PrintQueue: Performance Diagnosis via Queue Measurement in the Data Plane

Yiran Lei[†], Liangcheng Yu[‡], Vincent Liu[‡], Mingwei Xu[†]
[†]Tsinghua University, BNRist, Zhongguancun Laboratory [‡]University of Pennsylvania
{leiyr20@mails., xumw@}tsinghua.edu.cn, {leoyu, liuv}@seas.upenn.edu

ABSTRACT

When diagnosing performance anomalies, it is often useful to reason about *why* a packet experienced the queuing that it did. To that end, we observe that queuing is both a result of historical effects and the current state of the network. Further, both factors involve short and long timescales by nature. Existing work fails to provide insight that satisfies all of these needs.

This paper presents PrintQueue, a practical data-plane monitoring system for tracking the provenance of packet-level delays at both small and large timescales. We propose a set of metrics for describing 'congestion regimes' and present a set of novel data-plane data structures that accurately track those metrics over arbitrary time spans. We implement PrintQueue on a Tofino switch and evaluate it with multiple network traces. Our evaluation shows that the accuracy of PrintQueue is up to 3× times higher while the overhead is 20× times smaller than existing work.

CCS CONCEPTS

• **Networks** → **Data path algorithms**; **Programmable networks**; *In-network processing*; *Network monitoring*.

KEYWORDS

Queue measurement, Programmable networks, Data plane

ACM Reference Format:
Yiran Lei, Liangcheng Yu, Vincent Liu, and Mingwei Xu. 2022. PrintQueue: Performance Diagnosis via Queue Measurement in the Data Plane. In *ACM SIGCOMM 2022 Conference (SIGCOMM '22), August 22–26, 2022, Amsterdam, Netherlands.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3544216.3544257

1 INTRODUCTION

In today's networks, performance issues can come from many different sources, whether a DoS attack, an ECMP misconfiguration, TCP incast, or just an unlucky confluence of application flows converging at a single link. Performance issues can also yield different impacts, e.g., dropped packets, SLA violations, or a degraded user experience. However, almost all performance issues boil down to a packet getting to its destination late or not at all.

As a result, visibility into a network's queues is critical for diagnosing performance issues and answering questions such as: which other flows caused this packet to sit in this particular queue? Unlike other possible sources of delay like routing issues or failures, queues are simultaneously hard to predict (because of non-deterministic packet arrival timing) and hard to reason about after the fact (due to the high volume of traffic involved). What is more, the original *causes* of the delays can span arbitrary time scales. For shorter time scales, prior work has found that, in some large networks, microbursts as brief as 10s to 100s of microseconds are the norm, not the exception [35]. For longer time scales, differentiated classes of service, mechanisms like Layer-2 pause frames, and the cascading nature of queuing delays mean that the original causes of a delay can be far in the past. In fact, in the extreme case where a low-priority packet p is continuously delayed by higher priority traffic, the set of flows that caused p's delay is unbounded.

Existing work in flow measurement tends to perform poorly at the extreme timescales needed by queue diagnosis. For example, approaches like sketch-based heavy-hitter analysis [12, 14–17, 23–26, 34] typically operate over fixed windows of time. If a packet enters and exits the queue on these fixed window boundaries, the above class of systems can track concurrent flows precisely. If the packet does not, especially if it only spends a short time in the queue, then fixed-window approaches can grossly overestimate the presence, size, and impact of other flows. Packet-sampling approaches [10, 13, 18, 25, 37] suffer from similar issues, either necessitating heavy sampling or failing to scale to longer periods of congestion.

Work in queue monitoring is slightly more relevant but still lacks sufficient information to attribute delay precisely. For example, Conquest [6] is able to query whether a flow is a primary contributor to the current queue whenever the flow's packets enqueue. Unfortunately, it does not permit the reverse lookup: given a victim, determine the culprits in its queuing.

In this work, we argue that when trying to determine the causes of per-switch queuing delay, we must consider the current congestion regime holistically. For example, in a microburst, the early packets in the burst are, in some ways, just as culpable as the packet immediately prior to the victim—if either did not exist, the victim would be sent sooner. To that end, we present a taxonomy of the low-level causes of per-packet queuing delay. Our taxonomy consists of three types of culprits: packets that *directly* delay a victim packet, packets in the current congestion regime that *indirectly* delay the victim, and the *original* causes of the current congestion regime. Together, these categories paint a picture of the current period of congestion in its entirety: the first one captures the current causes of the network's congestion, and the last two capture its historical roots.

This paper presents PrintQueue, a monitoring framework that tracks the causes of queuing delays across an entire congestion regime. PrintQueue leverages the flexibility of modern programmable switch data/control planes to implement two novel mechanisms

SIGCOMM '22, August 22–26, 2022, Amsterdam, Netherlands

ACM ISBN 978-1-4503-9420-8/22/08.
https://doi.org/10.1145/3544216.3544257

that, combined, track each of the causes of delay outlined above. The mechanisms are designed for the specific challenges of each effect. When tracking flows that directly or indirectly impact a victim packet, PrintQueue effectively handles both nanosecond-level queuing and super-BDP impacts using a hierarchical, probabilistic flow-tracking data structure. When tracking the historical roots of congestion, PrintQueue efficiently handles the unbounded nature of the congestion regimes by formulating a simple summarization scheme that can implicate culprits at a packet-level granularity.

We implement PrintQueue on a Tofino switch and evaluate it with multiple network traces. More specifically, this paper makes the following contributions:

- We propose a set of metrics for describing a congestion regime. We classify the entire collection of responsible packets into three groups, i.e., direct, indirect, and original culprits. We show the necessity to track each group with real-world examples.
- We present PrintQueue, the first system to efficiently track an entire congestion regime. We design novel data structures, i.e., time windows and the queue monitor, for this purpose. The data structures are compatible with non-FIFO queuing policies.
- We validate PrintQueue with a hardware prototype. Our evaluation shows the accuracy of PrintQueue is up to 3× times higher than existing work while keeping the overhead 20× times smaller.

This work does not raise any ethical issues.

2 DESCRIBING A CONGESTION REGIME

Modern networks operate on increasingly tight deadlines. Both users and massively distributed computations expect and rely on low latency. For these networks, queuing can have a major impact on tail latency. Fundamentally, queuing delay is caused by congestion and its resulting queue buildups. In modern networks, these buildups tend to happen in waves. For example, consider microbursts, short-lived periods of high utilization that typically last for less than a millisecond and cause the majority of congestion in some data center networks [35].

In a microburst, the delivery of a packet p is based on the send time of the previous packet p_1, denoted as $p \leftarrow p_1$. However, the victim p 's delay is not just the previous packet p_1's fault. p_1's send time is decided by its previous packet p_2, i.e., $p_1 \leftarrow p_2$. Similarly, $p_2 \leftarrow p_3$, and so on. In each case, eliminating the other packets would have led to an earlier send time for p, and thus, they contributed to p's delay. So the culprits for p's delay are the packets that directly or indirectly point to p ($\leftarrow p_1 \leftarrow p_2 \leftarrow p_3 \cdots$). This chain of blame can extend all the way to the beginning of the microburst. Notably, one cannot assign blame to any packet from before the start of the microburst.

In this work, we argue that when trying to attribute the cause of a delay, one must consider the entire congestion regime, i.e., the period extending from when the victim packet finally leaves the queue back to when the queuing first began. More formally, consider a queue with an arbitrary packet scheduling algorithm and the burst of packets depicted in Figure 1. All packets depicted are at least partially culpable in the victim's queuing delay, not just those in the queue when the victim arrives at $t = 4$.

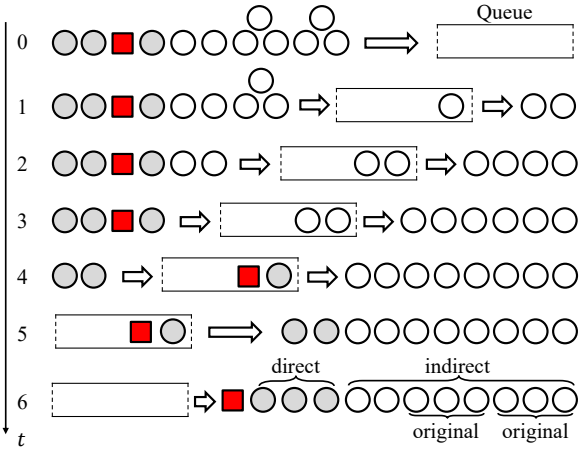


Figure 1: The congestion regime of a single burst of packets. The red square represents a lower-priority victim packet; all other packets are higher priority. The white packets increase the queue depth to 2 and sustain the level. The grey packets also impact the victim because of their high priority.

We further argue that we can comprehensively categorized these packet-level causes of queuing delay into three groups:

Packets that *directly* delay the victim. For a victim packet that is enqueued at t_1 and is dequeued at t_2, the packets that directly contribute to the delay of t_2 are precisely those that were dequeued between t_1 and t_2. This definition is independent of the packet scheduling algorithm. In Figure 1, directly culpable packets are marked in grey. In essence, the switch chooses to deliver these grey packets instead of the victim.

Identifying direct culprits is essential to diagnosing many real performance issues. For example, knowing the makeup of these flows can reveal which flows are competing with the victim flow and, e.g., whether those flows are just a few heavy hitters or a collection of smaller higher-priority requests.

