Flow Event Telemetry on Programmable Data Plane

Yu Zhou[★†], Chen Sun[★], Hongqiang Harry Liu[★], Rui Miao[★], Shi Bai[★], Bo Li[★], Zhilong Zheng[★†], Lingjun Zhu[★], Zhen Shen[★], Yongqing Xi[★], Pengcheng Zhang[★], Dennis Cai[★], Ming Zhang[★], Mingwei Xu[†]
[★]*Alibaba Group* [†]*INSC and BNRist, Tsinghua University*

ABSTRACT

Network performance anomalies (NPAs), *e.g.* long-tailed latency, bandwidth decline, *etc.*, are increasingly crucial to cloud providers as applications are getting more sensitive to performance. The fundamental difficulty to quickly mitigate NPAs lies in the limitations of state-of-the-art network monitoring solutions — coarse-grained counters, active probing, or packet telemetry either cannot provide enough insights on flows or incur too much overhead. This paper presents `NetSeer`, a flow event telemetry (FET) monitor which aims to discover and record all performance-critical data plane events, *e.g.* packet drops, congestion, path change, and packet pause. `NetSeer` is efficiently realized on the programmable data plane. It has a high *coverage* on flow events including inter-switch packet drop/corruption which is critical but also challenging to retrieve the original flow information, with novel intra- and inter-switch event detection algorithms running on data plane; `NetSeer` also achieves high *scalability* and *accuracy* with innovative designs of event aggregation, information compression, and message batching that mainly run on data plane, using switch CPU as complement. `NetSeer` has been implemented on commodity programmable switches and NICs. With real case studies and extensive experiments, we show `NetSeer` can reduce NPA mitigation time by 61%–99% with only 0.01% overhead of monitoring traffic.

CCS CONCEPTS

• **Networks** → **Network monitoring**; *Programmable networks*;

KEYWORDS

Flow event telemetry, programmable data plane, monitoring

ACM Reference Format:
Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, Pengcheng Zhang, Dennis Cai, Ming Zhang, Mingwei Xu. 2020. Flow Event Telemetry on Programmable Data Plane. In *Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM '20), August 10–14, 2020, Virtual Event, NY, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3387514.3406214

Yu Zhou and Chen Sun are the co-primary authors.

SIGCOMM '20, August 10–14, 2020, Virtual Event, NY, USA

ACM ISBN 978-1-4503-7955-7/20/08…$15.00
https://doi.org/10.1145/3387514.3406214

1 INTRODUCTION

Cloud providers own networks that can comprise thousands of switches and hundreds of thousands of links. In such large and complex networks, bugs and failures in configuration, software, and hardware are inevitable. Therefore, applications often suffer from unceasing network performance anomalies (NPAs), such as bandwidth decline, long-tailed latency, bandwidth/latency jitters, and so forth.

NPA is increasingly critical to cloud providers as applications become more sensitive to performance. On the one hand, the rapidly ascending network speed remarkably raises the expectation of network performance [15, 34, 40, 71]. For instance, after block storage migrates to RDMA from kernel TCP, the expected average end-to-end network latency inside data centers becomes $20\mu s$ instead of $2ms$. As a result, an occasional $200\mu s$ in-network latency becomes unacceptable, even though it is negligible for kernel TCP, because it makes the user experience unpredictable and significantly slows down batch I/O. On the other hand, interactive applications are quickly emerging and have become a major driving force of the cloud business growth. For instance, applications like algorithmic stock trading, cloud gaming, and VR/AR (virtual reality/augmented reality) are highly profitable, while they also have strong requirements, *e.g.* 99.9% SLA (service level agreement), on the stability and sufficiency of bandwidth and latency [25]. Failing to deliver the SLAs can result in severe customer, reputation and revenue loss.

However, quickly mitigating NPAs is extremely challenging, because it has an exceedingly high standard on the coverage, speed, and accuracy of network monitoring. In most NPA cases, the bottleneck to mitigate NPAs is the time spent on locating the causes of NPAs (§2.2). To quickly locate the causes, cloud providers must be able to instantly discover all performance-critical events that happen on application traffic, *e.g.* packet drops, congestion, *etc.*, which are direct triggers of NPAs. Furthermore, the granularity of these events should be at flow-level rather than the traditional interface- or device-level, because precise information about where and how the victim application's traffic gets disturbed can significantly accelerate the cause location (§5.1). Therefore, cloud providers must watch the data plane for *flow events* and make the event detection as responsive as possible.

Unfortunately, existing network monitoring methodologies are far from satisfying the preceding requirements for troubleshooting NPAs, since they are not designed for this purpose. For instance, off-the-shelf switches merely provide counters that are aggregated into per-interface, per-device, or per-sampled-flow granularity [9, 11, 54]; Probe-based monitoring systems [19, 62] only probe the network with time granularity of 10+ seconds and cannot directly detect the events of original application traffic. As a result, cloud providers usually have to combine coarse-grained information from multiple sources and guess the causes, which takes a long time even

for experienced operators (§2.1). Recently, as data plane becomes more configurable, packet telemetry (PT), such as ERSPAN [72] and INT (in-band network telemetry) [30], are proposed for obtaining fine-grained data plane information. Nonetheless, PT solutions are faced with essential trade-offs between granularity and cost — They either compromise cost and scalability by processing all packets (with truncated payload) [21, 51] or sacrifice granularity by packet sampling [49, 54, 65, 72] or aggregation [32, 38, 69].

The fundamental limitation of existing network monitoring solutions for troubleshooting NPAs lies in the rigidity of the traditional data plane – off-the-shelf switching ASICs can only provide simplistic functions to compute fixed and aggregated counters or mechanically mirror packets to the management plane. Hence, despite the data plane is the origin of flow events, it has to rely on the remote management plane to perform sophisticated monitoring logics, *e.g.* event detection, data cleaning, and compression, *etc.*, resulting in huge traffic transmission and computation overhead.

One promising way to achieve both fine-granularity and cost-efficiency is performing monitoring logics directly in the origin – the switch data plane itself. Fortunately, recent advances on the commodity programmable data plane (PDP) have provided us a new foundation to realize this vision.

This paper presents `NetSeer`, a flow event telemetry (FET) based network monitor which continuously and simultaneously watches all individual flows and comprehensively detects flow events, including packet drops, congestion, path change, and packet pause, with moderate cost. The key idea of `NetSeer` is to detect flow events and compress event data entirely inside PDP before uploading data to the management plane, so it can significantly reduce the monitoring overhead as well as preserving the fine-granularity of data.

However, realizing a FET system is challenging in three major aspects: high coverage of flow events, high scalability with network sizes, and high data accuracy with extremely low false-negative and false-positive rates. To address these challenges, `NetSeer` proposes the following innovative designs.

Coverage: `NetSeer` traces the entire packet processing logic in PDP to comprehensively detect intra-switch flow events. Nevertheless, detecting inter-switch packet drop/corruption events is challenging, since we have no visibility into links, and cannot easily retrieve original flow information from dropped/corrupted packets. `NetSeer` leverages the programmability at both sides of a link to collaboratively discover inter-switch drop events and recover the flow information of dropped or corrupted packets (§3.3).

Scalability: `NetSeer` achieves high scalability with three designs. First, `NetSeer` fully utilizes data plane programmability to precisely identify packets that encounter events, aggregate packets experiencing the same flow-level event, and losslessly compress the flow events, which minimizes the produced event data volume (§3.4). Second, `NetSeer` batches small-sized event messages into large packets with a novel in-data-plane design of circular-packet-driven event collection to reduce transmission and processing overhead in switches (§3.5). Finally, `NetSeer` performs FET in a distributed manner, making it linearly scalable with network size.

Accuracy: `NetSeer` ensures zero false negatives in event generation and uses switch CPU with ASIC offloading to discover and eliminate false positives with small overhead (§3.6).

`NetSeer` has been fully implemented with Barefoot Tofino switches and Netronome SmartNICs. Studies on real NPA cases and extensive testbed-based evaluations show that `NetSeer` can reduce the cause location time of NPAs by 61%-99%, incurring only 0.01% traffic overhead and around 20% hardware resource occupation. `NetSeer` has sufficient capacity to fully cover all targeted flow events unless PDP itself is malfunctioning or some extreme cases happen, *e.g.* >1,000 consecutive packet corruptions or >40Gbps congestion drops.

2 BACKGROUND AND MOTIVATION

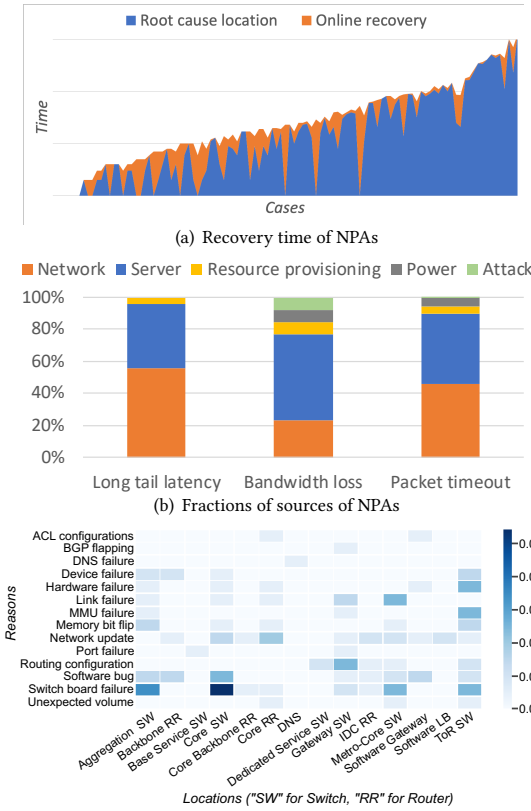
In today's clouds, NPAs can last for hours or longer and severely affect application performance. The ability of NPA troubleshooting is vital for cloud providers. In this section, we use our network operation experience to concretely demonstrate the current status of NPAs in production and motivate our proposal. *Alibaba* is one of the largest global cloud providers, offering tens of diverse services to billions of customers with data centers across 21 areas around the globe. Regarding its physical network, *Alibaba* has a comprehensive network monitoring system, which contains not only well-known techniques, e.g. a PingMesh-like probing system, Syslog and counter collectors, but also a traffic generator and a packet telemetry system based on ERSPAN. Even so, we still encounter hundreds of NPAs yearly and suffer from long recovery time.

