet *1* Not Tapped on Egress
Programmable Switch

Figure 7: Packet *1* appears on tapped ingress links but not the tapped egress link; it did not contribute to congestion in the monitored queue. Packet *2* appears on tapped egress link but not the ingress links; we don't know its arrival time or queuing delay, but still count it towards congestion analysis for packets of the same flow. Packet *3* appears on both tapped ingress and egress so we can match its two occurrences to know (a_3, d_3, f_3).

When a collision happens, the egress packet i will be coupled with an ingress packet entry (ID_j, a_j) with a different packet identifier $ID_j \neq ID_i$ and fail to match, thus a_i will be missing, and queuing delay $(d_i - a_i)$ becomes unknown. This is similar to the "Unmonitored Ingress" situation discussed in the next subsection. We ignore this packet for analyzing queuing delay but still record it for the current time window, and therefore, occasional collisions would not affect our congestion analysis.

4.3 Robustness to Unmatched Packets

Unmonitored egress. We may not see all ingress packets again at the tapped egress link. Some of them may be routed to an egress link that is not being monitored, and some of them may be dropped due to congestion. For example, in Figure 7, packet *1* was tapped on an ingress link, but was routed to an egress port not being tapped. These packets would fill up the register memory array with packet IDs and arrival timestamps that would not be matched later and are therefore useless. Meanwhile, the register memory array cannot garbage collect by itself, and would gradually get filled by these entries. We solve this issue by implicitly expiring entries: we allow an entry to be evicted from the array once its arrival timestamp has aged more than the maximum possible queuing delay and can thus be considered expired.

A flow may exit the switch from multiple egress links. If only a subset of a flow's packets appears on the tapped egress link, only this subset contributes to the congestion in the monitored queue. ConQuest only analyzes congestion for this subset of packets.

Unmonitored ingress. Symmetrically, some packets ob-

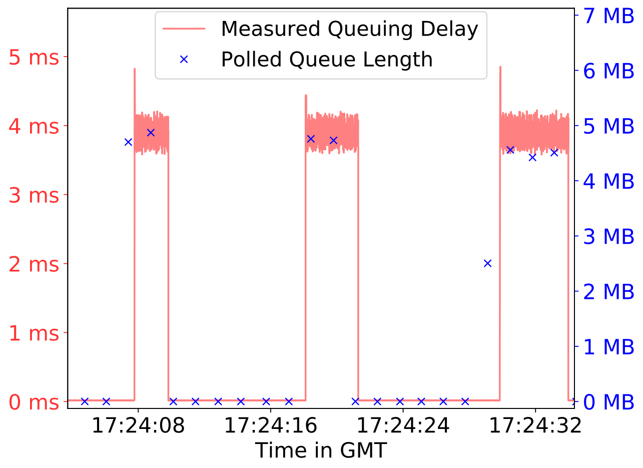

Figure 8: Queuing delay measured by our prototype matches the ground-truth queue-depth reported by the legacy switch.

served on an egress link might have entered the switch from an ingress link that is not being monitored. In Figure 7, packet *2* comes in from an untapped ingress port, but appears on the tapped egress port. For these packets, we do not have their arrival timestamp. However, we still insert them into the current snapshot as usual, as they contributed to the congestion at our monitored egress port and snapshot insertion only requires the departure timestamp.

A flow may also enter the switch from multiple ingress links. If only a subset of the flow's packets is coming from tapped ingress links, only this subset has a valid arrival timestamp in the system. Thus, ConQuest can only identify whether such a packet is a culprit for packets in this subset. However, since all packets in this flow are inserted to the snapshots, regardless of if their ingress link is tapped or not, they are all be accounted for when determining the flow's size in the queue.

