dDrops: Detecting Silent Packet Drops on Programmable Data Plane

Mimi Qian[a], Lin Cui[a,*], Xiaoquan Zhang[a], Fung Po Tso[b], Yuhui Deng[a]

[a] *Department of Computer Science, Jinan University, Guangzhou, China*
[b] *Department of Computer Science, Loughborough University, LE11 3TU, UK*

Abstract

Silent packet drops are common in data center networks, and are a major cause of network performance anomalies (NPAs) that have significant impacts on application performance and network management. However, existing solutions using coarse-grained statistics and flow-level telemetry either fail to provide precise location of packet drops or incur large overhead. This paper presents *dDrops*, a packet-level telemetry based on programmable data plane to detect and retrieve details of silent packet drops immediately when they happen. *dDrops* can dynamically adapt to varying ratios of silent packet drops for different ports on a switch to improve performance of silent packet drops detection. Moreover, a dynamic memory management scheme is also designed to efficiently use the limited memory on the data plane of switch. *dDrops* has been implemented on both P4 hardware programmable switches (based on Intel Tofino ASIC) and BMv2. Extensive experiment results show that *dDrops* is able to detect and locate the silent packet drops within $5ms$ (including detailed information of dropped packets), and reduce the memory consumption by up to 50%.

Keywords: Silent packet drop, Packet-level telemetry, Programmable data plane

1. Introduction

Large-scale datacenter networks (DCNs) are comprised of hundreds of thousands of servers which are interconnected via a significant number of network switches and links. In such large systems, network performance anomalies (NPAs), such as packet loss, bandwidth decline, long-tailed latency and bandwidth/latency jitters, are inevitable [1].

Recent studies [2] show that packet drop is a major cause of NPAs. More than 15% of NPAs are attributed by silent packet drops, which are packet drops or packet corruptions over the link between two independent forwarding pipelines. For example, one common occurrence of silent packet drops is frame checksum (FCS) errors. Previous study [3] shows that 13% of the failures in DCNs are caused by FCS errors due to damaged or faulty cable. Such packet drops not only significantly degrade application performance,

*Corresponding author
Email address: tcuilin@jnu.edu.cn (Lin Cui)

but also affect network management such as understanding flow size distribution and anomaly detection [4, 5, 6]. However, due to the difficulty of detection (e.g., they are usually invisible to switches), locating silent packet drops usually takes longer time than other NPAs, e.g., about 161 minutes on average [2]. Thus, it is important to detect silent packet drop fast (e.g., in tens of milliseconds) so that operators can intervene in time to take actions to eliminate these losses and mitigate their impacts. Moreover, in order to diagnose the root cause and mitigate the impact on flows/applications, detailed information of silent packet drops are required, for example, details of lost packets (e.g., packet headers and other useful contents), locations (e.g., ingress & egress ports of switches) and the timing of losses (e.g., the last hop switch what time the victim packets appear).

Unfortunately, traditional solutions fail to satisfy all these requirements for silent packet drops detection. Some methods use **counters** (e.g., [7, 8, 9]) to aggregate number of packets on the granularity of each interface, device or flow, failing to provide detailed information of individual lost packet. **Probe-based** solutions (e.g, [5, 10]) are typically used to obtain end-to-end statistics, such as latency and loss, from an application point of view, but they only probe the network rather than track the actual traffic, which can capture losses but cannot identify the exact switch or link causing silent packet drops and affected flows/applications. **Sampling** solutions (e.g., [11, 12, 13, 14, 15]) measure a subset of the network traffic and inevitably miss some silent packet drop events. Thus, above schemes are not fine-grained enough for silent packet drop detection.

For detailed packet information of silent packet drops, **packet-level** telemetry is required, which can obtain fine-grained information based on data plane tracing. Nevertheless, existing packet-level solutions suffer from trade-off between granularity and cost. For example, NetSight [16] can provide packet-level detection for silent packet drops by making switches to mirror all packets to controllers. It proposes an optimization scheme by sending compressed aggregates of postcards[1] to controllers, which still brings 7% bandwidth overhead. NetSeer [2] uses coordination between upstream and downstream switches to detect silent packet drops in the packet-level. It allocates a fixed memory for each port of switches, causing large memory consumption and low accuracy when just one or a few ports experience continuous losses (e.g., last for several minutes due to damaged optical fiber or dirty optical connectors).

Therefore, in this paper, *in order to achieve fast and fine-grained detection of silent packet drop with low overhead*, we propose *dDrops* which is a packet-level telemetry based on programmable data plane (PDP). *dDrops* leverages neighboring switches for packet-level drop detection, a pattern that is already adopted by several production switch functions such as NetSeer [2] and BFD [17]. *dDrops* can well adapt to dynamic failure and packet rate on different ports of switches, and constantly monitor all individual packets to discover silent packet drops. Considering limited memory resources on data plane, a novel dynamic memory management scheme is designed to enable memory sharing across all ports of a switch. As a result, *dDrops* is able to handle a large numbers of continuous silent packet drops with low memory consumption. In addition, *dDrops* can detect silent packet drops quickly within $5ms$ while providing detailed information of individual lost packet, such as location of packet loss (e.g., ingress & egress ports of switches), packet header information and timing information.

The major contributions of this paper are three-folded:

[1] Postcards are event records created whenever a packet traverses a switch.

1. We investigate the issue of silent packet drops and propose *dDrops* which achieves both low memory consumption and fine-grained detection. To the best of our knowledge, this is the first paper considering adaption to dynamic characteristics among ports for silent packet drop detection on programmable data planes.
2. A dynamic memory management scheme is designed for programmable data planes. It allows on-demand allocation/release of memory and efficient content access, enabling sharing buffer across different ports on a switch.
3. We have implemented *dDrops* on both hardware P4 switches (equipped with Intel Tofino ASIC) and software switches of BMv2. Extensive evaluations demonstrate that it can detect and locate silent packet drop within $5ms$. *dDrops* saves 50% SRAM compared with NetSeer, and uses only 0.013% of the bandwidth overhead when loss rate is 0.01%.

The rest of paper is organized as follows. Section 2 discusses motivation of this paper through analyzing the major challenges of detecting silent packet drops. The major design of *dDrops* are provided in Section 3. Section 4 discusses the implementation of *dDrops* on programmable switch. Section 5 evaluates *dDrops* on both BMv2 and hardware P4 switches with Tofino ASIC. Finally, Section 7 concludes this paper.

2. BACKGROUND AND MOTIVATIONS

2.1. Silent packet drop and its impacts

Silent packet drops refer to packet drops or packet corruptions over the link between two independent forwarding pipelines. The causes for silent packet drops include faulty cables, damaged/dirty connectors or optical fiber, bad transceivers (e.g., devices that convert between optical and electrical signals), poorly installed hardware (e.g., not-well-seated linecards) or decaying transmitters [2, 5, 3, 18]. Today's DCNs are usually comprised of hundreds of thousands of hardware and software components, including multiple types of switches and routers, load balancers, and millions of cables and fibers. Silent packet drops may occur frequently at any of those components. When they happen, applications suffer from increased latency and packet drops, while switches for various reasons (such as without reporting [5, 11]) can not show information about these packet drops and seem innocent.

Moreover, silent packet drops will typically occur for a period time (e.g., tens of minutes). One reason is that such packet drops or corruptions are usually invisible to switches, making them difficult to detect and mitigate its impact, for example, it can take up to several hours [2] to locate. In addition, when silent packet drops happen, operators may need to reboot switches because of poorly install hardware or replace some damaged components. These interventions are costly as they usually take a long period of time to complete. It may need several days or even weeks in the worst case to mitigate and repair critical failures [3]. Thus, silent packet drops may affect a large number of flows, significantly increase perceived latency of users since dropped packets need to be retransmitted, and increase the difficulty for network management such as traffic monitoring and diagnosis [4, 5, 6].

2.2. Motivations

Table 1 compares existing solutions with *dDrops* (more detailed description can be found in Section 6). Existing works detect silent packet drops via maintaining per-packet or per-flow information on programmable switches or match-and-mirror packets to a remote collector. However, these solutions consume large overhead in order to capture details of all packet loss and can not adapt to dynamic network characteristics as a result of changing packet rate or network failures variation on different ports and links.

(1) Memory consumption for silent packet drops detection *vs.* limited SRAM on programmable data plane

To help operators analyze the root cause of silent packet drops, identifying affected applications and taking intervention actions, following information are required: (1) All lost packet information such as 5-tuples (source, destination IP addresses, ports, and protocol) which can indicate the applications that are affected by silent packet drop and help operators to take corresponding actions. (2) Location including switches, link or port. (3) Timing information that can determine whether the problem is transient (which could be ignored) or persistent (which requires attention). It is very important to obtain these information for as many lost packets as possible, because operators can see affected flows or applications based on these information, so that more fine-grained operations (e.g, re-schedule flows or select better servers) can be performed to mitigate or solve the impact of silent packet drops. However, a large amount of packet loss may occur when continuous packet drop happens. Previous studies have found that an average failure rate of 40.8 links per day in a production data center, which can cause a median silent packet drop of 59,000 packets per failure [19]. Collecting above information will generate a large amount of data, while switches only have limited memory (only tens of MB for all the counters and match-action tables).

FlowRadar [9] keeps information about all per-flow traffic and counters at each switch and compares them at two nearby hops. Its memory usage is proportional to the number of the ongoing flows (e.g., 29.7MB per switch for SingleDecode and 24.8MB per switch for NetDecode with 1M flows). NetSeer [2] leverages coordination of both upstream and downstream switches to detect silent packet drops by allocating fixed memory to each port. Thus, its memory overhead is proportional to number of ports upon continuous losses. For example, for a switch with 64×100Gbps ports, $5ms$ of continuous silent packet drops requires over 50MB SRAM in total (see Figure 1).

Therefore, being able to keep detailed information of all silent packet drops with limited memory in data plane is highly desirable.

(2) Dynamic differences in packet loss and flow rate among ports of a switch.

Previous studies [3] show that the number of simultaneous faulty links is small. This means it is unlikely that all ports or links of a switch experience silent packet drops simultaneously. In addition, the packet arrival rates of different ports can vary drastically [20, 21, 22]. These factors lead to widely varying number of silent packet drops on different ports of the same switch. Ignoring these characteristics of switch ports will degrade performance of silent packet drops detection. It will either waste large amount of memory in order to capture all lost packets (e.g., allocating a fixed buffer for each port of a switch), or reduce detection accuracy due to overflow of buffer when continuous silent packet drops happen on a port (e.g., $5ms$ of continuous silent packet drops require over 50MB in Figure 1).

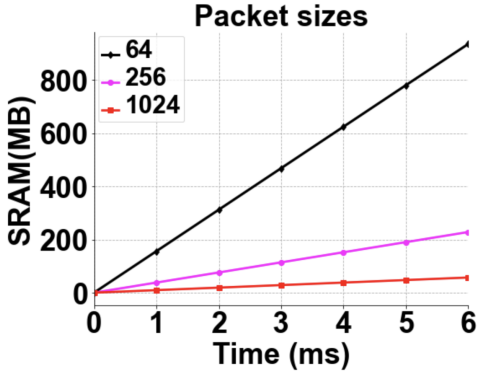


Figure 1: Memory consumption on different duration of continuous silent packet drops. Packets of 1024 bytes are transmitted back-to-back on 100Gbps.

Table 1: State-of-the-art datacenter detection solutions.

Methods	Detection time	Location	Packet-level info	Overhead	Dynamic adaption
Host monitoring [5]	10s of seconds	Infer location	✗	Host CPU	✓
Mirroring [11]	10s of ms	✓	✓	B/w prop to packet	✗
Flow-level counters [9, 6]	After a flow(let) ends	✓	✗	Memory prop.to flows	✗
NetSeer [2]	in milliseconds[1]	✓	✓	Medium (fixed memory allocation)	✗
dDrops	$< 5ms$	✓	✓	Low (on-demand dynamic memory allocation)	✓

[1] The upstream switch can detect silent packet drop within $6ms$ in the experiment. But NetSeer batches 50 events together in one packet to the controller, so the actual detection time may extend to few more seconds.

Therefore, the characteristic of dynamic differences in both packet loss and flow rate among ports should be considered when designing detection methods for silent packet drops.

3. *dDrops* Design

Leveraging the programmability of programmable data plane (PDP), we propose *dDrops*, an adaptive detection scheme which can constantly detect silent packet drops and report their features to controller.

3.1. Overview

3.1.1. Goals of dDrops

dDrops should detect silent packet drops immediately when they happen, and be able to retrieve detailed information of lost packets, such as critical header fields, locations and timing information, under limited memory in data plane. Furthermore, it should achieve low overhead and good memory efficiency taking into account the diversity of traffic and failures among different ports. There are three goals for *dDrops*:

1. **Low memory consumption.** When silent packet drops happen, detailed information of individual lost packet (e.g., the 5 tuples and other useful data) are important for troubleshooting by network administrators. Due to the invisibility of

silent packet drops on switches, it is inevitable to store information of all packets transmitted out of a switch. However, doing so is very memory intensive. For example, considering a switch with 64×100Gbps (e.g., about $10^6 pps$), $5ms$ of continuous silent packet drops detection will require over 50 MB memory[2]. This is challenging for programmable switches as they usually have small memory size (e.g., 20∼30 MB) to be shared among different functions (e.g., forwarding and encapsulation). Thus, minimizing memory consumption is crucial for the success and feasibility of *dDrops* in practice.

2. **Dynamic adaptation to port differences.** The occurrence and duration of silent packet drops happen in a random manner [3]. Moreover, the traffic rate of different ports on a switch also varies and changes dynamically, leading to different amounts of lost packets when continuous silent packet drop occurs. For these reasons, allocating a fixed amount of memory to each port is inefficient. Thus, a flexible on-demand memory allocation/release scheme is required to adapt to the dynamic diversity of both traffic rate and failure on each port of a switch.
3. **Fast detection.** Modern datacenter applications have strong requirements for SLA (e.g., 99.9%) as the rapid development of network speed. Failing to deliver the SLAs can result in severe customer, reputation and revenue loss [2]. Thus, to mitigate impact on performance of applications, the latency between a failure occurrence and its detection should be as low as possible [23]. Therefore, *dDrops* should be able to detect silent packet drops and report the detailed information of lost packets to the controller (i.e., network administrators) immediately when they happen, so that network administrators can intervene quickly.

3.1.2. Challenges and limitations

Achieving all goals above in the PDP is challenging. For example, in order to detect all silent packet drops, *dDrops* needs to leverage the coordination of nearby programmable switches to cache packets information before being forwarded. However, caching all packets in data plane is impossible and will increase memory consumption. Considering dynamic port differences, the amount of memory required by each port is different and can change dramatically, e.g., when continuous silent packet drops happen. Therefore, how to adapt to the dynamic differences of ports is very challenging, and it still remains unexplored in data plane yet.

Moreover, in order to process packets at guaranteed line rates, PDP significantly restricts operations in the pipeline [24]. For example, a stateful ALU can only support basic mathematical operations (e.g., addition, subtraction and bitwise operation) which makes it challenging to implement more complex applications for *dDrops*. For example, multiplication and division are required when implementing dynamic memory allocation/release for *dDrops*.

Registers in P4 programs are stored in SRAM blanks which are local to each match+action processor. They have three limitations. First, a program can only read or write a register by match+action model implemented in the same stage. Second, a register can only be accessed once for each packet to meet timing requirements. Finally, all register operations in the stage must be accessed in parallel. These three limitations can be challenging to

[2]Assuming only 5 tuples are stored for each packet, i.e., 13 bytes for each packet.

dDrops which take advantage of PDP for implementation of dynamic memory allocation or release.

3.2. dDrops architecture

dDrops is a PDP telemetry system that supports *low memory consumption*, *dynamic adaption to port difference* and *fast detection* for network measurement. It can obtain detailed information such as 5-tuples, location and timing information of silent packet drops (referred as packet features in the following of this paper). Unlike existing works [2, 6], *dDrops* introduces a dynamic memory management mechanism, which can dynamically allocate and release memory on demand for each port rather than allocating fixed memory per port. Considering the invisibility of silent packet drops on single switch, *dDrops* lets two adjacent switches cooperate with each other to detect silent packet drops between them. For ease of description, the two adjacent switches are referred as upstream switch and downstream switch respectively. Figure 2 depicts an overview of *dDrops* and how it is deployed in a network infrastructure.

dDrops consists of three main components as follows:

- **Drop detection** is used to detect silent packet drops. It saves the packet ID (which is used to identify each packet) of the last received packet for each port using an array (which is initialized to 0), and compares it with the currently received packet to identify whether packet loss happens. If so, it constructs a loss notification packet to notify the upstream switch. The upstream switch performs the lookup operation for the positions of lost packets in the buffer, sends the positions to *packet features management* for reporting and notifies releasable blocks (a block refers to a small fixed size memory, e.g., 1024Bytes) to *memory management*.

- **Packet features management** is used to report lost packets and store packet features of all packets that have been sent out recently through each port. It obtains the position from *drop detection*, retrieves lost packets from memory and reports them to remote controller. In addition, it extracts the 5-tuple information from each packet, copies metadata including forwarding latency and port, and generates a packet ID, then inserts them into the buffer space of the port before packets are transmitted out. *Packet features management* will apply for a new block via *memory management* when the buffer space of the port is almost full.

- **Memory management** enables dynamic memory allocation and release using a sequence of *match + action* tables in PDP. It implements operations of both *buffer allocation* and *buffer release*. *Buffer allocation* can allocate a block from a global buffer (A fixed size buffer pool, which is used to cache packet features in a switch) to a port, e.g., when buffer space of the port is almost full. *Buffer release* can release specific blocks, e.g., when packets in the block are all reported to controller as packet loss or received by downstream switch.

3.3. Drop detection

To capture all silent packet drops, *dDrops* uses an individual packet ID to identify all packets transmitted by each port. Initially, a block-based buffer is allocated for each

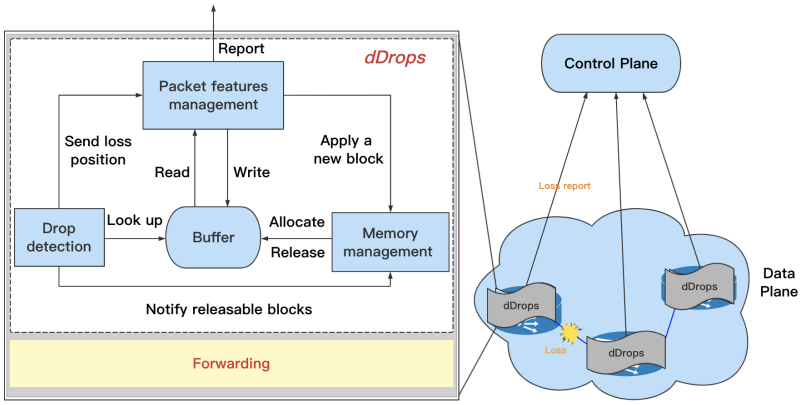


Figure 2: Overview of *dDrops*: The data plane nodes perform the monitoring operations and report lost packets to the control plane which uses the packet features of losses from *dDrops* to locate where losses happen in the network and mitigate the impact of losses.

port to temporarily keep packets that are not confirmed for successful transmission yet. As shown Figure 3, *dDrops* takes 7 steps to detect silent packet drops:

Step 1: For each packet that arrives at the upstream switch, it will identify whether the packet is a normal transmitted packet or a notification packet via packet header type.

Step 2: The egress of upstream switch embeds a packet ID into each transmitted normal packet. The packet ID is increased by 1 for each packet. It can be embedded into unused header fields (e.g., VLAN tags and DSCP) or customized header.

Step 3: The egress of upstream switch inserts packet features and the packet ID into the block buffer allocated to the egress port. If the block buffer is full, a new block buffer will be allocated by *memory management* from a global buffer that is shared by all ports of a switch (see Section 3.4).

Step 4: The ingress of downstream switch will extract the packet ID and compare it with the last received packet. Discontinuous packet IDs are considered as a signal of packet loss.

Step 5: In case of packet loss, the downstream switch sends back a notification packet (i.e., loss notification packet) which contains the current received packet ID and the last received packet ID to upstream switch.

Step 6: The upstream switch performs the lookup operation for retrieving packet features of lost packets whose packet IDs fall into the missing interval from buffer, and reports them to the controller by *packet features management*. Then, the upstream switch releases the blocks, whose packets have been reported to the controller or confirmed to be received successfully, through *memory management*.

Step 7: If more than K packets are received without any drops (see detailed discuss on K in Section 3.5), the downstream switch would proactively send a notification packet (i.e., a proactive notification packet) to the upstream switch, which contains the last received packet ID. That means all packets are received prior to the packet ID. Thus, the upstream switch can release those blocks whose packets have been received by the downstream switch.

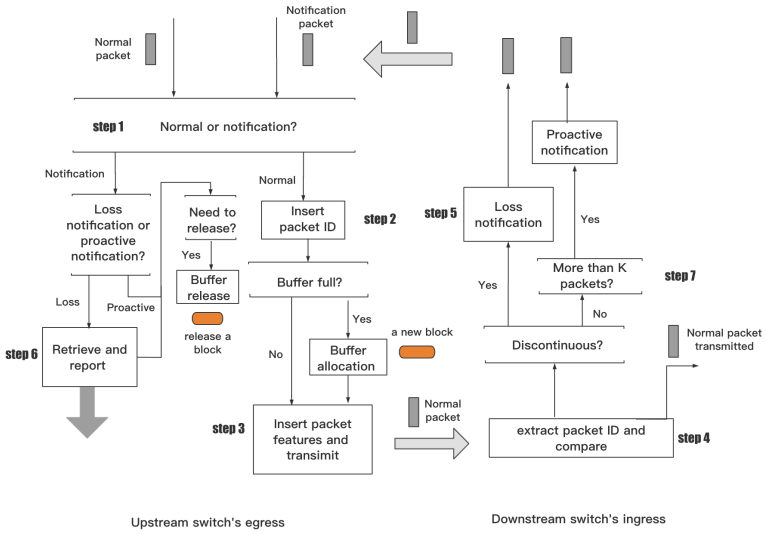


Figure 3: Silent packet drop detection.

The packet ID can either be placed in an existing header field or transported via a custom header. Both approaches need new metadata to store the packet ID. Compared to leveraging an existing header field, inserting packet ID via a custom header will produce extra bandwidth overhead and create more VLIW resources for modifying a new packet header. In the implementation, dDrops chooses to insert the packet ID in an existing header field (using VLAN tags).

dDrops generates notification packet by cloning[3]. To avoid notification packet being dropped, it can be sent three times like prior works [2, 25] at the cost of more bandwidth usage. Another effective optimization is that the discontinuous packet IDs can be piggybacked in headers of normal packets from downstream to upstream switch. For example, discontinuous packet IDs can be inserted to several normal packets, whose packet header type is modified to a notification packet. Upon receiving these packets, the upstream switch will parse packets, extract packet IDs and change them back to normal packet without affecting their normal forwarding.

Upon receiving a loss notification packet, the upstream switch triggers the lookup operation for lost packets and reports them to controller. Note that currently available PDP does not support loops inside a stage. To report multiple continuous lost packets, *dDrops* uses each subsequent packet that arrives at the switch to perform the lookup operation once, until all lost packets are reported (the report of the first dropped packet can be triggered by the loss notification packet).

Due to the limitation of PDP memory and buffer size, if too many packets are dropped (i.e., continuous silent packet drops for a long period of time), more blocks will be allocated constantly and the global buffer will be filled up quickly. When the global buffer is full, *buffer allocation* for new blocks will fail and information of subsequent

[3]The *clone* operation can be used to generate a copy of the original packet in the pipleline.

packets can not be stored in the buffer any more. In this case, lost packets can not be detected correctly, resulting in false negative for detection. Therefore, under the buffer sharing mechanism, the capacity of each port to detect continuous packet loss is within a certain range, which is discussed in Section 3.5.

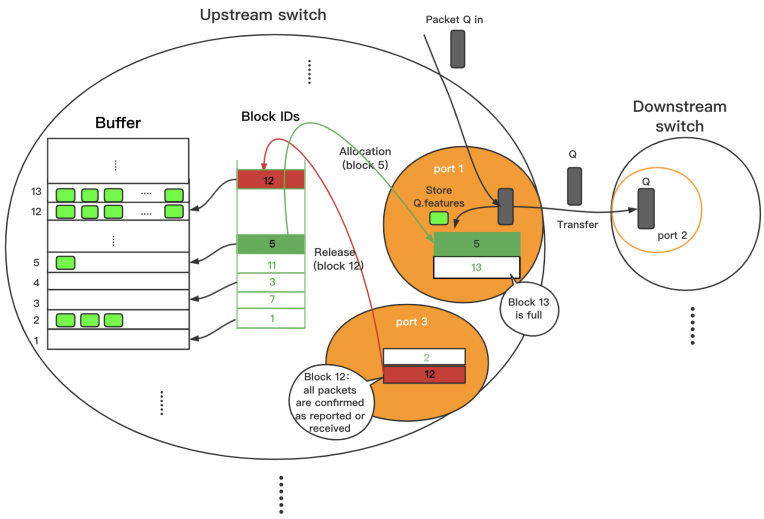


Figure 4: Dynamic allocation and release

3.4. Memory and packet features management

dDrops introduces buffer sharing on a switch by *memory management* to adapt the diversity of traffic and failure among different ports. Initially, *memory management* allocates a block for each port from a global buffer, which stores the packet ID and packet features. When buffer of a port is almost full, *packet features management* will apply for a new block via *memory management*, which will call *buffer allocation* operations to assign buffer space. When all packets of a block have been received by downstream switch or reported to controller due to packet loss (via *packet features management*), the block will be released by *memory management*.

Block-based allocation and store. To allow all ports of switch to share the limited memory in data plane, *dDrops* proposes an innovative block-based buffer sharing mechanism (Figure 4). A global buffer $buffer$ is divided into many blocks, each containing a fixed number of slots, which is denoted by C (a block size). A slot is used to store packet features and packet ID of a packet. Each $block \in buffer$ has a unique $block.id$. *dDrops* allows the buffer to be shared by all ports of a switch through an effective sharing buffer algorithm in Algorithm 1. It maintains a global stack S_{block} containing all blocks. The packet ID is split into $logical.id$ and offset, and uses an array $map[]_i$ for port i to save actual block which is allocated by *memory management*. The function CONVERTID() implements a block buffer table (similar to the page table of virtual memory in operation system) to convert packet ID $pktid$ to $index$, which is the position of the packet in the global buffer. The $index$ is obtained by concatenating the offset W with the actual block id (line 1∼5, see Figure 5), using simple mathematical operations (e.g., *shift*, *and*, *or*).

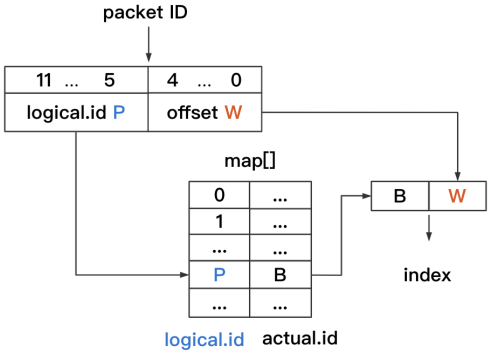


Figure 5: Converting packet ID to buffer index

Specifically, upon transmitting a packet Q on port i, *dDrops* first checks current used block *curblock*. If *curblock* is full and S_{block} is not empty (line 9∼10), *dDrops* would pop a block from S_{block} and allocates it to port i for extension (line 11∼12). Then *dDrops* uses CONVERTID() to calculate *index* and insert packet features to this new block (line 13∼14). The counter of this block is increased by 1 (line 15). If *curblock* is not full, *dDrops* inserts features of packet Q to *curblock* and increase *curblock.counter* (line 17∼19).

Report and release. There are two cases that need to perform *buffer release*. First, when a lost packet of one port is reported, it implies that all packets whose packet ID is smaller than this lost packet are confirmed as reported or received. Thus their space can be released. Second, when the upstream switch receives a proactive notification packet, it means all packets whose packet ID is smaller than the packet ID carried in notification packet have been received. So their space also can be released.

The operations of reporting and releasing buffer are shown in Algorithm 2. *dDrops* maintains a stack S_{lost} to keep lost packet IDs extracted from loss notification packets. For each subsequent packet that arrives at port i, *dDrops* first pops out a lost packet id *pktid* and get *logical.id* from *pktid* (line 2∼3). Then, it uses CONVERTID() to calculate *index* and retrieve features from buffer with *pktid*, reporting them to controller (line $4 \sim 5$). For buffer release, $maxfreeId$ is updated to *logical.id* (line 6). *dDrops* performs *buffer release* operation if $curfreeId$ (which is initialized to 0) is less than $maxfreeId$. Specifically, *dDrops* gets the actual block that can be released by $map[curfreeId]_i$, then pushes it back to S_{block}; $curfreeId$ is increased by 1(line 7∼9). For packets that are confirmed to be received, *dDrops* uses the same operation to release their buffer.

3.5. Discussions

Block size. For an n-ports switch, let the link bandwidth-delay product (BDP) for the ith port be BD_i. Suppose the probability of silent packets drop is ρ. Each packet contains p bytes. Considering packets are transmitted from the ith port of the upstream switch to downstream switch back-to-back, if a packet Q_j is dropped (jth packet of the port), before the loss notification packet is received, the upstream switch would transmit

Algorithm 1 Block-based buffer allocation

Input: Global stack S_{block}; a packet Q to be transmitted by port i.

```
1: function CONVERTID(pktid)
2:     obtain logical.id and offset with pktid
3:     actual.id ← map[logical.id]_i
4:     index ← concatenating offset and actual.id
5:     return index
6: procedure ALLOCATION(Q)
7:     obtain Q.pktid retrieve from Q and get logical.id with Q.pktid
8:     curblock ← read current block used by port i
9:     if curblock.counter = C then
10:        if S_block not empty then
11:            curblock ← S_block.pop()
12:            map[logical.id]_i ← curblock.id
13:            index ←CONVERTID(Q.pktid)
14:            buffer[index] ← Q.features
15:            curblock.counter++
16:    else
17:        index ←CONVERTID(Q.pktid)
18:        buffer[index] ← Q.features
19:        curblock.counter++
```

N_i packets:

$$
\begin{aligned}
N_i &= 2\frac{BD_i}{8p} + 1 + 1(1-\rho) + 2\rho(1-\rho) + 3\rho^2(1-\rho) + ... \\
&= 2\frac{BD_i}{8p} + 1 + \sum_{k=1}^{\infty} k\rho^{k-1}(1-\rho) \\
&= 2\frac{BD_i}{8p} + 1 + \frac{1}{1-\rho} \\
&< 2\frac{BD_i}{8p} + 1 + 2 = 2\frac{BD_i}{8p} + 3 \qquad\qquad 0 \leq \rho < 0.01.
\end{aligned}
$$

To obtain N_i, we first need to know how many packets can hold on the link during the period from the upstream switch to the downstream switch of Q_j. This capacity is determined by the bandwidth (in bits per second or *bps*) multiplied by the one-way transmission time. We call it BD_i and the number of packets is $\frac{BD_i}{8p}$. If the upstream switch sends packets continuously and receives a loss notification packet within the round-trip time, the minimum number of packets that the upstream switch will transmit continuously is $2\frac{BD_i}{8p}$. The "+1" is because the loss notification packet will not be sent back until the entire packet is received. The remaining term after $2\frac{BD_i}{8p} + 1$ is the expected number of packets that the upstream switch is expected to send following Q_j until the first successfully sent packet. It is a geometric distribution that counts the number of successive failures before a success is obtained in a Bernoulli trial with parameter $1 - \rho$. Thus, it should be equal to $\frac{1}{1-\rho}$. This is an increasing function of ρ. Also, since packet

Algorithm 2 Report packet features and release buffer

Input: Global stack S_{lost}, S_{block}.

1: **if** S_{lost} not empty **then**
2: $pktid \leftarrow S_{lost}.pop()$
3: $logical.id \leftarrow$ to shift $pktid$ ▷ refer to Figure 5
4: $index \leftarrow$CONVERTID($pktid$)
5: $report(buffer[index])$ ▷ report features of lost packets to controller
6: $maxfreeId \leftarrow logical.id$ ▷ $maxfreeId$ is the max logical block id that can be released
7: **if** $curfreeId$<$maxfreeId$ **then** ▷ $curfreeId$ is initialized to zero
8: $S_{block}.push(map[curfreeId]_i)$
9: $curfreeId++$

loss rate is very low (e.g., less than 0.01%) in data centers [5], we can approximate N_i to $2\frac{BD_i}{8p}+3$ without the knowledge of the drop rate.

All those packets should be kept in the buffer of the ith port until the loss notification packet is received. Thus, we define the block size C to be at least the maximum of N_i for all ports:

$$C \geq \max(N_1, N_2, \ldots, N_n)$$

Proactive notification. If there is no packet drop for a long time, the downstream switch would not send any drop packet notification and the buffer cannot be released on upstream switch. Thus, *dDrops* lets the downstream switch send a proactive notification if more than K packets are received without any drops. Proactive notification will introduce additional network overhead. To reduce this overhead, *dDrops* requires $K \geq C$. The additional network overhead will decrease as K increases, which is also shown in experiment results in Section 5.3. However, K cannot be too large considering the limited available memory in data plane. We define the maximum of K as K_{max}. *dDrops* needs to ensure all buffer can be released timely under the multi-ports sharing mechanism. Therefore, K_{max} should satisfy the Equation (1). Suppose the size of buffer (total slots) is R and the switch has n ports:

$$(\lceil \frac{K_{max}+2\frac{BD_i}{8p}+1}{C} \rceil + C)n = R \tag{1}$$

$$K_{max} = \lfloor (\frac{R}{n} - C)C - 2\frac{BD_i}{8p} - 1 \rfloor \tag{2}$$

where $\lceil \frac{K_{max}+\frac{2BD_i}{8p}+1}{C} \rceil$ is the number of blocks used by a port when it receives a proactive notification packet from downstream switch that have been received K packets without any drops. One block ("$+C$") is used to store the subsequent packets for reporting and releasing.

Detection capacity of silent packet drops. Detection capacity refers to the maximum number of lost packets that can be detected when continuous silent packet drops happen. The detection capacity of a port is determined by the maximum buffer that the port may obtain. *dDrops* utilize the diversity of network traffic and failure

across different ports of a switch to obtain a large detection capacity. Considering two extreme cases, if continuous packet loss occurs on all ports at the same time, a port has the least capacity to detect continuous packet loss. On the other extreme, if only one port experiences continuous packet loss and others work well, the port has the largest capacity of detecting continuous packet loss. About the detection capacity:

- *Min*: In the first case, all ports suffer from continuous packet drops until the buffer is full. The detection capacity of *dDrops* is similar to allocating the same fixed buffer to each port, which is $\frac{R}{n} - C$. C is subtracted to maintain a block for storing the subsequent packets that are used for reporting and releasing.

- *Max*: When continuous packet drops only happen on one port, each other port only occupies one block. Therefore, the maximum detection capacity would be $R - (n-1) \times C - C = R - nC$. The "$-C$" sharing the same reason as the first case.

4. Implementation

We have implemented *dDrops* on both BMv2 [26] and the P4 hardware switch (Flnet S9180-32X) with a 3.2Tb/s Barefoot Tofino 32D ASIC[24], using $P4_{14}$ language[4].

The detection module is implemented in ingress that is shown in Figure 3. It extracts packet ID and packet features from each packet, maps the port to a register array using a hash of the ingress port, and then triggers a notification by cloning if loss happens or there are more than K packets received without any drops. Other functions, such as reporting packet features of lost packets, allocating and releasing buffer, are implemented in egress. To avoid affecting normal packet forwarding, *dDrops* sends notification packets and features of lost by *clone_ingress_pkt_to_egress* and *clone_egress_pkt_to_egress* respectively.

The dynamic memory management module is implemented using a pointer and register stack. When a block is filled up, *dDrops* checks the pointer of the stack for allocation. If the stack is not empty, *dDrops* moves the pointer and pops a block from the stack. When all packets are reported or received in one block, *dDrops* pushes the block back to the stack and updates the pointer. This design requires a simultaneous *read* and *write* register operations when the port needs both allocation and release block. However, each packet can only access a register once and can not move "backwards" in the pipeline of Tofino ASIC. To solve the problem, *dDrops* uses *clone* packet to complete the release operation.

In the design of block buffer table, each port uses an array *map* to store actual block. There may be three situations that need to *read* or *write* this array for each packet. First, when *packet features management* stores a packet, it would read the array to determine the position of buffer in order to keep the packet's features. Second, when packet losses occur, *drop detection* component needs to read the array to look up their positions in the buffer. Third, when *memory management* allocates a block for the port, it would write actual block id to the array. Therefore, *dDrops* needs to access *map* at most three times for each packet. In order to solve the limitation that only one read and write operation

[4] https://github.com/AntLab-Repo/dDrops

can be performed in a pipeline, *dDrops clones* a normal packet multiple times to achieve above operations.

5. Evaluation

In this section, we evaluate performance of *dDrops* on *drop detection*, *packet features management* and *memory management*. First, we analyze detecting capacity, the latency to obtain features of lost packet and the PDP resource requirements of *dDrops* to demonstrate its feasibility and performance on real hardware switches. Second, we show the performance of *dDrops* on both bandwidth overhead and detection latency through Mininet [27]. Third, we evaluate *dDrops* under realistic network traffic trace to verify its effectiveness on silent packet drop detection and memory management.

5.1. Experiment setup

We have setup a testbed composed of two connected P4 hardware switches (Flnet S9180-32X with Tofino ASIC). Each switch is connected with a server, which has 2.90 GHz CPU cores, 16 GB RAM, and 10Gbps NIC. All network links are 10Gbps fibers. In addition to *dDrops*, we also deploy NetSeer [2] for comparison. NetSeer detects packet loss with coordination of both the upstream and downstream switches by allocating a fixed memory for each port.

In addition to testbed experiments, we also conduct extensive evaluations in Mininet with a network topology that consists of 6 servers, 3 switches and 9 links (see Figure 6). All switches are interconnected, while each switch is connected with two servers. Unless otherwise stated, the host sends a packet every $10ms$, and the average size of packets is 64B. We tune the drop rate during the experiment for comparison with existing state-of-the-art solutions including NetSeer [2], Everflow [11] and NetSight [16]. Both Everflow and NetSight can detect packet loss by sending the packet headers of all packets at every hop to a centralized controller in the entire network.

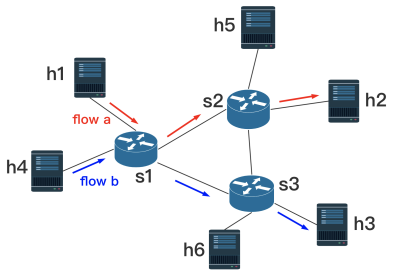
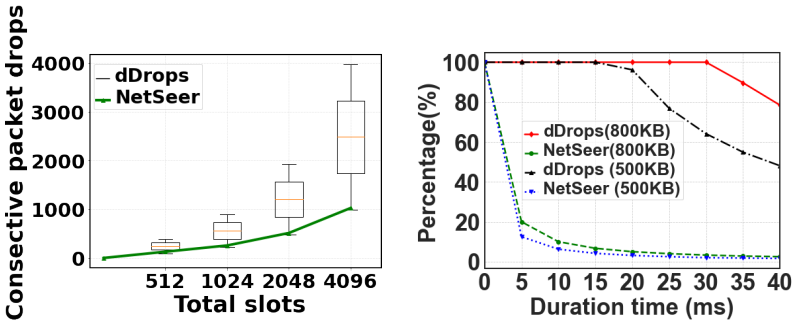


Figure 6: Network topology in Mininet.

5.2. Performance in P4 hardware switch

Detecting capacity. We evaluate the capacity of detecting continuous packet drops for both *dDrops* and NetSeer. We first calculate the upper and lower bounds of the capacity of detecting continuous packet drops of *dDrops* according to the formula in Section 3.5. In addition, a port may be assigned any number of blocks between the



(a) Continuous packet drop v.s. Slots (b) Percentage of detecting drops.

Figure 7: Detection capacity of silent packet drops

upper and lower bounds under the multi-port sharing mechanism due to the randomness of packet rate and loss rate in network [3]. Thus, we assume the probability of any block being allocated for one port is equal for simplicity. For NetSeer, the capacity of detecting continuous packet drops is fixed at 1/4 of the total buffer. Thus, we can obtain the maximum number of continuous silent packet drops that can be detected with different total buffer size, which is shown in Figure 7(a) (four ports are used for each switch). Figure 7(a) shows that the capacity of *dDrops* is always larger than that of NetSeer with different buffer sizes. Specifically, the capacity of *dDrops* increases by 118% and 134% on average for 1024 and 2048 of slots respectively when compared to NetSeer. The maximum detectable number of continuous packet drops of *dDrops* is close to the size of total buffer. Figure 7(b) shows the percentage of dropped packets can be detected correctly when long continuous silent packet drop happens. With 800KB SRAM in total, NetSeer can only detect 20% and 3.3% of lost packets when it lasts for $5ms$ and $30ms$ respectively, while *dDrops* can detect 100%. It means that *dDrops* achieves higher accuracy than NetSeer. The reason for this result is the inherent benefit of dynamic memory allocation/release. Considering the diversity of both traffic rate and failure on each port, *dDrops* will allocate more memory from the global buffer to those ports experiencing continuous silent packet drops, while the memory of each port is fixed for NetSeer.

PDP Resource Usage. Table 2 summarizes the memory and computational resource usage in the pipeline of Tofino ASIC for both *dDrops* and NetSeer [2]. *dDrops* uses 18.33% SRAM, while NetSeer consumes 36.98% SRAM with the same buffer size (65536 slots). Thus, *dDrops* saves 50% of the memory compared with NetSeer. NetSeer maintains fixed size ring buffer per-port at each switch which results in high memory consumption, because the number of simultaneous faulty links is small [3]. In addition, *dDrops* also consumes fewer resources than NetSeer in terms of stateful ALUs, stages and VLIWs.

Latency to obtain packet features. Figure 8 shows the detection latency for obtaining packet features of silent packet drops. We define the detection latency as $t_{interval}$ = $t_{receive}$ - t_{send}, where $t_{receive}$ is the time of packet features received by controller and t_{send} is the time of a subsequent packet sent by upstream switch which will triggers the lookup operation for reporting. The average and 99th percentile of latency for *dDrops* are

Table 2: Resource consumption on Tofino ASIC.

	dDrops	NetSeer
Memory		
SRAM	18.33%	36.98%
TCAM	0%	0%
Computational		
sALUs	43.75%	47.92%
VLIWs	6.77%	7.81%
Stages	9	12

31us and 45.57us respectively, while they are 30.2us and 43.08us for NetSeer. Therefore, the detection latency for obtaining packet features of *dDrops* is close to NetSeer, while *dDrops* obtains larger capacity of detection with lower SRAM than NetSeer.

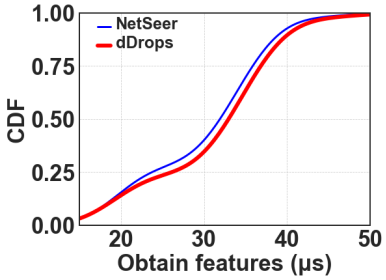
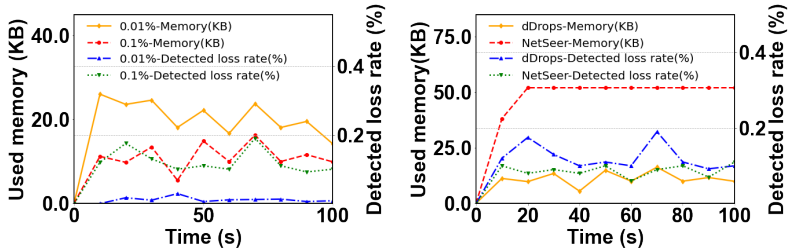


Figure 8: Latency to obtain packet features on P4 hardware switches.

5.3. Performance in P4 software switch

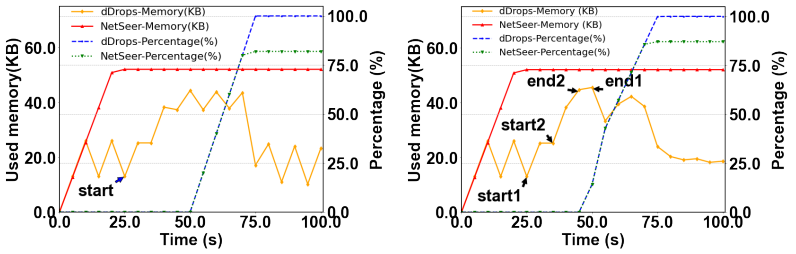
Memory usage of multiple ports. We use the dataset collected in 2010 from a data center trace (UN2) [28] to evaluate the memory usage and performance on lost packets detection on a switch with multiple ports (see Figure 9). It contains different types of packets, such as TCP and UDP. Each host emits these traffic with a certain rate using *tcpreplay* (which is a tool for replaying network traffic from files). There are two flows (flow a and b in Figure 6), which are sent to two different destination hosts through different ports of the switch. The data rate of flow a is 200pps and flow b is 100pps. Different drop rates on links are used for evaluation. The total buffer size is set to be 52KB for *dDrops* and NetSeer.

Figure 9(a) shows that the average of consumed memory remain stable around 11.15KB and 20.65KB at drop rate 0.1% and 0.01%, respectively. While Figure 9(b) shows that the ring buffer of NetSeer is exhausted quickly (reached 52KB within 20 seconds) as more and more packets arrive at the ports. Since NetSeer have no active release mechanism, it needs to overwrite existing data when buffer is full. In contrast, *dDrops* can release the blocks for packets that have been received and reported. These results also show that both methods can detect packet loss rate of links well (i.e., 0.01% and 0.1%), but *dDrops* can detect more packet drops than NetSeer (e.g., 20s and 70s in Figure 9(b)).



(a) *dDrops* memory usage and detected drop rate.

(b) *dDrops* vs. NetSeer (0.1% loss rate).

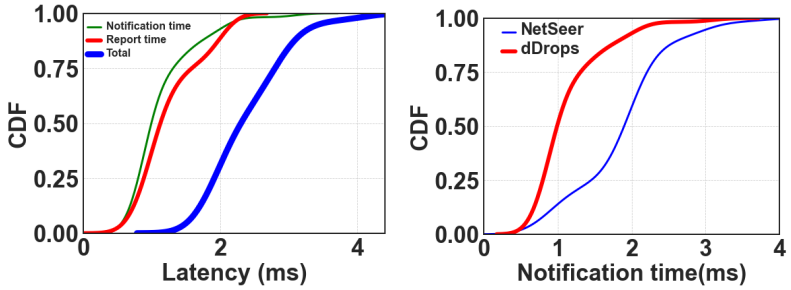


(c) *dDrops* vs. NetSeer under continuous packet drop.

(d) *dDrops* vs. NetSeer under continuous packet drop (two ports).

Figure 9: Memory usage and detected packets with different drop rate

Memory usage under continuous packet drop. We also evaluate the memory usage and ratio of correctly detected lost packets when continuous packet drops happen in Figure 9(c) and Figure 9(d). In Figure 9(c), we simulate continuous silent packet drops by disabling one of the two ports used in the experiments (from $25s$ to $50s$). The data rate of both flows a and b are $100pps$. Since *dDrops* adopts dynamic memory allocation/release mechanism, its total memory usage starts to increase at $25s$ when continuous silent packet drops happen. At $50s$, the first packet after continuous silent packet drops arrives at the downstream switch. Upon receiving the loss notification packet from downstream switch, lost packets are reported to the controller by subsequent packets. During this process, blocks of reported packets will be released and the total memory usage of *dDrops* is gradually reduced to normal level. Particularly, *dDrops* detects and reports all lost packets during continuous silent packet drops. For NetSeer, it fills up all memories rapidly (within $25s$) and only detects 81% of all lost packets as a result of buffer overflow. Furthermore, we also disabled another port to simulate the case of concurrent continuous packet drops at two ports (from $35s$ to $45s$). Results in Figure 9(d) show that, during the period $35s$ to $50s$, the maximum memory usage of *dDrops* is increased by 7.38KB, compared to the scenario in Figure 9(c), due to continuous packet drops of another port. At $55s$, the blocks of this port are released since lost packets are reported, and the total memory usage of *dDrops* significantly reduced for the first time (the memory usage in Figure 9(c) at this point is larger because the occupied memory has not been released without any drops or proactive notification packets). At $75s$, all lost packets of the two ports are reported and memory is released to normal levels.



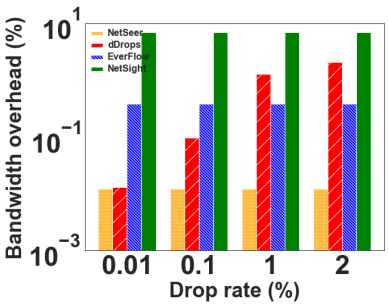
(a) Breakdown of detection latency. (b) Latency to get notification packet.

Figure 10: Latency of detection

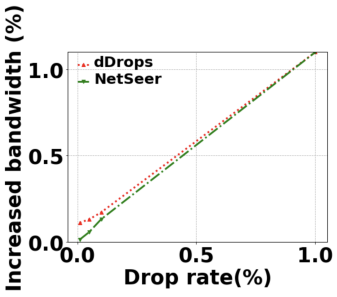
Detection Latency. Figure 10(a) shows the result of detection latency of silent packet drops of *dDrops*. In this experiment, TCP traffic is used and loss rate is set to be 0.01%. The detection latency (blue curve) is broken down into notification time and report time. Notification time refers to the interval between when the upstream switch sends a subsequent packet, which will trigger a notification packet in the downstream switch, and receives the notification packet. Report time consists of the time to look up features of lost packets on the upstream switch and the time to send them to the controller. Results show that the $99th$ percentage of lost packets can be detected and reported within $1ms$ to $4ms$. Figure 10(b) shows that average notification time of *dDrops* and NetSeer are around $1.1ms$ and $1.8ms$ respectively. Notification time of NetSeer is slightly longer because it maintains independent registers and tables/actions for each port to store or access packet features that requires more memory and computational resource under multiple ports.

Bandwidth overhead. Figure 11(a) shows how the drop rate affects the bandwidth overhead between switches and controller. Both Everflow [11] and NetSight [16] capture packets at every hop in the entire network and send them to a centralized controller. Everflow mirrors raw packets and NetSight introduces compression to reduce the overhead. Their overhead are proportional to the ongoing traffic. Both *dDrops* and NetSeer use coordination between neighbor switches and report lost packet features to controller. Compared with NetSight, *dDrops* reduces 99.8% and 98.6% bandwidth usage for 0.01% and 0.1% drop rate respectively. When packet drop rate is larger than 1%, the bandwidth overhead of *dDrops* (2.09%) becomes higher than Everflow. The reason is that *dDrops* uses clones of packets to report features of lost packets to controller and its overhead is proportional to the number of lost packets. However, previous studies [5] show that the packet loss rate is very low (less than 0.01%) in data centers. Thus, such overhead of *dDrops* is acceptable in practice. Overhead of NetSeer is 0.012%, which is lower than *dDrops* because NetSeer optimizes it by compressing and batching loss events.

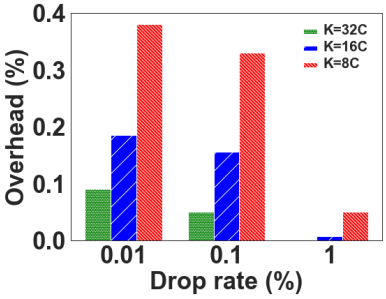
Figure 11(b) shows how the drop rate affect the overhead between adjacent switches. When packet loss rate is relatively high, the overhead of *dDrops* is close to NetSeer. When packet loss rate is low, the overhead of *dDrops* is slightly higher than NetSeer, which is necessary because the downstream switch needs to proactively send notification packets to the upstream switch to release buffer.



(a) Bandwidth overhead of switch-to-controller.



(b) Bandwidth overhead between switches[6].



(c) Bandwidth overhead of *dDrops* on different K.

Figure 11: Overall bandwidth overhead

We also compare the additional bandwidth overhead generated by proactive notification packets with different parameter K and packet drop rate. Figure 11(c) shows that the additional overhead decreases as the packet loss rate increases. Moreover, the overhead also decreases when value of K increases. This is because *dDrops* needs to send a proactive notification packet when there are more than K packets are received without any drops. In Section 3.5, we have discussed the maximum value of K considering available memory to minimize the bandwidth overhead.

dDrops uses the programmability of PDP to clone packets for notification and report. The bandwidth overhead of cloning includes overhead of switch-to-controller and overhead between adjacent switches. According to above analysis on Figure 11, the bandwidth overhead of switch-to-controller depends on silent packet drop rate, while the bandwidth overhead between adjacent switches is determined by both K and loss rate.

6. Related works

In data centers, there have already been many network monitoring systems, but none of them can meet design goals for silent packet drops detection.

[6]No extra bandwidth is introduced between adjacent switches for NetSight and EverFlow.

Host-based telemetry solution. Host-based solutions [5, 29, 10] send probes into the network to infer failures or measure the states of devices or links according to responses. However, they cannot precisely locate causes of losses in the network with data solely from end hosts, especially small-scale packet drops.

Pingmesh [5] sends probes into the network to locate causes of lost packets, but can not infer exact switch within the leaf or spine groups suffering from losses. Furthermore, to reduce its bandwidth overhead, the frequency of probing is at least 10 seconds, it may result in many losses which cannot be detected between two probes.

Sketch-based. Recent works [30, 15, 20, 25, 31] adopt sketches for network measurement in hardware or software switches. Network operators could collect the information of every packet in a compact data structure and analyze problems such as switch offline or DDoS attacks. However, they are not robust to network failures, and suffer from the trade off between memory efficiency and provable accuracy.

FlowRadar [9] encodes per-flow counters with a small memory at switches, then leverages the computing power at the remote collector to perform network-wide decoding and analysis of the flow counters. It identifies packet losses by comparing the counters at two nearby hops. However, it needs to wait for synchronization of counters, which may result in coarse-grained time scale (e.g., 10s of milliseconds).

Packet telemetry. NetSeer [2] leverages coordination between upstream and downstream switches to detect silent packet drops in packet-level. It allocates a fixed memory for each port of switches, causing large memory consumption and low accuracy. The design of *dDrops* takes inspiration from NetSeer, but differs in three aspects: (1) Instead of allocating fixed memory to each port, *dDrops* introduces a novel dynamic memory management to enable memory sharing across all ports of a switch, which reduces memory consumption on the data plane. (2) *dDrops* can well adapt to dynamic failure and packet rate on different ports, which improves memory usage efficiency and accuracy of detection. (3) *dDrops* allows downstream switch to trigger proactive notifications, which can further reduce memory consumption. Some other previous works have also used dynamic memory management in the data plane. For example, *Flow [32] uses a cache to group packet-level telemetry information according to the flow IDs. It allocates a narrow ring buffer for each flow, attempts to allocate a wider buffer from a buffer pool for active flows when the narrow ring buffer is filled up, then the active flows keep the buffer until collision happen or it is full. Instead of just allocating once, *dDrops* can allocate blocks of memory for each port multiple times dynamically. Another difference is that the memory of *dDrops* can be proactively released after ensuring the packets are received by downstream switch.

ERSPAN [33, 34, 11] and INT [16, 35] can potentially achieve full-visibility, but suffer from granularity-cost trade-off. Their bandwidth and processing overhead grow quickly with the scale of the network. Thus, they cannot fully capture such random events (e.g., silent packet drop). NetSight [16] can detect silent packet drops by mirroring all packets at every hop in entire network to a centralized controller. It requires 31% bandwidth overhead and proposes techniques to reduce the bandwidth overhead to 7%. But these techniques require heavy modifications on switching ASICs and are difficult to deploy. To reduce bandwidth overhead, EverFlow [11] selects packets based-on pre-selection filers, and then mirrors these packets to remote controller. The controller extracts per flow information and performs static analysis. since it only traces a subset of packets, some lost packets may be missed (see Table 1).

7. Conclusions and future works

The measurement of silent packet drops is significant on application performance of data centers and today's clouds network. We presented *dDrops*, a packet-level telemetry system that supports *low memory consumption*, *dynamic adaption to port difference* and *fast detection* network measurement. It leverages the PDP to report the detail information of silent packet drops to remote controller. Experiments on both software and hardware switches show that *dDrops* can detect silent packet drops efficiently with low memory through the dynamic memory management scheme.

Since the probability of continuous silent packet drops has not been studied yet, we use independent probability of packet loss in *dDrops*, which is commonly used in many existing works [36]. In the future work, we will conduct more deep investigation and study on continuous silent packet drops such as probability analysis. Also, a memory sharing scheme on the data plane is proposed in this paper. We will extend this on-demand allocation and dynamic release mechanism to other measurement tasks, such as flow distribution measurement, which will be part of our future work.

8. Acknowledgments

This work has been partially supported by National Natural Science Foundation of China (NSFC) No. 62172189 and 61772235; Natural Science Foundation of Guangdong Province No. 2020A1515010771; Science and Technology Program of Guangzhou No. 202002030372; The UK Engineering and Physical Sciences Research Council (EPSRC) grants EP/P004407/2 and EP/P004024/1, and InnovateUK grant 106199-47198.

References

[1] G. Dimopoulos, P. Barlet-Ros, C. Dovrolis, I. Leontiadis, Detecting network performance anomalies with contextual anomaly detection, in: 2017 IEEE International Workshop on Measurement and Networking (MN), 2017, pp. 1–6.

[2] Y. Zhou, C. Sun, H. H. Liu, R. Miao, S. Bai, B. Li, Z. Zheng, L. Zhu, Z. Shen, Y. Xi, et al., Flow event telemetry on programmable data plane, in: Proceedings of the Conference of the ACM Special Interest Group on Data Communication, 2020, pp. 76–89.

[3] X. Wu, D. Turner, C.-C. Chen, D. A. Maltz, X. Yang, L. Yuan, M. Zhang, NetPilot: Automating datacenter network failure mitigation, in: Proceedings of the Conference of the ACM Special Interest Group on Data Communication, 2012, pp. 419–430.

[4] S. Sheng, Q. Huang, P. P. Lee, Deltaint: Toward general in-band network telemetry with extremely low bandwidth overhead, in: 2021 IEEE 29th International Conference on Network Protocols (ICNP), IEEE, 2021, pp. 1–11.

[5] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, et al., Pingmesh: A large-scale system for data center network latency measurement and analysis, in: Proceedings of the Conference of the ACM Special Interest Group on Data Communication, 2015, pp. 139–152.

[6] Y. Li, R. Miao, C. Kim, M. Yu, Lossradar: Fast detection of lost packets in data center networks, in: Proceedings of the 12th International on Conference on emerging Networking Experiments and Technologies, 2016, pp. 481–495.

[7] J. D. Case, M. S. Fedor, M. L. Schoffstall, J. R. Davin, Simple network management protocol (SNMP), RFC (1990) 1–36, Accessed on: Feb 25, 2022. `https://datatracker.ietf.org/doc/html/rfc1157`.

[8] B. Claise, Cisco systems netflow services export version 9, RFC (2004) 1–33, Accessed on: Feb 25, 2022. `https://www.ietf.org/rfc/rfc3954.txt`.

[9] Y. Li, R. Miao, C. Kim, M. Yu, FlowRadar: A better NetFlow for data centers, in: 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2016, pp. 311–324.

[10] C. Tan, Z. Jin, C. Guo, T. Zhang, H. Wu, K. Deng, D. Bi, D. Xiang, Netbouncer: Active device and link failure localization in data center networks, in: 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2019, pp. 599–614.

[11] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, et al., Packet-level telemetry in large datacenter networks, in: Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication, 2015, pp. 479–491.

[12] V. Sekar, M. K. Reiter, W. Willinger, H. Zhang, R. R. Kompella, cSamp: A system for Network-Wide flow monitoring, in: 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2008.

[13] M. Yu, L. Jose, R. Miao, Software defined traffic measurement with openSketch, in: 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2013, pp. 29–42.

[14] J. Suh, T. T. Kwon, C. Dixon, W. Felter, J. Carter, OpenSample: A low-latency, sampling-based measurement platform for commodity SDN, in: 2014 IEEE 34th International Conference on Distributed Computing Systems, 2014, pp. 228–237.

[15] Z. Liu, R. Ben-Basat, G. Einziger, Y. Kassner, V. Braverman, R. Friedman, V. Sekar, Nitrosketch: Robust and general sketch-based monitoring in software switches, in: Proceedings of the Conference of the ACM Special Interest Group on Data Communication, 2019, pp. 334–350.

[16] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, N. McKeown, 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), 2014, pp. 71–85.

[17] D. Katz, D. Ward, Bidirectional forwarding detection (bfd)(rfc 5880), IETF, Accessed on: Feb 25, 2022. `https://datatracker.ietf.org/doc/html/rfc5880`.

[18] D. Zhuo, M. Ghobadi, R. Mahajan, K.-T. Förster, A. Krishnamurthy, T. Anderson, Understanding and mitigating packet corruption in data center networks, in: Proceedings of the Conference of the ACM Special Interest Group on Data Communication, 2017, pp. 362–375.

[19] P. Bailis, K. Kingsbury, The network is reliable: An informal survey of real-world communications failures, Queue 12 (7) (2014) 20–32.

[20] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, S. Uhlig, Elastic Sketch: Adaptive and fast network-wide measurements, in: Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication, 2018, pp. 561–575.

[21] V. T. Lam, M. Mitzenmacher, G. Varghese, Carousel: Scalable logging for intrusion prevention systems, in: 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2010.

[22] W. Leland, M. Taqqu, W. Willinger, D. Wilson, On the self-similar nature of ethernet traffic (extended version), IEEE/ACM Transactions on Networking 2 (1994) 1–15.

[23] M. Kalewski, A. Kobusinska, J. Kobusinski, Fast failure detection service for large scale distributed systems, in: 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing, IEEE, 2009, pp. 229–236.

[24] Intel tofino series programmable ethernet switch ASIC, `https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html`, accessed on: Feb 16, 2022.

[25] Y. Zhao, K. Yang, Z. Liu, T. Yang, L. Chen, S. Liu, N. Zheng, R. Wang, H. Wu, Y. Wang, et al., LightGuardian: A Full-Visibility, Lightweight, In-band Telemetry System Using Sketchlets, in: 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2021, pp. 991–1010.

[26] p4lang/behavioral-model, `https://github.com/p4lang/behavioral-model`, accessed on: Feb 16, 2022.

[27] B. Lantz, B. Heller, N. McKeown, A network in a laptop: rapid prototyping for software-defined networks, in: Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks, 2010, pp. 1–6.

[28] T. Benson, A. Akella, D. A. Maltz, Network traffic characteristics of data centers in the wild, in: Proceedings of the 10th ACM SIGCOMM conference on Internet measurement, 2010, pp. 267–280.

[29] Y. Peng, J. Yang, C. Wu, C. Guo, C. Hu, Z. Li, deTector: a topology-aware monitoring system for data center networks, in: 2017 USENIX Annual Technical Conference (USENIX ATC), 2017, pp. 55–68.

[30] Q. Huang, P. P. Lee, Y. Bao, Sketchlearn: relieving user burdens in approximate measurement with automated statistical inference, in: Proceedings of the Conference of the ACM Special Interest

Group on Data Communication, 2018, pp. 576–590.

[31] Q. Huang, S. Sheng, P.-H. Wang, X. Chen, C. Zhang, I. Pedisich, Y. Bao, Z. Han, D. Bharadia, R. Zhang, et al., Toward nearly-zero-error sketching via compressive sensing, in: 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2021, pp. 1027–1044.

[32] J. Sonchack, O. Michel, A. J. Aviv, E. Keller, J. M. Smith, Scaling hardware accelerated network monitoring to concurrent and dynamic queries with * Flow, in: 2018 USENIX Annual Technical Conference (USENIX ATC 18), 2018, pp. 823–835.

[33] D. Yu, Y. Zhu, B. Arzani, R. Fonseca, T. Zhang, K. Deng, L. Yuan, dShark: A general, easy to program and scalable framework for analyzing in-network packet traces, in: 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2019, pp. 207–220.

[34] P. Nikolopoulos, C. Pappas, K. Argyraki, A. Perrig, Retroactive packet sampling for traffic receipts, Proceedings of the ACM on Measurement and Analysis of Computing Systems (2019) 1–39.

[35] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, L. J. Wobker, In-band network telemetry via programmable dataplanes, in: Proceedings of the Conference of the ACM Special Interest Group on Data Communication, Vol. 15, 2015.

[36] B. Arzani, S. Ciraci, L. Chamon, Y. Zhu, H. H. Liu, J. Padhye, B. T. Loo, G. Outhred, 007: Democratically finding the cause of packet drops, in: 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2018, pp. 419–435.