2.1 Impacts of NPAs on Cloud Networks

There are generally two types of network faults: connectivity loss and NPA. The former means applications lose the network connectivity for minutes or even longer, and the latter means applications suffer from less violent performance anomalies, such as bandwidth drop, long-tailed latency, or performance jitters. Relatively speaking, NPA is easier to happen and harder to mitigate than connectivity loss. This is because NPA usually happens in a shorter time-scale with randomness, leaving minor fingerprints in networks. The proportion of NPAs in *Alibaba*'s network faults kept increasing in the last three years and reached 80% in 2019.

To understand why NPA is a significant concern of cloud providers, we selected two representative cases to show how an ordinary ticket from customers eventually turned into influential incidents because of the long mitigation time.

Case-1: Is the cause in the network or not? A customer reported unexpected latency glitches (100s of ms, sometimes above 1s) between a virtual machine (VM) and cloud storage service. To diagnose this problem, operators first checked SNMP [9] data and found that *minute-level* switch throughput and buffer usage remained low; Secondly, a *minute-level* probing system similar to Pingmesh [19] did not discover latency anomalies; Finally, operators used the header of customer request packets to construct TCP pings and sent them as probes [72]. Nevertheless, the latency appeared normal. Even though network monitoring systems did not discover any network faults, operators still could not claim with confidence that the network was not responsible, since there might exist fine timescale events such as microbursts that were missed. The search for the cause continued until the storage team discovered disk I/O blocking when processing bursty requests, providing a side proof for "network innocence". The troubleshooting



(a) Recovery time of NPAs

(b) Fractions of sources of NPAs

(c) Heatmap for reasons and locations of network faults

Figure 1: NPA statistics in production from O(100) real service tickets that record the entire NPA troubleshooting details in *Alibaba*'s network (2018-2019).

lasted for about 2 days because of the back-and-forth suspects on networks and applications.

Case-2: Is the lossy switch causing the NPA? One customer reported occasional packet drop between two VMs. Network operators first retrieved switch drop statistics. Data showed that the ToR switch of one VM indeed dropped packets during that period. However, operators were not sure whether these dropped packets included customer's packets. Thus, operators tried to reproduce the packet drop. Operators sent a fixed number of imitated customer packets between the two VMs and counted the number of received packets in each switch alongside the path [72]. By repeating the above operation multiple times, operators successfully captured a packet drop event at the ToR. Meanwhile, SNMP showed that the ToR downlink port indeed dropped packets. Combined with other information, operators suspected that customer packet drops were likely to be caused by occasional bursty incasts. The entire process took over an hour.

The above two cases are relatively easy to handle since their NPAs last persistently. Some NPAs happen occasionally in production and cannot be reproduced in testing environments, and they disturb applications for days or even weeks before the eventual mitigation, causing significant revenue loss.

2.2 Challenges of NPA Cause Location

From the two cases above, we can see that operators spend lots of time locating the causes of NPAs. In fact, most of the efforts for NPA mitigation are devoted to NPA cause location according to general statistics shown in Figure 1(a). This figure shows a realistic distribution of the NPA mitigation time of *Alibaba* with state-of-the-art network monitoring systems, including counters, Syslog, probings, and PT (ERSPAN). About half of the NPAs took more than 10 minutes to recover, and the longest recovery time for one NPA reached above 12 hours, which is unsatisfactory to cloud providers. It also illustrates that the cause location is usually the bottleneck that costs 90% time or more in most cases. In contrast, the actual recovery operations after finding the location of causes are typically fast, because cloud networks and applications often have enough redundancy.

From the two cases above, we can also see that the challenges of NPA cause location stem from the variety of the locations of NPA causes. Firstly, applications usually consider NPAs as the consequences of network faults. But in reality, only a fraction of NPAs are caused by networks. Figure 1(b) shows the fractions of different sources of causes for three types of NPAs. Besides networks, problems in server software/hardware, resource provisioning, power supply, and security attacks can also be the reasons for NPAs. As a result, the diagnosis of NPAs often starts from the network but ping-pongs between applications, software stacks, and other potential issues, wasting tremendous time (*e.g.* Case-1). Secondly, even if the network is indeed causing a given NPA, there are numerous potential root reasons and locations across the whole cloud network that can lead to the NPA, as shown in Figure 1(c). Therefore, operators heavily rely on the coverage and accuracy of network monitoring systems to locate the causes (*e.g.* Case-2).

Given the huge variety of cause reasons and locations, operators always struggle with the coarse time and aggregation granularity of network monitors. For instance, in Case-1, what made operators hesitate to make a judgment of "network innocence" was the sub-second congestion events that could be missed; For another example, in Case-2, the interface based drop counters could not directly indicate whether and where the victim application's packets were dropped. Therefore, operators always try to combine multiple coarse-grained counters to infer where the traffic of the victim application gets disturbed, which wastes a lot of time and is error-prone. Besides, the reason why operators did not use ERSPAN in both cases was the worry that fine-grained traffic mirroring could introduce congestion and packet drops, which would further confuse the troubleshooting process.

Therefore, a powerful network monitoring system is required by network operators for troubleshooting NPAs. This monitor should have the ability to fully catch sub-second level events in the data plane and distinguish among all individual flows.

2.3 Our Proposal: Flow Event Telemetry

Data plane events include packet drop, congestion, path change, and PFC pause on lossless fabrics, which are the direct reasons for NPAs experienced by applications. Our goal is to design a network monitoring system which can provide comprehensive and accurate

flow-level data plane event information, which we call "flow event telemetry (FET)".

It is non-trivial to extract all flow events from the raw traffic. The fundamental reason why traditional counters and PT cannot achieve our goal is that their data plane cannot directly hold the algorithms and data structures for FET. Traditional fixed-function data plane only provides aggregated counters and loses flow level details; PT has to rely on management plane to perform FET logic, resulting in huge resource overhead (31% bandwidth overhead [21]).

With the development of PDP, we see the opportunity of completely realizing FET inside the switch data plane. We believe this choice is rational because of two unique advantages to realize FET on PDP. First, directly running FET in the data plane can significantly reduce monitoring traffic volume (0.01% bandwidth overhead) and data processing overhead. Second, different from CPU, ASICs in PDP can keep line rate when running customized packet processing logics, so the performance impact of FET can be minimized. Therefore, `NetSeer` is built to realize this vision.

`NetSeer` is designed as an always-on monitoring system that comprehensively captures events of all network traffic. Only in this way can network operators have full confidence in network exoneration or fast NPA location. To make such comprehensiveness practical, we propose multiple novel designs to minimize the transmission and processing overhead of monitoring traffic. Considering the limited processing capacity of switch CPU, most of our optimizations are performed in the switch data plane. With stronger switch CPU, we can further reduce the overhead by implementing data encoding and compression algorithms. Finally, a partial deployment of `NetSeer` to monitor flows of specific applications can also enable fine-grained network monitoring for these applications.

3 DESIGN

This section first outlines the challenges and design principles of `NetSeer`. It then elaborates `NetSeer`'s concrete design to achieve full event coverage, good scalability, and high accuracy, and discusses the preconditions of the above features.

3.1 Overview

There are mainly four types of flow events that are the direct causes of NPAs in the network data plane:

- *Packet drop*: It usually results in timeout, retransmission, and slowing down at senders. It is a major cause of NPAs. There are numerous reasons for packet drops. A packet drop that causes NPA can take hours to locate (Figure 3);
- *Packet queuing*: It can delay the transmission and arrival of packets, and it is usually caused by congestion;
- *Packet out-of-order*: It leads to deep buffering or triggers negative acknowledgments (NAKs) at receivers, and it is often induced by flow path changes;
- *Packet pause*: It postpones switches from forwarding packets in a priority queue, and it is brought by priority flow control (PFC [1]) pauses in lossless Ethernet;

Therefore, the crux to make operators quickly and confidently judge *whether* and *where* an application's traffic gets disturbed inside the network is to track the flow events that happen to application's traffic flows – If no flow event is happening during an NPA's period, the network is innocent to the NPA. Otherwise, the network may be guilty, and events are good starting points for cause location. Hence, tracking all flow events can significantly facilitate timely troubleshooting and mitigation of NPAs.

Design goals. `NetSeer` is a network monitor that constantly discovers and records all flow events in networks.