5 P4 Prototype on Hardware Switch

We implemented a four-snapshot prototype of ConQuest in P4 on a programmable Barefoot Tofino Wedge-100 switch ("programmable switch") under the tapping setup described in Section 4, to analyze the traffic on a Cisco CRS 16-Slot Single-Shelf System ("legacy switch"). We tapped 3 ingress links and 1 egress link, all running at 10 Gbps. To evaluate ConQuest, We use an IXIA traffic generator to generate traffic to feed to the 3 ingress ports. The legacy switch is configured to route all traffic to the same egress port, into a single FIFO queue.

The P4 program used in the testbed also includes the logic for matching ingress/egress packets and computing ground-truth statistics for evaluation. The combined P4 program has around $1,200$ lines of code.

Estimating queuing delay. Since our P4 program needs to first match two appearances of the same packet and compute queuing delay before maintaining snapshots, we verified such matching by comparing the measured queuing delay with the ground truth maximum queue depth reported by the legacy switch. The legacy switch is capable of reporting maximum queue depth approximately every 0.5 seconds. We sent bursty traffic into the legacy switch to maximize congestion and observe the queue depth. As shown in Figure 8, the queuing delay computed by our P4 program nicely aligns with the queue depth reported by the legacy switch (divided by line rate 10Gbps). Later, we use the queuing delay measured by our P4 program to approximate ground truth queuing delay. Since the measured maximum queuing delay is around $4ms$, we set our time window to be $T = 1ms$ accordingly, which is $1/h$ of maximum queuing delay.

Estimating queue constituents. To evaluate ConQuest on the testbed, we need to know the ground truth of whether each packet is actually a culprit packet, based on α. Fortunately, since the testbed uses a FIFO queue, the set of packets that departed while packet i is waiting is equivalent to the packets that were in the queue when i entered. Thus, we simply need to know the size of flow f_i in the queue.

Although we cannot know exactly the ground truth per-flow size in the queue in real time, we maintained a best-effort estimate. For each flow, we first configure IXIA to generate packets with a sequentially increasing sequence number (using TCP SEQ header field, conveniently), and observe the difference between this sequence number as it appears on the ingress and egress links. This allows us to estimate fairly accurately how many packets are in the queue for each flow in real-time, notwithstanding packet drops. We report this estimated flow size in the queue for every egress packet and use this to approximate the ground truth flow size. Note that this estimation technique requires maintaining per-flow state in the PISA switch and is made possible by only having tens of flows in our evaluation testbed, while ConQuest itself works with any number of flows.

Evaluating the prototype. The P4 prototype of ConQuest uses a two-row Count-Min Sketch data structure for each snapshot with 64 counters per row. Since the testbed setup can only send tens of flows, we won't observe estimation error caused by hash collisions. The estimation error will be mostly caused by rounding error of time windows.

To maintain consistency with the approximated ground truth flow size measured by the testbed, we configure ConQuest to count packets instead of bytes in the testbed setup.

We evaluate the performance of the ConQuest prototype by looking at the Precision-Recall curve for identifying culprit flows, defined as follows. First, whenever queuing delay exceeds τ =0.5ms (1/8 of the maximum queuing delay), ConQuest reports the result of its per-packet analysis. Now, for each packet i, ConQuest queries the number of packets that departed between a_i and d_i that also belong to f_i, and compares the fraction of those packets (among all packets

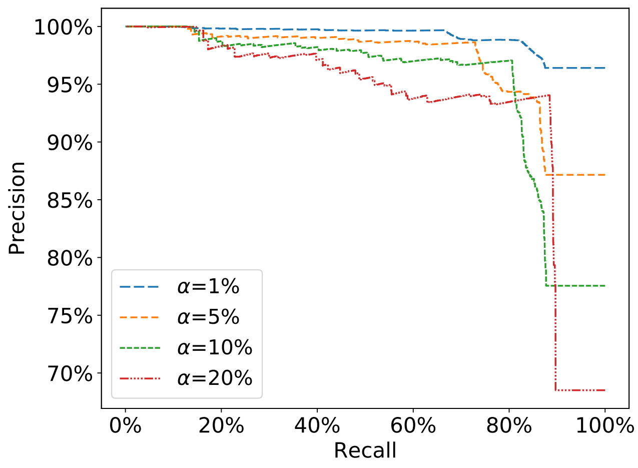

Figure 9: Precision and Recall for ConQuest's P4 prototype.