Packets that *indirectly* delay the victim. Using the same scenario, packets that indirectly impact the victim are those that do not directly delay the victim but may have (indirectly) caused the queuing of a packet that did. More precisely, these are packets whose dequeue time, t_2' is before the victim's enqueue time, t_1, and where the queue depth is greater than zero for the entire period $[t_2', t_1]$. The union of direct and indirect culprits equals the complete congestion regime.

Identifying indirectly culpable packets is also important. For instance, in the case of TCP incast or otherwise synchronized traffic patterns [30], these congestion regimes are characterized by the entire burst containing a single application's traffic. In the light of that, knowing indirect culprits can help identify the synchronized behavior and the fact that there is sufficient capacity surrounding the burst, which can be utilized by de-synchronizing the sends.

Packets that are the *original* causes of the congestion. Finally, out of the indirectly culpable packets, a subset of packets have slightly more blame—the packets that brought the queue to its current level. Specifically, for a queue depth of n packets, there are

at least n packets whose arrival increased the depth of the queue. In Figure 1, these are the packets that enqueue during $t = [0, 1]$.

Identifying these historical causes of queue buildup can also be essential to differentiate specific types of behavior. Consider a scenario where several large TCP WAN connections are sharing a link but properly managing the queue. If a sudden burst of UDP datagrams arrives, the queue will quickly balloon and stay high before TCP has time to react. For a subsequent victim packet, the direct culprits will not contain the burst—it has long since left the network. The indirect culprits will also be misleading—the majority (by volume) are the other TCP flows. The queue buildup will properly implicate the datagram burst.

3 DESIGN OVERVIEW

For a victim packet, PrintQueue identifies each of the above types of culprit packets. It further aggregates the culprits according to their flow ID to get per-flow packet counts. The packet counts serve as a measure of the flows' contribution to the victim's queuing delay and a guide for the network operator's subsequent actions. Specifically, PrintQueue identifies each culprit flow with its:

- Flow ID, expressed as 5-Tuple (i.e., source and destination IP addresses, ports, and protocol ID).
- Contribution, expressed as the total number of culprit packets.

Three key ideas underlie PrintQueue's design.

(1) To accurately track culprits at fine-granularity, PrintQueue should store individual packets and their dequeue times: For both direct and indirect culprits, their definition and discernment rely on their dequeue timestamps and their relation to the victim packets' queuing time. PrintQueue designs a data structure that enables networks and operators to query an arbitrary time range (small or large) for queuing-delay culprits. This *query interval* may correspond to the total queuing time of a victim packet or the period of a single burst.

As the query interval is arbitrary, PrintQueue can serve as a general framework for higher-level queue diagnosis tasks. For example, operators can trigger a query when they receive a customer complaint about high delays. Alternatively, the egress pipeline in the data plane can automatically trigger a local query when it detects high queuing for important traffic.

(2) To ensure scalability to unbounded time scales, PrintQueue must compress packets to reduce overhead: While (1) provides the ability to query for culprit packets at fine granularity after-the-fact, its storage requirements are intractable. PrintQueue instead seeks to store packets belonging to exponentially-growing periods in only linearly-growing space. The data structure we propose is a time window.

While time windows necessarily sacrifice accuracy, PrintQueue leverages a hierarchical approach to create tiers of tracking accuracy that are ordered by recency. In fact, for the most recent 'time window,' every packet is tracked precisely. As the packet ages, PrintQueue compresses it into successively more approximated structures that mirror packets to increasingly long intervals. Figure 2(b) depicts this process and compares it to the more linear approach taken by existing work.

(3) To prioritize tracking of the original causes of congestion and do so efficiently, PrintQueue should maintain a distinct list of packet-level

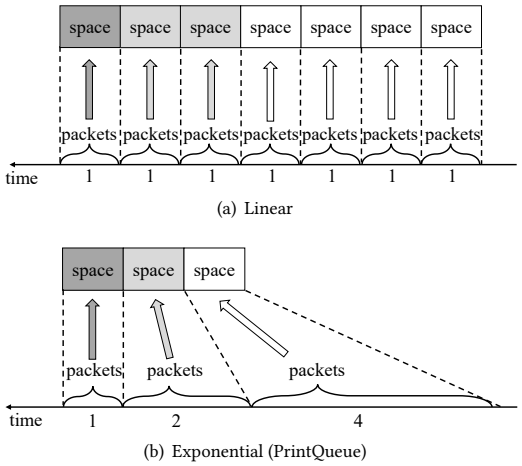


Figure 2: The linear storage of most existing work versus the exponential storage approach of PrintQueue's time windows mechanism. In PrintQueue, the most recent time period (darkest) stores packets in full fidelity. Note that PrintQueue's space advantages hold even if the linear storage is not full fidelity (but is still proportional), e.g., in the case of a sketch.

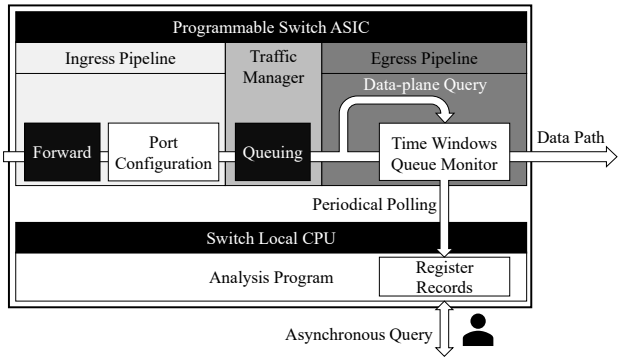


Figure 3: The per-switch PrintQueue architecture. PrintQueue is activated in specific ports by port configuration. It tracks culprits with time windows and the queue monitor, which are periodically checkpointed by the control plane. Queries can be initiated either remotely (via an asynchronous request) or locally (in the egress pipeline).

'high-water' marks: Finally, to find the original culprits, PrintQueue reserves a unique data structure that stores packets causing queue growth and evicts packets as the queue drains. The insertion and eviction of packets track queue variations precisely. The data structure, queue monitor, explains which packets in history brought the queue depth to its current level.

Architecture. As illustrated in Figure 3, modern switches are split into ingress and egress packet processing pipelines, with the switch buffer placed between the two. While different ports may share buffer space or the same physical pipeline, queuing delay is almost entirely a function of the activity on each independent egress port.

Network operators first enable PrintQueue on a per-egress-port basis. On those ports, PrintQueue tracks culprits with two data-plane components: time windows (for tracking both direct and

Metadata	Description
`egress_spec`	The output port where a packet is forwarded.
`enq_timestamp`	The timestamp when a packet enqueues.
`deq_timedelta`	The time that a packet spends in the queue.
`enq_qdepth`	The queue depth when a packet enqueues.

Table 1: Metadata required by PrintQueue.

indirect culprits) and the queue monitor (for tracking the original causes of congestion).

In the control plane, the analysis program periodically polls and stores the contents of the time windows and queue monitor. Finally, higher-layer applications query the culprits either by sending a request to the analysis program or initiating a synchronous query in the data-plane egress pipeline. The advantage of the latter is that it can query the time windows before they age the directly culpable packets into a less accurate window.

Metadata requirements. PrintQueue uses the metadata fields in Table 1. This information is provided by both Tofino chip [2] and BMv2 Simple Switch target [1]. The flow ID can be derived directly from packet header contents.

We describe PrintQueue's components in detail below.

4 TIME WINDOWS

Time windows are a hierarchical and probabilistic data structure whose purpose is to answer queries of the form: for a given egress port and query interval, which flows are occupying the port, and how many packets do they contribute?

4.1 Physical Layout

Figure 4 depicts a set of T time windows (indexed from 0 to $T-1$). Each time window is implemented with a stateful register array consisting of 2^k *cells*. Cells are the smallest building block of a time window and always hold the information of just a single packet.

The entire set of T time windows covers a fixed, contiguous timespan, called the *set period*. Each time window also covers a contiguous subset of the set period called a *window period*. Different time windows may cover differently sized window periods. Similarly, each cell covers a contiguous subset of a window period called a *cell period*. Like window periods, cell periods can also differ in size, although all cells in a single time window will have the same cell period. In this way, every point in time within the set period can be attributed to precisely one time window and cell, although a cell and its single packet's worth of information can represent a relatively large span of time (see Section 4.3 for why that is okay).

A key feature of time windows is that each successive window period (and, thus, their cell periods) is exponentially larger than the last. As hinted in Figure 2, if the length of window period 0 is 1 unit, the length of window period i is $2^{\alpha i}$ units, where α is a configurable compression parameter.

In each time window, the cell array functions as a ring buffer. As time passes, PrintQueue continually writes packet information into the time windows, evicting and overwriting the oldest cells of each. Writing begins at cell 0 and proceeds to cell 2^k-1 before it loops back to 0. To handle ring-buffer overflows, each cell stores a monotonically increasing *Cycle ID* to distinguish cell periods from

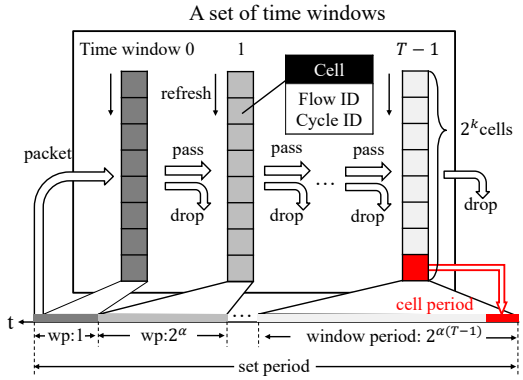


Figure 4: The physical layout of a set of T time windows. The bottom of the figure illustrates the timeline of packets passing through this set of time windows and the relevant granularities of time in PrintQueue.

32-bit timestamp – 0xAAA9105A

13bits	12bits	
Trimmed Timestamp (TTS)		
Cycle ID	Index	
1010101010101	001000100000	1011010

Figure 5: An example to calculate cycle ID and index of timestamp 0xAAA9105A. The raw bits of the timestamp are shown in the bottom row.

different cycles. As newer packets evict older ones, PrintQueue combines the oldest 2^α cells from time window i and stores the combined value as the newest cell in time window $i+1$. This process occurs recursively, sliding all of the windows as it goes.

To ensure packet-level granularity, PrintQueue sets the cell period of time window 0 to a value less than the transmission delay of a minimum-sized packet in the target network (e.g., 64 B). This means that in time window 0, there will be no cell-level collisions—each cell has at most one packet to store every cycle. Time window 0 typically has thousands of cells, making the window period more than 100 µs. Note that this means queries for the culprits in microbursts (lasting for 10s to 100s of microseconds) are often guaranteed to have full precision and recall. Queries triggered soon after the incurred delay are similarly advantaged.

4.2 Per-packet Procedure

We now delve deeper into the per-packet procedure and PrintQueue's method for retrieving culprits across arbitrary query intervals. PrintQueue begins with the packet's flow ID and dequeue timestamp. The dequeue timestamp is computed as (`enq_timestamp` + `deq_timedelta`).

Every packet then enters time windows at time window 0. The cell index in time window 0 is based on the low bits of the dequeue timestamp. In the beginning, time window 0 is empty, and all packets are placed directly. Over time, however, PrintQueue may need to evict older packets. These packets are either dropped or passed to the next time window as new inputs, with all time windows repeating the process recursively. The rules for mapping packets to certain positions in a window and deciding whether to pass packets to the next time window are as follows.

Mapping rule. For a given time window, the mapping rule is used to compute the cell index and cycle ID of the target packet. Both can be computed from the dequeue timestamp using simple bitshifts.

For the first time window, PrintQueue shifts the timestamp to the right by $m_0 = \lfloor \log_2(min\_pkt\_tx\_delay) \rfloor$ bits to obtain a trimmed time-stamp (TTS). In today's switches with nanosecond clocks, $m_0 = 6$ (64 ns) is typically sufficient. As each time window has 2^k cells, the k least-significant bits of the TTS are used to index into the appropriate cell, and the remainder of the TTS is stored as a cycle ID. Figure 5 gives an example of this breakdown for an example timestamp with $m_0 = 7$, $k = 12$.

Recall that, when evicting, 2^α cell periods are combined and passed to the next window. Thus, each subsequent time window shifts the TTS by an additional α bits; again, the lowest k bits of new TTS are the index and the rest are the cycle ID. For example, suppose $\alpha = 1$ and $k = 12$. In window 0, two cells with TTS `0x3fff000` and `0x3fff001` are mapped into the same cell of window 1, whose TTS is `0x1fff800`. Algorithm 1 shows the procedure, in context.

More formally, time window i compresses all the packets in a period of length $2^{m_0+\alpha i}$ to a single cell, and a span of $2^{m_0+\alpha i} \times 2^k = 2^{m_0+\alpha i+k}$ into a window period. All together, the set period lasts for a span of $\sum_{i=0}^{T-1} 2^{m_0+\alpha i+k} = \frac{2^{\alpha T}-1}{2^\alpha-1} 2^{m_0+k}$.

Passing rule. Whenever there is a collision in a cell, PrintQueue always chooses the newer one. PrintQueue applies a passing rule to determine the fate of evicted packet record, i.e., whether it is dropped or carried to the next time window.

Algorithm 1 (lines 6–11) shows the passing rule logic. When a new packet arrives at a time window, PrintQueue always stores it, whether the cell is empty or not. If the cycle ID of the new packet is larger than that of the evicted packet by exactly one, it passes the evicted packet to the next time window as a new packet. Intuitively, this means that PrintQueue only has one shot to pass each packet to the next window—in the next window period immediately following the packet's arrival—and it will only pass the packet if it encounters a packet sharing the same cell index during that window period. It will not pass competing packets with the same cycle ID and cell index. Packets that it does not pass will be deleted asynchronously when another packet arrives in a future cycle.

Thus, deeper time windows correspond to older and larger periods of time. When a packet is passed into a given time window, it is guaranteed to be the newest one.

Example. Figure 6 shows a concrete example of both rules in action. During time step 1, cell 0 and cell 1 of time window 0 are both passed to cell 0 of time window 1. Flow A's packet arrives first, but is evicted when flow B's packet arrives. Because the two packets have the same cycle ID, flow A's packet is directly dropped instead of being passed to the next time window.

At the end of time step 2, A's incoming packet in cell 3 evicts D's packet in window 0. D's packet will *not* be passed, as its cycle ID is too far in the past.

At the end of time step 3, B's packet in cell 0 of time window 0 is pushed out by the incoming A packet and passed to cell 0 of time window 1. Because the cycle ID of the window 1 packet is exactly one less than the incoming packet's cycle ID, the window 1 packet *will* be passed to cell 0 of window 2. The newly added packet in cell 0 of window 1 will be replaced in time step 5.

Algorithm 1: Time windows data-plane algorithm

```
   Input: packet p, m_0, windows, k, α, T
1  i = 0                                          (i is the time window index)
2  p.TTS = p.dequeue_timestamp >> m_0
3  while i < T do
4      p.Index = p.TTS & (2^k − 1)
5      p.CycleID = p.TTS >> k
6      e = windows[i][p.Index]
7      windows[i][p.Index] = p
8      if p.CycleID - e.CycleID == 1 then
9          p = e                                  (pass the evicted packet)
10     else
11         break                                  (drop and stop)
12     p.TTS = p.TTS >> α
13     i++                                        (to the next window)
14 end
```

4.3 Analysis and Proofs

Time windows intentionally drop some packets to better compress the data. To recover from this loss, they leverage several attributes.

Theorem 1. If the probability of a packet arriving at cell j in a window period is z_j, then the probability that *no* packet is passed from cell j during the next window period is $1 - z_j^2$.

Proof. To pass a packet in the next window period, there must be two incoming packets in the current and next window period, respectively. The probability of this occurring is z_j^2. Otherwise, the probability of no passing is $1 - z_j^2$. □

If z_j is i.i.d., let $z = z_0 = \cdots = z_{2^k-1}$ and $p = 1 - z^2$.

Theorem 2. If z_j is i.i.d. and there are n new packets stored in the cells during the current window period, then:

- During the next window period, the subsequent window is expected to store $(\frac{1}{2^\alpha} z \frac{1-p^{2^\alpha}}{1-p})n$ new packets, passed from the current window.
- Subsequent time window's z_j, the probability that cell j stores a new packet in every window period, is $1 - p^{2^\alpha}$.
- Subsequent time window's z_j is i.i.d.

Proof. For simplicity but without loss of generality, randomly select one from the new packets that are stored during the current window period. For it to be stored in the next window period, there are two requirements. First, the cell must have an incoming packet during the next window period. The probability is z. Second, after the selected packet is passed, no later packets of the same window period push it out. 2^α cells, indexed from 0 to $2^\alpha - 1$, of the current window are mapped to the single cell of the subsequent window. Assume the selected packet falls into cell m ($0 \le m \le 2^\alpha - 1$). Cells b ($m + 1 \le b \le 2^\alpha - 1$) should not pass any packets during the next period. According to Theorem 1, the probability is $p^{2^\alpha-1-m}$. Since the selected packet has equal probability to fall into any of the 2^α cells, the probability that no competing packets push out the selected packet is $\frac{1}{2^\alpha} \sum_{m=0}^{2^\alpha-1} p^{2^\alpha-1-m} = \frac{1}{2^\alpha} \frac{1-p^{2^\alpha}}{1-p}$. With two requirements satisfied simultaneously, the above probability that

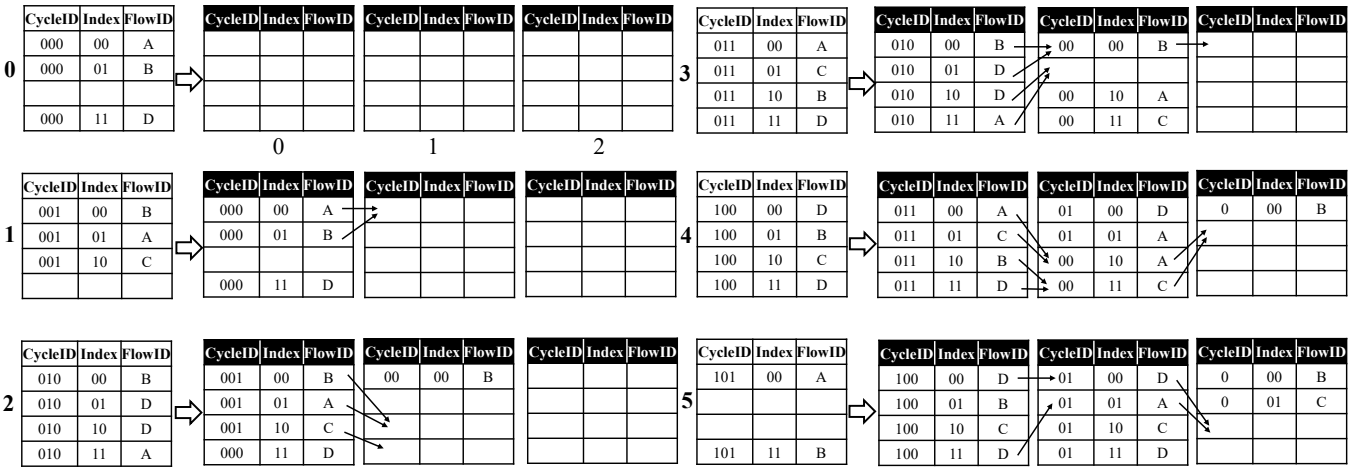


Figure 6: Example of time windows in action; $k = 2$, $T = 3$, $\alpha = 1$. The diagram shows 6 time steps. In each, the incoming packets are shown on the left; tables with black headings are the time windows. Arrows show packet movements during time steps.

the selected packet is stored in the subsequent window becomes $\frac{1}{2^\alpha} z \frac{1-p^{2^\alpha}}{1-p}$. Since we select a packet randomly, the probability is equivalent for all the new packets of the current window period. Hence the subsequent window is expected to store $(\frac{1}{2^\alpha} z \frac{1-p^{2^\alpha}}{1-p})n$ packets.

The packet in each cell of the subsequent window comes from any of 2^α cells of the current window. The probability that no cell passes packets is p^{2^α}. Otherwise, the probability that a cell in the subsequent window stores a new packet is $1 - p^{2^\alpha}$.

The above proof applies to all cells in the subsequent window so the subsequent window's z_j is i.i.d. □

PrintQueue uses the **proportional property** of Theorem 2 to recover the original packet counts from the compressed data. In the beginning, a flow with n packets is stored in the current time window. As time goes by, the current window stores new packets, dropping some of the n packets and passing the rest. In the subsequent window, the packet count of the flow is compressed as we can only observe some of the n packets. Theorem 2 says that the observed number is proportional to the original number n in the preceding time window. We can easily recover the original number by dividing the observed number by the ratio $\frac{1}{2^\alpha} z \frac{1-p^{2^\alpha}}{1-p}$.

Theorem 2 also shows that from the first time window, the proportional property extends to all the time windows, each with new z, p calculated from preceding windows'. We recover the packet number all the way back to the first time window by repeating the process: divide the number by the ratio between neighbor time windows. Recall that the first time window tracks packets precisely. Therefore, the estimated packet count in the first window is our target value.

PrintQueue introduces *coefficient* to simplify the recovery process. PrintQueue first computes the ratio between neighbor windows. Then, PrintQueue defines coefficient[i] as the ratio of packet count in window i to the packet count in the first window. Apparently, coefficient[0] is 1. For deeper windows, multiply the ratios recursively to get coefficient[i] as shown in Algorithm 2. Finally, if we observe a flow with n packets in window i, the flow's real packet number in that period is expected to be n / coefficient[i].

The proportional property only provides an expected value without any error bounds. Ideally, if the packets of all the flows fall randomly into every 2^α cells, the errors are minimal, because there is no bias on passing specific flows' packets. The errors still exist, because for extremely small flows, none of their packets will survive when traversing through windows multiple times. In practice, after queuing, packets enter time windows not in the ideal way, but near randomly. Before packets causing a congestion get enqueued, each one is likely to experience small random delays when traversing network ends, links, and switches. So packets of different flows are slightly randomized in the queue, making near-random entry into time windows in the egress pipeline. We show in Section 7 that under different workloads the errors are limited.

Next, we need a concrete value of z in the **first** window to apply Theorem 2 to all the time windows. Theorem 3 describes the necessary assumptions to get the value and proves their sufficiency. Suppose the transmission delay of minimal-sized packets at line rate is d. The length of cell periods in time window 0 is 2^{m_0}. $2^{m_0} \leq d$.

Theorem 3. If the port of switch forwards packets at line rate and the number of cells, 2^k, is large, then:

- The first window's z_j, the probability of cell j storing a new packet during every window period, is $\frac{2^{m_0}}{d}$.
- The first window's z_j is i.i.d.

Proof. The window period 0 is 2^{k+m_0}. The number of new packets in that period is $\frac{2^{k+m_0}}{d}$. With 2^k cells and no packet collisions in the first window, the probability that a cell stores a new packet every window period is $\frac{2^{k+m_0}}{d} \div 2^k = \frac{2^{m_0}}{d}$. Suppose a cell has already stored a new packet. There are still $(\frac{2^{k+m_0}}{d} - 1)$ packets coming in the window period. Each of the $(2^k - 1)$ unoccupied cells has the probability of $(\frac{2^{k+m_0}}{d} - 1) \div (2^k - 1) \approx \frac{2^{m_0}}{d}$ ($\approx$ because k is large) to store a new packet. Therefore, whether a cell has already stored a new packet does not affect the probability of the rest, proving z_j is i.i.d. □

Algorithm 2: Coefficient algorithm

Input: transmission delay of minimal-sized packets d, m_0, α, T
Output: coefficient

```
coefficient[0] = 1
i = 1
z = 2^{m_0} / d
acc = 1
while i < T do
    p = 1 − z^2
    acc= acc × (z × (1 − p^{2^α}) / (1 − p) / 2^α)
    coefficient[i] = acc
    z = 1 − p^{2^α}
    i++
end
```

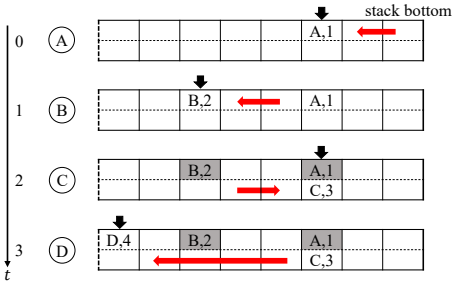


Figure 7: Example of queue monitor in action. The queue monitor is updated with each incoming packet in circle; black arrows represent stack top pointers; red arrows indicate stack increase/decrease; grey entries are stale.

In practice, PrintQueue sets the number of cells per window, 2^k, to be a large number, typically 4096. Besides, time windows diagnose performance issues at the time of congestion, indicating the switch is forwarding packets at line rate in specific ports. Therefore, with $z = \frac{2^{m_0}}{d}$ in the first time window, PrintQueue calculates the z, p and coefficients recursively and estimates per-flow packet counts from all the time windows.

We note that time windows have higher accuracy in estimating the packet counts of recent traffic that just enters. Their packets are located in the initial time windows, with only a small portion of dropped ones. Old traffic, on the contrary, is heavily compressed in the deep windows, causing larger errors. Therefore, time windows' accuracy is biased on traffic's recency. PrintQueue leverages the feature and designs the data-plane query to improve accuracy. We introduce queries in Section 6.

5 QUEUE MONITOR

We augment the time window mechanism with a queue monitor that tracks the original causes of the current congestion regime for each port. The queue monitor uses `enq_qdepth` packet metadata to learn the queue depth observed by every packet. For ease of exposition, we discuss tracking for a single port and class of service; multiple queues are tracked individually.

The primary challenge in the design of the queue monitor is that the goal of the mechanism—to keep the original causes of congestion—is fundamentally opposed to the recency bias of time windows and switch queues. Instead, to retain packets' influence

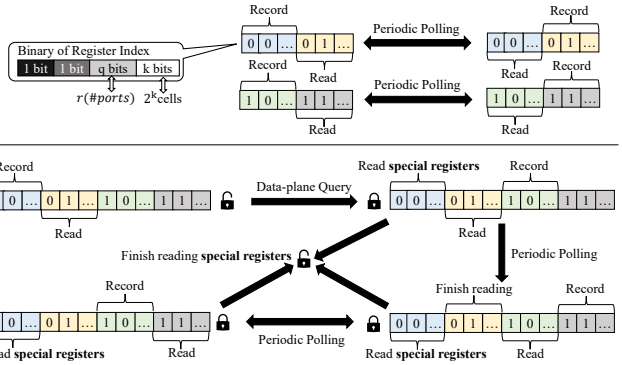


Figure 8: The decomposition of register index is shown in the top left corner. As shown in the upper part, PrintQueue flips the second-highest-order bit for asynchronous query. In the lower part, PrintQueue flips the highest-order bit for data-plane query and locks itself until the completion of reading the special registers.

for an arbitrary time, PrintQueue's queue monitor is structured as a *sparse stack*.

Conceptually, the queue monitor is a register array with length equal to the maximum length of the queue divided by the buffer allocation granularity. Another register, acting as a 'stack top' pointer, stores the latest queue depth at the time of enqueue. In the egress stages, whenever a packet changes the queue depth ($l_1 \rightarrow l_2$), the packet's flow ID will be added to the l_2 register entry along with a monotonically increasing sequence number. Each entry consists of two parts. The upper half stores metadata for depth increases, and the lower half stores decreases. PrintQueue updates the top pointer in both cases.

In this mechanism, some entries (even those 'under' the top pointer) may be empty or filled with stale packets. Consider the instance in Figure 7: (1) at $t = 1$, packet B brings the queue from a depth of 2 to 5 units, (2) at $t = 2$, the queue drains back to 2, and (3) at $t = 3$ packet D brings the queue up to 7 units. Entries at 2, 5, and 7 record depth increases, but the entry at 5 is from a previous peak. PrintQueue can correct for this using the aforementioned sequence numbers. Specifically, the analysis program can, after the fact, walk the array starting from 0 to the current value of the top pointer and make note of the largest sequence number observed thus far. Entries are only considered if they have a higher sequence number than previous entries.

The above algorithm may generalize to other scheduling algorithms. In particular, we note that efficient queue management at high bandwidth puts certain restrictions on feasible hardware. Others have observed this and created general frameworks for constructing advanced scheduling out of smaller FIFO queues [20, 22, 32, 33]. The queue monitor can track each priority or rank separately.

6 ANALYSIS PROGRAM

The control-plane analysis program runs on the switch's control CPU. The analysis program has three main functions: (1) configure PrintQueue on specific ports, (2) checkpoint/collect time window and queue monitor data-plane state, and (3) execute queries.

6.1 Port Configuration

Users can activate the time windows and queue monitor on a per-port basis, and they will track each port's queues separately. Under the hood, users first specify the number of ports activating PrintQueue, denoted as #*ports*. PrintQueue rounds up #*ports* to the nearest power of 2, denoted as $r(\#ports)$. PrintQueue then allocates several large register arrays (one for the queue monitor and T for the time windows). Every register array consists of $r(\#ports)$ partitions, each intended for the use of a single port. The size of the queue monitor array is a function of the number of ports and maximum queue depth; the sizes of the time window arrays are a function of the number of ports and the size of the time windows structure.

PrintQueue contains a flow table in the ingress stages that matches on the egress port and gates activation of PrintQueue's mechanisms. Specifically, the flow table matches the destination port and returns the prefix of the port's registers (i.e., the value of the q bits in Figure 8). If no matching is found, the packet is ignored.

6.2 Frozen Register Reads

While the time windows and queue monitor are updated on every packet, the analysis program reads them on a much coarser granularity. Its reads are triggered in two ways: periodically (to checkpoint the state) and on-demand (e.g., as a result of a data-plane triggered query).

Periodic reads. The analysis program saves the values in the time windows and queue monitor every set time in order to ensure that it has culprit information for any possible query interval. More specifically, a set of time windows covers a contiguous time span of $t_{set} = \frac{2^{\alpha T}-1}{2^{\alpha}-1} 2^{m_0+k}$. PrintQueue must capture a snapshot of the register state at least once per t_{set} before oldest unread values are aged out of the time windows.

To ensure atomic and serializable reads of all of the data, PrintQueue borrows a technique from Mantis [31] and periodically 'freezes' the full set of time windows and queue monitor. While PrintQueue reads the frozen copy, the data plane continues to update a second set of registers. As shown in Figure 8, PrintQueue implements this by flipping the second-highest-order bit in the register index every t_{set}, when the register set is fully loaded.

On-demand reads. Reads can also be triggered on-demand to take advantage of time windows' recency bias, i.e., that recovery from the initial time windows tends to be more accurate. Examples of on-demand triggers include packets with unusually high queuing delay, sampled members of a high-priority flow, or a special end-host-generated probe.

In these cases, when PrintQueue sees a packet that requires diagnosis, the data plane immediately freezes the current data, directs subsequent per-packet updates to a third set of registers, and sends a notification to the control-plane analysis program. Periodic updates will flip between the two unused sets of registers. The analysis program, upon receiving the notification, knows the existence of the on-demand read and starts to read the recently frozen register set (we call it the 'special' registers). The notification contains the triggering packets' enqueue and dequeue timestamps, which can act as the query interval. As shown in Figure 8, PrintQueue implements this by flipping the highest-order bit of register index in the data plane. Note that only a single on-demand read can be in progress at any point. Concurrent reads will be temporarily ignored until PrintQueue can finish reading the special register set.

We note that the time periods covered by the periodically polled registers and special registers do not overlap, because packet at any time point would belong to only one register set.

Algorithm 3: Filter algorithm for time windows

```
   Input: windows, T, k, α
1  i = 0
2  TTS, CID, Idx = LatestCell(windows[0])
3  for i < T do
4      j = 0
5      for j ≤ Idx do
6          if windows[i][j].CycleID ≠ CID then
7              windows[i][j] = nil
8          j++
9      end
10     for j < 2^k do
11         if windows[i][j].CycleID + 1 ≠ CID then
12             windows[i][j] = nil
13         j++
14     end
15     TTS = (TTS −2^k) >> α                (the most recently passed cell)
16     Idx = TTS & (2^k − 1)
17     CID = TTS >> k
18     i++
19 end
```

6.3 Query Execution

After reading the registers, the analysis program stores the values for use in query execution. Queries are distinguished by whether they target information in the time windows or the queue monitor, which accept different inputs and return different results:

- Time window queries accept a query interval as input and return an estimate of the per-flow packet counts over that period, whether for direct or indirect culprits.
- Queue monitor queries accept a query point as input and return the list of original causes of congestion at the time instant closest to the input time.

The queries are also classified into two types: asynchronous and data-plane queries. The former accept arbitrary query intervals/points in the control plane and retrieve packets from all the registers. The latter, however, are initiated by packets in the data plane, leverage the on-demand reads, and retrieve packets from the special registers. Both are eventually executed by the analysis program.

Time window queries. Querying time windows involves two steps: filtering out stale cells and accumulating packet counts. Filtering (Algorithm 3) is applied once to remove old packets that have not yet been evicted from the raw time windows. **LatestCell**() in line 2 iterates through all the cells in a window, finds the latest one, and returns its TTS, cycle ID, and cell index. PrintQueue only retains cells that are either (1) in the same cycle ID or (2) in the previous cycle ID with an index greater than the latest cell, i.e., within one window period of the most recent cell.

When a query arrives, PrintQueue first determines the set of applicable time windows. If the query interval crosses multiple

windows or window sets, PrintQueue splits it into disjoint pieces. In each time window, it divides the per-flow packet counts by the corresponding coefficient$[i]$. Finally, PrintQueue aggregates the results from each window.

Queue monitor queries. Queue monitor queries also involve a filtering and a retrieval step. Filtering is necessary to remove stale entries that arise from the combination of fluctuating queue depths and the sparse layout of the data structure. It occurs exactly as described at the end of Section 5. As mentioned, when a query arrives, PrintQueue returns the queue monitor snapshot closest to the query time.

7 EVALUATION

We implement a prototype of PrintQueue on a Tofino programmable switch. Time windows need 4 MAU stages for preparations and two additional stages for each time window. The queue monitor uses six, but these can be overlapped with the above. PrintQueue consists of ~5000 lines of code in total.

7.1 Time Windows Performance

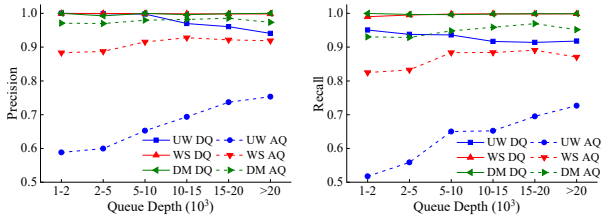
Testbed and workload. To evaluate the time windows mechanism, we use a hardware testbed consisting of a single Tofino switch and 4 Linux servers. Each server has 2× 2.40 GHz Xeon E5-2620 v3 CPU and 64 GB RAM. Two servers send traffic through 40 Gbps links, while the other two receive the traffic through 10 Gbps links.

For workloads, we utilize the University of Wisconsin Data Center Trace [4] (abbrev. UW) and two synthetic traces modeled after well-known flow size distributions. The first pattern is from web search tasks [3] (abbrev. WS), while the second is from a data mining cluster [9] (abbrev. DM). Flows and packets arrive according to Poisson processes. We use `tcpreplay` to emulate the TCP packet traces. To scale up the traces to today's link speeds, we leverage the `tcpreplay` multiplier option and the Netmap [19] driver to ensure the kernel can keep up. The two senders replay different pcap files.

In order to capture the ground-truth, the switch inserts a telemetry header into every packet that contains the enqueue/dequeue timestamps and queue depth at the packet's enqueue time. This header is not required in a real PrintQueue deployment—only to compute our evaluation metrics. On the receiver, the server leverages DPDK [7] to process packets at line rate and store the telemetry headers in files. The ground-truth per-flow packet counts are later computed by parsing the files for their dequeue timestamps.

Methodology. We evaluate a range of configurations and examine several classes of packets in each. To evaluate worst-case performance, we assume asynchronous queries on periodically read data unless otherwise specified. When we evaluate on-demand queries, we examine performance for the packet that triggered the lookup. Regardless of the query type, we choose a victim packet and provide its enqueue and dequeue time to the analysis program as the query interval. Note that this corresponds to a query for the directly culpable packets, but queries for indirect culprits are identical.

Separately, we examine the logged telemetry headers to compute the ground truth of which packets were dequeued during the target period. With both the time windows and the ground truth per-flow packet counts, we use precision and recall to calculate the accuracy of PrintQueue. We first compute, for every flow in the query period, the true positives of PrintQueue. Precision is the sum of the true positives over PrintQueue's cumulative packet count estimate. Recall is the sum of the true positives over the ground truth's cumulative estimate. The time window result is equivalent to the ground truth if and only if both precision and recall are 1.



(a) precision (b) recall

Figure 9: Precision and recall versus queue depth under different workloads.

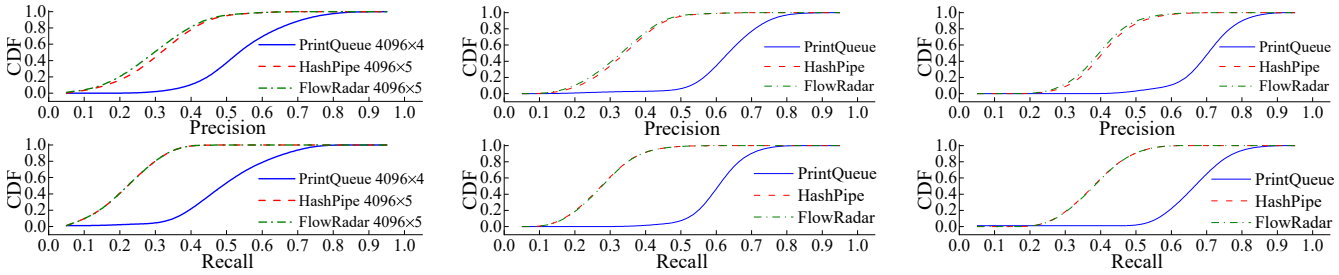
Accuracy versus queue depth. We begin by analyzing accuracy as a function of queue depth under our three workloads.

For a given victim packet, we classify its query into six groups based on the queuing it encounters: 1k to 2k, 2k to 5k, 5k to 10k, 10k to 15k, 15k to 20k, and above 20k. For asynchronous queries (abbrev. AQ), we randomly sample 100 victim packets experiencing each queue depth, query their direct causes of congestion, and compute precision and recall of the results (larger sample sizes produced similar results). For on-demand data-plane queries (abbrev. DQ), we add a threshold in the data plane that initiates a query if they observe each queue depth.

Figure 9 shows the average accuracy for each queue-depth group. For data-plane queries, the accuracy is consistently high (>90%) because the queries are predominantly touching the least compressed time windows. Accuracy decreases slightly for longer query intervals as the first time window can no longer hold all the packets of the target interval, pushing some culprits into deeper windows. Somewhat surprisingly, for asynchronous queries, we see the opposite trend: the accuracy is higher for longer query intervals, but decreases for shorter intervals as intervals have a chance of falling into a more heavily approximated time window, which has a disproportionate effect on short query intervals. Note that while data-plane queries are always more accurate than asynchronous queries, they must read an extra set of registers, which has a rate limited by the efficiency of control plane polling. Thus, operators should be judicious about initiating data-plane queries.

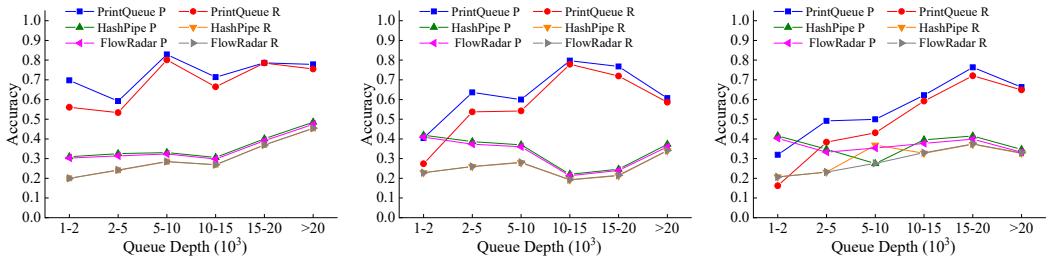
The accuracy differences among the three traces primarily stem from packet size (UW: around 100 bytes, WS/DM: near MTU). With a constant link rate of 10 Gbps, packets are forwarded at different rates (UW: 9.1 Mpps with average packet interval 110 ns, WS/DM: 0.84 Mpps with average packet interval 1200 ns). We choose $m_0 = 10$ and a smaller compression factor $\alpha = 1$ for WS/DM while $m_0 = 6$, $\alpha = 2$ for UW. $T = 4$ and $k = 12$ for all. Fundamentally, the accuracy of UW is lower because the number of packets to track is near 10× times larger than in WS/DM. Because of that, UW has to use a bigger compression factor $\alpha = 2$, leading to bigger errors.

Our `Python` analysis program front end can execute ~100 queries per second.



(a) 1k – 5k (b) 5k – 15k (c) >15k

Figure 10: PrintQueue versus HashPipe and FlowRadar with different queue-depth-based query intervals under UW traces. The resource consumption of the primary data structures of each approach are listed in the graphs of the left-most column.



(a) $\alpha = 2$, $k = 12$, $T = 4$ (b) $\alpha = 2$, $k = 12$, $T = 5$ (c) $\alpha = 3$, $k = 12$, $T = 4$

Figure 11: PrintQueue versus related works with different parameters under UW traces.

Trace	PrintQueue	HashPipe	FlowRadar
UW	0.684/0.634	0.396/0.341	0.391/0.350
WS	0.909/0.864	0.801/0.582	0.763/0.582
DM	0.977/0.948	0.838/0.671	0.838/0.671

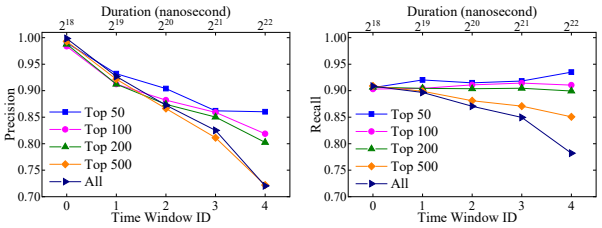
Table 2: Average precision/recall of PrintQueue, HashPipe, and FlowRadar under different traces.

PrintQueue versus other systems. The above accuracy numbers significantly outperform existing work, which tends to collect and reset the data structures at fixed intervals. To provide a fair comparison, we use two recent proposals for flow-size estimation, HashPipe [23] and FlowRadar [15], and set their reset intervals to the set period of PrintQueue (as the periodic control plane polling interval is the common bottleneck). These configurations result in comparable SRAM requirements: HashPipe and FlowRadar use 4096 register entries in each of five stages, while PrintQueue uses 4096 cells in each of four time windows. We note that HashPipe and FlowRadar are only queryable on the granularity of a reset period. We, therefore, improve their estimations by prorating packet counts using a multiplier equal to the length of the query interval over the length of the total period. For fairness, we also only show PrintQueue results on asynchronous queries, as data-plane queries have much higher accuracy. We do not compare to sketches as they cannot provide flow IDs, only aggregate byte counts.

As shown in Table 2, the average precision and recall of PrintQueue is significantly higher than either HashPipe or FlowRadar under all three workloads. We dig further into the UW traces, which are the most challenging. Figure 10 shows the results for a few categories of queue depths (i.e., query intervals): low occupancy (1k to 5k), medium occupancy (5k to 15k), and high occupancy (>15k). The median accuracy of PrintQueue is up to 3× times higher than that of existing work. The results of HashPipe and FlowRadar are similar, as they both capture the heavy hitters over the entire monitoring interval. We note that these inaccuracies are not caused by hash collisions or other factors that are traditionally the target of heavy-hitter accuracy improvements. Rather, it is because they run in fixed monitoring intervals, and proportional prorating of the results can greatly over- or under-estimate reality.

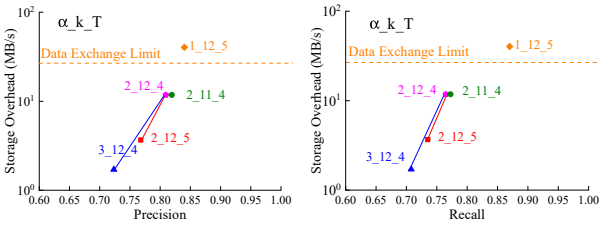
PrintQueue versus related work with different parameters. We repeat the comparisons under UW traces while varying the parameters α, k, and T. Each subgraph of Figure 11 shows the median accuracy of the sampled packets for different queue depths.

Across all evaluated parameter sets, PrintQueue outperforms existing work at larger query intervals. PrintQueue can also outperform existing work at small query intervals, but its accuracy can drop with higher values of α and T. For the former, it is because the compression ratio becomes too large. In particular, the queuing period of 1k to 2k depth is approximately 60 µs to 120 µs. With $\alpha = 3, T = 4$, the cell periods of the four windows are 64 ns, 512 ns, 4 µs, and 32 µs. If the query interval falls into the last window–a common occurrence in asynchronous queries—time windows must estimate the per-flow packet counts with *only four cells total*. A similar effect occurs when we increase T and add a time window with lower accuracy. Larger query intervals decrease the probability of this worst-case scenario and enable queries to leverage the advantages of PrintQueue's exponential storage. Data-plane queries do not suffer from either issue.



(a) precision (b) recall

Figure 12: Top-K flows from a single time window under UW traces.



(a) storage versus precision (b) storage versus recall

Figure 13: Storage versus accuracy with α, k, T under UW traces.

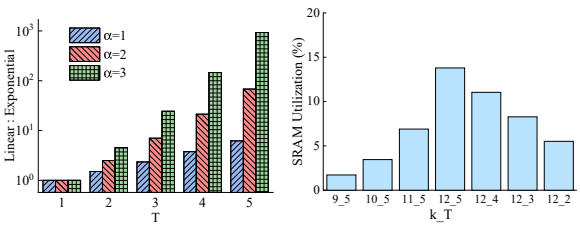
In practice, network operators should choose the lowest values of α and T that are feasible for their networks. We evaluate and discuss relevant constraints later in this section.

Accuracy versus different windows for Top-K flows. We next evaluate the relative accuracy of individual time window with the metric of Top-K flow packet counts. We again focus on the UW traces. We use $\alpha = 1$, $k = 12$, $T = 5$, and set the query interval to be the full window period.

As shown in Figure 12, the accuracy of 5 windows varies. As expected, the precision drops with the depth of windows, with the first window achieving precision near 1 because it is un-compressed. Any errors are due to mismatches between the packet size and cell granularities. As in previous experiments, errors accumulate in deeper windows. We note that since each packet has an approximate probability of being passed across windows, PrintQueue tends to store flows with more packets and so the top-k results remain relatively accurate. For reference, during most window periods, the flow number is on the order of thousands.

We observe that the UW traces [4] have an extreme long-tailed distribution. In fact, the packet count of the 100th largest flow is less than 1% of the packet count of the largest flow. When moving to the Top-500 flows, the mice begin to overwhelm the elephants, dropping the accuracy in larger time windows.

Accuracy versus control-plane overhead. One underlying constraint on the configuration of PrintQueue is the control plane's ability to extract results frequently enough to ensure no gaps in time window coverage. Fundamentally, the control plane is limited by analysis program I/O throughput and PCIe bandwidth. Thus, the limitation can be quantified in terms of the number of register entries that can be read per second.



(a) linear versus exponential (b) SRAM usage

Figure 14: Storage overhead comparison and SRAM.

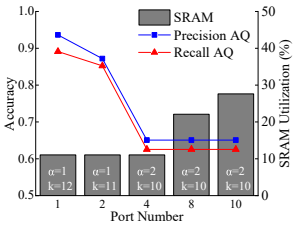


Figure 15: Accuracy versus port number under WS traces.

Figure 13 shows the required PCIe bandwidth in MB/s versus precision and recall for different configurations of PrintQueue under UW traces. We plot a rough estimate of the maximum capabilities of our current analysis program implementation. When the data size per second is above the line, the time needed to read the registers is longer than PrintQueue's set period, which leads to packets getting evicted before they are successfully read and stored.

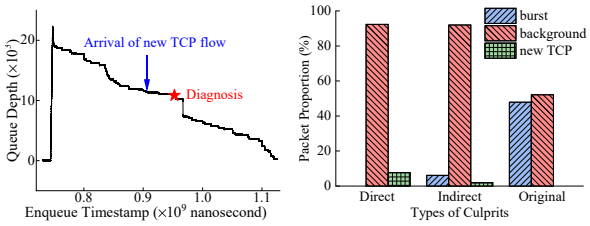
With larger α, the compression of PrintQueue becomes more aggressive, reducing the I/O requirements of the system. At the same time, larger α leads to reduced precision and recall. T has a similar effect as each additional window has exponentially more compression, but here too, more compression translates to less accuracy.

The parameter k does not influence parameter feasibility, as the set period and the number of registers are multiplied by the same factor. Our experiment also shows that k has little impact on the accuracy for asynchronous queries under UW traces. Larger values of k are, however, preferred for data-plane queries as they mean that longer query intervals fit within the initial time windows. The configurations we chose in the preceding sections related to queries are all below the feasibility line.

Linear storage versus exponential storage. We also compare PrintQueue's storage overhead with techniques like NetSight [10] and BurstRadar [13]—two systems with linear storage requirements. Figure 14(a) shows the ratio of the linear storage overhead to PrintQueue's overhead with different α, T. PrintQueue's overhead is up to three orders of magnitude less than linear storage methods.

SRAM overhead. We evaluate the data-plane SRAM overhead of time windows across a range of k and T parameters. α does not affect resource consumption. As shown in Figure 14(b), across different parameters, time windows consume only a moderate amount of resources, making the system practical in real networks.

Port parallelism. We activate PrintQueue on several of ports simultaneously and evaluate the accuracy for a single one. Naturally, we can activate more ports if the SRAM usage grows linearly with



(a) queue depth (b) direct, indirect, and original

Figure 16: Time windows versus queue monitor for tracking the burst flow.

the port number. But the method does not scale as the port number continues growing. Instead, we adjust parameters α and k to reduce the total SRAM cost. Figure 15 shows accuracy of asynchronous queries against the total data-plane SRAM utilization. With $\alpha = 2$, at most 10 ports can run PrintQueue in parallel. A further increase is constrained by the PCIe bandwidth limit of the local interface.

7.2 Queue Monitor Case Study

We show the effectiveness of the queue monitor qualitatively using a case study. Specifically, we let one server send a background TCP flow limited to ~90% of the link capacity (9 Gbps). Another server first sends a burst of 10000 datagrams at a rate of 4 Gbps. After a short time, it then begins a TCP flow at a low rate (0.5 Gbps).

As shown in Figure 16(a), the burst flow causes a rapid increase in queue depth. While the burst flow lasts for only around 5 ms, the queuing caused by the burst lasts for 376 ms (i.e., 76× times longer than the burst period itself)! The new TCP flow arrives at the blue arrow in Figure 16(a). At the star, PrintQueue leverages time windows and the queue monitor to query the direct, indirect, and original culprits to diagnose the high queuing delay of the new TCP flow. In this setting, we expect to be able to implicate the burst flow because, without it, the queuing would not exist or be nearly as severe.

As shown in Figure 16, direct culprits consider the background traffic the most significant contributor. They do not include any packets of burst flow, as the packets have long before left the queue. Indirect culprits have captured all the packets since the beginning of the congestion. The burst flow can be found, but it is indistinguishable from a normal mouse flow. The results of the query for the original culprits, instead correctly show that the culpability of the burst flow is comparable to that of the background traffic (5597:6096) despite their differences in total size.

The SRAM usage of queue monitor for a single port is 12.81% of data-plane resources.

8 RELATED WORK

PrintQueue is related to a rich body of prior work in queue and performance monitoring. In this section, we discuss the most relevant work in these areas.

Queue measurement techniques. Others have previously noted the importance of queue-based performance monitoring and proposed methods to do so. Many of the earlier instances in this set focus on the length of the queue rather than its contents [29, 35]. Many others rely on raw flow sampling [10, 11, 13, 18, 25, 37] to reconstruct queue contents; compared to these approaches, PrintQueue requires significantly less space and pipeline overhead.

One particularly relevant work to time windows is ConQuest [5, 6], which also tracks queue composition in the data plane using a special snapshot-based data structure. However, ConQuest solves a different problem. It judges whether the current packet's flow is the main contributor to queuing. To implicate the causes of delay in a specific victim packet's queuing, ConQuest would need offline storage space linear to the total packets in the network. Further, ConQuest only supports FIFO queues while PrintQueue's time windows are agnostic to the packet scheduling policy.

We also note that Microscope [8] makes a similar observation about the importance of historical causes of queuing, but in the context of network function performance. The specifics of packet queuing delay and PrintQueue's implementation on programmable data planes introduce novel constraints.

Flow counting techniques. Prior work, e.g., FlowRadar [15], TurboFlow [24], and CounterBraids [16], develops accurate per-flow traffic counters. Heavy hitter detection techniques, e.g., HashPipe [23], DOVE [14], and others [17, 26, 34], only track the traffic of large flows. Flow counter techniques can provide flow information along with its size like PrintQueue. But they work under fixed time periods, failing to retrieve flows in arbitrary query intervals.

Bandwidth measurement techniques. Work [27] measures bandwidth at all time scales. But it calculates total rates without the knowledge of each flow's contribution. The algorithms modify the network stacks of end hosts and can not be applied in today's programmable switches.

Provenance. Prior work, e.g., Dapper [21], DTaP [36], Zeno [28], gives detailed explanations of event causes in the distributed system. PrintQueue expands the concept of provenance to packet queuing. PrintQueue's results can be incorporated into these higher-level frameworks.

9 CONCLUSION

In this paper, we systematically classify the culprit packets of queuing in switches. We present PrintQueue, a practical data-plane monitoring system for tracking the provenance of packet-level queuing delays at both small and large timescales. We design time windows to capture direct and indirect culprits over any time span, and queue monitor to track original culprit packets. We implement PrintQueue on a Tofino switch and evaluate it with multiple network traces. Through evaluations, we show that PrintQueue achieves high accuracy with limited overhead.

ACKNOWLEDGMENTS

We thank our shepherd Aurojit Panda and all the anonymous SIGCOMM reviewers for their helpful and thoughtful comments. This work was supported by the National Natural Science Foundation of China under Grant 61832013. It was also funded in part by Google, Meta, VMWare, and NSF grant CNS-1845749. Mingwei Xu is the corresponding author.

REFERENCES

[1] [n. d.]. The BMv2 Simple Switch target. Website. ([n. d.]). https://github.com/p4lang/behavioral-model/blob/main/docs/simple_switch.md.

[2] [n. d.]. Intel Open-Tofino. Website. ([n. d.]). https://github.com/barefootnetworks/Open-Tofino.

[3] Mohammad Alizadeh, Albert Greenberg, David Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data center TCP (DCTCP). *ACM SIGCOMM Computer Communication Review* 40, 63–74. https://doi.org/10.1145/1851182.1851192

[4] Theophilus Benson, Aditya Akella, and Dave Maltz. 2010. Network Traffic Characteristics of Data Centers in the Wild. In *Internet Measurement Conference* (internet measurement conference ed.). Association for Computing Machinery, Inc. https://www.microsoft.com/en-us/research/publication/network-traffic-characteristics-of-data-centers-in-the-wild/

[5] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, and Ori Rottenstreich. 2018. Catching the Microburst Culprits with Snappy. In *Proceedings of the Afternoon Workshop on Self-Driving Networks (SelfDN 2018)*. Association for Computing Machinery, New York, NY, USA, 22–28. https://doi.org/10.1145/3229584.3229586

[6] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, Ori Rottenstreich, Steven A Monetti, and Tzuu-Yi Wang. 2019. Fine-Grained Queue Measurement in the Data Plane. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies (CoNEXT '19)*. Association for Computing Machinery, New York, NY, USA, 15–29. https://doi.org/10.1145/3359989.3365408

[7] Linux Foundation. 2015. Data Plane Development Kit (DPDK). (2015). http://www.dpdk.org

[8] Junzhi Gong, Yuliang Li, Bilal Anwer, Aman Shaikh, and Minlan Yu. 2020. Microscope: Queue-Based Performance Diagnosis for Network Functions. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 390–403. https://doi.org/10.1145/3387514.3405876

[9] Albert Greenberg, James Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David Maltz, Parveen Patel, and Sudipta Sengupta. 2011. VL2: A scalable and flexible data center network. *ACM SIGCOMM Computer Communication Review* 39 (01 2011), 51–62. https://doi.org/10.1145/1594977.1592576

[10] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. 2014. I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 71–85. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/handigol

[11] Vimalkumar Jeyakumar, Mohammad Alizadeh, Yilong Geng, Changhoon Kim, and David Mazières. 2014. Millions of Little Minions: Using Packets for Low Latency Network Programming and Visibility. *SIGCOMM Comput. Commun. Rev.* 44, 4 (Aug. 2014), 3–14. https://doi.org/10.1145/2740070.2626292

[12] Lavanya Jose and Minlan Yu. 2011. Online Measurement of Large Traffic Aggregates on Commodity Switches. In *Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE 11)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/hot-ice11/online-measurement-large-traffic-aggregates-commodity-switches

[13] Raj Joshi, Ting Qu, Mun Choon Chan, Ben Leong, and Boon Thau Loo. 2018. BurstRadar: Practical Real-Time Microburst Monitoring for Datacenter Networks. In *Proceedings of the 9th Asia-Pacific Workshop on Systems (APSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 8, 8 pages. https://doi.org/10.1145/3265723.3265731

[14] Yiran Lei, Yu Zhou, Yunsenxiao Lin, Mingwei Xu, and Yangyang Wang. 2021. DOVE: Diagnosis-driven SLO Violation Detection. In *2021 IEEE 29th International Conference on Network Protocols (ICNP)*. 1–11. https://doi.org/10.1109/ICNP52444.2021.9651986

[15] 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)*. USENIX Association, Santa Clara, CA, 311–324. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/li-yuliang

[16] Yi Lu, Andrea Montanari, Balaji Prabhakar, Sarang Dharmapurikar, and Abdul Kabbani. 2008. Counter Braids: A Novel Counter Architecture for per-Flow Measurement. In *Proceedings of the 2008 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '08)*. Association for Computing Machinery, New York, NY, USA, 121–132. https://doi.org/10.1145/1375457.1375472

[17] Jonatas Marques, Kirill Levchenko, and Luciano Gaspary. 2020. IntSight: Diagnosing SLO Violations with in-Band Network Telemetry. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '20)*. Association for Computing Machinery, New York, NY, USA, 421–434. https://doi.org/10.1145/3386367.3431306

[18] Jeff Rasley, Brent Stephens, Colin Dixon, Eric Rozner, Wes Felter, Kanak Agarwal, John Carter, and Rodrigo Fonseca. 2014. Planck: Millisecond-Scale Monitoring and Control for Commodity Networks. *SIGCOMM Comput. Commun. Rev.* 44, 4 (Aug. 2014), 407–418. https://doi.org/10.1145/2740070.2626310

[19] Luigi Rizzo. 2012. Netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC'12)*. USENIX Association, USA, 9.

[20] Vishal Shrivastav. 2019. Fast, Scalable, and Programmable Packet Scheduler in Hardware. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 367–379. https://doi.org/10.1145/3341302.3342090

[21] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Technical Report. Google, Inc. https://research.google.com/archive/papers/dapper-2010-1.pdf

[22] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. 2016. Programmable Packet Scheduling at Line Rate. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 44–57. https://doi.org/10.1145/2934872.2934899

[23] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. 2017. Heavy-Hitter Detection Entirely in the Data Plane. In *Proceedings of the Symposium on SDN Research (SOSR '17)*. Association for Computing Machinery, New York, NY, USA, 164–176. https://doi.org/10.1145/3050220.3063772

[24] John Sonchack, Adam J. Aviv, Eric Keller, and Jonathan M. Smith. 2018. Turboflow: Information Rich Flow Record Generation on Commodity Switches. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 11, 16 pages. https://doi.org/10.1145/3190508.3190558

[25] John Sonchack, Oliver Michel, Adam J. Aviv, Eric Keller, and Jonathan M. Smith. 2018. Scaling Hardware Accelerated Network Monitoring to Concurrent and Dynamic Queries with *flow *(USENIX ATC '18)*. USENIX Association, USA, 823–835.

[26] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. 2018. Distributed Network Monitoring and Debugging with Switchpointer *(NSDI'18)*. USENIX Association, USA, 453–466.

[27] Frank Uyeda, Luca Foschini, Fred Baker, Subhash Suri, and George Varghese. 2011. Efficiently Measuring Bandwidth at All Time Scales. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/nsdi11/efficiently-measuring-bandwidth-all-time-scales

[28] Yang Wu, Ang Chen, and Linh Thi Xuan Phan. 2019. Zeno: Diagnosing Performance Problems with Temporal Provenance. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 395–420. https://www.usenix.org/conference/nsdi19/presentation/wu

[29] Nofel Yaseen, John Sonchack, and Vincent Liu. 2018. Synchronized Network Snapshots *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 402–416. https://doi.org/10.1145/3230543.3230552

[30] Nofel Yaseen, John Sonchack, and Vincent Liu. 2020. tpprof: A Network Traffic Pattern Profiler. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 1015–1030. https://www.usenix.org/conference/nsdi20/presentation/yaseen

[31] Liangcheng Yu, John Sonchack, and Vincent Liu. 2020. Mantis: Reactive Programmable Switches. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 296–309. https://doi.org/10.1145/3387514.3405870

[32] Liangcheng Yu, John Sonchack, and Vincent Liu. 2022. Cebinae: Scalable In-network Fairness Augmentation. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '22)*. Association for Computing Machinery, Amsterdam, Netherlands. https://doi.org/10.1145/3544216.3544240

[33] Zhuolong Yu, Chuheng Hu, Jingfeng Wu, Xiao Sun, Vladimir Braverman, Mosharaf Chowdhury, Zhenhua Liu, and Xin Jin. 2021. Programmable Packet Scheduling with a Single Queue. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 179–193. https://doi.org/10.1145/3452296.3472887

[34] Lihua Yuan, Chen-Nee Chuah, and Prasant Mohapatra. 2011. ProgME: Towards Programmable Network Measurement. *IEEE/ACM Trans. Netw.* 19, 1 (feb 2011), 115–128. https://doi.org/10.1109/TNET.2010.2066987

[35] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. 2017. High-Resolution Measurement of Data Center Microbursts. In *Proceedings of the 2017 Internet Measurement Conference (IMC '17)*. Association for Computing Machinery,

New York, NY, USA, 78–85. https://doi.org/10.1145/3131365.3131375
[36] Wenchao Zhou, Suyog Mapara, Yiqing Ren, Yang Li, Andreas Haeberlen, Zachary Ives, Boon Thau Loo, and Micah Sherr. 2012. Distributed Time-Aware Provenance. *Proc. VLDB Endow.* 6, 2 (Dec. 2012), 49–60. https://doi.org/10.14778/2535568.2448939
[37] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. 2015. Packet-Level Telemetry in Large Datacenter Networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*. Association for Computing Machinery, New York, NY, USA, 479–491.

A ARTIFACT APPENDIX

Abstract

PrintQueue's artifact is publicly available, including the source code and documents for all the mentioned components in the paper. The artifact can reproduce the paper results. The detailed instructions to build, deploy, and operate the system are introduced in the `Github` repository.

Scope

The artifact is used to reproduce all the major results of PrintQueue.

Contents

The artifact includes the source code of PrintQueue, consisting of:

- P_4 code running at Intel Tofino programmable switch, including the data-plane code (implementation of time windows and queue monitor) and control-plane code (read and filter registers; execute queries).
- DPDK code running at receiver server, extracting and storing PrintQueue telemetry headers.
- Code to simulate traces modelled after DCTCP and VL2 flow distribution.
- Experiment data collected from our testing and script to reproduce the paper results.

Hosting

The aritifact is accessible via `Github` (please refer to the *master* branch and the *latest* commit) and `Zenodo`.

- `Github` link: https://github.com/A-Dying-Pig/PrintQueue/tree/master
- `Zenodo` DOI: 10.5281/zenodo.6789638

Requirements

PrintQueue requires specific hardware and software environments:

- The switch code functioned on the Intel Tofino switch.
- The receiver code required DPDK-compatible NIC and DPDK library.
- The packages required by `Python` scripts were listed in the documents.