Specifically, *coverage*, *scalability*, and *accuracy* are the three primary requirements in the design of `NetSeer`. First, full coverage means that `NetSeer` should be able to discover all flow events from high-speed on-going traffic as long as the PDP is correctly functioning and the event generation speed does not exceed `NetSeer`'s capacity. Second, `NetSeer` should be scalable enough to work effectively even in large scale production cloud networks. Finally, `NetSeer` should have zero false negatives and very few false positives, which is critical for operators' confidence and efficiency in debugging NPAs.

Challenges. It is highly challenging to achieve the above requirements on coverage, scalability, and accuracy.

- *Coverage*: For full coverage, `NetSeer` needs to identify not only events in ASICs but also events on interfaces and fibers. Especially, for silent packet drops and packet corruptions over links, `NetSeer` also needs to recover the flow identifiers from the completely lost or partial distorted packets.
- *Scalability*: There can be thousands of switches, hundreds of thousands of links, millions of flows and hundreds of Tbps traffic inside a cloud network. `NetSeer` needs to effectively compress event data to avoid impacting applications' traffic or introducing too much overhead. Moreover, encapsulating each individually compressed event into one small packet is costly to transmit and process. `NetSeer` needs to reorganize small-sized events to reduce the overhead.
- *Accuracy*: Due to the limitations on the computations that can be performed on programmable switching ASICs, false-positive may be unavoidable when we push a "no false negative" guarantee (which is more critical for operators to exonerate or debug the network with full confidence). Therefore, `NetSeer` must be able to tolerate these limitations to fulfill the accuracy requirement.

Design principles. `NetSeer` takes full advantage of PDP to address the challenges and achieve the goals.

- *For coverage*, it traces the entire forwarding logic inside the programmable ASICs to detect events inside switches and runs in-data-plane algorithms on the neighboring switches/NICs together to discover drops and corruptions over links and recover the flow information of the events.
- *For scalability*, it implements efficient data structures and algorithms to compress and batch packet-level events into flow-level events while keeping the information integrity for troubleshooting NPAs. `NetSeer` fully utilizes the resources of switch PDP to process information locally as much as possible, so it can linearly grow with the network size.
- *For accuracy*, it jointly leverages programmable ASIC, CPU, and RAM resources in switches to avoid false negatives, eliminate false positives with high efficiency, and reliably transmit flow events to the backend storage.

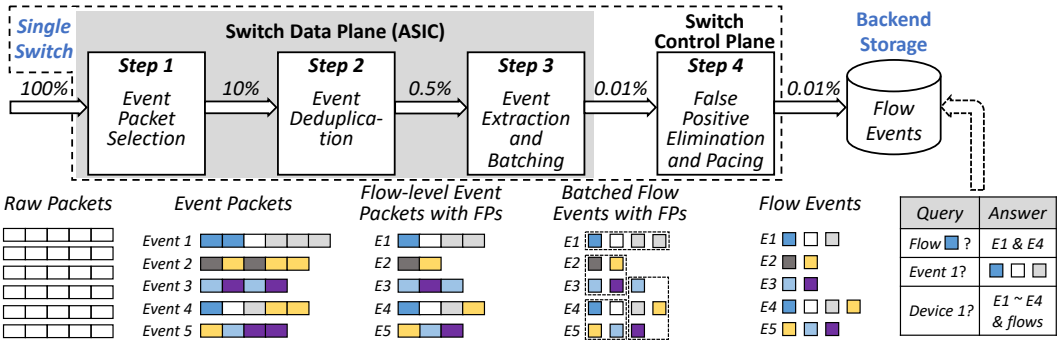


Figure 2: The architecture and workflow of `NetSeer`. (*FP* stands for false positive.)

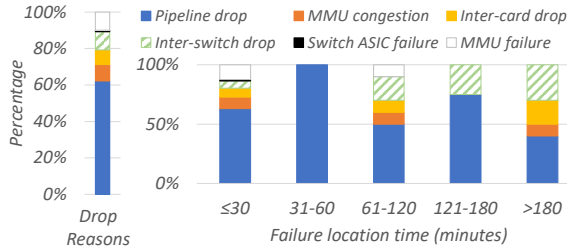


Figure 3: Packet drops that cause NPAs (from *Alibaba*).

3.2 Architecture and Workflow

We present the system architecture and workflow of `NetSeer` in Figure 2. Each programmable switch takes 4 steps to derive comprehensive and compact flow events from original traffic.

Step 1: `NetSeer` uses logics in PDP to precisely *select* (not sample) packets that experience flow events. According to our evaluation, the event packets only occupy <10% of all network traffic.

Step 2: `NetSeer` leverages the programmability of data plane to remove redundant event packets, such as merging consecutive congestion packets of a single flow into one flow-level event, which could reduce the data volume by 95% and also smooth event bursts under continuous congestion or drops.

Step 3: `NetSeer` extracts event information and batches the events to further reduce the transmission and processing overhead. Doing so could eliminate unnecessary information and reduce the data volume by two orders of magnitude.

Step 4: `NetSeer` leverages the ASIC, CPU, and RAM in switches to eliminate false positives, and performs pacing and reliable transmission to deliver events (0.01% of the original traffic) to the backend storage. Operators could flexibly query the storage by specifying a flow, event, device, or period and obtain related flow events. Flow events explicitly correlate flows and abnormal network forwarding behaviors, which enables operators to quickly locate NPAs if any.

3.3 Event Packet Detection

Generally, `NetSeer` traces each step in packet processing in the data plane to detect all events that happen to each packet. While it is not hard to detect congestion, path change or pause, it is challenging to fully detect all packet drops.

Understanding packet drops. Packet drop is a major cause of NPAs. According to *Alibaba*'s records, 86% NPAs are caused by various types of packet drops. Figure 3 presents the overall fractions of different packet drop types and breakdown distribution with network fault location time in *Alibaba*'s networks. Despite intuitively packet drops due to congestion should be the most common type, Figure 3 shows that pipeline drops (*e.g.* due to routing blackholes, ACL rules, zero TTL, larger-than-MTU packet size, *etc.*) cause more than 60% NPAs. Congestion drop takes about 10%, and most cases are large scale incasts. Inter-switch and inter-card drops are silent packet drops or corruption drops over the link between two independent forwarding pipelines. There are also about 10% NPAs caused by malfunctioning hardware, *e.g.* ASIC failures or MMU failures. Since all types of packet drops can result in long cause location time, `NetSeer` must cover as many types as possible.

Detecting intra-switch packet drops. The difficulty to fully detect packet drops inside a switch lies in the limitations of programmable switching ASICs.

A straightforward general solution to detect intra-switch packet drops is to record the appearance of packets at the beginning of the programmable pipeline and confirm the exit of packets at the pipeline tail. Effective as it seems, this solution has several problems. First, maintaining such a record requires unacceptable large memory. If we use compact algorithms such as hash tables, unavoidable hash collisions could result in missed or wrongly identified packet drops and deteriorate event coverage. Second, due to network events, packets could be delayed, dropped, or disorganized. To verify the exit of every single packet, we have to maintain a timer for each packet, introducing unacceptable overhead. Finally, we have to check the packet record at the beginning and end of the pipeline simultaneously, demanding a synchronization mechanism that harms performance.

Therefore, instead of a general solution that treats a switch as a black box, we have to go deeper into the packet forwarding logic to detect packet drop events. Figure 4 summarizes the reasons for intra-switch packet drops according to cloud provider *Alibaba*'s historical records and our conversation with some prominent switch vendors. Intra-switch packet drop has mainly two types – pipeline drop and congestion drop. Pipeline drop is caused by pathological packet format, unavailable route or next-hop interface, or simply the blockage from ACL rules. Congestion drops happen when queues are full and have to drop incoming packets. `NetSeer` embeds drop

Switch status	Drop type	Drop reason (partial)	Detection method
Functional Fully covered by NetSeer	Pipeline drop	Parity error	Report a packet when table lookup miss happens to this packet in the pipeline
		Port / Link down	Report a packet when the target port / link / switch for the packet is down
		ACL config error	Report a packet when it is dropped by an ACL rule
		Forwarding loop	Report a packet when its TTL reaches 0
	MMU Congestion drop	Uneven ECMP	Redirect packets to be dropped by MMU to a dedicated internal port, and report in egress
		Unexpected volume	
	Inter-switch drop or corruption	Link corruption	1. Record & number packets in upstream switch 2. Transmit packets 3. Detect discrete sequence numbers of received packets in downstream switch 4. Inform upstream switch of loss 5. Report drop in upstream
		Transmitter failure	
	Inter-card drop	Backplane drop	Similar to inter-switch drop detection with programmable switch boards or cards
		Communication drop	
Malfunctioning	ASIC failure	Switch ASIC failure	Advanced switches could detect ASIC failures and produce Syslog
	MMU failure	MMU block / failure	A switch cannot forward packets, which can be detected through active probing

Figure 4: The types and reasons of packet drops, and the methods NetSeer uses to detect them.

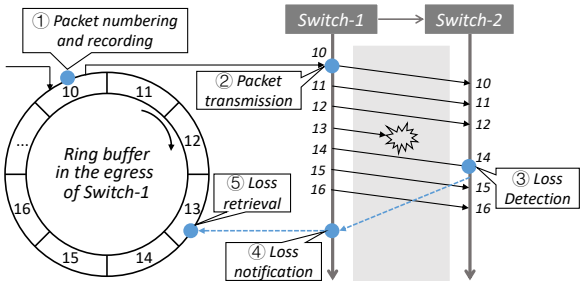


Figure 5: Inter-switch drop/corruption detection.

detection logics into the entire packet processing logic in ASICs to report all packet drops, as shown in Figure 4.

Detecting inter-switch packet drops. Silent packet drops or corruptions due to connector contamination, damaged or bent fiber, decaying transmitters, bad transceivers or shared-component failure are common in production networks [19, 66, 73]. If they happen, usually the upstream switch cannot detect any problems, while the downstream switch can see nothing under silent packet drops and cannot correctly recover flow information under packet corruptions. Therefore, this type of packet drops is hard to locate. As Figure 3 shows, inter-switch/card drops together occupy 18% of all drops that cause NPAs. Nevertheless, 50% of packet drops that take over 180 minutes to locate are caused by them. The average location time for inter-switch/card drops reaches about 161 minutes, which is longer than the others.

The challenge to detect inter-switch packet drops lies in *the lack of direct visibility in the electrical and optical components between the two neighboring switches*. NetSeer leverages the switch programmability and coordination of both the upstream and downstream switches to build a novel detection mechanism for inter-switch packet drops. The key idea is to use a four-byte consecutive packet ID between two neighboring switches to detect packet loss. As shown in Figure 5, packets are sent from Switch-1's egress to Switch-2's ingress. ① Switch-1 inserts a private sequence number, which increases by 1 per packet, into each packet it sends out to Switch-2 (②), and Switch-2 removes the sequence numbers. A ring buffer in Switch-1 caches the 5-tuple (or other flow identifies that can be flexibly defined) and packet IDs of the recent N packets that have been sent to Switch-2. ③ At the downstream side, Switch-2 treats inconsecutive sequence numbers of incoming packets as a sign of packet drops (Switch-2 drops corrupted packets directly at MAC layer). ④ To find the flow information of lost packets, Switch-2 constructs a packet which contains the starting and ending of missing sequence numbers and sends it to Switch-1. To avoid this notification packet from being dropped again on the link, we produce three copies of it in Switch-2 and send them all to Switch-1 via an independent queue in high priority. NetSeer sends redundant notification packets to protect their arrival at the upstream switch while keeping the communication overhead under control. ⑤ Switch-1 looks up its ring buffer for the packets whose sequence numbers fall into the missing interval and reports them as dropped packets. The entire logic runs in PDP at line speed.

NetSeer realizes inter-switch drop detection through *inter-switch coordination*, a pattern that is already followed by several production switch functions such as BFD [28]. Therefore, such coordination can be implemented with acceptable complexity. Regarding overhead, despite the entire detection process is performed in the switch data plane at line rate, NetSeer needs to insert a packet ID field into every single packet. We could leverage existing unused bits in packets such as VLAN tags or IP options to minimize the overhead.

Since current programmable ASICs do not support loops inside a stage, to report multiple consecutive packet drops, we use subsequent packets from Switch-1 to Switch-2 to trigger the lookup of lost packets. For each subsequent packet that arrives at the ring buffer, Switch-1 caches the packet and performs the lookup once, until all lost packets are reported.

Note that limited by the PDP memory and ring buffer size, if too many consecutive packets are dropped, Switch-2 may not timely inform Switch-1, and subsequent packets will replace the dropped packets in the ring buffer. Thus, this mechanism has a certain capacity of detecting consecutive packet drops and might miss some packet drops, which we discuss in §3.7. However, even in the case of ring buffer overriding, NetSeer will not report the wrong packets, since we compare the packet IDs carried by the notification packets with those recorded in the ring buffer before retrieving the packets.

In multi-board (card) switches, we use a similar idea to detect inter-card packet drop, though a trend in the industry is to gradually deprecate this type of switches in data centers.

Congestion, path change and pause detection. For congestion, we measure the queuing delay inside a switch with ingress and egress meta timestamps created for each packet in the data plane, and select packets whose queuing delay exceeds a threshold.

For path change, NetSeer learns incoming flows and remembers the ingress and egress ports of each flow in each switch's data plane. It selects the first packet of a new flow or an old flow whose ports are changed as a path change event packet. To cope with limited hardware resource capacity, we quickly expire old flows and maintain the path information of every single new flow. In this way, we could report the path information of all flows within limited resources, with slightly more flows reported as new ones.

For pause, we design a *queue status detector* in the *ingress* pipeline that matches PFC frames to understand whether a queue is paused or not. NetSeer will check PFC (PAUSE or RESUME) messages to recognize the status of each queue. For each arriving packet, NetSeer looks up the corresponding queue status in ingress, and identifies a packet as a pause event packet if the queue is paused.

3.4 Flow Event Generation & Compression

With event packet detection, NetSeer reduces the monitoring traffic volume by precisely selecting packets that experience network events. However, extreme situations such as severe congestion can result in massive congested or dropped packets. The key idea to solve this problem is to aggregate sequential event packets that belong to one flow into a single flow event. A flow event only contains the flow's five-tuple, the number of packets involved, event type, and event details.

Event packets to flow events. We observe that instead of capturing every event packet, operators only need to capture the *flows* involved in *events*. Therefore, for all events except path change which is flow-level by nature, we define *redundant event packets* as those belonging to the same flow and encountering the same event, and try to eliminate redundancy by aggregating event packets into flow events, and maintaining one counter for each flow event. This could reduce the monitoring traffic volume from $O(\#event\ packets)$ to $O(\#event\ flows)$. Event deduplication should achieve the following two goals. (1) *Zero false-negative:* It should not miss flow events that should be reported, which is crucial for network exoneration and fast NPA location. (2) *Minimal false positives:* It should minimize the duplication of reported flow events.

Algorithm 1: Deduplication based on group caching

```
   Input: Event packet P; Group caching table cache[]
1  Function event_packet_deduplication (P, cache[])
2      index ← calculate_hash(P.flow_info);
3      if cache[index].flow_info is equal to P.flow_info then
4          cache[index].counter ++;
5          if cache[index].counter ≥ cache[index].target then
6              produce_event(cache[index]);
7              cache[index].target ← cache[index].target + C;
8      else
9          cache[index].flow_info ← P.flow_info;
10         cache[index].counter ← 1;
11         cache[index].target ← C; // C is a constant;
12         produce_event(cache[index]);
```

The state-of-the-art data deduplication method is hashing based approaches such as bloom filters [38, 69]. The benefits of a bloom filter are its high memory efficiency and provable worst-case accuracy guarantees. Unfortunately, due to hash collisions, the bloom filter has an unavoidable possibility of false negatives, making it vulnerable to miss event flows. Specifically, when two different flow events collide in the same entry of the bloom filter, the later one will not be reported. Another candidate approach is identifying heavy hitters through Hashpipe [55] or *count-min sketches*, and blocking subsequent packets of the heavy flows. Nevertheless, finding and blocking heavy flows require counter accumulation and the switch control plane to install the exact match rule of heavy flows. For events at fine timescale such as microbursts, the events may have ended before the rules are installed, making this approach helpless.

To realize event deduplication, we design an effective *group caching* mechanism presented in Algorithm 1. For each type of event, we maintain a hash table with each entry recording an exact flow 5-tuple (or other flow identifiers that can be flexibly defined), in order to avoid false negative. We hash each event packet to a specific entry (line 2) and perform a simple comparison. If the packet belongs to the recorded flow, the counter of this entry increases by 1, and the packet will not be reported (lines 3–4). Otherwise, the entry is replaced with the packet's 5-tuple, and the evicted flow as well as the new flow will both be reported (lines 8–12). In this way, the first packet of a flow event will always be reported, and thus false negative is eliminated. Also, an event is reported every time the entry's counter passes a new threshold (lines 5–7). Constant C controls the interval of the counter-report. The evaluation shows that group caching could reduce the overhead by ~95% (§5.2).

Note that NetSeer particularly aggregates packet drops by ACL at the granularity of ACL rule rather than a flow, because most ACL drops are normal, creating massive flow drop events. For ACL aggregation, NetSeer assigns a unique rule ID for each ACL rule and maintains a drop counter for each rule ID. The switch CPU can find the ACL rule corresponding to the ID, and report the original ACL rule and the counter. Note that despite we do not directly report the flow information of dropped packets, the ACL rule itself already contains a match field that describes target flows.

Event information extraction. NetSeer further compresses the monitoring traffic volume by only extracting the necessary information from flow events, which includes flow headers such as 5-tuple, switch-port-queue, and event-specific data such as queuing latency for congestion events and drop reason for packet drop events. 24 bytes is adequate to store any type of network event (§4). Considering that the average packet length in data centers is around 724 bytes [8], this approach can further reduce the traffic by about 97%.

3.5 Circulating Flow Event Batching

After event extraction, flow events are sent to the switch CPU via data plane generated packets. Since each event only occupies 24 bytes and an Ethernet packet length is at least 64 bytes, sending one event with one packet will result in 62.5% overhead and low CPU capacity to process small packets.

To address this challenge, NetSeer proposes to *batch* event packets, i.e. packing more than one event within each packet, to reduce the bandwidth overhead and be friendly for CPUs. Considering the

packet length constraint of Ethernet packets, we recommend batching 50 events together in one packet. However, enabling batching is non-trivial due to the limited stage and memory resource in PDP. A straightforward approach is concatenating events in the switch memory and report when there are enough events. Nevertheless, as the memory bit width of each stage is limited [20], one event cannot be entirely accommodated in one stage, let alone 50.

We propose an innovative circulating event batching mechanism. We design a stack data structure across multiple stages. The length of each stack entry is 24 bytes, i.e. the size of an event. Each incoming event is pushed into the stack for temporal caching. Meanwhile, we leverage the capability of PDP to generate circulating event batching packets (CEBPs) that constantly recirculate within the pipeline via a separate internal port. When a CEBP hits the stack, it pops one event and append the event into its payload. Once the payload length exceeds a threshold or all events have been collected, the CEBP is forwarded to CPU and cloned at the same time with an empty payload to collect latter events. Essentially, we leverage switch circulation capability and internal port bandwidth to compensate for stage memory width limitation without affecting the forwarding performance of normal traffic. Event batching is required to produce adequate CEBPs to batch massive flow events. Evaluation in §5.2 shows that we can satisfy this requirement within PDP capacity. Furthermore, circulating event batching could prolong the event response latency by a few microseconds.

3.6 False Positive Elimination

We define data false positive as repetitive event reports for the same flow event. False-positive happens due to hash collisions in group caching tables. If two flow events are hashed to the same table entry, the later one will replace the former one. If the former one has not ended, it will replace the later one again. As a result, hash collisions can result in multiple initial event reports of a single flow event. False positives not only increase data volume to be transmitted and processed but also mislead network operators as if multiple independent events have happened to the same flow.

To fully eliminate false positive in received events, the switch CPU maintains a hash map data structure and removes duplicated events. However, due to the limited processing power of switch CPU, performing hash calculation takes lots of CPU cycles and could reduce the processing capacity of CPU. Our solution is to enable the switch pipeline to calculate the hash value in advance and attach it to the event. Switch CPU could directly retrieve the value for indexing. Doing so could improve CPU processing capacity by 2.5× (§5.2). We use TCP for reliable transmission of network events from switch CPU to backend storage. Report traffic is also monitored by `NetSeer` similar to normal traffic to ensure event coverage.

3.7 Preconditions of `NetSeer`'s Coverage

`NetSeer`'s effectiveness depends on the correctness of switch hardware and software. As shown in Figure 4, `NetSeer` cannot cover drops due to ASIC or MMU hardware failures. Fortunately, modern switches have self-checking mechanisms to detect such failures and actively raise alerts to operators.

Moreover, resource in programmable switches is limited [20, 40]. Thus, there is a capacity for event detection within each switch. We discuss the capacity in §4 and illustrate why `NetSeer`'s capacity can satisfy most(~90%) real-world situations.

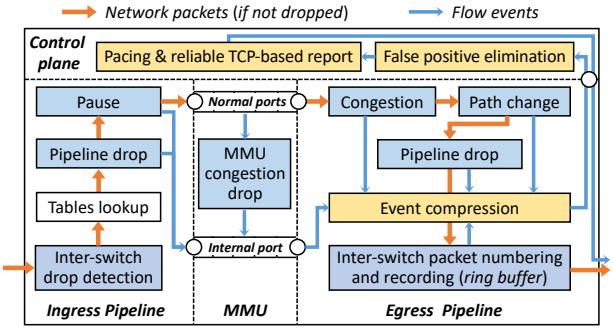


Figure 6: `NetSeer` switch implementation.

Finally, middleboxes such as Firewall and Load Balancer are also potential sources of NPAs. However, middleboxes are implemented on various platforms such as commodity servers [58], programmable hardware devices [40], or dedicated hardware, requiring corresponding customized monitoring mechanisms. Based on the design experience of `NetSeer`, we propose three principles to monitor middleboxes.

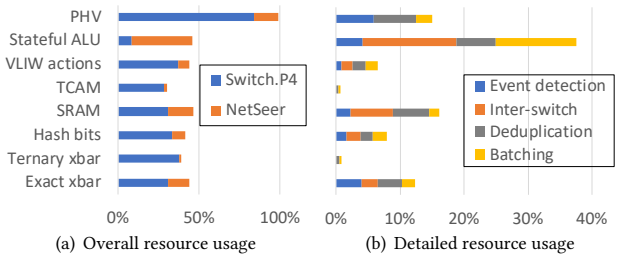
- *Inter-device drop awareness.* Middleboxes' NICs or software stacks should be able to collaborate with network switches to detect inter-device drops to ensure coverage.
- *Event-based anomaly detection.* Middleboxes should be able to detect local events such as packet drop or buffer overflow according to its components and functionality. Detecting events instead of coarse-grained statistics is helpful for failure location with scalability in mind.
- *Reliable report.* Middleboxes should be able to report events reliably to maintain data accuracy.

4 IMPLEMENTATION

We have implemented `NetSeer` on switches with Tofino 32D ASIC and x86 CPU, and Netronome NFP-4000 smartNIC [47], with ~12,000 LoC, as an extension of the regular data plane.

Switch ASIC. Figure 6 presents the implementation of `NetSeer` ASIC logic on programmable switches. `NetSeer` logic can be embedded into original switch programs (like switch.p4 in our experiment) as an extension. The inter-switch drop detection modules are at the beginning of ingress and the ending of egress. We layout the pipeline drop and pause detection modules in ingress, enable congestion drop detection in the MMU and detect congestion, path change, pipeline drop in egress. Among them, flow events from ingress and MMU are redirected to an internal port to avoid affecting normal packet forwarding. All events are processed by the event compression module and finally delivered to the CPU.

Switch CPU. As shown in Figure 6, we have implemented false positive elimination and pacing in switch CPU and used reliable TCP to deliver batched events to the backend. To maximize capacity, we implement the logic using DPDK. However, ports bound by DPDK cannot directly expose IP to the network or answer ARP and ICMP, making it challenging to communicate with the backend over



(a) Overall resource usage (b) Detailed resource usage

Figure 7: Overall and detailed PDP resource usage.

TCP. Our solution is to create a native Linux socket and employ a kernel protocol stack to communicate with the backend. Events are highly aggregated and could be handled easily by the kernel.

Event formats. We extract flow and event information in the switch data plane before sending it to switch CPU. Specifically, we maintain the following information for each type of event.

- *Flow* (13B): <*5-tuple*> for TCP/UDP packets. Flow fields can be flexibly defined and extended according to packet formats.
- *Congestion* (5B): <*egress port, egress queue, queue latency*>.
- *Path change* (2B): <*ingress port, egress port*>.
- *Pause* (2B): <*egress port, egress queue*>
- *Drop* (3B): <*ingress port, egress port, drop code*>. Packet drop reasons are encoded into the *drop code* field.

In addition, we attach a 2-byte counter and a 4-byte pre-calculated hashing value into each event report. Thus, we need ≤24 bytes to report an event for a TCP flow (identified by 5-tuple).

NIC. We implement the inter-switch drop detection and inter-switch packet ordering and recording modules (shown in Figure 6) in the NICs, and store detected events into local logs. Doing so guarantees event coverage on all network components including the edge links and interfaces.

PDP resource occupation. Regarding `NetSeer`'s PDP resource usage, the usage of all resource types except stateful ALU is merely below 20%. As for stateful ALU (40%), the event batcher and the inter-switch packet drop detection contribute to the most usage (28% in total), as they need to frequently conduct stateful operations over cross-packet states.

On NICs, `NetSeer` brings 5% additional usage of memory and on-board memory external to ASIC, and about 33% external memory cache to accelerate access to external memory.

Capacity. Due to the limited hardware resources, `NetSeer`'s capacity for event detection and report is also limited. We summarize the capacity of different events and resource bottleneck below. Moreover, according to statistics from *Alibaba*, we observe that `NetSeer` can ensure full event coverage under most (~90%) situations, and can work together with existing monitoring systems to quickly mitigate NPAs under extreme situations.

- *Inter-switch drop:* limited by the size of the ring buffer. Our detection mechanism can detect 1,000 continuous 1024-byte packet drops with merely 800KB SRAM (§5.2). Existing researches [73] show that 87.33% corrupted links in data centers have a packet corruption ratio of <0.1%. Under such a low ratio, the probability of 1,000 consecutive losses is extremely low. When corruption ratio is high, network operators could closely monitor the counters and statistics of packet corruption and perform actions such as isolating a port under anomaly to quickly mitigate NPAs.
- *MMU Drop:* limited by switch MMU's capability to redirect packets to be dropped (~40Gbps). This means that `NetSeer` could capture every packet drop in MMU if the total drop speed of all ports is ≤40Gbps. We derive per-second MMU drop counters from *Alibaba* and observe that the drop rate at the 99th percentile is around 2.9e-5, which means 186Mbps for a 6.4Tbps switch. This proves that `NetSeer` ensures full MMU drop coverage under 99% situations. Under rare cases where a switch is heavily dropping packets, counters and statistics could timely reveal the problem to operators.
- *Pause, ingress pipeline drop, and MMU drop:* jointly limited by the bandwidth of switch's internal port (100Gbps). This capacity can easily be increased if we use or add more internal or physical ports to transmit these events.
- *All events:* jointly limited by the capacity of PCIe between the pipeline and CPU (18Gbps), and the power of switch CPU (13.4Gbps with 2 cores). Evaluation in §5.2 shows the bandwidth overhead of `NetSeer` is 0.01% of the original traffic. For a 6.4Tbps switch, the produced flow event volume is <1Gbps, which is within the capacity of PCIe and CPU.

We present more details regarding capacity in §5.2.

5 EVALUATION

We evaluate `NetSeer` on a testbed with a 4-ary Fat-Tree topology [59] composed of 10 Tofino 32D switches (3.2T) and 8 servers. Each server has 192 Intel Xeon Platinum 8163 2.5GHz CPU cores, 64GB RAM, and a Netronome NFP-4000 smartNIC (4×25G) [47]. Switches are interconnected with 100G links, while each server is connected to ToR with 4×25G links. From a networking point of view, the testbed can be treated to have 32 servers, each with a 25Gbps uplink to ToR.