that departed during $[a_i, d_i]$) to a threshold. When this fraction is larger than a predefined culprit threshold α, we call the packet a culprit packet. ConQuest's *Precision* is defined as the number of culprit packets correctly estimated by ConQuest divided by all packets ConQuest believed to be culprit, and its *Recall* is defined as the number of culprit packets correctly estimated divided by the ground truth number of culprit packets. As a standard metric for evaluating a binary classifier, the Precision and Recall curve captures how ConQuest trades false positives for false negatives and achieves balanced accuracy.

In our experiment, we generate 10 background flows with various throughput, ranging from 1% to 50% of line rate, and 3 periodically bursty flows, with varying burst duration from 50 µs to 5 ms. We varied the culprit threshold from α=1% to 20%. For evaluation purpose, we forward all tapped packets to a collection server (alongside ConQuest's estimate) to enable Precision-Recall analysis. In actual deployment, ConQuest would only report to a collector when it detects a culprit flow according to its estimate.

The Precision-Recall curves for the experiment result is shown in Figure 9. As we can see, ConQuest performs reasonably well achieving both above 90% Precision and 80% Recall for identifying packets belonging to culprit flows, for various thresholds α. We present a more in-depth analysis of ConQuest's Precision and Recall in the next section.

6 Experimental Results

Simulation experiments allows us to freely tune all parameters of ConQuest that practical hardware may not permit, and gives us full detail about the queuing dynamics at any given time. We now describe the setup of our simulation experiments and present our results.

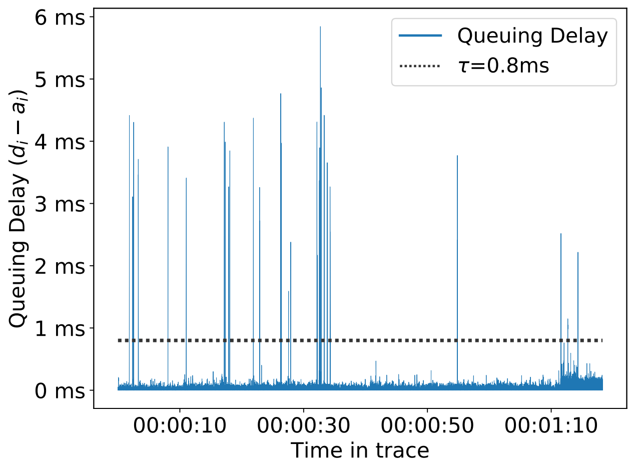

Figure 10: Simulated Queue buildup on the UW Trace shows low average utilization with occasional bursts, as in Figure 1.

6.1 Dataset and Implementation

To simulate congestion, we utilize the publicly available University of Wisconsin Data Center Measurement trace *UNI1* (UW trace) [3], by feeding the trace through a single FIFO queue with constant 10Gbps drain rate and unlimited queuing buffer. Since the original trace has an average throughput of only 25.3 Mbps, we replay the trace 50x faster. As we can see from Figure 10, the queue length exhibits a bursty pattern over time. Similar pattern arise when we use different drain rates ranging from 5Gbps to 20Gbps.

We simulate ConQuest using Python. Contrary to the testbed setup, the simulated queuing can provide us ground truth per-flow size measurements in bytes. Therefore, we also have ConQuest report in bytes instead of number of packets. We use an otherwise similar setup as the one described in Section 5: ConQuest reports per-packet analysis results whenever queuing delay grows over τ =0.8ms (about 1/8 of maximum queue depth observed). When a large fraction of the packets that departed within $[a_i, d_i]$ are from the same flow f_i as packet i, and this fraction exceeded a culprit threshold α=1%, we define packet i as belonging to culprit flow. We again use Precision and Recall metrics to evaluate ConQuest (see Section 5).

