Torp: Full-Coverage and Low-Overhead Profiling of Host-Side Latency

Xiang Chen[1,2,3], Hongyan Liu[1], Junyi Guo[2], Xinyue Jiang[1], Qun Huang[2],
Dong Zhang[3], Chunming Wu[1], Haifeng Zhou[1]
[1]Zhejiang University [2]Peking University [3]Fuzhou University

IEEE INFOCOM 2022 - IEEE Conference on Computer Communications | 978-1-6654-5822-1/22/$31.00 ©2022 IEEE | DOI: 10.1109/INFOCOM48880.2022.9796758

***Abstract*—In data center networks (DCNs), host-side packet processing accounts for a large portion of the end-to-end latency of TCP flows. Thus, the profiling of host-side latency anomalies has been considered as a crucial part in DCN performance diagnosis and troubleshooting. In particular, such profiling requires *full coverage* (i.e., profiling every TCP packet handled by end-hosts) and *low overhead* (i.e., profiling should avoid high CPU consumption in end-hosts). However, existing solutions *fully* rely on end-hosts to implement host-side latency profiling, leading to low coverage or high overhead. In this paper, we propose Torp, a framework that offers full-coverage and low-overhead profiling of host-side latency. Our key idea is to *offload* profiling operations to top-of-rack (ToR) switches, which inherently offer full coverage and line-rate packet processing performance. Specifically, Torp *selectively* offloads profiling operations to the ToR switch based on switch limitations. It *efficiently coordinates* the ToR switch and end-hosts to execute the entire latency profiling task. We have implemented Torp on 32×100 Gbps Tofino switches. Testbed experiments indicate that Torp achieves full coverage and orders of magnitude lower host-side overhead compared to other solutions.**

I. Introduction

Modern data center networks (DCNs) support numerous latency-sensitive applications such as web surfing, payment services, and online social networks. These applications generate TCP flows among DCN end-hosts. They also issue strict service-level objectives (SLOs) on the end-to-end latency of TCP flows. For example, an SLO requires 99% flows to be completed within 1 ms [1]. In this case, an occasional anomaly of 5 ms end-to-end latency is unacceptable since it degrades user experience and reduces revenue. Thus, DCN administrators are strongly required to detect latency anomalies in a timely manner, such that they can quickly troubleshoot to recover SLOs and thus minimize revenue loss.

Previous studies show that the *host-side latency* dominates the overall end-to-end latency of TCP flows [2]. It denotes the time between when a TCP packet enters an end-host and when the end-host returns the corresponding response. According to our testbed experiments (§II), the host-side latency contributes more than 80% of end-to-end latency. Thus, profiling host-side latency is considered as an essential building block in DCNs. Such profiling requires both *full coverage* and *low overhead*. (1) For full coverage, the profiling should measure the latency of all the packets to offer a complete view of host-side status. (2) For low overhead, the profiling itself should not incur high CPU consumption that affects host-side performance.

However, existing solutions are *fully* built on end-hosts so that they suffer from either high overhead or low coverage. First, some solutions frequently invoke system calls or kernel-space tracepoints to measure host-side latency for each packet [3, 4, 5, 6, 7, 8]. They are able to measure every packet handled by end-hosts, thus achieving full coverage. However, they incur significant CPU consumption due to frequent system calls or numerous kernel-space tracepoints. Second, some solutions adopt sampling techniques [9, 10, 11, 12] to select a subset from all packets. They only profile the latency of selected packets to reduce host-side CPU consumption. However, adopting sampling unavoidably sacrifices full coverage.

In this paper, we propose Torp, a framework that offers full-coverage and low-overhead profiling of host-side latency in DCNs. Our main idea is to *offload* the operations of latency profiling to top-of-rack (ToR) switches. This is because modern ToR switches are programmable and allow administrators to customize packet processing logic. They also guarantee line-rate performance and inherently offer full coverage of every packet sent to end-hosts. Thus, the offloading can significantly relieve host-side burdens without losing full coverage. At a high level, when a new TCP packet arrives at the ToR switch, Torp records a timestamp t_{in} indicating the time when the packet arrives the end-host. When receiving the corresponding response from the end-host, it records another timestamp t_{out} indicating the time when the end-host finishes its processing. It calculates $t = t_{out} - t_{in}$ as the host-side latency.

However, it is challenging to realize Torp due to the limited memory capacity of ToR switch, which is typically less than 10 MB memory [13, 14]. This limitation prevents Torp from storing the per-packet timestamp t_{in} in the switch data plane, which requires GB-level memory. For example, suppose that (1) a ToR switch connects to 64 end-hosts via 40 Gbps links and is fully loaded; (2) the size of each packet is 1,000 B [15]; (3) the measurement epoch is 5 s since most DCN flows last less than a second [16]; and (4) the size of t_{in} is 48 bits. In this case, the ToR switch processes packets at $64 \times \frac{40\,\text{Gbps}}{1000\,\text{bytes}} = 320\,\text{Mpps}$. Thus, the required memory size of storing timestamps is $320\,\text{Mpps} \times 5\,\text{s} \times 48\,\text{bits} = 9.6\,\text{GB}$, which far exceeds the capacity of 10 MB.

In response, we design two Torp components to *selectively* offload profiling operations with respect to switch limitations. One component is the Torp agent that retains the operation of storing the timestamp t_{in} in the end-host since the end-host offers enough memory. Another component, the Torp handler, executes other profiling operations in the switch data plane to reduce host-side overhead. Such separation requires careful *coordination* between the profiling operation executed by the

Haifeng Zhou, Qun Huang, and Chunming Wu are corresponding authors.

end-host and that executed by the ToR switch. To this end, Torp temporarily records t_{in} in packet header fields to transfer t_{in} between the ToR switch and end-hosts. We design Torp to record t_{in} in the IP option as changing the IP option does not affect host-side processing. After latency profiling, Torp will reset the modified fields to avoid affecting flow routing.

We implement Torp in P4 [17]. Torp supports kernel-based end-host applications and kernel-bypassing ones (e.g., user-space stacks [18, 19]). Also, it is incrementally deployable and compatible with modern DCNs. We only need to program ToR switches to add the Torp handler and run the plug-and-play Torp agent in end-hosts, while other parts of DCN remain unmodified. Our experiments conducted on a testbed with 32×100 Gbps Tofino switches [20] indicate that (1) Torp achieves full coverage and outperforms state-of-the-art systems with orders of magnitude lower CPU consumption; (2) the switch resource usage of Torp is small ($<2\%$), such that Torp can easily co-exist with other functions of ToR switches.

In addition, the capability of Torp can also bring benefits to many DCN use cases. This paper demonstrates a concrete use case: *profiling end-host applications*. Testbed experiments in §V-C indicate that Torp offers accurate and real-time profiling of three typical applications, including key-value stores [21], network function virtualization (NFV) frameworks [22, 23], and user-space network stacks [18, 19].

II. Motivation

Host-side latency. This paper aims to profile host-side latency. In particular, we target TCP flows because TCP flows accounts for more than 99% DCN flows [24]. We leave the profiling of UDP traffic in our future work. In this paper, the host-side latency refers to the sum of the processing latency of the host-side network stack and that of end-host applications. It contributes a large portion of end-to-end latency and can even reach thousands of microseconds, which is significantly higher than other types of latency (e.g., DCN propagation latency) [25, 2]. Thus, profiling host-side latency is essential for the performance diagnosis of TCP flows in DCNs.

We validate that host-side latency is dominant via testbed experiments. Our testbed uses a Tofino switch [20] that directly connects a client with an end-host. The switch performs layer-2 switching. Since a packet typically traverses 5-6 switches in DCN fabric [26], we let the switch to repeat its processing five times for each packet via packet recirculation to simulate DCN fabric transmission. The testbed configurations are detailed in §V-A. In the end-host, we run Memcached 1.6.7 [21] and httpd 2.4.46 [27], respectively, as the application. In the client, we use memaslap [28] and Apache benchmark [29] to stress-test Memcached and httpd, respectively, with 80% of the maximum load supported by the end-host. Our results indicate that the average end-to-end latency of Memcached and that of httpd are 32 μs and 28 μs, respectively. For Memcached, the 32 μs contain 27 μs in the end-host. For httpd, the 28 μs include 23 μs in the end-host. In both cases, the host-side latency contributes more than 80% of end-to-end latency.

Goals. We aim to accurately profile the host-side latency for every *TCP request*, i.e., a TCP packet sent to the target end-host, during TCP flow transmission. Specifically, we aim to measure the time between when a request enters the target end-host and when the end-host returns the corresponding response (i.e., the ACK of the request). Such profiling requires both *full coverage* and *low overhead*. By full coverage, we mean to profile the latency of every packet processed by the end-host, which provisions a full view of host-side performance. Achieving full coverage is very crucial for capturing host-side latency anomalies due to diagnose congestion or software bugs. With low coverage, the diagnosis will lose significant clues in troubleshooting due to the miss of latency anomalies. By low overhead, we mean that the profiling should not incur high CPU consumption that affects host-side processing.

Note that many methods have been proposed to locate fine-grained host-side performance bottlenecks (e.g., DETER [30]). Our goal is *not* to surpass these methods in the contexts for which they have been extensively optimized. Instead, our study complements these methods. Specifically, our latency profiling identifies which flows suffer from high latency. Administrators can associate the identified flows with the applications processing them to pinpoint performance bottlenecks. With full coverage and low overhead as the first-class design goals, our work alleviates the efforts of finding out latency anomalies.

Limitations of existing solutions. However, existing solutions suffer from either low coverage or high overhead. Here, we classify existing solutions into four types, including system call-based solutions, tracing-based solutions, sampling-based solutions, and offline replaying-based solutions. We elaborate on each type of solutions and discuss limitations as follows.

- **System call-based solutions**. These solutions [3, 4, 31, 32] run on the user space of end-hosts. They invoke system calls to collect system logs and TCP connection information from the network stack resided in the kernel space. To achieve full coverage, they have to frequently invoke system calls to obtain real-time statistics (e.g., polling TCP statistics every 1 ms [4]). Doing so unavoidably incurs high CPU consumption and degrades host-side performance.
- **Path tracing-based solutions**. These solutions [6, 7, 8, 33] work on the kernel space of end-hosts. They instrument the functions of the kernel-space network stack to insert tracepoints that trace the path of every packet during host-side processing. Thus, they are able to achieve full coverage in latency profiling. However, using numerous tracepoints also results in non-trivial CPU consumption [34].
- **Sampling-based solutions**. Some solutions alleviate the overhead of aforementioned solutions via packet sampling [9, 10, 11, 12]. They select some packets as samples with respect to a sampling rate (e.g., 1% or 0.1% of total packets), and only measure the latency of samples. Although these solutions achieve low overhead, sampling inherently suffers from low coverage and poor accuracy due to selective profiling. This makes sampling inadequate for most monitoring tasks, including the profiling of host-side latency.

- **Offline replay-based solutions**. This category of solutions [35, 36, 37, 38, 30] records packet traces at runtime, and conducts offline analysis on these traces to find performance issues. However, such offline processing is too slow for traffic monitoring and does not suit real-time latency profiling.

III. Torp Design

A. Design Overview

We propose Torp, a framework for full-coverage and low-overhead profiling of host-side latency.

Key idea. Existing solutions *fully* rely on the capability of end-hosts to perform *all* the profiling operations. In contrast, Torp offloads profiling operations to the ToR switch so as to reduce host-side overhead while retaining full coverage. Here, the ToR switch brings benefits in three aspects. First, it delivers traffic between end-hosts so that it naturally achieves full coverage of every packet handled by end-hosts. Second, existing ToR switches are programmable and allow administrators to customize the switch data plane [39]. Thus, it is feasible to realize profiling operations in the switch data plane. Third, since ToR switches guarantee line-rate packet processing performance, the performance overhead of executing profiling operations in the data plane of ToR switches is relatively small.

Challenge. Unfortunately, it is non-trivial to exploit the above benefits due to the memory limitation imposed by the ToR switch [13]. As mentioned in §I, a ToR switch typically offers a memory capacity of no more than 10 MB [13, 14], making it infeasible to store massive timestamps in the data plane.

Selective operation offloading. To this end, Torp carefully selects the profiling operations to be offloaded to the ToR switch. For a fine-grained design, Torp decomposes the entire profiling task into six partial operations: (1) identifying the type (request or response) of every arrival TCP packet, (2) recording the timestamp t_{in} when a TCP request arrives at the end-host, (3) storing the timestamp t_{in} during host-side processing, (4) recording the timestamp t_{out} when the end-host returns the corresponding response, (5) calculating the latency $t = t_{out} - t_{in}$, and (6) reporting t to the control plane. Among the six operations, Torp only keeps the third operation of storing the timestamp t_{in} in end-hosts, while offloading other operations to the switch data plane. The reasons are two-fold. First, storing per-packet timestamps requires GB-level memory so that it cannot be offloaded to the restrictive switch data plane. Second, other operations mainly consume computational resources. According to our experiments in §V, these operations consume thousands of CPU cycles in the end-host. Also, offloading these operations only occupies a small memory region in the switch data plane. Thus, we offload these operations to the ToR switch to reduce host-side overhead.

Switch-host coordination. However, one critical issue is how to coordinate the profiling operations executed by the end-host and that executed by the switch data plane. In particular, such coordination requires Torp to transfer the per-packet timestamp t_{in} between the switch data plane and end-hosts

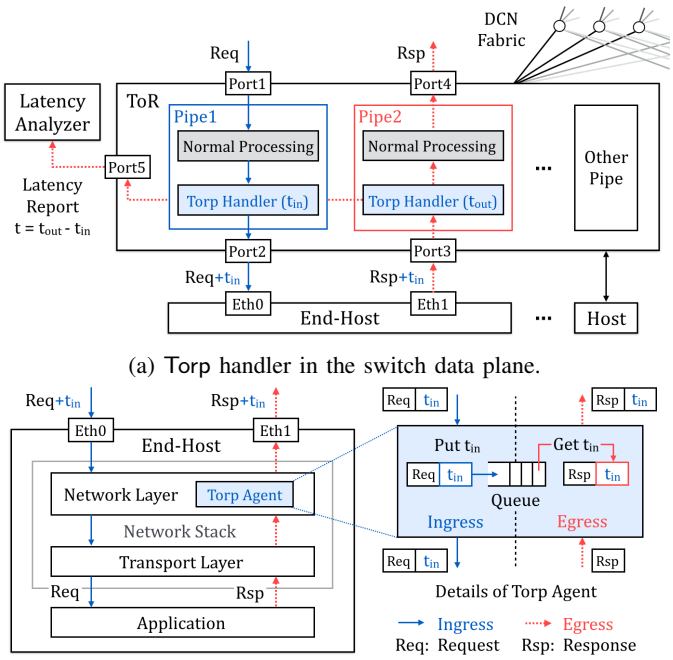


Fig. 1: Torp architecture.

in both directions. First, when sending a request to an end-host, Torp should transfer t_{in} from the switch data plane to the end-host in order to execute the operation of storing t_{in} in the end-host. Second, when the end-host returns the response, Torp should transfer t_{in} back to the switch data plane so as to perform the subsequent operations in the switch data plane.

To address this issue, Torp leverages the programmability of the ToR switch to attach t_{in} inside the transferred packet. A naive choice is to insert a new field to every packet [40, 41, 42, 39]. By that means, both the switch data plane and end-host can access t_{in} in the new field. However, adding a new field needs to augment the host-side network stack with the parsing of the new field, which is non-trivial. In response, Torp uses the IP option of every packet as the carrier of t_{in} among the switch data plane and the end-host. This enables switch-host coordination without adding new fields. Here, we use the IP option field due to the following reasons. First, we can include two extra 32-bit words (i.e., 8 bytes) in the IP option to store the entire 48-bit t_{in} without augmenting host-side stacks. Second, the overhead of using the IP option in packet headers is small (less than 0.01%) compared to a general maximum transmission unit (MTU) of 1500 bytes. Third, since the IP option is rarely used in modern DCNs, changing it does not affect the host-side processing of TCP packets. Thus, Torp incurs minimum modifications to end-host applications.

Architecture and Workload. As shown in Figure 1, Torp consists of two components: a Torp handler in the switch data plane (§III-B), and a Torp agent in the end-host (§III-C).

- **Phase#1** (Pipe1 in Figure 1(a)): The ToR switch receives a TCP request req. It first processes req with normal switch functionalities such as MAC learning. When req reaches the end of switch data plane, the Torp handler records the

current system time t_{in}, which indicates the time when req was sent to the end-host, on the IP option of req.

- **Phase#2** (Ingress in Figure 1(b)): The end-host receives req and invokes its network stack to process req. When req arrives at the network layer in the stack, the Torp agent intercepts req and extracts t_{in} from the IP option. Then it delivers req to upper stack layers while storing t_{in} in a queue during host-side processing.
- **Phase#3** (Egress in Figure 1(b)): The Torp agent receives the response rsp, which corresponds to the request req, from the transport layer. It locates the queue that stores t_{in} of req and dequeues t_{in} from the queue. Then it records t_{in} on the IP option, and sends rsp to the ToR switch.
- **Phase#4** (Pipe2 in Figure 1(a)): When the ToR switch receives rsp, the Torp handler extracts t_{in} from rsp and records the current system time t_{out} indicating the time when rsp is received. It calculates $t = t_{out} - t_{in}$ as the host-side latency of req, and reports t to the control plane. In particular, Torp employs a dedicated analyzer that directly connects to the ToR switch to receive t and perform further analysis. Next, it resets the IP option of rsp and transfers rsp to other functionalities in the switch data plane.

Deployment. The deployment of Torp is practical in DCNs as it only needs to activate the Torp handler in ToR switches and run the Torp agent in end-hosts. This neither affects end-host applications nor interrupts other switches outside DCN racks. Moreover, Torp does not interfere with host-side virtualization like virtual machines since it avoids modifying essential packet fields like MAC addresses, IP addresses, or TCP port numbers.

B. Torp Handler

As shown in Figure 1(a), the Torp handler runs in the data plane of the ToR switch. Its workflow shown in Algorithm 1 performs five profiling operations. First, it identifies the type of every arrival TCP packet (i.e., request or response). Second, for each request, it records the timestamp t_{in} when sending the request to the end-host. Third, when receiving a response, it records another timestamp t_{out}. Fourth, it calculates the host-side latency as $t = t_{out} - t_{in}$. Finally, it reports the latency data to the analyzer in the control plane.

Identifying packet type. For each arrival TCP packet, the Torp handler first executes the operation of identifying the packet type. Specifically, it ignores SYN and FIN packets to avoid affecting the establishment and termination of TCP connections. For any other TCP packet, it checks the egress switch port ρ that will forward the packet. If ρ connects to outside switches in the DCN fabric, it ignores the packet. Otherwise, the Torp handler checks the IP option. If the IP option is empty, the packet has not been processed by Torp so that it is identified as a new request. Otherwise, the packet is identified as a response returned by an end-host.

Processing TCP requests and responses. According to the type of arrival packet, the Torp handler performs corresponding subsequent operations. Specifically, if the packet is a TCP request, it performs the profiling operation of recording the timestamp t_{in} before delivering the request to the end-host. Since the memory in the ToR switch is limited, it does not store t_{in} in the switch data plane. Instead, it records t_{in} in the IP option of the request. In the end-host, Torp agent will be invoked to store t_{in} during host-side processing. After host-side processing, t_{in} will be stored in the IP option of the corresponding response returned by the end-host (see §III-C).

On the other hand, if the packet is a TCP response returned by an end-host, the Torp handler performs another three profiling operations. First, it records the timestamp t_{out} after receiving the response from the end-host. Second, it retrieves t_{in} from the IP option, calculates the host-side latency $t = t_{out} - t_{in}$, and resets the IP option. Third, it selectively reports the latency data to the analyzer, which detects latency anomalies based on received data.

Algorithm 1 Torp handler.

Input: packet pkt, threshold θ, ingress port φ, egress port ρ
Variables: latency t, in-switch timestamps t_{in}, t_{out}

```
1:  function IS_REQUEST(pkt, ρ)
2:      if pkt.ip.option == NULL then
3:          return True
4:      else                          ▷ pkt has been processed by the end-host
5:          return False
6:  function TORP_HANDLER(pkt, ρ, θ)
7:      if Valid(pkt.tcp) and (pkt.tcp.syn & pkt.tcp.fin == 0) then
8:          if Port ρ connects to other switches in DCN fabric then
9:              Exit()
10:         if Is_Request(pkt, ρ) then                ▷ pkt is a request
11:             t_in ← Get_System_Timestamp()
12:             pkt.ip.option ← t_in
13:         else                                      ▷ pkt is a response
14:             t_in ← pkt.ip.option
15:             t_out ← Get_System_Timestamp()
16:             t ← t_out − t_in
17:             if t > θ then                 ▷ Detect a latency anomaly
18:                 Report (t, pkt.fiveTuple, φ) to analyzer
19:             Remove pkt.ip.option
```

Reporting latency data. The Torp handler exports a user-configurable threshold θ for administrators to make a trade-off between profiling coverage and bandwidth overhead. Specifically, only when the latency $t > \theta$, it generates a latency report and sends the report to the analyzer. The report contains the latency t, the packet's five-tuple, and the identifier φ of the ingress switch port that receives the packet. Here, φ is used to identify which end-host experiences the anomaly. By default, θ is set to zero to retain full coverage. In this case, the overhead is acceptable (see Analysis). Administrators can tune θ (e.g., 100 μs) to filter out specific anomalies with respect to the SLOs attached to TCP flows. This further reduces the bandwidth overhead incurred by data transferring.

Workflow. Algorithm 1 shows the workflow of Torp handler. First, if the incoming TCP packet is an SYN or FIN packet, or the packet is sent to other switches, the Torp handler ignores the packet (lines 7-9). Otherwise, it identifies the packet type (line 10). If the packet is a request, it records the timestamp t_{in} on the IP option (lines 11-12). If the packet is a response, it extracts t_{in} from the IP option, and obtains the timestamp t_{out} (lines 14-15). It calculates $t = t_{out} - t_{in}$ (line 16). If $t > \theta$, it

sends a latency report to the analyzer (lines 17-18). $\theta = 0$ by default. Finally, it removes the response's IP option (line 19), and delivers the response to other switch functionalities.

Analysis. We keep the operations of Torp handler simple and hardware-compatible so that these operations can be directly deployed on the ToR switch. Also, doing so only consumes a small portion of switch resources. According to Exp#3 in §V, Torp consumes less than 2% switch resources. Moreover, the Torp handler has no impact on flow routing since it avoids modifying ARP, SYN, and FIN packets.

In addition, Torp incurs limited overhead for the control plane. Consider the example mentioned in §I where the ToR switch processes packets at 320 Mpps. For each packet, Torp will generate a 20-byte report including a 48-bit timestamp, a 104-bit five-tuple, and an 8-bit switch port identifier. Thus, Torp's rate of emitting reports is 320 Mpps×20 bytes/packet = 51.2 Gbps, which only needs one 100 Gbps switch port. Moreover, a two-socket server can process up to 100 Gbps traffic [43]. Thus, allocating one server per DCN rack is sufficient for latency analysis. Note that Torp also enables administrators to *only* monitor latency anomalies. In this case, since latency anomalies rarely occur, the resource consumption of processing anomalies is limited.

C. Torp Agent

The Torp agent stores the timestamp t_{in} in the end-host. It first extracts the timestamp t_{in} from the arrival request and stores t_{in} until receiving the end-host response. Next, when receiving the response, it matches the response with the corresponding timestamp t_{in}, and embeds t_{in} on the response before sending the response to the switch.

Design choice. When a request arrives at the end-host, it will be delivered to the host-side network stack (which resides either in the kernel or user space). The stack parses packet headers and provisions the payload data for high-level applications to receive. When the request enters the transport layer, its IP header will be discarded by default. Recall that the Torp handler embeds the timestamp t_{in} on the IP option of each request. So when the IP header is discarded, t_{in} recorded in the IP option will also be lost. Thus, the key issue is to identify the appropriate location for the Torp agent to intercept the request and store t_{in} before the IP header is discarded. To this end, we choose to place the Torp agent in the network layer, which guarantees its profiling operation to be normally executed.

Managing timestamps with per-flow queues. The Torp agent creates a set Q of queues to store timestamps in the network layer of the host-side network stack. Each queue is associated with a specific flow identified by the five-tuple of the flow. It manages the timestamps attached to the requests of the flow. When the Torp agent receives a request, it locates the queue $Q[key]$ based on the hashing result key of the request's five-tuple. Then it extracts t_{in} from the IP option of the request and records t_{in} in $Q[key]$.

When receiving a response from the transport layer, the Torp agent needs to find the corresponding timestamp since there could be multiple timestamps in the queue. It exploits the *equivalence* between the TCP acknowledge number of a request and the TCP sequence number of the corresponding response. Specifically, when storing t_{in} of a request, it also stores the request's acknowledge number in the queue. When the end-host returns a response, it enumerates every item from the head of the queue. If the sequence number of the response matches one acknowledgment number of an item in the queue, it retrieves t_{in} from the queue. Then it uses the IP option of the response to store t_{in}, which is the same as the Torp handler in §III-B. Thus, Torp *exactly* matches responses and timestamps, making it ignostic of packet loss or reordering.

Algorithm 2 Torp agent.

Input: packet pkt
Variables: index key, queues Q, queue item T, timeout Φ

```
1: function DEQUEUE(Q[key], seq_num)
2:     for T ∈ Q[key] do
3:         t_deq ← current system time
4:         if T.ack_num == seq_num then
5:             Q[key].pop(T)
6:             return T
7:         else if t_deq − T.t_enq > Φ then          ▷ T is expired
8:             Q[key].pop(T)
9:         else                                      ▷ No matched queue items
10:            return NULL
11: function TORP_AGENT(pkt)
12:    if Valid(pkt.tcp) and (pkt.tcp.syn & pkt.tcp.fin == 0) then
13:        if pkt is sent by ToR switch then         ▷ Ingress
14:            t_in ← pkt.ip.option
15:            key ← Hash(pkt.fiveTuple)
16:            t_enq ← current system time
17:            T ← (pkt.tcp.ack_num, t_in, t_enq)
18:            Q[key].push_back(T)
19:        else if pkt is sent by the end-host then  ▷ Egress
20:            key ← Hash(Swap(pkt.fiveTuple))
21:            T ← Dequeue(Q[key], pkt.tcp.seq_num)
22:            if T == NULL then                     ▷ None of items matches pkt
23:                Exit()
24:            pkt.ip.option ← T.t_in
```

In particular, end-host applications may only reply a subset of incoming requests, making other requests *outstanding*, i.e., these requests will never receive replies. For example, when the end-host activates the TCP delayed ACK mechanism, it only replies one response for every two consecutive requests. Thus, the timestamps associated with outstanding requests will never be removed from the queues maintained by the Torp agent. To this end, the strawman method is to set a timeout Φ (e.g., 500 ms) for all stored timestamps. When a timestamp was not matched after the timeout, it removes the timestamp. However, this method incurs excessive overhead as it *actively* monitors all timestamps. To this end, the Torp agent adopts a *passive* strategy. Specifically, when storing a timestamp, it also records the current system time t_{enq}. When receiving an end-host response, it locates the target queue $Q[key]$ and enumerates the items from the front to the end of $Q[key]$. For the current item T, it records the current system time t_{deq}, and checks if $t_{deq} - t_{enq}$ exceeds the timeout Φ. If so, the timestamp in the item is considered to be attached to an outstanding request so that it will be evicted from $Q[key]$. In

this way, the Torp agent handles outstanding requests without imposing significant overhead on host-side processing.

Workflow. As shown in Algorithm 2, the Torp agent intercepts non-SYN and non-FIN TCP packets in the network layer. For the packet sent by the ToR switch, it identifies the packet as a request and extracts t_{in} from the request (line 14). It hashes the request's five-tuple to obtain key and locate $Q[key]$. Then it puts the item T recording t_{in}, the acknowledge number, and the enqueuing time t_{enq} into $Q[key]$ (lines 15-18). Moreover, when receiving a packet from the transport layer, it swaps the source fields and destination fields of the packet's five-tuple, and obtains key via hashing (line 20). It lookups the corresponding item in $Q[key]$ using the packet's sequence number (line 21). If the lookup fails, the packet is a new request sent by the end-host and has not been profiled by Torp. Thus, the Torp agent ignores the packet. Otherwise, it embeds t_{in} on the packet (lines 22-24).

Analysis. Even though Torp still performs the profiling operation of storing per-packet timestamps in the end-host, its overhead are limited. First, the CPU consumption is small since other profiling operations are offloaded to the ToR switch. Second, the host-side memory consumption is limited. Specifically, for each packet sent by the ToR switch, the Torp agent temporarily records a 16-byte item comprising a 32-bit TCP acknowledgement number, a 48-bit t_{in}, and a 48-bit t_{enq}. Thus, the memory consumption M (in GB) can be calculated based on the maximum duration w (in seconds) of storing each timestamp and incoming traffic rate R (in Mpps):

$$M = (16\,\text{bytes / packet}) \cdot w \cdot R$$

Here we consider the worst case:

- Since the host-side latency of processing each packet is up to tens of milliseconds, we set w to one second.
- We set R to be the maximum processing speed supported by the end-host. Suppose that an end-host processes packets at the maximum speed of 40 Gbp while all the packets are 64-byte. In this case, R equals $\frac{40\,\text{Gbps}}{64\,\text{bytes}}$=312.5 Mpps.

In the above extreme case, the usage M is calculated as:

$$M = 16\,\text{bytes/packet} \cdot 1\,\text{second} \cdot 312.5\,\text{Mpps} = 5\,\text{GB}$$

Such consumption is small since a server typically offers tens of GB RAM [44]. Also, w and R are relatively small in practice, e.g., R is no more than 40 Mpps in a Facebook DCN [16]. Thus, the general memory consumption is tens of MB. Our experiments (§V) show that the Torp agent uses up to 52.31 MB RAM. Compared to sufficient host-side memory, such consumption is acceptable.

D. Profiling End-Host Applications

Torp supports profiling kernel-based or kernel-bypassing end-host applications. Here, the kernel-based applications are built on the kernel network stack. They use sockets to establish TCP connections and transfer packets. Many applications, such as RPC frameworks [45, 46] and key-value stores [21, 47, 48], are kernel-based. Torp naturally supports these applications as administrators only need to deploy the Torp agent in the kernel space without modifying applications.

Moreover, the kernel-bypassing applications are built on kernel-bypassing techniques. These applications directly transfer packets between the user space and NIC drivers, eliminating kernel overhead. For example, user-space network stacks implement the TCP/IP protocol stack in the user space to improve performance. For these applications, administrators need to deploy the Torp agent in the user space. To reduce their burdens, we provide three ready-to-use versions of Torp agent that support Linux kernel, DPDK [18], and PF_RING ZC [19], respectively. Administrators can flexibly select a version of Torp agent based on the type of applications.

IV. Implementation

We implement Torp on 32×100 Gbps Tofino switches [20].

Torp **handler**. The Torp handler is realized with 326 LoC in P4. It contains two match-action tables (MATs). The two MATs execute Phase#1 (Pipe1 in Figure 1(a)) and Phase#4 (Pipe2 in Figure 1(a)) of Torp, respectively.

Torp **agent**. We implement the Torp agent as two plug-and-play kernel modules, including an ingress module and an egress module, with 729 LoC in C. The two modules execute the ingress and egress datapaths of Torp agent, respectively. We realize the queues that store timestamps with linked lists. In particular, the two modules can be easily installed and removed at runtime without the need to recompile the kernel. Also, we implement another two kernel-bypassing versions of Torp agent based on DPDK [18] and PF_RING ZC [19] to enable the profiling of kernel-bypassing applications.

Compiler. We build a compiler that automatically integrates the Torp handler into original switch programs (e.g., switch.p4 [49]). The compiler first inserts the P4 code of Torp handler into the original program. It then augments the control flows of the original program to connect the MATs of Torp handler with normal functionalities. Our compiler supports the switch programs written in $P4_{14}$ or $P4_{16}$.

Analyzer. We implement an analyzer program in C and DPDK [18]. The analyzer parses latency reports to obtain latency data. It analyzes the data to obtain real-time latency statistics of every flow. We use these statistics to generate cumulative distribution functions (CDFs) in our experiments.

V. Evaluation

In this section, we evaluate Torp with testbed experiments. In each experiment, we present the average after 100 runs. We highlight our results and findings as follows.

- Torp achieves full coverage in latency profiling (Exp#1).
- Torp consumes an average of 380 CPU cycles per packet, which is orders of magnitude smaller than the overhead (thousands of cycles) of state-of-the-art systems (Exp#2).
- Torp occupies less than 2% switch resources. It only increases per-packet processing latency of ToR switch by 0.23 μs without affecting switch throughput (Exp#3).

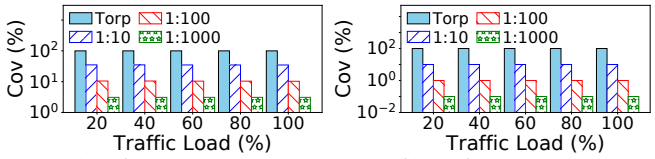
- The measurement error of Torp is less than 0.4 μs, which is acceptable compared to the normal host-side latency (up to hundreds of microseconds) (Exp#4).
- Torp outperforms the comparison methods with over one order of magnitude lower error when measuring the latency of a popular key-value store, Memcached [21] (Exp#5).
- Through the profiling of Torp, we observe that the processing latency of run-to-completion model is lower than that of pipeline model in NFV due to the elimination of inter-core packet transfers (Exp#6).
- Torp supports profiling user-space network stacks. We observe that these stacks achieve up to 71% latency reduction by removing kernel-space overhead (Exp#7).

A. Methodology

Testbed. We build a linear topology with three 32×100 Gbps Barefoot Tofino switches [20]. The switch in the middle is the core while the switches at the edge are ToR switches. These switches are directly connected via 100 Gbps links. Next, we connect three servers to the left edge switch via 40 Gbps links and use them as the clients to generate traffic workload. Another two servers are connected to the right edge switch via 40 Gbps links. They are used as the end-host and analyzer, respectively. Each server has 36-core Intel(R) Xeon(R) Gold 6240C CPU (2.60 GHz) and 128 GB RAM.

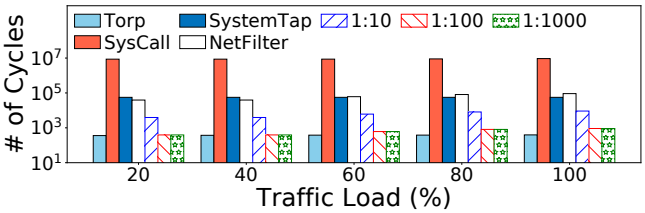
Comparison. We build a system call system (SysCall), two path tracing systems (SystemTap [7] and NetFilter [50]), and a sampling version of NetFilter (Sampling). First, SysCall invokes libpcap [51] to capture all the arrival TCP requests and outgoing responses. It performs the same operations as Torp in the user space. Second, we implement SystemTap and activate an ingress tracepoint and an egress tracepoint in the kernel stack. When a TCP packet arrives the network layer, the ingress tracepoint records the arrival time and reports the time to the user space. Then the egress tracepoint will report the departure time to the user space before the response leaves the network layer. Third, NetFilter invokes two layer-3 hooks to perform the same operations as SystemTap. Note that although SystemTap and NetFilter provide several tracepoints or hooks, we only activate two of them for a fair comparison. Fourth, Sampling is a variant of NetFilter that selects a subset of packets and only records timestamps for selected packets. We set the sampling rate to 1:10, 1:100, and 1:1000, respectively. In particular, for SystemTap, NetFilter, and Sampling, we use mmap [52] to transfer their timestamps from the stack to user space. In the user space, we run a process that calculates host-side latency based on received timestamps. Note that we do not compare Torp with offline replay-based solutions since the latter does not support real-time profiling.

Also, we compare Torp with two measurement tools, MoonGen [53] and Mutilate [54]. MoonGen generates synthetic traffic to evaluate the target at a high speed, while Mutilate is a tool dedicated to evaluating the key-value store, Memcached [21]. Unlike the above systems, the two tools measure the round-trip time of their synthetic packets to deduce host-side

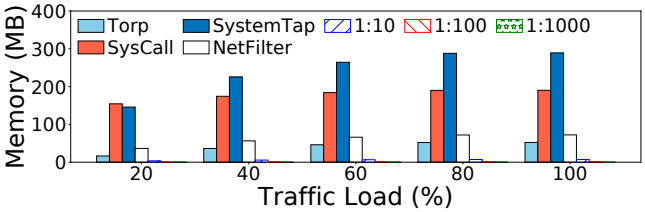


(a) Flow coverage. (b) Packet coverage.

Fig. 2: (Exp#1) Coverage (Cov).



(a) CPU consumption (# of CPU cycles).



(b) Memory overhead (MB).

Fig. 3: (Exp#2) Host-side overhead.

performance. Thus, they inherently achieve near-zero host-side overhead. However, they cannot measure the impact of real traffic on the performance of end-host applications. Thus, we only compare Torp with the two tools when using Torp to profile end-host applications in our experiments (Exp#4-5).

Settings. In Exp#1-4, we build an echo application, which simply returns a response for each TCP request based on TCP socket, as the end-host application. Moreover, in Exp#5-7, we vary the end-host application to demonstrate the benefits of Torp in different use cases. Among our experiments, we set the threshold θ of Torp to zero to achieve full coverage.

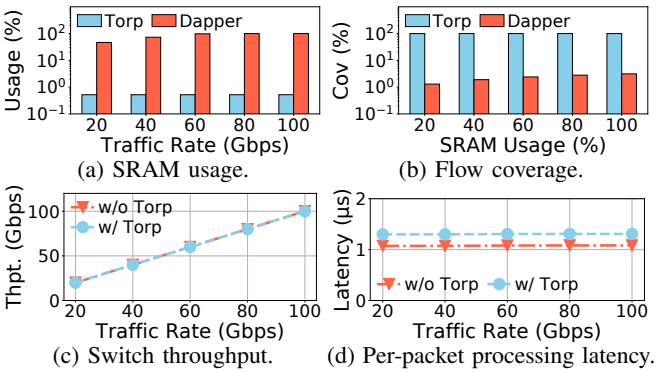
B. Microbenchmarks

(Exp#1) Coverage. We evaluate the coverage of Torp using two coverage ratios: (1) *flow coverage ratio* that represents the ratio of the number of flows that have been profiled to the total number of flows; and (2) *packet coverage ratio* that indicates the ratio of the number of packets that have been profiled to the total number of packets. We select a CAIDA 2018 trace [55] due to the absence of DCN traces. We input the trace to PktGen [56] to generate a workload in which 85% of flows are edge-local to simulate the locality of DCN traffic [16]. We use the workload to test Torp and configure the clients to emit workload to reach 20%∼100% of the maximum load supported by the end-host. As shown in Figure 2, since Torp provides per-packet latency profiling, it achieves full coverage. In contrast, Sampling misses 70%∼90% data. Note that other systems also achieve full coverage, which is elided here.

(Exp#2) Host-side overhead. We evaluate the host-side overhead of Torp. We use the same workload as Exp#1 and measure the CPU consumption and memory overhead during

TABLE I: (Exp#3) Switch resource overhead of Torp.

Name	PHV	VLIW	ALU	TCAM	SRAM	Hbits	Txbar	Exbar
w/o Torp	36.06%	35.94%	22.92%	32.29%	28.02%	31.91%	43.18%	29.17%
w/ Torp	37.76%	36.98%	22.92%	32.29%	28.54%	32.89%	43.18%	30.08%
Usage	**1.7%**	**1.04%**	**0%**	**0%**	**0.52%**	**0.98%**	**0%**	**0.91%**



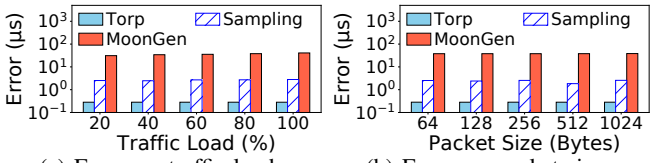
(a) SRAM usage. (b) Flow coverage.
(c) Switch throughput. (d) Per-packet processing latency.

Fig. 4: (Exp#3) Switch-side overhead.

profiling. We quantify the CPU consumption by measuring the number of CPU cycles [57]. Figure 3(a) shows the average number of consumed CPU cycles. The overhead of Torp (∼380 cycles) is orders of magnitude lower than that of SysCall (up to 9.3 M), PathTrace (∼56 K), and NetFilter (up to 91 K) since Torp offloads most profiling operations to the ToR switch. Figure 3(b) indicates that Torp consumes up to 52.31 MB RAM during profiling, which is small compared to GB-level host-side memory. Although Torp incurs more overhead than Sampling, it is the only system that achieves both full coverage and low overhead.

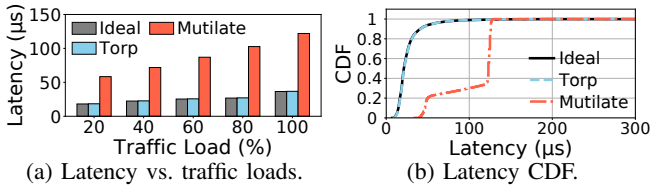
(Exp#3) Switch-side overhead. First, we measure the amount of switch resources consumed by Torp. We consider three types of resources: (1) packet header vectors (PHVs) that transfer packet headers and metadata across the switch data plane, (2) TCAM and SRAM that build the MATs of Torp, and (3) the computational resources that execute the operations of Torp, including very long instruction words (VLIWs), ALUs, hash bits (Hbits), ternary crossbar (Txbar), and exact xbar (Exbar). We measure the resource usage of switch.p4 [49], a common program for ToR switch, as the baseline. Table I shows that Torp uses less than 2% resources, which retains sufficient resources for other logic in the switch data plane.

Second, we compare Torp with Dapper [58]. Dapper stores per-flow information in the switch data plane to infer TCP congestion and receive windows. We modify Dapper to support the profiling of host-side latency. We use Torp and Dapper to process the workload used by Exp#1, respectively. We measure their switch resource usage under various traffic rates. As shown in Figure 4(a), Dapper exhausts SRAM resources when the input traffic rate reaches 80 Gbps. The reason is that it stores timestamps in the switch data plane, leading to serious resource overhead. In contrast, Torp avoids high overhead as it chooses to store timestamps in end-hosts, which saves scarce switch resources. Also, it is infeasible for Dapper to achieve high profiling coverage due to limited switch memory. Figure 4(b) shows that even using all the SRAM resources of a



(a) Error vs. traffic loads. (b) Error vs. packet sizes.

Fig. 5: (Exp#4) Measurement accuracy.



(a) Latency vs. traffic loads. (b) Latency CDF.

Fig. 6: (Exp#5) Key-value store.

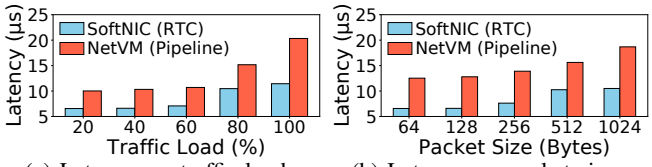
ToR switch, Dapper achieves a small flow coverage of 3.13%. Instead, Torp maintains full coverage in latency profiling.

Third, we evaluate the performance overhead of Torp. We measure the performance of port forwarding as the baseline. We vary the input rate of the workload from 20 Gbps to 100 Gbps to measure the overhead of Torp under pressure. Figure 4(c)-(d) show that Torp has no impact on switch throughput and adds up to 0.23 μs to per-packet processing latency, which is acceptable compared to the normal end-to-end latency (tens of or even hundreds of microseconds) [25, 2].

C. Case Studies: Profiling End-Host Applications

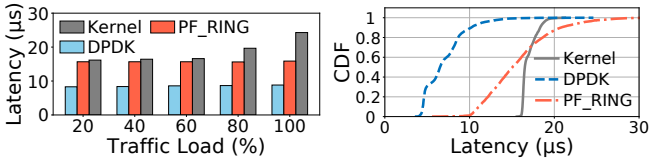
(Exp#4) Measurement accuracy. We first evaluate the measurement accuracy of Torp. We send a TCP flow from the client to the end-host. We change the flow rate from 20% to 100% of the maximum load or the packet size from 64 B to 1500 B. We fix the packet size to 700 bytes based on real-world data [16] as changing the load, while fixing the load to 80% as changing the packet size. We use Torp, Sampling with a sampling rate of 1:10, and MoonGen to measure host-side latency, respectively. We use the results of SysCall as baseline since SysCall provides accurate results. To quantify the accuracy, we measure the mean absolute error, i.e., the mean of absolute differences between the measured data and baseline. Figure 5 shows that the error of Torp is below 0.4 μs and is over one order of magnitude lower than other methods. Such a small error is acceptable as the normal host-side latency can reach hundreds of microseconds. In contrast, MoonGen involves high latency bias incurred by the client kernel, while Sampling loses numerous latency data.

(Exp#5) Key-value store. We evaluate that Torp can be used as an accurate latency measurement tool for Memcached [21]. We run Memcached 1.6.7 as the end-host application. In the client, we send *get* requests to Memcached. We run both Torp and Mutilate, and compare their results with the same baseline as Exp#4. Figure 6(a) shows the average host-side latency under various loads. The results indicate that the error of Torp is below 0.4 μs, which is small. In contrast, due to the overhead incurred by the client-side operating system, the latency measured by Mutilate highly diverges from the baseline. Figure 6(b) presents the CDFs of latency samples



(a) Latency vs. traffic loads. (b) Latency vs. packet sizes.

Fig. 7: (Exp#6) Comparison of NFV execution models.



(a) Latency vs. traffic loads. (b) Latency CDF.

Fig. 8: (Exp#7) User-space network stacks.

measured by Torp and Mutilate under 80% of the maximum load. We observe that Torp aligns better with the baseline, while Mutilate suffers from poor accuracy due to client bias.

(Exp#6) Comparison of NFV execution models. We use Torp to compare the performance of NFV execution models. Specifically, in NFV, NFs are chained as a sequential service function chain (SFC). There are two execution models for running an SFC: (1) *pipeline* model [59, 22] that assigns each NF in the SFC to a single CPU core, with traffic steered to each NF in the SFC one by one; and (2) *run-to-completion* (RTC) model [23, 60, 61] that executes all the NFs on a single CPU core. To compare the two models, we build two NFV frameworks, NetVM [22] based on the pipeline model, and SoftNIC [23] based on the RTC model. We implement an SFC, firewall ⇒ network address translator ⇒ L3 router, based on NetVM and SoftNIC, respectively. We run the SFC on the end-host and configure each NF with 100 rules. We use the same workload as Exp#4. Figure 7 presents the average per-packet processing latency versus various loads or packet sizes. The results measured by Torp indicate that SoftNIC outperforms NetVM with up to 48% latency reduction since RTC avoids transferring packets among CPU cores.

(Exp#7) User-space network stacks. We show that Torp supports profiling user-space network stacks. To do this, we implement a web server based on two user-space network stacks that implement the parsing of Ethernet, IPv4, and TCP headers. The two stacks are built on DPDK 20.08.0 [18] and PF_RING ZC 7.8.0 [19], respectively. We use Torp to profile per-packet processing latency of the server. Also, we profile the latency of the server based on Linux kernel 5.9.0 as the baseline. For workload, we apply the same settings as Exp#4. As shown in Figure 8, compared to the baseline, user-space network stacks achieve up to 71% latency reduction by eliminating kernel-space packet copies.

Summary. These use cases demonstrate that Torp is able to profile various types of end-host applications. In the future, we plan to further evaluate Torp with more types of applications.

VI. Related Work

Host-side latency profiling. Existing solutions for profiling host-side latency pose the trade-off between full coverage and low overhead. The solutions based on system calls [3, 4, 31, 32] and path tracing [33, 6, 7, 8] can be used to profile the processing latency of every packet in the end-host, but incur excessive CPU consumption. Sampling-based solutions [9, 10, 11, 12] achieve low overhead, but inevitably lose profiling coverage and accuracy. SLOG [62] attempts to exploit the capability of programmable switches to achieve low-overhead profiling while preserving full coverage. However, compared to Torp, its design remains preliminary and lacks a comprehensive evaluation.

In-network processing. There have been many efforts leveraging the power of programmable switches for distributed systems, including key-value stores [63], network measurement [64, 39, 65], coordination and consensus [66]. Torp is complementary to these efforts. Dapper [58] identifies misbehaving flows by checking TCP metrics on the ToR switch, while X. Chen *et al.* [67] measures RTT in the data plane. They do not provide the capability of host-side latency profiling, which is addressed by Torp. Moreover, TEA [44] exploits remote direct memory access (RDMA) to extend the resources of ToR switches with the memory available in external servers. However, it requires host-side NICs to support RDMA, which limits its practicality. Instead, Torp does not require specific hardware, thus making it readily deployable in DCNs.

DCN performance diagnosis. Recent systems for DCN performance diagnosis focus on RTT measurement [2, 67], network event monitoring [68, 69, 70], network statistic collection and analysis [71, 72], and so on. 007 [68] detects packet loss and locates detailed problems at runtime. OmniMon [39] achieves both resource efficiency and high accuracy in diagnosing DCN fabric. Torp can be deployed with 007 and OmniMon as it only imposes small changes and resource overhead to DCNs. It also complements the above systems by providing a full view of host-side performance at runtime.

VII. Conclusion

We propose Torp, a framework that profiles host-side latency while achieving full coverage and low overhead. Its key idea is to selectively offload profiling operations to ToR switches with respect to switch limitations. Extensive experiments in a real-world testbed show that Torp achieves full coverage and orders-of-magnitude lower overhead compared to state-of-the-art systems. We also demonstrate the benefits of using Torp via the use cases of profiling end-host applications.

The authors have provided public access to their code and/or data at https://github.com/Wasdns/Torp-INFOCOM22.

Acknowledgement

We thank our reviewers for their constructive comments. This work is supported by the National Key R&D Program of China (2018YFB1800601), the Science and Technology Development Funds of China (2021ZY1025), the National Natural Science Foundation of China (No. 61902362, No. 62172007), Joint Funds of the National Natural Science Foundation of China (U20A20179), and the Key R&D Program of Zhejiang Province (2021C01036, 2022C01085, 2022C01078).

REFERENCE

[1] A. Belay, G. Prekas, A. Klimovic *et al.*, "Ix: A protected dataplane operating system for high throughput and low latency," in *OSDI*, 2014, pp. 49–65.

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

[3] S. Kandula, R. Mahajan, P. Verkaik *et al.*, "Detailed diagnosis in computer networks," in *SIGCOMM*, 2009, p. 243254.

[4] M. Yu, A. G. Greenberg, D. A. Maltz, J. Rexford, L. Yuan, S. Kandula, and C. Kim, "Profiling network performance for multi-tier data center applications." in *NSDI*, 2011, p. 5770.

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

[6] Tcp probe. https://wiki.linuxfoundation.org/networking/tcpprobe.

[7] systemtap. https://sourceware.org/systemtap/.

[8] Lttng. https://lttng.org/.

[9] C. Hare, "Simple network management protocol (SNMP)," 2011.

[10] Netflow. https://www.ietf.org/rfc/rfc3954.txt.

[11] sflow. http://sflow.org/about/index.php.

[12] J. Suh, T. T. Kwon, C. Dixon, W. Felter, and J. Carter, "Opensample: A low-latency, sampling-based measurement platform for commodity sdn," in *ICDCS*, 2014, pp. 228–237.

[13] P. Bosshart, G. Gibb, H.-S. Kim *et al.*, "Forwarding metamorphosis: fast programmable match-action processing in hardware for sdn," in *SIGCOMM*, 2013, pp. 99–110.

[14] A. Gupta, R. Harrison, M. Canini *et al.*, "Sonata: Query-driven streaming network telemetry," in *SIGCOMM*, 2018, pp. 357–371.

[15] Y. Zhu, N. Kang, J. Cao *et al.*, "Packet-level telemetry in large datacenter networks," in *SIGCOMM*, 2015, pp. 479–491.

[16] A. Roy, H. Zeng, J. Bagga *et al.*, "Inside the social network's (datacenter) network," in *SIGCOMM*, 2015, pp. 123–137.

[17] P. Bosshart, D. Daly, G. Gibb *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[18] Intel. Data Plane Development Kit. http://dpdk.org.

[19] PF_RING ZC. https://bit.ly/3z8wgXo.

[20] Tofino. https://www.barefootnetworks.com/technology/#tofino.

[21] Memcached. http://memcached.org/.

[22] J. Hwang, K. K. Ramakrishnan, and T. Wood, "Netvm: high performance and flexible networking using virtualization on commodity platforms," *IEEE Transactions on Network and Service Management*, vol. 12, no. 1, pp. 34–47, 2015.

[23] S. Han, K. Jang, A. Panda *et al.*, "Softnic: A software nic to augment hardware," *UCB/EECS-2015-155*, 2015.

[24] M. Alizadeh, A. Greenberg, D. A. Maltz *et al.*, "Data center TCP (DCTCP)," in *SIGCOMM*, 2010, pp. 63–74.

[25] R. Kapoor, G. Porter, M. Tewari *et al.*, "Chronos: Predictable low latency for data center applications," in *SOCC*, 2012, pp. 1–14.

[26] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4. ACM, 2008, pp. 63–74.

[27] httpd. https://httpd.apache.org/.

[28] memaslap. http://docs.libmemcached.org/bin/memaslap.html.

[29] Apache Benchmark. https://httpd.apache.org/docs/2.4/programs/ab.html.

[30] Y. Li, R. Miao, M. Alizadeh, and M. Yu, "Deter: Deterministic tcp replay for performance diagnosis," in *NSDI*, 2019, pp. 437–452.

[31] H. Nguyen, Z. Shen, Y. Tan *et al.*, "Fchain: Toward black-box online fault localization for cloud systems," in *ICDCS*, 2013, pp. 21–30.

[32] H. Chen, N. Foster, J. Silverman, M. Whittaker, B. Zhang, and R. Zhang, "Felix: Implementing traffic measurement on end hosts using program analysis," in *SOSR*, 2016, pp. 1–12.

[33] M. Tariq, A. Zeitoun, V. Valancius, N. