This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Buffer-Based High-Coverage and Low-Overhead Request Event Monitoring in the Cloud

Kaihui Gao, Chen Sun, Shuai Wang, Dan Li, *Senior Member, IEEE*, Yu Zhou, Hongqiang Harry Liu, Lingjun Zhu, Ming Zhang, Xiang Deng, Cheng Zhou, and Lu Lu

***Abstract*—Request latency directly affects the performance of modern cloud applications. Due to various causes in hosts and networks, requests can suffer from request latency anomalies (RLAs), which may violate the Service-Level Agreement. However, existing performance monitoring tools have incomplete coverage and inconsistent semantics for monitoring requests and cannot accurately diagnose RLAs. This paper presents BufScope, a high-coverage and low-overhead request event monitoring system, which monitors *buffers* to capture most RLA-related abnormal events with consistent request-level semantics in the end-to-end datapath of request. First, BufScope models the datapath of request as a buffer chain and defines events based on three properties of buffers, so as to *end-to-end monitor* the root causes of RLA. Then, to achieve *consistent semantics* for captured events, BufScope designs a request-level semantics injection mechanism to make events captured in networks have the victim requests' ID. Finally, BufScope offloads the semantics operations and event collection in software to SmartNICs for *low CPU overhead*. We have implemented BufScope on commodity SmartNICs and programmable switches. Evaluation results show that BufScope can diagnose 98% RLAs with <0.08% network bandwidth overhead and 0.6% application throughput decline.**

***Index Terms*—Performance monitoring, buffer, programmable switch.**

I. Introduction

MODERN cloud applications are embracing Cloud Native [2] by deconstructing monolithic applications into microservices. Application-layer requests (*e.g.,* RPC, HTTP, and RESTful requests) are widely used for communication between microservices and have become a fundamental component in the cloud [3].

The request latency is the total elapsed time across an end-to-end request datapath, including the application, the end-host network stack, and the underlying network. Since request latency directly affects the performance of many distributed applications [4], it has become a crucial concern [5], [6], [7] for cloud providers. Besides, request-level information is the tie between the tenants and the cloud providers. For instance, when a request (*e.g.,* search, storage I/O) encounters a surge of latency, the tenant will provide the operators with the request-level descriptive information to diagnose the anomaly [8].

Request latency anomalies (RLAs), which cause long-tailed request latency, are common in the cloud. According to the data of a block storage cluster with over 40,000 servers from a prominent global cloud provider *Alibaba Cloud*, we observe that across all the 440 million RPCs in one hour, 0.01% of them (44K) suffer from a latency of $>$ 100ms, which violates the Service-Level Agreement (SLA) of the storage service [1]. Cloud providers need to diagnose RLAs to explain SLA violations accurately. Otherwise, revenue loss will be caused [9].

However, accurate diagnosis of RLAs is challenging, because it requires high coverage for request-level abnormal events (*i.e., request events*). Specifically, cloud providers must be able to capture abnormal events in the *end-to-end* datapath of all requests, *e.g.,* data pause, congestion, drop, which directly trigger RLAs. Moreover, the cloud provider needs to correlate all events experienced by the same request. To this end, all the captured events should be mapped to *request-level semantics* (*e.g.,* RPC ID) rather than the flow- or packet-level. In practice, it is non-trivial to extract the request-level semantics from flow- or packet-level data.

Unfortunately, existing performance or latency monitoring tools are far from satisfying the preceding requirements for RLA diagnosis. Specifically, though distributed application performance tracing tools [10], [11], [12], [13], [14], [15], [16], [17] can provide request-level timing data, high CPU overhead prevents them from monitoring all requests. Besides, they cannot capture the request events below the application layer. Network performance monitoring tools [9], [18], [19], [20], [21], [22], [23], [24], [25], [26] can capture flow-level events that happen in the network stack or the underlying networks, such as delayed ACK, packet drop, and path change. However, the captured events have no request-level information, thus the events captured in applications and networks cannot be correlated.

Manuscript received 23 April 2022; revised 12 October 2022; accepted 15 November 2022; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor D. Han. This work was supported in part by the Key-Area Research and Development Program of Guangdong Province under Grant 2021B0101400001; in part by the National Natural Science Foundation of China under Grant U21B2022; in part by Tsinghua University-China Mobile Communications Group Company Ltd., Joint Institute; and in part by Tsinghua University (Department of Computer Science and Technology)-Siemens Ltd., China Joint Research Center for Industrial Intelligence and Internet of Things. Part of this work was published in USENIX NSDI 2022 [1]. In this journal article, the authors add the SmartNIC offloaded event collection, new design for semantics injection, and more experiments. *(Corresponding author: Dan Li.)*

Kaihui Gao and Dan Li are with the Department of Computer Science and Technology, Tsinghua University, Beijing 100190, China (e-mail: kaihuigao@163.com; tolidan@tsinghua.edu.cn).

Chen Sun, Yu Zhou, Hongqiang Harry Liu, Lingjun Zhu, and Ming Zhang are with Alibaba Group, Hangzhou 310052, China.

Shuai Wang is with the Zhongguancun Laboratory, Beijing 100081, China.

Xiang Deng is with Siemens Ltd., Beijing 100016, China.

Cheng Zhou and Lu Lu are with China Mobile Research Institute, Beijing 100053, China.

Digital Object Identifier 10.1109/TNET.2022.3224610

1558-2566 © 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Tsinghua University. Downloaded on January 15,2023 at 04:54:38 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Since these existing performance monitoring tools have incomplete coverage and inconsistent semantics, the cloud providers often combine multiple tools when diagnosing RLAs and get into trouble. For instance, based on the RLA information (request-level) reported by the tracing tool, the application owners or tenants often naturally question the server and network team [9], [25]. However, due to the mismatch of monitoring semantics, these teams have to correlate the events (flow- or packet-level) obtained from their monitoring tools with the RLA ticket through coarse-grained time-correlation methods [25], which is not only inefficient but also inaccurate (§II-A).

The fundamental reason why existing tools fail to achieve the high coverage is that the traditional data plane in datacenter networks is a black box [9]. The request events which happen in the data plane cannot be detected and parsed as flexibly as those in the host software. Consequently, network monitoring tools typically capture accessible and coarse-grained flow- or packet-level events. They are making it difficult to end-to-end monitor RLAs with consistent request-level semantics. Fortunately, recent advances in the commodity programmable data plane provide a new foundation to improve the situation.

This paper presents BufScope, a high-coverage and low-overhead request event monitoring system. The main idea of BufScope is to translate most RLAs to *buffer*-related abnormal events, monitor all buffers in the request's end-to-end datapath, and capture all buffer-related abnormal events with consistent request-level semantics. Specifically, BufScope achieves *end-to-end monitoring*, *consistent request-level semantics*, and *low overhead* through three key designs.

(i) Buffer event modeling. The primary purpose of buffer is to handle the speed mismatch between upstream and downstream logic. If upstream or downstream logic has an anomaly, the buffer will reveal the corresponding abnormal events, such as queue buildup, pause, and out-of-order [27]. Based on the operational experience in *Alibaba Cloud*, we observe that most (>90%) RLAs reveal abnormal events in buffers. The remaining (<10%) RLAs come from NIC flapping, link corruption, bugs, *etc.*, which are very difficult and inefficient to cover. For uniform modeling, in this work we only cover RLAs with buffer events. Thus, BufScope models the end-to-end datapath of request as a buffer chain, including the application, network stack, NIC, and switch, and monitors RLA-related abnormal events that happen in all the buffers.

However, different buffers may have different RLA-related abnormal events, and defining all events for various buffers is challenging. For example, in lossy Ethernet, packets may be dropped before entering the buffer. While in lossless Ethernet, the upstream switch will pause packet forwarding to avoid the packet drop in the downstream switch. In response, BufScope uniformly classifies all buffers in both hosts and networks according to three properties: priority awareness, order sensitivity, and enqueue feature. Based on these properties, BufScope defines a complete buffer event library: packet drop, congestion, pause, out-of-order, and priority contention. Then, operators can monitor the corresponding events for one buffer based on its properties.

(ii) Request-level semantics injection. The lack of request-level semantics in the network is because the request header may not exist in the packet payload; even if it exists, its location in the payload is not fixed. This causes the commodity programmable switch to fail to extract the request identifiers when generating abnormal events. In response, BufScope designs a concise semantics injection mechanism, which just inserts the offset of the first request header at the end of the packet header. Then, based on the location-specific information, programmable switches can iteratively parse all request identifiers in a packet.

(iii) SmartNIC offloading. The semantics injection and event collection in software consume too many CPU resources and degrade application performance [11]. According to our experiments, recording latency data for all requests reduces application throughput by up to 45%. To save the host CPU, BufScope offloads the event collection and semantics injection tasks into SmartNICs. However, it is not easy to offload event collection, transmitting events directly to the NIC also consumes CPU. In response, we transfer events based on the idea of resource reuse, *i.e.,* we attach events into outgoing packets, to carry them into the SmartNIC. Finally, SmartNIC decodes and strips the events from packets.

We have integrated BufScope in an open-source RPC framework and *Alibaba Cloud*'s storage application with Broadcom SmartNICs and Barefoot Tofino switches. Testbed-based evaluation shows that BufScope can diagnose 98% RLAs (64% for the combined solution of existing state-of-the-art monitoring tools [9], [16], [22]) with <0.08% network bandwidth overhead (>4% for the baseline) and 0.6% application throughput decline (4.3% for the baseline).

II. Background and Motivation

In this section, we firstly use the representative experiences of *Alibaba Cloud* to demonstrate the current status of RLAs in production. Then, we analyze the limitations of existing monitoring tools, and motivate our proposal.

A. RLAs in the Cloud

Performance anomalies to a request usually involve two aspects: connectivity loss and RLA. The former means that the client loses connection to the remote server, due to hardware failure, a program corruption, or network outage. Such anomalies are more severe, exposing more clear and detectable signals. The latter anomaly means that, even though the request can be completed, its latency is larger than expected, violating the SLA. The RLA is easy to happen but very difficult to mitigate, because RLAs are often caused by random abnormal events in a shorter time scale, leaving fewer fingerprints for monitoring and locating. These abnormal events include polling hang in hosts, congestion in networks [9], [18], [25], *etc.*

To understand the impact of RLA on SLA, we experimented with *Alibaba Cloud*'s storage application, which utilizes the RPC framework, user-level network stack, and run-to-completion (RTC) model. One front-end server continuously performs 4KB read operations from the back-end storage via

Authorized licensed use limited to: Tsinghua University. Downloaded on January 15,2023 at 04:54:38 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

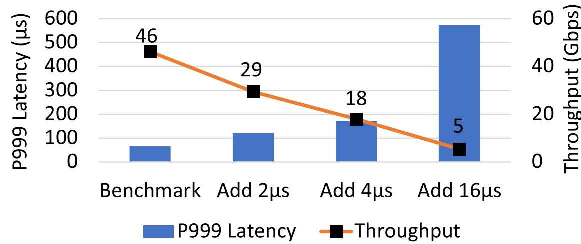


Fig. 1. Impact of RLAs on application performance.

RPCs. We simulate RLAs by adding μs-level delay in RPC processing logic and evaluate its impact on the end-to-end latency and throughput of the application.

As shown in Figure 1, a $2\mu s$ jitter in every single RPC could increase about $50\mu s$ for end-to-end tail latency, and reduce the overall throughput by up to 36.9%, which are even worse under longer delays. The reason why RLAs seriously affect the application performance is that once the logic processes RPCs slower than the NIC bandwidth (50Gbps), then the TCP receiver buffer gets filled up. Meanwhile, there is a large amount of inflight traffic in the network, and this traffic will jam the NIC buffer before TCP flow control takes effect, resulting in packet loss. The network stack needs to retransmit packets, which deteriorates the performance.

Such a performance degradation would violate the SLA of the cloud service, which needs to be explained in practice. However, cloud providers often encounter difficulties in RLA diagnosis. We present two representative real cases to illustrate this.

Case-1: Is the network congestion causing the RLA? A tenant observed a consistently low transmission rate between two VMs. The tenant naturally questioned the network because network congestion could slow down the sending rate. Then, network operators retrieved the switch statistics about queuing and drop. Data showed that the network utilization remained low, and no packets were lost during that period. They have to reproduce the case using the VM's traces. However, the real cause is the limited TCP receive buffer in the host, probably due to a CPU polling hang. It is hard for the network operators to claim their innocence unless they can monitor the packets end-to-end.

Case-2: Is the cause in the network or end-host? A tenant reported a weird latency glitch (100s of ms) of a read request. To diagnose the RLA, the storage application owners first checked the trace obtained by their tracing tool and found that the interval between sending and receiving the request is abnormal. Then, the application owners passed the ticket to the server and network teams. Based on the timestamp in the ticket, the server team operators look for the *second-level* logs of the kernel, CPU, *etc.* The network team operator queried the *flow-level* monitoring system to see if there was any packet loss or network failure in the request flow. Even if these queries have results, neither team is confident enough to claim innocence due to the mismatch in monitoring granularity between these teams.

The two examples above show that cloud providers need an end-to-end monitoring tool with consistent semantics to improve the efficiency of interpreting SLA violations.

B. Limitations of Existing Monitoring Tools

However, existing monitoring tools in both academia and industry have three limitations to meet the requirements.

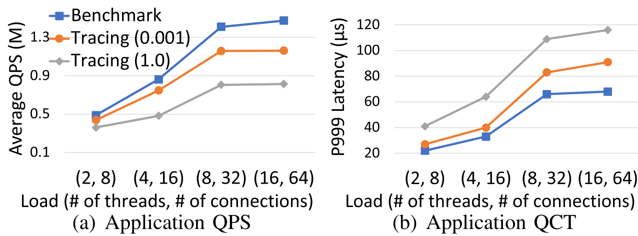
(i) Incomplete coverage. Existing tools monitor partial datapath of requests, either focusing on application-layer tracing [4], [10], [11], [12], [17], [28], [29], [30], transport-layer monitoring [18], [22], [25], [26], [31], network monitoring [9], [19], [20], [27], [32], [33], [34], [35], or partial combination [34], [36], [37], [38]. In addition, these tools define separate abnormal events based on their monitoring objectives. For example, Dapper [22] analyzes packet statistics to infer TCP performance events (such as non-backlogged, congestion and delayed ACK). Trumpet [24] uses triggers on the host to monitor network-wide events (such as burst, heavy flow and congestion). NetSeer [9] monitors flow-level events on the network (such as packet drop, queuing, path changing and pause). Performance profiling tools (*e.g.,* Perf [39]) can analyze events during program execution (such as CPU cycles, page errors, and cache misses). The tracing tool records request time data and provides APIs to monitor application-specific annotations. Until now, no tool could capture all RLA-related events in the end-to-end datapath of request, causing that operators have no confidence to claim innocence (*e.g.,* Case-1).

(ii) Inconsistent semantics. Because these existing tools have different concerns, cloud providers have to combine multiple monitoring tools in production to cover the request's datapath as fully as possible. For example, tracing tools are used in the application layer to monitor the performance of requests. Combined with network monitoring tools, they can capture flow-level events in the underlying network. However, the semantics and granularity of these monitoring tools are inconsistent, and the abnormal events they captured cannot be correlated, resulting in such combinations failing (*e.g.,* Case-2). We ran the storage application and anonymous request traces on our testbed (§V-A), where we use a combination of existing monitoring tools to capture abnormal events and attempt to diagnose slow RPCs. Unfortunately, we find that existing tools cannot explain most of the RLA. First, the request-level events and time data collected by the application tracing tool are too coarse-grained and incomplete, explaining only 28% of the RLAs. Then, based on the time-correlation approach, only 36% RLAs can be inferred by the network monitor, and the remaining 36% RLAs cannot be explained.

(iii) High CPU overhead. Existing performance monitoring schemes in software are inefficient. If the tracing tool is used for fine-grained monitoring in application, it will consume many CPU resources and severely degrade the application performance. To demonstrate this observation, we test the storage application with a tracing tool named Zipkin [16]. Then, we measure the QPS (queries per second) and tail QCT (query completion time) as we vary the sampling rate of the tracing tool. As illustrated in Figure 2, tracing (1.0), *i.e.,* monitoring every RPC, could decrease the QPS by up to 45% and increase the latency by up to 94%. Even if the sample rate is set as 0.001, the QPS can still be decreased by up to 21% (the latency increased by up to 34%), while theoretically missing 99.9% RLAs. The reason for such high overhead is because existing tools generate events and write disks using

Authorized licensed use limited to: Tsinghua University. Downloaded on January 15,2023 at 04:54:38 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.



(a) Application QPS (b) Application QCT

Fig. 2. Impact of tracing tool on application performance.

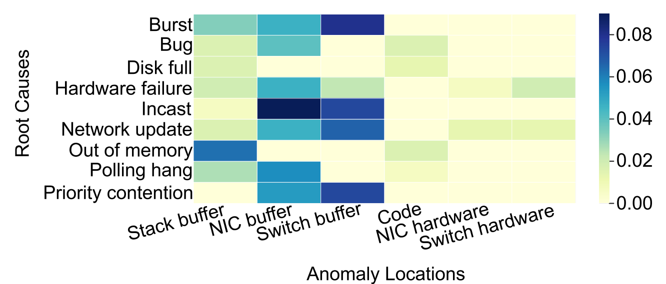


Fig. 3. Heatmap for root causes and anomaly locations of RLAs.

too much precious CPU, which was supposed to execute the application logic. The performance degradations could be more severe as NIC bandwidth increases from 10Gbps to 100Gbps and beyond.

C. Motivation

To accurately diagnose RLA, the entire life cycles of all requests must be monitored. Thus, our goal is to design a high-coverage request event monitoring system, which can monitor RLA-related events with consistent request-level semantics in the end-to-end datapath of request.

In summary, the fundamental limitation of existing performance monitoring tools is that they cannot uniformly model the data plane in network hardware and the datapath in end-host software. Unlike the software, the traditional fixed-function data plane in the network only provides limited accessibility for packets and black-box visibility [9].

As the commodity programmable hardware has been widely deployed in the modern cloud, we see the opportunity to realize our goal. We believe this choice is rational because of the following two unique advantages of programmable hardware. First, with programmable switches and NICs, fine-grained abnormal events in networks can be easily detected, parsed, and reported. It makes monitoring the data plane in networks as flexible as monitoring the datapath in software, and makes the unified modeling of the host and network possible. Second, SmartNICs show a promising capability to offload some trivial tasks during processing packets, which consume non-negligible CPU resources [40], [41], [42].

III. Design

This section first outlines the overview and workflow of BufScope, then elaborates BufScope's design to achieve high coverage for RLA monitoring with low overhead. Finally, it shows how cloud providers can use abnormal events captured by BufScope to diagnose RLAs.

A. Overview

The root causes of RLAs are varied and could be anywhere. Using an unified model to capture request events in all layers is a good starting point to locate the RLA.

Insight. To choose the correct end-to-end unified model for RLA locating, we analyzed each layer and abstracted the modules in each layer into two types, including logic that processes data, and buffers that handle speed mismatch between upstream and downstream logic. Different layers process data in different ways, and may encounter diversified and complex events. For example, in network stack, events like code bug and CPU contention may happen, while in the network, issues like wrong configuration or link failure may occur. It is unlikely that we build a unified model for logic in different layers. On the other hand, buffers are often homogeneous in all layers, and experience similar events like congestion or overflow. Moreover, events in buffer can indicate the location of RLA. For instance, buffer overflow often indicates slow downstream logic. So, it is highly intuitive to choose buffer for RLA Locating.

Production data. Based on the intuition, we further analyzed almost 500 typical RLA tickets from *Alibaba Cloud*'s block storage service, which operators manually troubleshot. We present the root causes of RLAs, and the locations exposed anomaly in Figure 3. The root causes spread across the datapath of request, such as polling hang in hosts, incast in NICs, and burst in switches. We observed that most (>90%) RLAs expose anomalies in *buffers*. The remaining (<10%) RLAs come from NIC flapping, network update, hardware failure, *etc.*, which requires hardware-specific monitoring, and is hard to cover using a unified model. Therefore, instead of monitoring the exact root causes case-by-case, BufScope chooses the buffer as the key object to capture the most RLA-related events.

Design goals. BufScope is a buffer-based request event monitoring system that aims to achieve high coverage for RLA monitoring with low overhead. Specifically, BufScope needs to achieve the following three requirements. First, BufScope should fully monitor buffers in the request's end-to-end datapath to cover RLA-related events. Second, all abnormal events captured by BufScope need to have consistent request-level semantics (*i.e.,* request identifier) to correlate with each other. Finally, to avoid using too much CPU and hurting the performance of the monitored application, BufScope must be low CPU overhead.

Challenges. It is highly challenging to achieve the above three requirements:

- *End-to-end monitoring:* BufScope needs to model a unified buffer chain for various communication frameworks and underlying networks for generality. However, buffers are varied and have different RLA-related events. Therefore, BufScope needs to understand buffers in-depth and define a complete event library, which contains all events that will occur in all types of buffers.

Authorized licensed use limited to: Tsinghua University. Downloaded on January 15,2023 at 04:54:38 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Fig. 4. Architecture of BufScope.

- *Consistent request-level semantics:* Uncertainty of location of request header in packet payload makes it hard for the commodity programmable switches to parse out the request identifiers when generating events. Thus, BufScope needs to design a novel mechanism to inject request-level semantics into the packet header.
- *Low CPU overhead:* The BufScope's semantics operations and event collection in software consume valuable CPU resources that was supposed to execute application logic [11]. Thus, BufScope needs to implement its all functions efficiently with low CPU consuming.

Design principles. BufScope takes full advantage of programmable data plane to address the challenges.

- *For end-to-end monitoring:* the programmable data plane makes it possible to model the end-to-end datapath of a request as a buffer chain, including application buffer, network stack buffer, NIC buffer, and switch buffer. Then, BufScope uniformly classifies all buffers according to three properties, and defines a complete event library to cover all types of buffers.
- *For consistent request-level semantics:* it inserts the offset of the first request's location at the end of packet header, which is performed on the sender side and recovered on the receiver side. So programmable switch can iteratively parse out all request identifiers in packet payload.
- *For low CPU overhead:* it takes full advantage of hardware offloading technology to release the host CPU. BufScope offloads the event collection, reporting, and semantics operation tasks into SmartNICs, which has enough underutilized processors.

B. Architecture and Workflow

We present BufScope's architecture in Figure 4. Buffers to be monitored by BufScope are specified by the buffer chain model (§III-C), and events to be monitored by BufScope are specified by the buffer event definition (§III-D). After that, BufScope takes 4 steps to monitor and collect the request events and diagnose the RLA.

Step 1 (in host): BufScope monitors buffers in the application and network stack. Then, the detected events are carried in outgoing packets (§III-F), which will arrive at the SmartNIC. So the host CPU is not responsible for event logging or reporting.

Step 2 (in SmartNIC): BufScope monitors buffers in the NIC. Besides, BufScope decodes the host events in the packets and collects them locally, then reports them to remote *Event Collector* in bulk. Finally, semantics injection (§III-E) is performed before the packet leaves the sender's SmartNIC. Symmetrically, packet recovery is performed before the packet leaves the receiver's SmartNIC.

Step 3 (in switch): BufScope monitors buffers in programmable switches and parses request identifiers when generating events. After request events are generated by the detection meter in the ingress and egress pipeline, the latter hands the events to the switch control plane, which performs event batching and reporting.

Step 4 (in Event Collector): Based on the request identifier, events from all layers are correlated in *Event Collector* to diagnose RLAs (§III-G).

C. Buffer Chain Modeling

Both research on performance diagnosis [27] and production data in *Alibaba Cloud* reveal that buffer is where the request's traffic stays and is the primary source of abnormal latency. Thus, instead of monitoring complex logic, function calls, and hardware failures, our crucial design choice is to monitor all the buffers in the end-to-end datapath of request to cover the most RLA-related events.

The first challenge is how to enumerate all the buffers in the datapath of highly diversified distributed frameworks [3], [7], [43], [44]. These distributed frameworks rely on different network stacks (such as kernel-bypass network stack [45] and kernel-based network stack [44]) and different threading models (such as run-to-completion model and pipeline model). For generality, BufScope needs to model the datapath of various distributed frameworks as a uniform composition of buffers.

We address the challenge by analyzing available distributed frameworks' logic, buffers, and their connections. We observe that all of their requests follow a unified chain of buffers across their entire life cycle. Therefore, we summarize a basic *buffer chain* model as shown in Figure 4 (the *Buffer* diagrams). Specifically, there exist 4 part buffers in the buffer chain. ① Application buffers are used to store application-layer data, like MessageQueue [46]. ② Network stack buffers are used to maintain requests (or responses) from/to the application and packets (logically) that are formed by decomposing requests

Authorized licensed use limited to: Tsinghua University. Downloaded on January 15,2023 at 04:54:38 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

or packets that will be constructed into requests. ③ Buffer in sender's NIC keeps packets delivered from the host and regularly schedules packets into networks. Conversely, buffer in the receiver's NIC stores packets from the network, and wait for the host to pull packets or actively write packets into host memory. ④ Network switch buffer holds packets for switching, and once the packets cannot be instantaneously forwarded, *i.e.,* a packet will be queued or dropped in the buffer.

Using different network I/O techniques (*e.g.,* zero-copy), the exact number of buffers a request passes may differ from this basic buffer chain model. Essentially, this model provides a guidance for monitoring the end-to-end datapath of requests. In addition to the buffer mentioned in the model, resource contention in other hardware or OS queues (*e.g.,* CPU cache, DRAM, and PCIe) will slow down data processing, resulting in congestion in the upstream buffer [47], which can also be detected by BufScope.

D. Buffer Event Definition

The buffer chain includes diverse types of buffers, which have different RLA-related abnormal events. Thus, a comprehensive and in-depth understanding of buffers is necessary to fully define buffer events.

Buffer classification. Some buffers are lossy, and some are lossless. For lossy Ethernet, when the queuing length of a switch buffer exceeds a certain threshold, subsequent arrival packets will be dropped instead of entering the buffer. For lossless Ethernet, when the buffer of a switch is congested, it will notify the upstream switch to pause packet forwarding until the downstream switch has enough buffer space for new packets. In addition, instead of discarding data, host buffers usually perform back pressure when the queue is deep. Another example, some buffers are order-sensitive, like TCP receive buffer, which may encounter head-of-line blocking (HOL), while order-insensitive buffers do not. There are many more examples. Therefore, BufScope is challenged to thoroughly analyze all types of buffers and define events for them respectively.

To address the challenge, we observe that though many types of buffers exist, they can be classified according to three properties: i) priority awareness, ii) order sensitivity, and iii) enqueue feature. Priority awareness is a property of buffers with multiple queues. These buffers usually enable a priority-based schedule across different queues (*i.e.,* priority-aware). Packets in a low priority queue will have to wait for the high priority queue to drain (in strict priority schedule). Conversely, priority-unaware means that the buffer schedules packets according to the FIFO principle. Order sensitivity represents whether a buffer maintains strict orders of arrived packets before popping them for subsequent processing. Enqueue feature refers to whether packets are likely to be dropped when enqueued.

Event definition. According to the specific type of the three properties, we define 6 kinds of buffer events that may cause RLAs, as shown in Table I. We not only consider the triggering condition of events, but also capture the exact reasons and information of the scene.

- *Priority contention.* This event is triggered in the priority-aware buffer when the length of higher-priority queues is too long, blocking the packets in low-priority queues for a long time. Inappropriate priority allocation may cause RLAs [48]. Conversely, the priority-unaware buffers always firstly forward the packets that arrive earlier and do not have this event.
- *Out-of-order.* This event is triggered in the order-sensitive buffers such as TCP receive buffer, where packets must be handed over to the application in a specified order. The out-of-order event refers to the fact that packets that have already been received must wait until packets are sent before they are received. In contrast, the order-insensitive buffers, such as switch buffer, do not have this event.
- *MMU drop.* This event will happen in the lossy buffers when the MMU is heavily congested. Packet drops would incur packet retransmission and usually result in RLAs.
- *Pause.* This event happens in the lossless buffer. When the buffer occupancy of the downstream switch exceeds a specific threshold, it sends a PAUSE signal to the upstream switch. The latter will pause packet forwarding until a RESUME signal is received. This will block packets in the upstream switch.
- *Congestion.* The event could happen to any buffer. Congestion event is defined as the situation where queue delay exceeds a threshold because the queue is too long rather than the PAUSE signal. This could be caused by the mismatch between upstream and downstream processing or transmission rates.
- *Pipeline drop.* This event refers to the active discarding of requests or packets by processing logic. The reasons include packet corruption, forwarding loop, or blockage from ACL rules. Considering that there are many reasons for pipeline drop, we record the drop code when capturing the events to indicate the specific cause.

In this way, we could predict the RLA-related events that will occur in any buffer, and deploy monitoring mechanism accordingly. In addition, events do not overlap in definition and are mostly mutually exclusive, but a request may encounter multiple events in a buffer simultaneously, such as congestion and priority contention, and BufScope will record all of them.

E. Request-Level Semantics Injection

The layered network architecture [49] is one of the reasons for the great success of the Internet and is used in the cloud. It provides a clear division of labour among the various teams (*e.g.,* network, server, and application) in the cloud. However, it presents a challenge for performance monitoring of distributed applications. Specifically, these teams are often confused for diagnosing RLAs due to the inconsistent semantics of their monitoring tools [9], [25]. To improve the accuracy of RLA diagnosis, BufScope needs to correlate all events by the request-level semantics.

Challenge of request-level semantics parsing. A request contains two parts, *i.e.,* header and payload. The header includes ID, type, date, length, and other metadata, while the payload holds the actual message content. The network stack

Authorized licensed use limited to: Tsinghua University. Downloaded on January 15,2023 at 04:54:38 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

TABLE I
BUFFER EVENT DEFINITION

Property	Type	Event	Triggering Condition	Event Information
Priority awareness	priority-unaware	-	-	-
	priority-aware	*priority contention*	Queuing delay exceeds a threshold & Lengths of higher-priority queues exceed a threshold	- Request ID - Egress queue - Lengths of higher-priority queues - Queuing delay
Order sensitivity	order-sensitive	*out-of-order*	Inconsecutive sequence number	- Request ID - ID of out-of-order request - Queuing delay
	order-insensitive	-	-	-
Enqueue feature	lossy	*MMU drop*	Queues are about to be full or already full	- Request ID - Egress queue - Egress port - Ingress port
	lossless	*pause*	Receiving a PAUSE signal	- Request ID - Egress queue - Egress port - Queuing delay
-	-	*congestion*	Queuing delay exceeds a threshold & no PAUSE signal	- Request ID - Egress queue - Egress port - Queuing delay
-	-	*pipeline drop*	Actively dropped by pipeline or processing logic	- Request ID - Ingress port - Egress port - Drop code (reason)

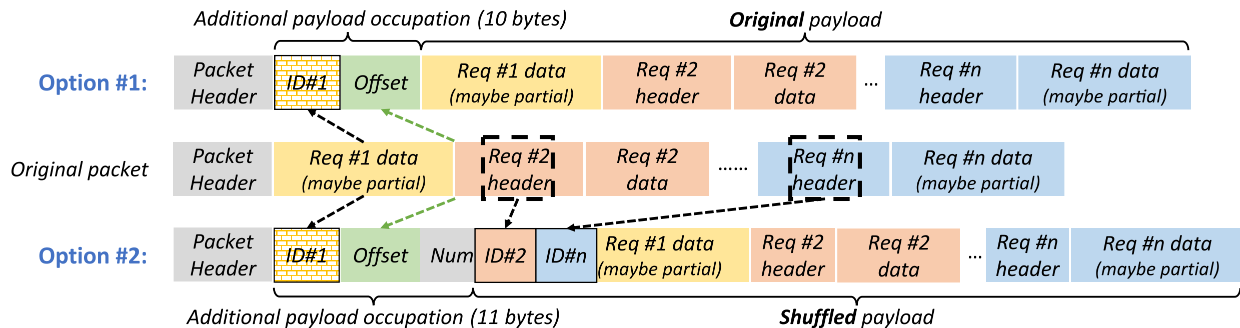


Fig. 5. Two design options of request-level semantics injection in sender. (*Req* stands for request.)

encapsulates requests into packets, where multiple requests are treated as a byte stream and packed into packets as payload. Typically, one large request could be carried by multiple (>1,000 for storage applications) packets, while multiple (>10) small requests may be encapsulated in one packet. Therefore, each packet may carry zero or multiple request headers in practice.

Based on this fact, realizing request-level semantics parsing in networks is challenging. The position of the request header in the packet payload is not fixed, which prevents the commodity switch from parsing the request identifiers. A straightforward solution is to insert all the request IDs before the packet payload, so that programmable switches can match and derive them. However, this solution introduces significant overhead. First, one request ID has 8 bytes in gRPC [44]. The standard MSS of TCP packets is 1460 bytes. This solution is not scalable, and in the extreme case, inserting 10 IDs would occupy ∼5.5% bandwidth. This overhead is ever-present and can degrade the system's goodput. Furthermore, performing too many memory copy operations would seriously waste CPU resources in hosts. An ideal solution should have a constant overhead that does not increase as the number of IDs increases.

Concise request-level semantics injection. We leverage *ID* and *length* fields in the request headers to reduce the overhead. Specifically, we choose to insert the *offset* (2 bytes) of the first complete request header at the end of the packet header (*i.e.,* TCP header), as shown in Figure 5 (option #1). So programmable switches (*e.g.,* P4-16 [50]) could iteratively derive all request indexes in the payload by using the *offset* and *length* fields. This way, we maintain the request-level semantics in a fixed location, which introduces very little overhead on performance and bandwidth.

Besides, the data segment at the beginning of the payload, *i.e.,* the partial *Req #1 data* in Figure 5, may not have the

Authorized licensed use limited to: Tsinghua University. Downloaded on January 15,2023 at 04:54:38 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Algorithm 1 Semantics Injection in Sender's NIC

```
Input: Packet, last_ID
1  offset ← NULL;
2  insert_len ← ID_len + offset_len;
3  buffer_append(Packet, insert_len);
4  buffer_move(payload, insert_len);
5  while index++ < payload_end do
6      if *index = Request.header then
7          if offset = NULL then
8              if index = payload_begin then
9                  ID#1 ← NULL;
10             else
11                 ID#1 ← last_ID;
12             offset ← index;
13         ending_ID ← Request.header.ID;
14 if ending_ID = NULL then
15     ID#1 ← last_ID;
16     offset ← NULL;
17 else
18     last_ID ← ending_ID;
```

corresponding header in the packet, because large request could be carried by multiple packets. Therefore, we should also maintain the identifier of the *Req#1 data*. To this end, we always insert the *ID#1* field (8 bytes) before the *offset* field, which copies the *Req #1 ID*. It requires maintaining a stateful variable, which records the ID of the recent request. Overall, the payload occupation of our injection solution is only 0.68% ($(8+2)/1460$), which is fixed and negligible.

We present the process of semantics injection (option #1) in Algorithm 1. For each packet, we first allocate a fixed space to store the *offset* and *ID#1* fields (line 1-4). Then we look at three possible scenarios. ① There is no partial *Req #1 data* and there is a full *Req #2 header* at the beginning of the payload (line 8-9), so the *ID#1* field is *NULL*; ② These is a partial *Req #1 data*, followed by a full *Req #2 header* (line 10-11), and the *offset* indicates the index of the first full header (line 13); ③ There are no request headers in this packet (line 14-16), then the stateful variable about the recent request ID does not need to be updated. Otherwise, in the first two scenarios, the variable records the ID of the end of this packet (line 17-18), which may becomes the recent request ID for the next packets.

Option #2 for semantics injection. The *option #1* does not work under two conditions: 1) There are pipelines in switch that don't support iteration, and 2) the length of the recyclable parser on the programmable switch is limited. If the payload contains too many request IDs, the switch cannot parse out all of them. Therefore, we provide another, more switch-friendly injection approach, as shown in Figure 5 (option #2). We *move* all the IDs into the end of the packet header. Please note that this is not copying. Note that we are using movement instead of copying. The latter will retain the original IDs and incur more payload occupation. This design option only adds a field

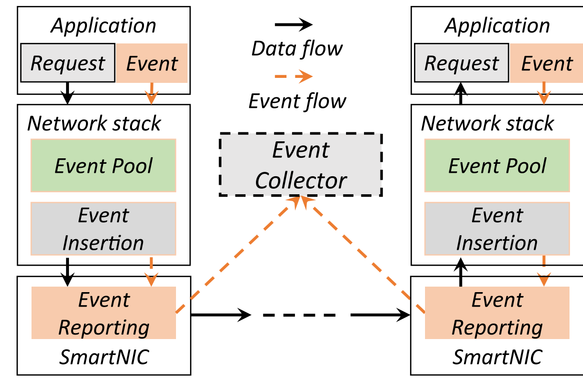


Fig. 6. SmartNIC-offloaded event collection.

Num to represent the number of the subsequent IDs, which is used for parsing in switch. The payload occupation of option #2 is only 0.75% ($(8+2+1)/1460$), where the *offset* is used for payload recovery. This way, all Req IDs are moved to a fixed location, the programmable switches do not need iteration to parse them.

Option #1 has more simple processing in the end-host and more complex parsing in switch, while option #2 needs more memory copies in the end-host, has simpler parsing without iteration in the switch. Cloud operators can choose it based on their capabilities.

SmartNIC-offloaded semantics injection and recovery. It seems that the most appropriate place to perform semantics injection is in the network stack, where requests are encapsulated as packets. However, such a simple solution still requires content parsing and memory copying (both of option #1 and #2), which can consume CPU and severely affect the performance of RTC applications. To save CPU in the hosts, we offload the semantics injection to SmartNIC.

We recover the packets in the receiver's SmartNIC, removing the inserted content. Through SmartNIC offloading instead of processing by host CPU, semantics injection and recovery incur little impact on the application performance.

F. SmartNIC-Offloaded Event Collection

Events could happen in hosts, NICs, and networks. In host software, event collection by own thread could consume CPU resources and degrade the monitored application performance due to its expensive operation, *e.g.,* generating unique ID, system interrupts, writing disk, *etc.* [11]. Besides, CPU cores in public cloud's server can be sold to tenants. Event collection should also avoid using additional CPUs. Thus, achieving persistent storage of events without consuming the own CPU of the thread or other CPU is an unsolved problem.

To address this problem, we innovatively design a SmartNIC-offloaded event collection. As shown in Figure 6, we firstly store events that happened in the application and network stack in memory (*i.e., Event Pool*). Note that the send direction and receive direction have the same *Event Pool*. In network stack, BufScope attaches events from the *Event Pool* into outgoing packets, delivers them to the SmartNIC following the direction of traffic streaming. Then, the SmartNIC

Authorized licensed use limited to: Tsinghua University. Downloaded on January 15,2023 at 04:54:38 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

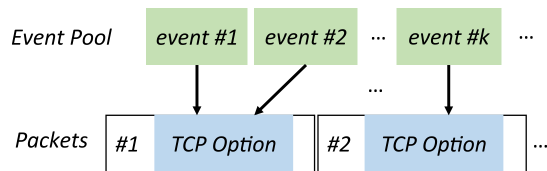


Fig. 7. Inserts events into the TCP option field.

decodes the events from packets, and logs or reports them asynchronously without affecting packet forwarding. In this way, the persistent storage of events only requires the host thread to perform a few memory copy operations. However, this solution faces a challenge.

Selecting the location in packet to attach events requires consideration. A naive solution is to treat the events as application data, and pack them into packet payload. However, TCP maintains a sequence number for reliability. If SmartNIC removes events from packet, the sequence number will be incorrect. The receiver's TCP will assume that a packet is lost and requires retransmission.

Instead of treating events equally as application data, we think of events as additional information that only needs to be passed to the SmartNIC. As shown in Figure 7, we insert events into the packet *header*. We consider the TCP option field as a reasonable position to carry events. The TCP option field has a maximum allowed length of 40 bytes. We disseminate events into different packets to jointly carry the events. To this end, the network stack checks the *Event Pool* when encapsulating packets. Finally, the SmartNIC will strip events and adjust checksum accordingly.

G. RLA Diagnosis and Mitigation

First, BufScope correlates events by request ID and generates an event trace in chronological order. Then, because the blocked time is also recorded in the event, the operators can clearly identify which event is the culprit, even if a request experiences multiple events. We also elaborate on how cloud providers diagnose and mitigate the RLAs according to the events captured by BufScope.

- *Priority contention.* This type of event suggests that RLAs occurred directly in the current buffer. It indicates that queues with higher priorities are congested, and the traffic with low priority is affected beyond expectation. This type of RLA can give cloud providers new insights into traffic patterns in their data centers, and force them to reassign the priorities or weights.
- *Out-of-order.* This type of event indicates that packets are dropped or detoured in previous buffers or logic. For instance, network packet drop and path change can cause out-of-order in TCP receive buffer. Cloud providers could check the MMU drop and pipeline drop events, refer to the following items, then ask network operators for help to check network device failures.
- *MMU drop.* This event can lead to time-consuming retransmission, which could directly cause RLAs. MMU drop is usually caused by burst and incast. Operators could consider optimizing the traffic pattern or updating link capacity, to mitigate TCP incast or congestion.
- *Pause and Congestion.* These events happen due to slow processing of packets in the current or downstream buffer. In this case, BufScope could identify the request that contributes the most to the congestion, *i.e.,* the heavy request, by using fine-grained queue measurement technology [51], [52]. Then, cloud providers need to evaluate the network architecture and application mixing model.
- *Pipeline drop.* This event occurs due to packet error, link corruption or ACL config error. The reason encoded in the event indicates the specific cause. Network operators need to replace the link or verify the network configuration. If this event occurs in the host's software, the processing logic must be debugged.

IV. Implementation

We have implemented BufScope in a kernel-based RPC application named Finagle [53] and the kernel-bypass-based *Alibaba Cloud*'s block storage application. We implement BufScope's functions in the data plane based on Barefoot Tofino switches and Broadcom PS225 SmartNICs.

Requirements and limitations. Because BufScope inserts the ID of request into the packet header, BufScope's implementation requires understanding the format of application-layer protocol and reducing MTU in the network stack. Besides, secondary development of the kernel is needed for monitoring buffers in the kernel-based network stack. Finally, BufScope monitors unencrypted traffic and request, which are widespread in the cloud (*e.g.,* east-west traffic).

Partial deployment. For end-to-end monitoring with consistent semantics, multiple independent teams (*e.g.,* server and network) in the cloud need to collaborate with each other and integrate BufScope. However, partial deployment of BufScope can also help with RLAs diagnosis. SmartNIC-offloaded event collection, semantic injection, and in-server monitoring can still be performed with separate support from the server team, which helps operators determine whether the root cause is in the server. With separate support from the network team, operators can blame or exonerate the network based on the packet- or flow-level events.

Buffer identification. Identifying buffers in the NIC and switch is apparent due to its generality. BufScope summarizes a basic buffer chain for different applications and network stacks. For kernel-based, there is at least one application buffer and one socket buffer. For kernel-bypass-based, zero-copy technology makes application may have only one *mbuf* array for DPDK [54]. However, the manual efforts to identify buffers only need to be done once.

Event capturing. We document the following necessary information for each type of events.

- *Priority contention* (15B): $<$ *ID, egress queue, length of higher-priority queues, queuing delay*$>$. To accurately measure the queuing delay, we record the timestamp when a packet enters and leaves the buffer. We also record the queuing length when the packet enters the buffer. Queuing length is used to distinguish priority contention from congestion

Authorized licensed use limited to: Tsinghua University. Downloaded on January 15,2023 at 04:54:38 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

event. The victim request ID, the timestamps and the length of multiple queues is obtained by INT.

- *Out-of-order* (20B): < *ID, ID of out-of-order request, queuing delay*>. We identify out-of-order event by observing inconsecutive sequence number in packets, and generate this type of events for all latter blocked requests. Recording the out-of-order request as the culprit.
- *MMU drop* (11B): < *ID, egress queue, egress port, ingress port*>. In switch, we redirect packets dropped by MMU to a dedicated internal port, and report in egress pipeline [9], then parse the request IDs in these packets.
- *Pipeline drop* (11B): < *ID, egress port, ingress port, drop code*>. we embed detection logics into the entire packet processing pipelines in ASICs to report all types of pipeline drops. Drop code represents the reason of the packet drop.
- *Pause* (14B): < *ID, egress queue, egress port, queuing delay*>. For a lossless network, the switch begins to generate pause events immediately after receiving a pause signal.
- *Congestion* (14B): < *ID, egress queue, egress port, queuing delay*>. Congestion events are produced when the queuing delay exceeds a threshold, meanwhile the length of the current queue is too long.

To reduce the bandwidth overhead of event collection, we leverage lossless *ZigZag Encoding* [55] to compress events. *ZigZag* takes advantage of the fact that small integers have a higher frequency of being used, which is true for these events. After compressing with *ZigZag*, the average length of events is shortened from 15 bytes to 8 bytes.

SmartNIC. We implement the offloading of event collection and request-level semantic injection in the ARM-based SmartNIC based on DPDK. Event decoding is efficient because the position of the TCP option is fixed. In addition to the cores required for packet forwarding, BufScope requires only one additional core for event reporting asynchronously. Since the ID of the partial *Req #1 data* in retransmitted packets is hard to get in NIC, we just set it as *NULL*. The side effects of this simplification are limited, because these ignored requests have already suffered anomalies (*e.g.,* drop or out-of-order) and have been detected by BufScope. When enabling TSO, semantic injection is performed after packets are segmented in SmartNIC.

Semantic injection. We apply option #1 of semantic injection in *Alibaba Cloud*'s block storage application based on the following considerations. First, the typical I/O sizes are 4KB, 16KB, and 64KB, which means there are no more than two requests in a packet, which the switch can fully parse. Then, doing too many memory copies in an ARM-based SmartNIC can impact performance. Therefore, we choose option #1, which has simpler processing in SmartNIC.

Switch. We embed BufScope's ASIC logic into original switch programs (*e.g., switch.p4*). The switch determines whether the incoming packet need to be monitored according to the DSCP field in the IP header. We detect the pause and pipeline drop in ingress pipelines, detect congestion drop in the MMU. For priority contention and congestion, we record the length of queues, ingress and egress timestamps, and judge whether the queuing delay and length exceed the thresholds in the egress pipeline. After packet events are detected, BufScope uses the group caching mechanism and hash table presented in NetSeer [9], to aggregate them into request events, without false negative and false positive. Then, request events are sent to the control plane via data plane generated packets. Finally, the control plane will report them to the *Event Collector* in bulk. Event detection, processing, and reporting in the switch are similar to that in the NetSeer [9]. We do not claim any novelty here.

Event Collector. We use the servers with 100Gbps NICs as the *Event Collector* in the cluster. The collector aggregates events in two phases to improve the readability and availability of monitoring data. First, the events from all components are aggregated at the granularity of the request. Sure, the aggregation granularity can be changed to the flow level as needed. Then, the events are marked in the request trace, which records the relationship among requests and generated by the distributed tracing system [10], [11].

V. Evaluation

Environment: We evaluate BufScope and existing monitoring tools on a testbed with a 4-ary and 3-tier Fat-Tree topology [56] composed of 10 Barefoot Tofino switches and 16 servers. Each server has 192 CPU cores, 64GB RAM, and one Broadcom PS225 SmartNICs (2×25G) [57]. Each SmartNIC possesses eight ARM Cortex-A72 3.0 GHz CPUs and 16GB memory. There are 4 ToR, 4 Aggregate and 2 Core switches. They are interconnected with 100G links, while each ToR connects four servers with 2×25G link.

Baselines: Given that none of the existing monitoring tools are designed to monitor the end-to-end datapath of request, we combine multiple state-of-the-art tools to fully cover it:

(i) Application tracing. We enable an open-source tracing tool [16] to capture the RPC timing data and application-specific abnormal events in application layer. Because of the large amount of captured data and non-negligible CPU overhead, tracing tools typically require sampling (*e.g.,* 0.1% under high-load services [11]) to reduce impact on the performance of the monitored application. Thus, we set the sampling rate of tracing as 0.1% and 100% for comparison.

(ii) TCP monitoring. Dapper [22] is used to diagnose performance problem of TCP in the end-host network stack.

(iii) Network monitoring. We deploy NetSeer [9] and packet sampling to capture events in switches and NICs. NetSeer is a flow-level event monitoring system based on programmable data plane. For packet sampling, we can parse the request IDs in the mirrored packet to get request events. Since it has a significant bandwidth overhead under a high sampling rate [9], we configure the sampling rate as 1%.

The evaluation needs to answer the following three questions.

- **Capability:** Can BufScope capture most (close to 100%) request events that happen in hosts, NICs, and network switches, and help accurately diagnose the real RLAs?
- **Scalability:** What's the bandwidth overhead and the hardware resources required by BufScope? Can it scale with the increasing datacenter size and bandwidth?

Authorized licensed use limited to: Tsinghua University. Downloaded on January 15,2023 at 04:54:38 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

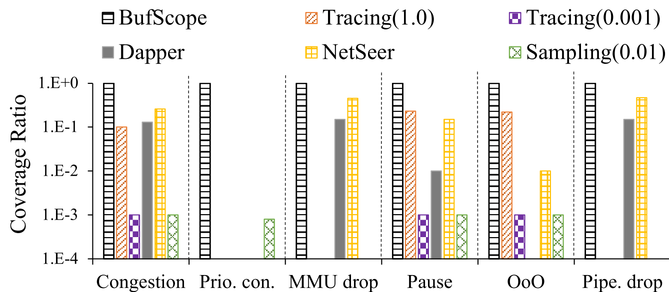


Fig. 8. Event coverage ratios. Tracing (sampling rate).

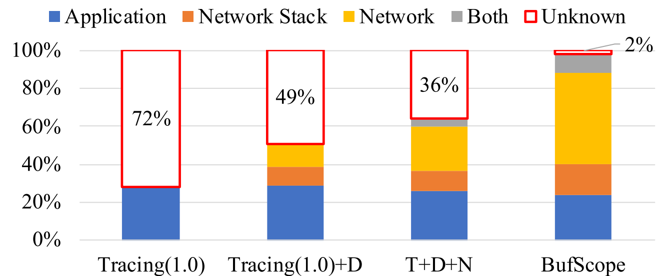
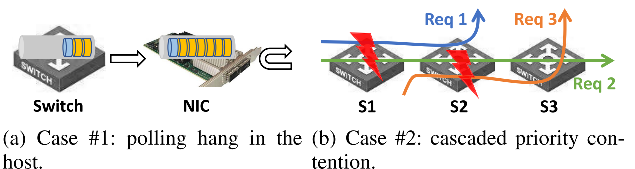


Fig. 9. Diagnosing RLAs by different monitoring tools (T:Tracing, D:Dapper, N:NetSeer).



(a) Case #1: polling hang in the host. (b) Case #2: cascaded priority contention.

Fig. 10. Real case study with BufScope.

- **Performance overhead:** How to choose an efficient threshold? How does BufScope affect the application performance? How much performance gain the hardware offloading brings?

A. Capability

Setup. We deploy the storage application (supports RDMA) and Finagle as the monitored applications, and run request traces based on four real-world traffic distributions including DCTCP [58], VL2 [59], storage and WEB [60] for 6 hours. We set the average link utilization as 80% to produce enough pressure. *Congestion*, *drop* and *out-of-order* (OoO) are naturally produced, while we manually inject pipeline drop. Besides, we configure multi-priority queues to trigger *priority contention*, and enable priority flow control (PFC [61]) in RDMA network to trigger *pause*, which do occur in clouds [9], [37]. We start by evaluating BufScope's capability to fully capture all events along the datapath of request. Next, we compare the proportion of unexplained RLAs of different monitoring tools, and study 2 real RLAs.

Event coverage. We enable tracing, Dapper, NetSeer, packet sampling and BufScope to capture events, respectively. We present the event coverage ratios for different types of events in Figure 8. For a fair comparison, we enable tracing tool in this experiment to monitor all events in host buffers that it can cover. Even so, the tracing tool only has visibility into applications, it cannot detect events in the network. Therefore, tracing(1.0) can only cover up to 23% events, while tracing(0.001) can only cover 0.1% events. Dapper analyzes TCP statistics to infer network events, such as congestion and drop, it can only cover up to 15% events.

NetSeer could capture flow-level events in networks, including congestion, path change, pause and drop, but leaves out the priority contention events. Besides, its captured events miss request-level semantics. Based on the time-correlation methods, NetSeer can only cover up to 47% request events. For packet sampling, if the mirrored packet is lucky enough to encounter an event, then we can parse out the event with request ID. Thus, it only cover <10% events which is always less than its sampling rate. In comparison, BufScope has full coverage for the six types of request events happened in both of the hosts and networks.

Diagnosing RLAs. We leverage the events captured by different monitoring tools to diagnose the root cause of the slow RPC (*i.e.,* RLAs) detected during that period. The locations and proportions of root causes are shown in Figure 9. Specifically, since tracing tool only has visibility in application, it with full sampling (*i.e.,* Tracing(1,0)) can only explain 28% RLAs, leaving 72% RLAs undetermined. Then, combining the server and network monitors, which capture flow-level events to diagnose RLAs based on the time-correlation methods, can only explain 23% and 13% more RLAs, respectively. In other words, enabling tracing, Dapper and NetSeer (*i.e.,* T+D+N) at the same time still leaves 36% RLAs inexplicable. Just enabling BufScope, we can tell whether and how much each component is responsible for each slow RPC, and explain much more (98%) RLAs, including those whose causes were unknown with existing monitors, and some RLAs that were caused by multiple components. The remaining 2% of the RLAs did not reveal any events. We speculate that they are caused by hardware-related anomalies. Compared with [1], we improved the diagnostic rate because we captured the *pipeline drop* events, which accounted for >60% in all packet drop.

Real case study. We reproduce 2 real *Alibaba Cloud*'s production RLAs on our testbed with inferred topology, requests pattern, and traffic rate during the incidents, which cannot be captured and explained by existing monitoring tools.

#1) Polling hang in the host. When a request encounters multiple anomalies, the challenge in diagnosing is to identify the one that has the greatest impact. For example, one RPC encountered congestion in the switch and the receiver NIC. Unfortunately, existing monitoring tools cannot capture how much delay each anomaly causes and treat them equally, resulting in the blame game between the server and network team. Conversely, BufScope can end-to-end capture latency-critical events with consistent semantics, which can associate events that occur in different components. As shown in Figure 10(a), BufScope found that it was blocked for $5ms$ at the receiver NIC, and only experienced $80\mu s$ of congestion in the switch. Therefore, the reason for the RLA was polling

Authorized licensed use limited to: Tsinghua University. Downloaded on January 15,2023 at 04:54:38 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

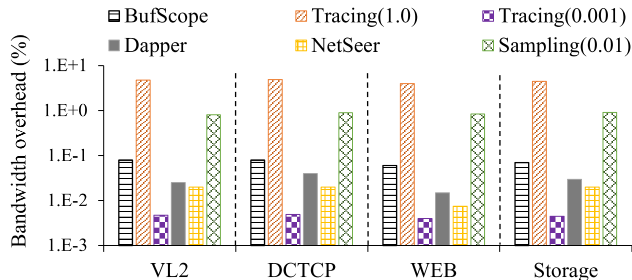


Fig. 11. Bandwidth overhead of event collection.

hang of thread. Based on this, application owners can further analyze its logic and the system log to solve the problem.

#2) Cascaded priority contention. In a priority-aware network, it is non-trivial to assign priorities to different applications. An inappropriate allocation can result in SLA miss for low-priority application. Worse still, a cascading effect would happen when there are multiple queues with diverse priorities. As shown in Figure 10(b), *Req#1*, *Req#2*, and *Req#3* have decreasing order of priority. *Req#2* and *Req#3* contend in a downstream switch S_2. However, *Req#2* is not the culprit. Because *Req#2* is blocked by *Req#1* in the upstream switch S_1. If they do not contend at S_1, the *Req#2* will depart from S_2 before *Req#3* arrives, resulting in no flow contention in the network. Thus, to debug the RLAs of *Req#3*, simply observing the priority contention in a switch is not enough. Since BufScope can capture all detailed contention events, we can analyze this cascade effect and find a more effective method to mitigate it.

B. Scalability

Bandwidth overhead. We compared the bandwidth overhead (BO, ratio of consumed bandwidth to total network bandwidth) required by BufScope and baselines to report events during that period. Figure 11 shows that BufScope only incurs <0.08% BO under various real-world workloads, of which 0.02% from host, 0.01% from NICs and 0.05% from switches. For link bandwidth at 100Gbps, the overhead is at most 80Mbps. In comparison, tracing(1.0) suffers from >4% BO, its each span (*i.e.,* every request) needs 400 bytes on average. Tracing(0.001) needs >0.004% BO. Dapper records only TCP abnormal events and consumes <0.04% BO. Network packet sampling (0.01) needs ~1% BO, which is similar to its sampling rate, because the payload also needs to be recorded to parse request semantics. Because NetSeer captures flow-level events, its event scale and fineness are not as high as BufScope. Thus, NetSeer only incurs ~0.02% BO. In summary, T+D+N consumes >4.06% BO.

To further understand the scalability of BufScope, we calculate the monitoring event traffic as well as the processing overhead of BufScope according to the configuration of *Alibaba Cloud*'s production datacenters. For a normal 3-tier datacenter, connecting 10,000 servers requires approximately 400 switches (3.2Tbps), which produce a maximum of $400 \times 3200Gbps \times 0.08\% = 1024Gbps$ monitoring traffic at most. Processing such traffic requires ~10 servers with 100Gbps NICs, which implies a 0.1% processing overhead.

TABLE II
HARDWARE RESOURCE CONSUMPTION

Resource	switch.p4	NetSeer	BufScope
PHV	83.4%	97.2%	98.3%
Hash Bits	32.2%	41.3%	42.4%
SRAM	30.7%	45.8%	47.2%
TCAM	28.5%	30.4%	32.9%
VLIW actions	35.0%	43.1%	45.5%
Stateful ALUs	8.7%	46.8%	11.2%

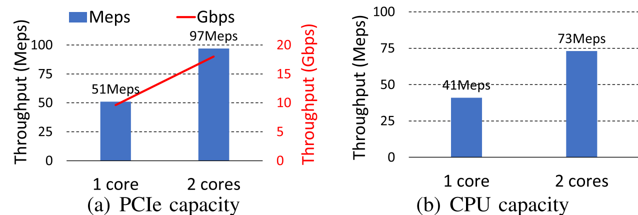


Fig. 12. Capacity of PCIe and switch CPU.

Hardware resource utilization. We integrate BufScope into the *switch.p4*. In Table II, we compare the hardware resources required by the *switch.p4*, BufScope +*switch.p4*, and NetSeer+*switch.p4*. Because BufScope parses out multiple request IDs as part of the packet header, BufScope consumes a large percentage of the Packet Header Vector (PHV). Compared to the *switch.p4*, BufScope consumes a relatively larger proportion (16.5%) of SRAM, which is used for the exact match-action tables and bloom-filters. BufScope's consumption is less than NetSeer's for stateful ALUs, because inter-switch packet drop detection in NetSeer needs to frequently conduct stateful operations over cross-packet states, BufScope treats it as a hardware failure and ignores it. In conclusion, the usage of all resources by BufScope is well below 100%, which means it can easily fit on top of switch.p4.

Module capacity. Request event data will pass several modules including data plane, PCIe, and switch CPU. To evaluate the capacity of the PCIe channel between pipeline and switch CPU, we fix the event batch size as 30, *i.e.,* we pack 30 request events together in one Ethernet packet, and vary cores of CPU for data processing. Figure 12(a) shows that the PCIe channel works at 9.6 Gbps or 51 Meps (events per second) with 1 CPU core, and 18 Gbps or 97 Meps with 2 CPU cores, which is enough for the maximum possible event volume (<7Meps) of a 6.4Tbps switch. Switch CPU is responsible for performing the event reporting and pacing. As shown in Figure 12(b), one or two cores can process request events at 41 Meps or 73 Meps, respectively. Thus, it can handle the maximum possible events in a switch.

C. Parameter Setting

In this experiment, we show a method for selecting the appropriate congestion event threshold in network.

Congestion event threshold determination. Congestion events appear when the queuing delay exceeds a certain threshold, defined as the congestion threshold. We pay special

Authorized licensed use limited to: Tsinghua University. Downloaded on January 15,2023 at 04:54:38 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

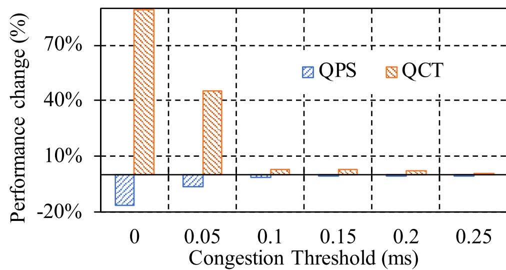


Fig. 13. Impact of congestion threshold on application performance.

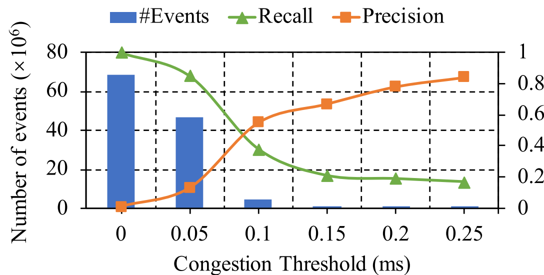
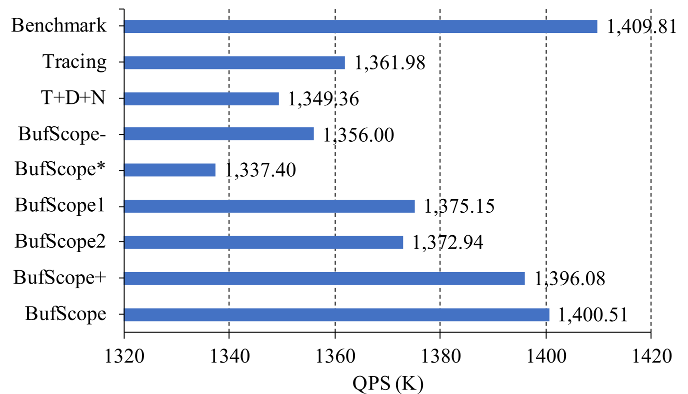


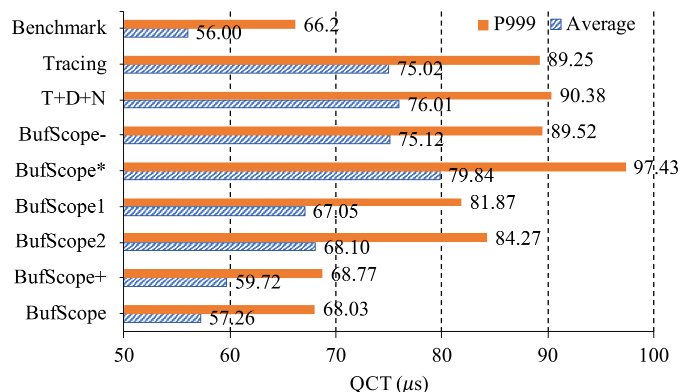
Fig. 14. Number of captured congestion events and the two indicators under different thresholds.

attention to congestion events as it occupies a significant portion of all events, which will occur in both the hosts and networks. BufScope can harm application performance if too many congestion events are collected. Thus, we run the application for 10 seconds with a full-mesh traffic pattern, and measure the impact on the QPS (Query per Second) and QCT (Query Completion Time) of the application as we vary the congestion threshold. As shown in Figure 13, the larger the threshold, the smaller the performance overhead, because it generates fewer congestion events. Thus, threshold selection is highly related to the efficiency of BufScope.

Since requests without RLA can also experience light congestion, not all captured congestion events are RLA-related. In this experiment, the captured events are RLA-related when the RLA request experiences only congestion and no other events. We use two indicators to evaluate the efficiency of the congestion threshold. `Recall` represents the proportion of the captured RLA-related congestion events in all actual RLA-related congestion events. In contrast, `precision` represents the proportion of the captured RLA-related congestion events in all captured congestion events. The higher the threshold, the lower the `recall` and the higher the `precision`. Thus, the selection of the threshold needs to balance these two indicators. Figure 14 shows the changes in the number of events collected, the `recall` and `precision`. We observe that as the threshold increases from 0 to 0.25ms, the `precision` increases from a shallow value to nearly 100%, and `recall` drops from 100% to a shallow value. Thus, application owners could choose to reduce the threshold if the performance overhead of BufScope is acceptable, or increase the threshold to capture events with more negligible overhead. In the following experiments, we take 0.1ms as the congestion threshold for high monitoring efficiency.



(a) Application QPS



(b) Application QCT

Fig. 15. Application performance under different monitoring tools.

Note that the appropriate threshold is related to the link rate and the end-to-end RTT. A larger threshold is required for slower networks (*e.g.,* 1ms for 10Gbps link). In addition, setting the congestion threshold for a buffer in software should consider its drain rate. Buffers in different layers can have different thresholds. In summary, this experiment reveals the impact of the congestion threshold. Cloud operators should comprehensively balance these factors when setting the threshold.

D. Performance Overhead

In this experiment, we use 16 threads (extreme performance has been achieved), which perform 4KB file read, to test the performance of the *Alibaba Cloud*'s storage application under different monitoring tools, and obtain the QPS, average and P999 QCT by running 30 seconds. We show BufScope's performance breakdown by incrementally adding its features.

Pure software implementation without SmartNIC offloading. To verify the performance benefits of hardware offloading, in this experiment, we evaluate the application performance by implementing BufScope without SmartNIC offloading. First, BufScope- represents the performance overhead *only* for semantics injection mechanism without event collection, it means monitoring events but not storing them. As shown in Figure 15. BufScope- decreases the QPS by 3.8% and increases the P999 QCT by 35.2% compared to

Authorized licensed use limited to: Tsinghua University. Downloaded on January 15,2023 at 04:54:38 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

the Benchmark, which represents the raw performance of the application without any monitoring tools. It shows that using the application's threads for semantics injection can seriously affect performance.

Second, BufScope* represents implementing semantics injection and event collection in end-host software. BufScope* decreases the QPS by 1.4% and increases the P999 QCT by 8.8% compared to the BufScope-. It shows that using the application's threads to handle events further degrades performance. Compared to T+D+N, BufScope* decreases the QPS by 0.9% and increases the P999 QCT by 7.7%. The results reveal that the CPU resources of the storage application threads are valuable. We need to use offload techniques to reduce the CPU overhead.

The benefits of SmartNIC-offloaded semantics injection. In this experiment, we offload the semantics injection and recovery into SmartNICs, and keep event collection in software. BufScope1 and BufScope2 represent option #1 and option #2, respectively. BufScope1 increases the QPS by 2.8% and decreases the P999 QCT by 16.0% compared to the BufScope*. It shows that the benefits of SmartNIC-offloaded semantics injection. BufScope2 slightly decreases the QPS by 0.2% and increases the P999 QCT by 2.9% compared to the BufScope1. It shows that the option #2's performance is lower than option #1's. The reason is option #2 has more memory copy operations in the ARM-based SmartNIC. Thus, we recommend using the option #1 if it can be implemented in the programmable switch.

The benefits of SmartNIC-offloaded event collection. In this experiment, we further offload the event collection into SmartNICs, so it is the complete BufScope, which uses the option #1. BufScope increases the QPS by 1.8% and decreases the P999 QCT by 16.9% compared to the BufScope1. The only difference between BufScope and BufScope1 is that the event collection is offloaded into NIC. The reason why BufScope1 has poor performance is that it requires 20% CPU clocks to write disk in this experiment. BufScope+ [1] stands for using an additional core in server to store events, while BufScope uses an ARM core in SmartNIC to handle events. Although BufScope has similar performance as BufScope+, it saves one server CPU. The results demonstrate that *SmartNIC-offloaded Event Collection* can save CPU resources of application threads and improve performance, with the cost of just one ARM core in NIC.

Overall BufScope. Compared with the benchmark, BufScope only decreases the QPS by 0.7% and increases the P999 QCT by 3.0%. Compared with T+D+N, BufScope improves the QPS by 3.8% and reduces the P999 QCT by 24.4%. Compared with BufScope*, BufScope improves the QPS by 4.7% and reduces the P999 QCT by 29.9%. This demonstrates that *SmartNIC-offloaded event collection, semantics injection and recovery* can significantly reduce the performance overhead. According to our analysis, part of the performance degradation caused by BufScope compared to the benchmark comes from our ARM-based SmartNIC. We will use FPGA-based SmartNIC in the future to further improve processing performance.

VI. Related Work

There has been a rich literature regarding application monitoring and diagnosis. We classify them into six categories according to their coverage for the request's datapath.

Tracing-based. Tracing-based monitoring tools are widely used for large-scale application performance tracing and debugging [10], [11], [12], [13], [14], [15], [17], [28], [30], [62], [63]. By inserting annotations into the execution path of the request, tracing tools can locate the problematic step in application layer, but has no visibility in the network stack and underlying networks. Besides, tracing could provide fine-grained latency statistics, but will actually degrade application performance [11]. Therefore, tracing are often used in an on-demand and sampling way.

Log-based. Log analysis is proven effective in many programs or performance debugging scenarios [4], [29], [64], [65], [66], [67], [68], [69]. However, logs are often created by CPU, which is proven inefficient and could waste much CPU resources. Therefore, log-based monitoring systems often use second-level monitoring granularity, which will miss a lot of RLAs.

Network stack-based. Many researches are trying to monitor network performance on the end-host network stack. For example, some research efforts propose to constantly monitor TCP performance by watching TCP statistics such as timeout and retransmission, and deduce the root cause of RLAs through statistical analytics [18], [22], [31], [70], replay [26], or machine learning [25]. Trumpet [24] leverages triggers at end-hosts to monitor every packet and network-wide events. However, they lack visibility into the network, leading to the incomplete coverage for RLAs. Moreover, they focus on packet- or flow-level event capturing and cannot correlate events to the corresponding requests.

NSight [71] reconstructs lifetimes of network messages within end-host stack, then diagnoses the precise causes of latency deviations in the microsecond regime. However, NSight only focuses on the end-host stack, and generating a timeline for each message is too expensive. BufScope and NSight are complementary and can work together. Based on the abnormal events (*e.g.,* congestion), BufScope can detect a message that experienced an anomaly in some part of the end-host, then NSight can reconstruct the timeline of this message and identify the root cause at function level.

NIC-based. Simon [35] collects statistics from NICs, and reconstruct flow queuing time, link utilization, link composition, and other statistics. Nevertheless, it could only obtain aggregated statistics with millisecond-level granularity and lose clues for events at fine-timescale such as microsecond-level microbursts. Similarly, it mainly focuses on network events and cannot fully detect host events.

Network-based. The network serves as the conjunction component among distributed servers. Thus, many efforts have been devoted to network monitoring by active probing [20], [23], telemetry [19], [32], *etc.* NetSeer [9] leverages programmable switch to monitor flow-level network abnormal events, without the request-level semantics. Retro [33] and Microscope [27] monitor the queue to identify anomalies, which is similar to BufScope's buffer model. However,

Authorized licensed use limited to: Tsinghua University. Downloaded on January 15,2023 at 04:54:38 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

network-based monitoring tools have no visibility into hosts. Moreover, their combination with tracing cannot improve the accuracy of RLAs diagnosis due to the inconsistent semantics.

Network and host collaboration. Recent researches use both the network and end-host to jointly collect, store and analyze data [34], [36], [37], [38]. In order to correlate packets' behaviour in end-hosts and networks, they often enable network switches to attach metadata to packets, and extract event in hosts, which will consume a lot of host CPU resources. Besides, these systems did not consider the request-level abnormal events and RLA diagnosis.

VII. Conclusion

By proposing BufScope, this paper presents a promising way to utilize the programmable data plane to achieve high coverage and low overhead for request monitoring and RLA diagnosis. Its core idea is to uniformly model the data plane in networks and the datapath in hosts using the buffer. We translate most RLAs to buffer-related events, and monitor them in the buffer chain to end-to-end cover the lifecycle of request. Then, we design a concise semantics injection mechanism so that all captured events have the victim requests' ID. Finally, BufScope offloads the event collection and semantics operations into SmartNICs to reduce the CPU and performance overhead. BufScope has been fully implemented on commodity SmartNICs and programmable switches. Testbed-based evaluations show that BufScope can diagnose 98% RLAs with negligible bandwidth and performance overhead.

References

[1] K. Gao et al., "Buffer-based end-to-end request event monitoring in the cloud," in *Proc. USENIX NSDI*, 2022, pp. 829–843.

[2] CNCF. (2021). *Cloud Native Computing Foundation*. [Online]. Available: https://cncf.io/

[3] P. Stuedi, A. Trivedi, B. Metzler, and J. Pfefferle, "DaRPC: Data center RPC," in *Proc. ACM SoCC*, 2014, pp. 1–13.

[4] X. Zhao et al., "*lprof*: A non-intrusive request flow profiler for distributed systems," in *Proc. USENIX OSDI*, 2014, pp. 629–644.

[5] M. Balakrishnan et al., "Tango: Distributed data structures over a shared log," in *Proc. ACM SOSP*, 2013, pp. 325–340.

[6] J. Ousterhout et al., "The RAMCloud storage system," in *Proc. ACM TOCS*, 2015, pp. 1–55.

[7] A. Kalia, M. Kaminsky, and D. Andersen, "Datacenter RPCs can be general and fast," in *Proc. USENIX NSDI*, 2019, pp. 1–16.

[8] Y. Gao et al., "When cloud storage meets RDMA," in *Proc. USENIX NSDI*, 2021, pp. 519–533.

[9] Y. Zhou et al., "Flow event telemetry on programmable data plane," in *Proc. ACM SIGCOMM*, 2020, pp. 76–89.

[10] R. Fonseca, G. Porter, R. H. Katz, and S. Shenker, "X-trace: A pervasive network tracing framework," in *Proc. USENIX NSDI*, 2007, pp. 271–284.

[11] B. H. Sigelman et al., "Dapper, a large-scale distributed systems tracing infrastructure," Google, 2010. [Online]. Available: https://research.google.com/archive/papers/dapper-2010-1.pdf

[12] J. Kaldor et al., "Canopy: An end-to-end performance tracing and analysis system," in *Proc. ACM SOSP*, Oct. 2017, pp. 34–50.

[13] A. Satish, T. Shiou, C. Zhang, K. Elmeleegy, and W. Zwaenepoel, "Scrub: Online troubleshooting for large mission-critical applications," in *Proc. 13th EuroSys Conf.*, Apr. 2018, pp. 1–15.

[14] D. Ardelean, A. Diwan, and C. Erdman, "Performance analysis of cloud applications," in *Proc. USENIX NSDI*, 2018, pp. 405–417.

[15] Uber Technologies. (2020). *Jaeger: Open Source, End-to-End Distributed Tracing*. [Online]. Available: https://www.jaegertracing.io/

[16] Twitter. (2021). *Zipkin*. [Online]. Available: http://zipkin.io/

[17] CNCF. (2021). *Opentelemetry*. [Online]. Available: http://opentelemetry.io/

[18] M. Yu et al., "Profiling network performance for multi-tier data center applications," in *Proc. USENIX NSDI*, 2011, pp. 1–14.

[19] Y. Zhu et al., "Packet-level telemetry in large datacenter networks," in *Proc. ACM SIGCOMM*, 2015, pp. 479–491.

[20] C. Guo et al., "Pingmesh: A large-scale system for data center network latency measurement and analysis," in *Proc. ACM SIGCOMM*, 2015, pp. 139–152.

[21] Y. Li, R. Miao, C. Kim, and M. Yu, "LossRadar: Fast detection of lost packets in data center networks," in *Proc. 12th Int. Conf. Emerg. Netw. EXperiments Technol.*, Dec. 2016, pp. 481–495.

[22] M. Ghasemi, T. Benson, and J. Rexford, "Dapper: Data plane performance diagnosis of TCP," in *Proc. Symp. SDN Res.*, Apr. 2017, pp. 61–74.

[23] Y. Peng, J. Yang, C. Wu, C. Guo, C. Hu, and Z. Li, "deTector: A topology-aware monitoring system for data center networks," in *Proc. USENIX ATC*, 2017, pp. 55–68.

[24] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "Trumpet: Timely and precise triggers in data centers," in *Proc. ACM SIGCOMM Conf.*, Aug. 2016, pp. 129–143.

[25] B. Arzani, S. Ciraci, B. T. Loo, A. Schuster, and G. Outhred, "Taking the blame game out of data centers operations with NetPoirot," in *Proc. ACM SIGCOMM Conf.*, Aug. 2016, pp. 440–453.

[26] Y. Li, R. Miao, M. Alizadeh, and M. Yu, "DETER: Deterministic TCP replay for performance diagnosis," in *Proc. USENIX NSDI*, 2019, pp. 437–452.

[27] J. Gong, Y. Li, B. Anwer, A. Shaikh, and M. Yu, "Microscope: Queue-based performance diagnosis for network functions," in *Proc. ACM SIGCOMM*, 2020, pp. 390–403.

[28] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using magpie for request extraction and workload modelling," in *Proc. USENIX OSDI*, 2004, p. 18.

[29] Y. Zhang, S. Makarov, X. Ren, D. Lion, and D. Yuan, "Pensieve: Non-intrusive failure reproduction for distributed systems using the event chaining approach," in *Proc. ACM SOSP*, 2017, pp. 1–14.

[30] Y. Zhang, K. Rodrigues, Y. Luo, M. Stumm, and D. Yuan, "The inflection point hypothesis: A principled debugging approach for locating the root cause of a failure," in *Proc. 27th ACM Symp. Operating Syst. Princ.*, Oct. 2019, pp. 131–146.

[31] A. Roy, H. Zeng, J. Bagga, and A. C. Snoeren, "Passive realtime datacenter fault detection and localization," in *Proc. USENIX NSDI*, 2017, pp. 595–612.

[32] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown, "I know what your packet did last hop: Using packet histories to troubleshoot networks," in *Proc. USENIX NSDI*, 2014, pp. 71–85.

[33] J. Mace, P. Bodik, R. Fonseca, and M. Musuvathi, "Retro: Targeted resource management in multi-tenant distributed systems," in *Proc. USENIX NSDI*, 2015, pp. 589–603.

[34] A. Khandelwal, R. Agarwal, and I. Stoica, "Confluo: Distributed monitoring and diagnosis stack for high-speed networks," in *Proc. USENIX NSDI*, 2019, pp. 421–436.

[35] Y. Geng et al., "SIMON: A simple and scalable method for sensing, inference and measurement in data center networks," in *Proc. USENIX NSDI*, 2019, pp. 549–564.

[36] P. Tammana, R. Agarwal, and M. Lee, "Simplifying datacenter network debugging with PathDump," in *Proc. USENIX OSDI*, 2016, pp. 233–248.

[37] P. Tammana, R. Agarwal, and M. Lee, "Distributed network monitoring and debugging with switchpointer," in *Proc. USENIX NSDI*, 2018, pp. 453–456.

[38] P. Huang et al., "Gray failure: The Achilles' heel of cloud-scale systems," in *Proc. USENIX HotOS*, 2017, pp. 150–155.

[39] A. C. D. Melo, "The new Linux' perf'tools," in *Proc. Slides Linux Kongress*, vol. 18, 2010, pp. 1–42.

[40] Y. Moon, S. Lee, M. A. Jamshed, and K. Park, "AccelTCP: Accelerating network applications with stateful TCP offloading," in *Proc. USENIX NSDI*, 2020, pp. 77–92.

[41] M. T. Arashloo, A. Lavrov, M. Ghobadi, J. Rexford, D. Walker, and D. Wentzlaff, "Enabling programmable transport protocols in high-speed nics," in *Proc. USENIX NSDI*, 2020, pp. 93–109.

[42] D. Firestone et al., "Azure accelerated networking: SmartNICs in the public cloud," in *Proc. USENIX NSDI*, 2018, pp. 51–66.

[43] Apache. (2020). *Thrift*. [Online]. Available: http://thrift.apache.org/

[44] Google. (2020). *GRPC: A High-Performance, Open Source Universal RPC Framework*. [Online]. Available: https://grpc.io/

[45] E. Jeong et al., "mTCP: A highly scalable user-level TCP stack for multicore systems," in *Proc. USENIX NSDI*, 2014, pp. 489–502.

[46] Apache. (2021). *Rocketmq*. [Online]. Available: https://rocketmq.apache.org/

Authorized licensed use limited to: Tsinghua University. Downloaded on January 15,2023 at 04:54:38 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

[47] G. Kumar et al., "Swift: Delay is simple and effective for congestion control in the datacenter," in *Proc. ACM SIGCOMM*, 2020, pp. 514–528.
[48] X. Chen et al., "Fine-grained queue measurement in the data plane," in *Proc. CoNEXT*, 2019, pp. 15–29.
[49] I Standardization, *Information Technology—Open Systems Interconnection—Basic Reference Model: The Basic Model*, International Standard ISO/IEC 7498-1: 1994, 1996, p. 59.
[50] M. Budiu and C. Dodd, "The p416 programming language," in *Proc. ACM SOSP*, 2017, pp. 5–14.
[51] X. Chen et al., "Fine-grained queue measurement in the data plane," in *Proc. CoNEXT*, 2019, pp. 15–29.
[52] R. Joshi, T. Qu, M. C. Chan, B. Leong, and B. T. Loo, "BurstRadar: Practical real-time microburst monitoring for datacenter networks," in *Proc. 9th Asia–Pacific Workshop Syst.*, Aug. 2018, pp. 1–8.
[53] Twitter. (2021). *Finagle*. [Online]. Available: http://twitter.github.io/finagle/
[54] D. Intel. (2014). *Data Plane Development Kit*. [Online]. Available: http://dpdk.org
[55] Google. (2021). *Protocol Buffers: Encoding: Signed Integers*. [Online]. Available: https://developers.google.com/protocol-buffers/docs/encoding#signed_integers
[56] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. ACM SIGCOMM Conf. Data Commun.*, 2008, pp. 63–74.
[57] Broadcom. (2020). *Stingray PS225 Smartnic*. [Online]. Available: https://www.broadcom.com/products/ethernet-connectivity/smartnic/ps225
[58] M. Alizadeh et al., "Data center TCP (DCTCP)," in *Proc. ACM SIGCOMM*, 2010.
[59] A. Greenberg et al., "Vl2: A scalable and flexible data center network," in *Proc. ACM SIGCOMM*, 2009, pp. 51–62.
[60] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *Proc. ACM Conf. Special Interest Group Data Commun.*, Aug. 2015, pp. 123–137.
[61] *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 Standard 802.1Qbb-2011 (Amendment to IEEE Standard 802.1Q-2011, IEEE Standard 802.1Qbe-2011, IEEE Standard 802.1Qbc-2011), 2011, pp. 1–40.
[62] E. rlingsson. ; M. Peinado, S. Peter, M. Budiu, and G. Mainar-Ruiz, "Fay: Extensible distributed tracing from kernels to clusters," in *Proc. ACM TOCS*, 2012, pp. 1–35.
[63] J. Mace, R. Roelke, and R. Fonseca, "Pivot tracing: Dynamic causal monitoring for distributed systems," in *Proc. 25th Symp. Operating Syst. Princ.*, Oct. 2015, pp. 378–393.
[64] X. Zhao, K. Rodrigues, Y. Luo, D. Yuan, and M. Stumm, "Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle," in *Proc. USENIX OSDI*, 2016, pp. 603–618.
[65] L. Luo, S. Nath, L. R. Sivalingam, M. Musuvathi, and L. Ceze, "Troubleshooting transiently-recurring errors in production systems with blame-proportional logging," in *Proc. USENIX ATC*, 2018, pp. 321–334.
[66] D. Yuan et al., "Be conservative: Enhancing failure diagnosis with proactive logging," in *Proc. USENIX OSDI*, 2012, pp. 293–306.
[67] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," in *Proc. Int. Conf. Dependable Syst. Netw.*, 2002, pp. 595–604.
[68] K. Nagaraj, C. Killian, and J. Neville, "Structured comparative analysis of systems logs to diagnose performance problems," in *Proc. USENIX NSDI*, 2012, pp. 353–366.
[69] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl, "Detailed diagnosis in enterprise networks," in *Proc. ACM SIGCOMM Conf. Data Commun.*, 2009, pp. 243–254.
[70] B. Arzani et al., "007: Democratically finding the cause of packet drops," in *Proc. USENIX NSDI*, 2018, pp. 419–435.
[71] R. Haecki et al., "How to diagnose nanosecond network latencies in rich end-host stacks," in *Proc. USENIX NSDI*, 2022, pp. 861–877.

Authorized licensed use limited to: Tsinghua University. Downloaded on January 15,2023 at 04:54:38 UTC from IEEE Xplore. Restrictions apply.