We define a flow based on the standard 5-tuple (source and destination IP address, protocol, and source and destination port). The UW trace has around 550,000 flows in total. In our queuing simulation, there are on average 63.6 distinct flows in the queue during each transient congestion, with an average of 3.7 culprit flows.

There are two primary design choices for ConQuest, the snapshot data structure's memory size and the snapshot time window size. Using more memory to construct larger Count-Min Sketch (CMS) data structures reduces collisions and improves accuracy. Using a smaller time window T provides better granularity when approximating the range $[a_i, d_i]$

by lowering the rounding error, at the cost of using more pipeline stages. We evaluate the effect of both choices on ConQuest's accuracy.

6.2 Effect of Limited Per-Snapshot Memory

We first evaluate the memory needed to achieve adequate accuracy. In each snapshot, we use a 4-row CMS to record and estimate the total flow size for each flow during each snapshot time window. When memory is insufficient, CMS suffers from hash collisions and over-estimates the size of flows, reporting more false positives and lowering Precision (but Recall does not change since CMS would not underestimate flow size). Figure 11 shows the effect of varying the total number of counters in the CMS on Precision. The Precision plateaus at 24 to 32 counters (6 to 8 columns per row) with diminishing returns for allocating additional counters.

6.3 Effect of Snapshot Time Window Size

Next, we evaluate the effect of snapshot window granularity on accuracy. We focus on improving Recall in this evaluation, since Figure 11 already demonstrated that the estimation yields high Precision when given enough memory. The multiple curves in Figure 12 almost overlap, since providing more than enough memory yields negligible difference on Recall, or even slightly decreases Recall; this is because hash collisions lead to overestimates, creating both more false positives and true positives simultaneously.

Increasing the number of snapshots h (therefore using a shorter time window T) reduces ConQuest's rounding error when computing $\lceil \frac{a_i}{T} \rceil$ and $\lfloor \frac{d_i}{T} \rfloor$. Using fewer snapshots (and larger windows) would cause bursts that departed immediately before a_i to be erroneously included in the $[a_i, d_i]$ range, thus the rounding error would lead to lower Recall. In the worst case, ConQuest can only look at one snapshot and cannot adapt to the change in queuing delay. As shown in Figure 12, by aggregating a maximum of $h = 8$ snapshots each spanning T=0.8ms of traffic, we can already achieve a high Recall, and have diminishing returns afterwards. Note that since the maximum queuing delay in the simulated queue is around 6ms, we choose $h \times T = 6.4ms$ in all combinations.

6.4 Estimating the Flow Size Distribution

ConQuest produces flow size estimates for all flows, not only the largest ones. Thus, we can use the snapshots to report the flow size distribution for all packets during a congestion. A network operator may use such a distribution to gain insights on the nature of congestion in a specific switch, and decide on the most appropriate action. For example, if there's usually only one large flow occupying 90% of the queue, then it may be sensible to mark or drop the heaviest flow.

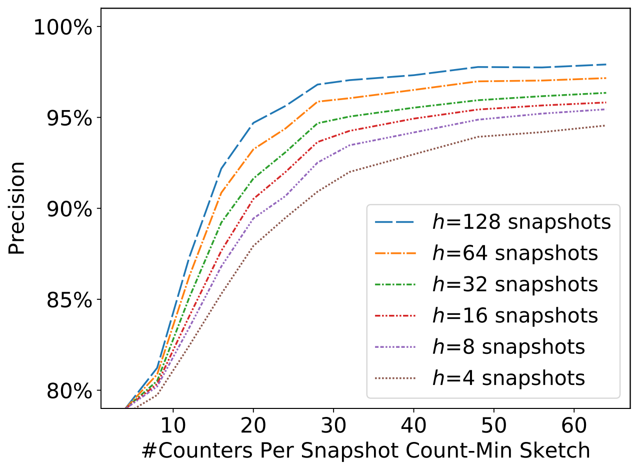

Figure 11: Precision vs. snapshot data structure size. Using 24-32 counters is adequate for achieving high Precision.

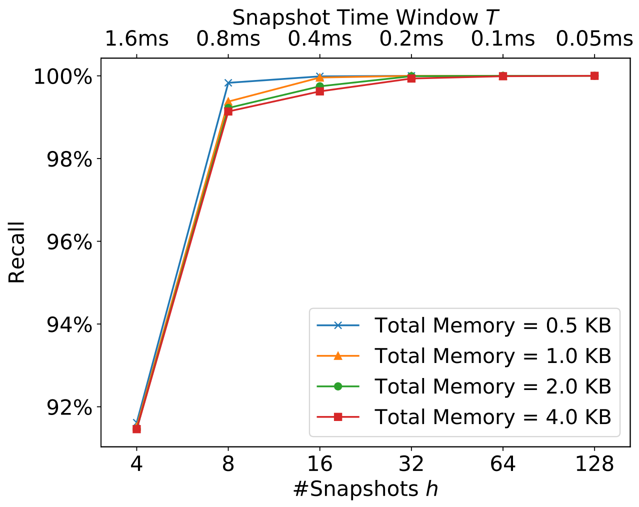

Figure 12: Recall vs. snapshot time window size. Using 8 snapshots gives sufficiently high Recall.

We evaluate the accuracy of this estimation by comparing all estimated flow sizes with the ground truth. In this evaluation, shown in Figure 13, we use parameters derived from previous experiments to achieve high accuracy using minimal resources: maintaining $h = 4$ to 64 snapshots, each accumulating traffic in a window of T=0.1 to 1.6ms, and each using a 32-counter CMS. Since the culprit flows occupy most of the queue, their estimated size are close to integer multiples of the snapshot window size, causing the presented staircase-like graph when ConQuest uses fewer snapshots. For the smaller flows, a small absolute error is normally achieved, with higher relative error. During congestion, the mean estimation error is 120KB while median estimation error is 79 KB for $h = 16$ snapshots, implying the estimation is adequately accurate for a majority of flows. For $h = 4$ snapshots, the mean and median errors increased

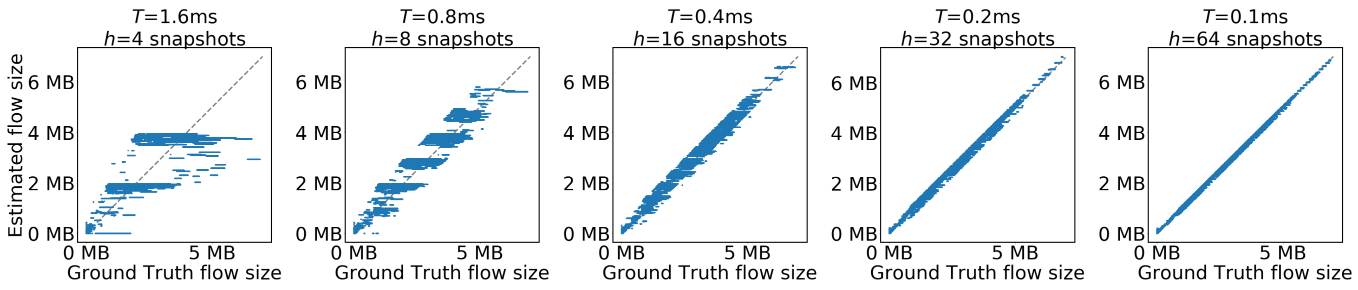

Figure 13: Estimated vs. actual flow sizes in the queue in simulation experiments, using different snapshot time window T. The staircase pattern observed for larger T is caused by rounding errors, and diminishes as we reduce T and use more snapshots.

to 461KB and 281KB respectively. As a reference, under the h =4 snapshots (T =1.6ms), 10Gbps line rate setup, the total traffic recorded in each time window is 2MB.