We also deploy five existing network monitoring systems for comparison purposes, including SNMP counters [9], packet sampling, Pingmesh [19], EverFlow [72], and NetSight [21] in the testbed. For sampling, we configure the sampling rate as 1:10, 1:100, and 1:1000. We enable Pingmesh to send one round of full-mesh probes every 1 second. For EverFlow, switches mirror SYN and FIN packets to the collector with ERSPAN. Meanwhile, to simulate Everflow's on-demand packet telemetry, we repeatedly choose 1,000 random flows and perform packet telemetry over these flows with a minute time interval. NetSight enables switches to mirror all packets with metadata including forwarding latency and ports, which is very similar to INT postcard mode [31]. All mirrored packets are truncated to 64 bytes. We evaluate `NetSeer` with case studies (§5.1) and performance benchmark (§5.2).

5.1 Real Case Study

To understand `NetSeer`'s effectiveness on troubleshooting NPAs, we study 5 real and representative NPAs and occasional SLA violations of a block storage system from *Alibaba*.

Troubleshooting network incidents. We reproduce 5 real *Alibaba*'s NPA related incidents on our testbed with inferred topology, number of flows, and flow traffic rate during the incidents.

We discuss and confirm the above parameters with an experienced operator from *Alibaba*, and invite him to locate the incident cause with NetSeer on our testbed. We introduce each case and explain why NetSeer can reduce the cause location time by 61%∼99%, as shown in Figure 8(a).

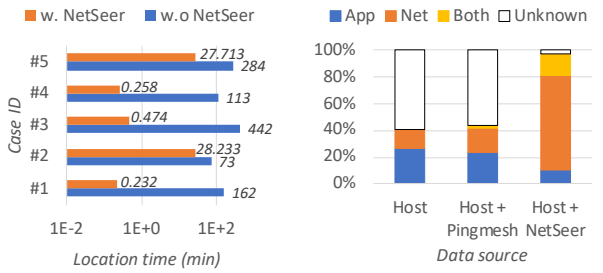
#1) Routing error due to network updates. A software engineering product reported network connectivity issues, which was caused by a problematic network update that installed a wrong routing entry into a core router. However, drop counters and host-based probing within the cluster appeared normal. Operators discovered a few 5-tuples that suffered packet drop, but could not locate the failure due to unknown flow paths. Thus, operators had to review possibly relevant network updates and finally located the faulty update. With NetSeer, once affected flows are found, operators could quickly identify path change events caused by the faulty update and reduce location time from 162 minutes to 14 seconds.

#2) ACL configuration error. A newly created VM in provider *Alibaba*'s cluster could not access the Internet. The actual cause was a misconfigured ACL rule. It took operators 28 minutes to communicate with the client to obtain the affected flows. However, cause location was surprisingly difficult, as existing network monitors cannot correlate flows to problematic statistics, resulting in so many possible causes like routing errors, switch failures, ACL errors, *etc.*. With NetSeer, after acquiring affected flows, we could immediately discover pipeline drops by ACL and cut the location time by 61%.

#3) Silent drop due to parity error. A Redis service received customer complaints about probabilistic request timeout under massive PHP connections, which was caused by random silent drop due to switch memory bit flip. The corrupted entry happened to reside outside the switch detection zone and therefore was not reported through Syslog. Most time (96.6%) was spent on identifying the 5-tuples that encountered failure and their paths. NetSeer can help operators catch pipeline drops due to table lookup miss, identify flows to or from Redis, and locate the problem within 30 seconds.

#4) Congestion due to unexpected volume. Multiple online VMs of some customers could not be visited through ssh or ping. The actual cause was an unexpected volume from another customer that congested a core switch. Statistics revealed that the throughput of the core switch reached its capacity. Operators' first idea was scheduling the large flows to a backup link to recover the connectivity of victims. However, operators spent a long time figuring out which flows to migrate, due to the lack of visibility into flows that caused congestion. With NetSeer, operators could find the flows that contributed the most to congestion by checking MMU congestion drop counters and perform scheduling accordingly.

#5) SSD firmware driver bug. *Alibaba*'s storage performance monitor discovered high latency on a remote SSD cloud disk. Continuous monitoring of the next 27 minutes showed that storage servers in an entire POD suffered from frequent TCP retransmission. The storage service owner immediately suspected that the network was dropping packets. However, network operators only discovered MMU drops in a ToR switch of that POD but were not sure whether dropped packets included storage traffic. They had to dig into switch logs to check whether the ToR encountered any problems. At the 284th minute, the storage service owner learned that SSD buffer could be exhausted and start dropping requests under mixed heavy read-write workloads due to a driver bug. Finally, the problematic storage servers were isolated, and the performance recovered. Meanwhile, operators were unable to exonerate the physical network due to the incapability of correlating flows to events. With NetSeer, operators could quickly find how many storage packets are dropped by which switches and accelerate the debugging process by clearly stating network responsibility within 42 seconds.



(a) Debugging network incidents (b) Debugging SLA violations

Figure 8: Real case study with NetSeer.

Summary. From our experience of analyzing and debugging the above cases, we observe that existing monitoring systems fail to gather necessary information for NPA location including victim 5-tuple, flow paths, and explicit correlations between flows and events. In comparison, once we obtain trivial clues such as src-IP, flow, or suspicious switch ID, NetSeer helps operators quickly find related events in seconds.

Troubleshooting occasional SLA violations. A block storage application is one of the most fundamental products of *Alibaba*. Customer VMs communicate with storage backend through RPC calls. The application has comprehensive end-host performance monitoring mechanisms that collect metrics with a 15s interval. Occasionally, RPC read or write latency is longer than expected, resulting in bad user experience. Slow RPC could be caused by (1) slow application processing due to software bugs or bursty traffic, or (2) physical network faults such as congestion or packet drops.

We obtain *Alibaba*'s production storage application and real-world traces of storage visits from an advertisement product and run it on our testbed for 6 hours. We collect monitoring data during that period and try to contribute detected slow RPC calls to the application or the network. Unfortunately, existing monitoring systems fail to explain a large portion of slow RPCs. First, metrics collected by host monitors are too coarse-grained with a 15s interval and can only explain 40.8% NPAs, as shown in Figure 8(b). Second, existing network monitors fail to capture network faults on-site with real-time pings or hindsight probes, and can only explain 44% NPAs.With NetSeer's help, we can tell whether and how much the network is responsible for each slow RPC, and explain much more (97%) NPAs, including those whose causes were *unknown* with existing monitors, and some NPAs that were considered as application-induced but were partially caused by the network as well (i.e. the *Both* legend in Figure 8(b)).

Summary. NetSeer can help network operators claim network innocence or quickly locate NPA causes. Without NetSeer, applications

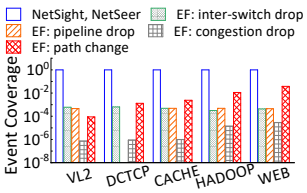


Figure 9: Event coverage ratios.

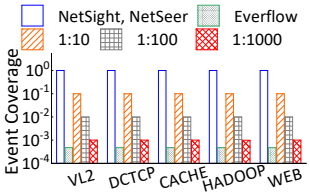


Figure 10: Congestion event coverage.

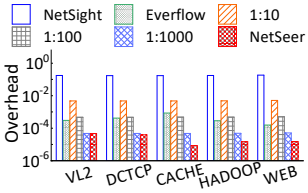


Figure 11: Overall bandwidth overhead.

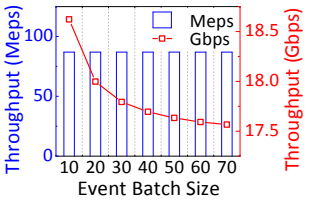


Figure 12: Event batching capacity.

naturally blame the network for connectivity issues or NPAs, and stop looking for other potential causes before network response. By comprehensively capturing events, NetSeer can clearly identify network responsibility and accelerate the NPA debugging process.

5.2 Performance Benchmark

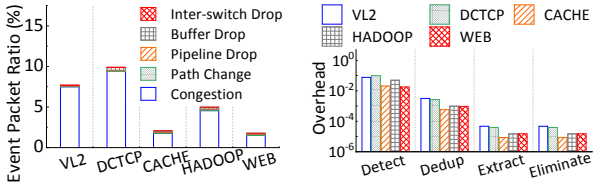
We evaluate NetSeer's coverage, accuracy, scalability, and capacity on our testbed. We start 8 clients to communicate with 32 servers. Each client has 100K flows and a fan-in ratio of 4. We produce traces based on five real-world traffic distributions including *DCTCP* [4], *VL2* [18], *CACHE*, *HADOOP*, and *WEB* [52]. We set the average link utilization as 70% to produce enough pressure. In the experiments, congestion and MMU drop are naturally produced, while we manually inject inter-switch drop, pipeline drop, and path change events. [1]

NetSeer ensures full event coverage and accuracy. We show the coverage ratios of path change, MMU drop, inter-switch drop, and pipeline drop in Figure 9, and congestion in Figure 10. Only NetSeer and NetSight have full event coverage, while other solutions merely cover <10% events. Specifically, sampling cannot capture packet drops. EverFlow's coverage is <1% for all event types. Pingmesh can detect the existence of 0.02% congestions, but cannot detect involved flows.

NetSight performs per-packet telemetry and can capture all events. We compare events derived from NetSight with events collected by NetSeer and find that when the data rate is within hardware capacity, NetSeer events are accurate with zero FP/FN.

NetSeer is scalable with 0.01% bandwidth overhead. Figure 11 shows that NetSeer only incurs <0.01% bandwidth overhead. For a 6.4Tbps switch, the overhead is at most 640Mbps or ~4M events per second (eps), which is comparable to EverFlow and 1:1000 sampling. Meanwhile, NetSight suffers from ~18% bandwidth overhead, which is three orders of magnitude higher than NetSeer. For processing overhead, our testbed produces 32 × 25Gbps × 70% / 1KB (average packet size) = 70Mpps traffic. For NetSight, one CPU core can process postcards at 240Kpps [21]. This means that we need up to 70Mpps × 5 hops / 240Kpps = 1458 cores, while all servers in our testbed have 1536 cores. In comparison, NetSeer achieves full coverage with merely 2 × 10 = 20 switch CPU cores.

Next, we evaluate how each step in NetSeer can reduce bandwidth overhead. We show in Figure 13(a) that the overall event



(a) Event packet ratio (b) NetSeer per-step overhead

Figure 13: Per-step bandwidth overhead.

packet ratio is <10% in our experiments, which brings >90% overhead reduction. Then, as presented in Figure 13(b), NetSeer conducts event packet deduplication that decreases overhead by 95%. Next, NetSeer extracts event information from event packets and further reduces overhead by 98%, resulting in an overall <0.01% overhead. Finally, false-positive elimination in switch CPU brings less than 7% overhead reduction.

To further understand the scalability of NetSeer, we calculate the monitoring traffic volume as well as the processing overhead of NetSeer according to the configuration of *Alibaba* production data centers. For a normal 3-tier data center network, connecting 10,000 servers requires approximately 400 switches, which produce a maximum of 400×640Mbps=256Gbps monitoring traffic at most. Processing such traffic requires 3 servers with 100Gbps NICs, which implies a 0.03% processing overhead.

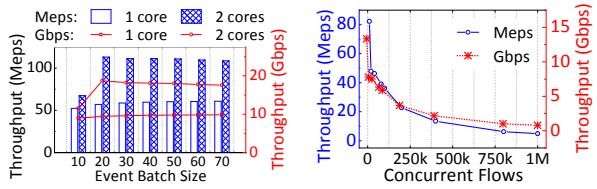
NetSeer's capacity within hardware resources. Event data will pass several modules including event detection, compression, PCIe, and switch CPU. We evaluate the processing capacity of each module within the switch resources and the maximal ratio of network events that can be detected.

Module capacity. Event extraction, deduplication, and batching compress data collectively in the switch data plane. The first two steps can work in line rate. However, batching performance is limited by the bandwidth of internal ports. Figure 12 shows that batching can report event messages at about 86Meps or 17.7Gbps, which is enough for the maximum possible event volume (~4Meps) of a 6.4Tbps switch.

To evaluate the capacity of PCIe channel between pipeline and switch CPU, we vary the batch size and #CPU cores for packet processing. Figure 14(a) shows that when the batch size is ≥20, the PCIe channel works at 9.5Gbps or 57Meps with 1 CPU core, and 18Gbps or 110Meps with 2 CPU cores.

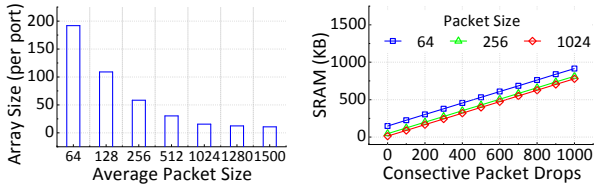
Switch CPU's capacity varies with the number of flows and packet rates. We allocate 2 CPU cores and set the batch size to 50. As shown in Figure 14(b), two cores can process events at 82Meps

[1] As our SmartNIC does not support PFC, we do not evaluate pauses. Instead, we use congestion and MMU drops to assess how NetSeer handles congestions.



(a) PCIe capacity. (b) CPU capacity.

Figure 14: Capacity of PCIe and switch CPU.



(a) Minimal array size per port (b) Overall SRAM consumption

Figure 15: Capacity of inter-switch drop detection.

when there are 1K concurrent flows and can work at 4.5Meps when there are 1M concurrent flows, which can handle the maximum possible events in a switch. Moreover, offloading hash calculation to PDP can save 71.4% CPU cycles and improve CPU processing capacity by 2.5×.

Event detection capacity. For inter-switch drops, Figure 15(a) shows that the ring buffer should have ≥25 slots to retrieve at least one 1024-byte dropped packet. As shown in Figure 15(b), with 800KB SRAM in total, `NetSeer` can detect 1,000 consecutive drops of 1024B packets for each port of a switch with 64×100Gbps ports.

As discussed in §4, We can detect all path change and congestion when forwarding at the line rate. Other events have to redirect event packets and are limited by the bandwidth of internal ports. The capacity is ~40Gbps for MMU drop, and 100Gbps for ingress pipeline drop, MMU drop, and pause.

6 RELATED WORK

Network monitoring has been widely and continuously researched over decades, but none of the existing solutions are designed for fast NPA cause location.

Host-based network monitoring. Host-based monitoring systems either send probes into the network and speculate network status according to responses [2, 7, 13, 19, 41, 48, 50, 62], or analyze transport layer or other I/O performance to inference potential network failures [5, 6, 10, 16, 29, 44, 45, 53, 67, 70]. Some works exploit moderate support from switches to execute measurement instructions [26, 36] or insert path information [60, 61]. These systems can infer the existence of NPAs from the perspective of end hosts. However, they cannot precisely locate causes of all NPAs in the network with data solely from end hosts, especially small-scale packet drops. Thus, they suffer from compromised coverage and accuracy when used to detect network events.

Network statistics collection. These systems obtain sampled or aggregated data or counters directly from network switches [9, 11, 54]. Operators could detect network failures by analyzing rules or counters [14, 17, 35, 39, 64, 66, 73]. Recent works adopt hash tables or sketches for network measurement in hardware or software switches for memory efficiency and provable accuracy [3, 22, 23, 37, 38, 42, 43, 68, 69]. Network operators could deeply understand the network status and detect severe problems such as switch offline or DDoS attacks with these systems. However, with the collected coarse-grained statistics, operators have difficulty discovering events at a fine timescale that could result in NPAs, or correlating flows to detected network events with high confidence, making them incapable of fast NPA mitigation.

Packet telemetry. ERSPAN [12, 49, 51, 72] and INT [21, 31] suffer from granularity-cost trade-off. Thus, they often work as on-demand debugging tools and cannot fully capture transient or random events on-site. NetSight [21] proposes a potential optimization by only encoding changes between successive postcards in switches. The authors estimate that the traffic overhead after this improvement is 7% (14% to further distinguish between intra- and inter-switch drops like `NetSeer`). In comparison, `NetSeer` has the same event coverage with NetSight but only incurs <0.01% bandwidth overhead, which is a huge improvement in scalability.

Programmable switch assisted measurement. The advent of programmable switches have enabled a series of researches to perform (1) flow measurement [24, 32, 56, 57], (2) specific event monitoring including microbursts [27], the existence of packet loss without any details regarding drop reasons [33], or heavy-hitters [55], and (3) monitoring frameworks that support flexible queries based on high-level primitives [20, 46, 63]. Especially, PacketScope [63] allows querying the lifecycle of every packet inside the programmable switch with a Spark-like dataflow language. However, these systems are not designed with the goal of comprehensively monitoring network events, such as inter-switch packet drops that are time-consuming to locate with existing tools. Thus, they cannot quickly mitigate NPAs with high event coverage and confidence.

7 CONCLUSION

In this paper, we present `NetSeer`, an in-data-plane network monitor that precisely catches flow-level data-plane events with advanced programmable data planes to facilitate the troubleshooting of NPAs. We show that `NetSeer` can achieve nearly-full flow event coverage, good scalability with network sizes, and high accuracy of flow event data with novel designs that fully explore the flexibility and performance of programmable data planes. With case studies and experiments, we reveal `NetSeer`'s powerful ability to accelerate the mitigations of various NPAs.

This work does not raise any ethical issues.

ACKNOWLEDGEMENT

We thank our shepherd Prof. Arpit Gupta, and the anonymous reviewers for their constructive comments. Prof. Mingwei Xu is the corresponding author. This work is supported by Alibaba Group through Alibaba Innovative Research (AIR) Program. Yu Zhou and Mingwei Xu are supported in part by the National Natural Science Foundation of China (61625203 and 61832013) and National Key R&D Program of China (2017YFB0801701).

REFERENCES

[1] 2011. IEEE Standard for Local and metropolitan area networks–Media Access Control (MAC) Bridges and Virtual Bridged Local Area Networks–Amendment 17: Priority-based Flow Control. *IEEE Std 802.1Qbb-2011 (Amendment to IEEE Std 802.1Q-2011 as amended by IEEE Std 802.1Qbe-2011 and IEEE Std 802.1Qbc-2011)* (2011).

[2] Aijay Adams, Petr Lapukhov, and J Hongyi Zeng. 2016. NetNORAD: Troubleshooting networks via end-to-end probing. *Facebook White Paper* (2016).

[3] Omid Alipourfard, Masoud Moshref, Yang Zhou, Tong Yang, and Minlan Yu. 2018. A comparison of performance and accuracy of measurement algorithms in software. In *SOSR*.

[4] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data Center TCP (DCTCP). In *SIGCOMM*.

[5] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 2018. 007: Democratically Finding the Cause of Packet Drops. In *NSDI*.

[6] Behnaz Arzani, Selim Ciraci, Boon Thau Loo, Assaf Schuster, and Geoff Outhred. 2016. Taking the Blame Game out of Data Centers Operations with NetPoirot. In *SIGCOMM*.

[7] Francois Aubry, David Lebrun, Stefano Vissicchio, Minh Thanh Khong, Yves Deville, and Olivier Bonaventure. 2016. SCMon: Leveraging segment routing to improve network monitoring. In *INFOCOM*.

[8] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. 2010. Understanding data center traffic characteristics. *SIGCOMM CCR* (2010).

[9] Jeffrey D Case, Mark Fedor, Martin L Schoffstall, and James Davin. 1990. *Simple network management protocol (SNMP).* Technical Report.

[10] David R Choffnes, Fabián E Bustamante, and Zihui Ge. 2010. Crowdsourcing service-level network event monitoring. In *SIGCOMM*.

[11] B Claise, G Sadasivan, V Valluri, and M Djernaes. 2007. Rfc 3954: Cisco systems netflow services export version 9 (2004). *Retreived online: http://www. ietf. org/rfc/rfc3954. txt* (2007).

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

[13] Amogh Dhamdhere, Renata Teixeira, Constantine Dovrolis, and Christophe Diot. 2007. NetDiagnoser: Troubleshooting network unreachabilities using end-to-end probes and routing data. In *CoNEXT*.

[14] Brian Eriksson, Paul Barford, Rhys Bowden, Nick Duffield, Joel Sommers, and Matthew Roughan. 2010. BasisDetect: A model-based network event detection framework. In *IMC*.

[15] Peter X Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network requirements for resource disaggregation. In *OSDI*.

[16] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. 2019. {SIMON}: A Simple and Scalable Method for Sensing, Inference and Measurement in Data Center Networks. In *NSDI*.

[17] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. 2011. Understanding network failures in data centers: measurement, analysis, and implications. In *SIGCOMM*.

[18] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. 2009. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM*.

[19] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. 2015. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *SIGCOMM*.

[20] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: query-driven streaming network telemetry. In *SIGCOMM*.

[21] 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 *NSDI*.

[22] Qun Huang, Xin Jin, Patrick PC Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. 2017. Sketchvisor: Robust network measurement for software packet processing. In *SIGCOMM*.

[23] Qun Huang, Patrick P. C. Lee, and Yungang Bao. 2018. Sketchlearn: Relieving User Burdens in Approximate Measurement with Automated Statistical Inference. In *SIGCOMM*.

[24] Nikita Ivkin, Zhuolong Yu, Vladimir Braverman, and Xin Jin. 2019. Qpipe: Quantiles sketch fully in the data plane. In *CoNEXT*.

[25] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. 2015. Silo: Predictable message latency in the cloud. In *SIGCOMM*.

[26] 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. In *SIGCOMM*.

[27] Raj Joshi, Ting Qu, Mun Choon Chan, Ben Leong, and Boon Thau Loo. 2018. BurstRadar: Practical real-time microburst monitoring for datacenter networks. In *APsys*.

[28] Dave Katz and David Ward. 2010. Bidirectional Forwarding Detection (BFD). RFC 5880. (June 2010). https://doi.org/10.17487/RFC5880

[29] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. 2019. Confluo: Distributed monitoring and diagnosis stack for high-speed networks. In *NSDI*.

[30] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. 2015. In-band network telemetry via programmable dataplanes. In *SIGCOMM Posters and Demos*.

[31] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. 2015. In-band network telemetry via programmable dataplanes. In *SIGCOMM industrial demo*.

[32] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. FlowRadar: A Better NetFlow for Data Centers. In *NSDI*.

[33] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. LossRadar: Fast Detection of Lost Packets in Data Center Networks. In *CoNEXT*.

[34] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. 2019. HPCC: high precision congestion control. In *SIGCOMM*.

[35] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. 2013. F10: A fault-tolerant engineered network. In *NSDI*.

[36] Xuemei Liu, Meral Shirazipour, Minlan Yu, and Ying Zhang. 2016. MOZART: Temporal coordination of measurement. In *SOSR*.

[37] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. 2019. Nitrosketch: Robust and general sketch-based monitoring in software switches. In *SIGCOMM*.

[38] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *SIGCOMM*.

[39] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P Brighten Godfrey, and Samuel Talmadge King. 2011. Debugging the data plane with anteater. In *SIGCOMM*.

[40] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *SIGCOMM*.

[41] Ivan Morandi, Francesco Bronzino, Renata Teixeira, and Srikanth Sundaresan. 2019. Service traceroute: tracing paths of application flows. In *PAM*.

[42] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. 2014. DREAM: dynamic resource allocation for software-defined measurement. In *SIGCOMM*.

[43] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. 2015. Scream: Sketch resource allocation for software-defined measurement. In *CoNEXT*.

[44] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. 2016. Trumpet: Timely and Precise Triggers in Data Centers. In *SIGCOMM*.

[45] Radhika Niranjan Mysore, Ratul Mahajan, Amin Vahdat, and George Varghese. 2018. Gestalt: Fast, Unified Fault Localization for Networked Systems. In *ATC*.

[46] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. 2017. Language-Directed Hardware Design for Network Performance Monitoring. In *SIGCOMM*.

[47] Netronome. 2019. Netronome Flow Processor. Website. (June 2019). https://netronome.com/product/nfp-6xxx/.

[48] Hung X Nguyen and Matthew Roughan. 2012. Rigorous statistical analysis of internet loss measurements. *ToN* (2012).

[49] Pavlos Nikolopoulos, Christos Pappas, Katerina Argyraki, and Adrian Perrig. 2019. Retroactive Packet Sampling for Traffic Receipts. *SIGMETRICS* (2019).

[50] Yanghua Peng, Ji Yang, Chuan Wu, Chuanxiong Guo, Chengchen Hu, and Zongpeng Li. 2018. deTector: a topology-aware monitoring system for data center networks. In *ATC*.

[51] 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. In *SIGCOMM*.

[52] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. 2015. Inside the Social Network's (Datacenter) Network. In *SIGCOMM*.

[53] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, and Alex C Snoeren. 2017. Passive realtime datacenter fault detection and localization. In *NSDI*.

[54] Vyas Sekar, Michael K Reiter, Walter Willinger, Hui Zhang, Ramana Rao Kompella, and David G Andersen. 2008. cSamp: A System for Network-Wide Flow Monitoring. In *NSDI*. USENIX.

[55] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S Muthukrishnan, and Jennifer Rexford. 2017. Heavy-hitter detection entirely in the data plane. In *SOSR*.

[56] John Sonchack, Adam J Aviv, Eric Keller, and Jonathan M Smith. 2018. Turboflow: Information rich flow record generation on commodity switches. In *Eurosys*.

[57] 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. In *ATC*.

[58] Chen Sun, Jun Bi, Zhilong Zheng, Heng Yu, and Hongxin Hu. 2017. Nfp: Enabling network function parallelism in nfv. In *SIGCOMM*.
[59] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. 2015. Cherrypick: Tracing packet trajectory in software-defined datacenter networks. In *SOSR*.
[60] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. 2016. Simplifying Datacenter Network Debugging with PathDump.. In *OSDI*.
[61] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. 2018. Distributed Network Monitoring and Debugging with SwitchPointer. In *NSDI*.
[62] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. 2019. NetBouncer: Active Device and Link Failure Localization in Data Center Networks. In *NSDI*.
[63] Ross Teixeira, Rob Harrison, Arpit Gupta, and Jennifer Rexford. 2020. Packetscope: Monitoring the packet lifecycle inside a switch. In *SOSR*.
[64] Frank Uyeda, Luca Foschini, Fred Baker, Subhash Suri, and George Varghese. 2011. Efficiently Measuring Bandwidth at All Time Scales.. In *NSDI*.
[65] Mea Wang, Baochun Li, and Zongpeng Li. 2004. sFlow: Towards resource-efficient and agile service federation in service overlay networks. In *ICDCS*.
[66] Xin Wu, Daniel Turner, Chao-Chih Chen, David A Maltz, Xiaowei Yang, Lihua Yuan, and Ming Zhang. 2012. NetPilot: automating datacenter network failure mitigation. In *SIGCOMM*.
[67] Yang Wu, Ang Chen, and Linh Thi Xuan Phan. 2019. Zeno: diagnosing performance problems with temporal provenance. In *NSDI*.
[68] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. 2018. Elastic Sketch: Adaptive and Fast Network-wide Measurements. In *SIGCOMM*.
[69] Minlan Yu, Lavanya Jose, and Rui Miao. 2013. Software Defined Traffic Measurement with OpenSketch. In *NSDI*.
[70] Yifei Yuan, Dong Lin, Ankit Mishra, Sajal Marwaha, Rajeev Alur, and Boon Thau Loo. 2010. Quantitative network monitoring with NetQRE. In *SIGCOMM*.
[71] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion control for large-scale RDMA deployments. *SIGCOMM* (2015).
[72] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. 2015. Packet-level telemetry in large datacenter networks. In *SIGCOMM*.
[73] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas Anderson. 2017. Understanding and mitigating packet corruption in data center networks. In *SIGCOMM*.