7 Control Actions on the Culprit Flows

In previous sections, we mainly focused on measuring a flow f's weight in the queue (w_f). Now we discuss the potential actions that may be taken to alleviate congestion.

Figure 14 depicts the same simulated queue shown in Figure 10. Here we highlight the portion of the queue that is occupied by the single *heaviest* flow in the queue. During periods of congestion, the top-1 flow occupies most of the queue, whereas all of the other flows do not surpass the marked threshold. Therefore, acting on the top-1 flow alone has the potential to significantly reduce the required buffer size (e.g., from $8MB$ to $2MB$), while avoiding loss and delay for the other flows. We now discuss what potential actions we can take to achieve these gains.

Act on the current packet. The switch data plane can mark the Early Congestion Notification (ECN) bit of the packets of a culprit flow. If queuing delay deteriorates further, the switch can go one step further by dropping such packets. Dropping all packets with a probability proportional to a small constant c that depends on the queue utilization, is equivalent to the primitive Random Early Detection (RED) queuing discipline. Instead, we could use flow f's weight w_f in the dropping probability. For example, dropping the packet with probability $P[drop] \propto \max(w_f - c, 0)^2$ is similar to the CHOKe [14] algorithm at its steady state for unresponsive flows. ConQuest therefore enables fast prototyping of active queue management algorithms that target culprit flows by using probabilistic dropping.

Act on future packets. Upon identifying a culprit flow, the switch can feed this information from the egress pipeline back to the ingress pipeline using packet recirculation. The ingress pipeline can then prevent this flow from exacerbating the imminent queue buildup, by re-routing, rate-limiting, or dropping the packets from this flow temporarily.

Report and aggregate flows. Transient congestion is sometimes not caused by individual culprit flows. In some

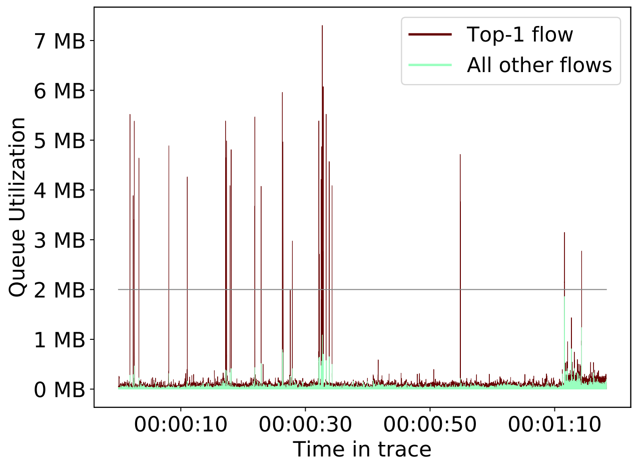

Figure 14: In the simulation on the UW trace, the top-1 flow in the queue occupies most of the queuing buffer during transient congestion.

cases, we can identify the cause of the congestion by defining flows at a coarser level of granularity. For example, TCP incast [6] is caused by many sources sending packets to the same destination simultaneously, and can be accurately captured by defining flows by destination IP address. In other cases, ConQuest can report packets from culprit flows to a software collector for further analysis, such as aggregating the reports to detect hierarchical heavy hitters or other groupings of flows belonging to a single distributed application (e.g., coflows). We leave these extensions as future work.

8 General Queuing Disciplines

Practical switch configuration sometimes incorporate non-FIFO queuing due to multiple traffic classes. The question we focus on, namely, for each packet i—"which packets departed while i was waiting"—is still well defined, as long as we have arrival and departure timestamps (a_i, d_i) for each packet. Therefore, we can still use ConQuest to analyze queuing and congestion in such settings.

However, unlike a FIFO queue, the queuing delay can be unbounded for packets in a general non-FIFO queuing setting. We can either manually specify a "realistic upper-bound" for the queuing delay $(d_i - a_i)$ based on practical experience, or maintain multiple hierarchical groups of snapshot with different timescales, similar to the design in [18]. For example, we can maintain three groups of snapshots, with the first group of snapshots each recording traffic in a time window of $T_1 = 100\mu s$. The second group can have a larger time window $T_2 = 1ms$, and the third group can have $T_3 = 10ms$. When a packet experienced a $32.1ms$ delay, we aggregate one snapshot from the first group, two from the second group, and three from the third group; this allows us to query $T_1 + 2T_2 + 3T_3 = 32.1ms$ duration of recent packet departures.

We can further generalize the queuing delay analysis beyond a single switch. As network operators, we may be interested in analyzing the end-to-end delay a packet experienced between the time it entered and the time it departed the network. Using this analysis we can learn which flows caused increased latency for other packets at which queue. This analysis can also be performed, as long as we can collect and analyze (a_i, d_i) pairs.

9 Related Work

Measuring microbursts. Zhang et al. [21] implemented a high-precision microburst measurement framework in datacenter networks, by polling multiple switches' queue depth counter at high frequency, and analyzing duration and inter-arrival time of microbursts. However, the system provides limited insight into the contents of the bursts, such as flow-size distribution and the ID of most significant flows. Speed-Light [19] is a system capable of recording synchronized traffic snapshots across multiple switches for offline analysis, when triggered by a network event such as high queue utilization. Similarly, BurstRadar [12] can also capture the traffic during microbursts for offline analysis. *Flow [17] uses programmable switches to aggregate per-packet statistics before sending them to remote hosts for further analysis. The aggregation at the switch reduces the computation burden for remote hosts caused by the need to analyze line rate traffic. Speedlight, BurstRadar and *Flow can all provide detailed analysis for transient congestion offline, by ferrying the information to switch controllers or remote hosts. They can generate almost immediate reports on a human timescale, but are still insufficient for taking actions to suppress the microburst directly in data plane.

Load balancing. We can prevent congestion in the network by load balancing. Existing solutions in data center networking such as Fastpass [15] offer a centralized traffic orchestration approach for treating queue buildup using scheduling methods. These attempts are too slow for analyzing and suppressing transient congestions, as most of the damage is already done by the time high delay or loss can be detected centrally. Presto [11], NDP [10], and Homa [13] minimize queuing delay at switches by requiring hosts in the network to participate in advanced congestion control schemes, and are thus less suitable for transit networks. Other solutions, such as DRILL [9] and CONGA [2], perform load balancing to disperse the load within the data plane. General solutions such as routing changes or load balancing may disrupt the well-behaved flows, not just the culprits. Instead, solving the microburst problem requires a better understanding of the cause of a transient congestion event as opposed to just detecting it. For instance, finding out that congestion is caused by a single bursty flow or by a certain application opens the opportunity for a targeted remediation such as packet marking, rate limiting, or selective dropping.

Fair queuing. An alternative method to prevent bursty flows from affecting other traffic is to use fair queuing. Sharma et al. [16] recently proposed an approximate per-flow fair queuing mechanism using programmable switches. Instead of enforcing fairness among all flows, ConQuest focuses on individual flows causing queue buildup, and only acts upon those flows during congestion.

Sliding window query. We note that our proposed framework continues a series of theoretical works presenting streaming algorithms for querying in a sliding window. For example, the work in [1, 5, 20] proposed algorithms for membership or heavy-hitter query in a *fixed-size* sliding window. However, our work deals with a dynamic query window size $d_i - a_i$, which varies across every query. ConQuest reads from a variable number of time window snapshots to solve the issue that query window size is unknown at insertion time. Additionally, as far as we know, our system is the first solution to be implemented within the computational model constraints of practical programmable switches.

10 Conclusion

We present ConQuest, a scalable framework for analyzing queuing and transient congestion in network switches in real time. ConQuest reports which flows contributed to the queue buildup, and enables direct per-packet action in the data plane to prevent transient congestion. We propose a novel framework to use ConQuest to monitor queuing in legacy network switches, and implement a P4-based prototype of ConQuest under this setup. Testbed evaluation and simulation experiments show ConQuest can achieve high accuracy in identifying the flows contributing to the queue buildup while using only $2KB$ register memory, a modest resource consumption on programmable switches.

References

[1] Yehuda Afek, Anat Bremler-Barr, Shir Landau Feibish, and Liron Schiff. Detecting heavy flows in the SDN match and action model. *Computer Networks*, 136:1–12, 2018.

[2] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. CONGA: Distributed congestion-aware load balancing for datacenters. In *ACM SIGCOMM Conference*, pages 503–514, 2014.

[3] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *ACM SIGCOMM Internet Measurement Conference*, pages 267–280, 2010.

[4] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *ACM SIGCOMM Conference*, pages 99–110, 2013.

[5] Vladimir Braverman, Ran Gelles, and Rafail Ostrovsky. How to catch L_2-heavy-hitters on sliding windows. *Theoretical Computer Science*, 554:82–94, 2014.

[6] Yanpei Chen, Rean Griffith, Junda Liu, Randy H Katz, and Anthony D Joseph. Understanding TCP incast throughput collapse in datacenter networks. In *ACM Workshop on Research on Enterprise Networking*, pages 73–82, 2009.

[7] The P4 Language Consortium. P4_{16} language specification. `https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html`, May 2017.

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

[9] Soudeh Ghorbani, Zibin Yang, Philip Brighten Godfrey, Yashar Ganjali, and Amin Firoozshahian. DRILL: Micro load balancing for low-latency data center networks. In *ACM SIGCOMM Conference*, pages 225–238, 2017.

[10] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *ACM SIGCOMM Conference*, pages 29–42, 2017.

[11] Keqiang He, Eric Rozner, Kanak Agarwal, Wes Felter, John Carter, and Aditya Akella. Presto: Edge-based load balancing for fast datacenter networks. In *ACM SIGCOMM Conference*, pages 465–478, 2015.

[12] Raj Joshi, Ting Qu, Mun Choon Chan, Ben Leong, and Boon Thau Loo. BurstRadar: Practical real-time microburst monitoring for datacenter networks. In *ACM SIGOPS Asia-Pacific Workshop on Systems (APSys)*, 2018.

[13] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *ACM SIGCOMM Conference*, pages 221–235, 2018.

[14] Rong Pan, Balaji Prabhakar, and Konstantinos Psounis. CHOKe-A stateless active queue management scheme for approximating fair bandwidth allocation. In *IEEE INFOCOM*, pages 942–951, 2000.

[15] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A centralized ”zero-queue” datacenter network. In *ACM SIGCOMM Conference*, pages 307–318, 2014.

[16] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating fair queueing on reconfigurable switches. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1–16, 2018.

[17] John Sonchack, Oliver Michel, Adam J Aviv, Eric Keller, and Jonathan M Smith. Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *Flow. In *USENIX Annual Technical Conference*, pages 823–835, 2018.

[18] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Distributed network monitoring and debugging with SwitchPointer. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 453–466, 2018.

[19] Nofel Yaseen, John Sonchack, and Vincent Liu. Synchronized network snapshots. In *ACM SIGCOMM Conference*, pages 402–416, 2018.

[20] MyungKeun Yoon. Aging bloom filter with two active buffers for dynamic sets. *IEEE Transactions on Knowledge and Data Engineering*, 22(1):134–138, 2010.

[21] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-resolution measurement of data center microbursts. In *ACM SIGCOMM Internet Measurement Conference*, pages 78–85, 2017.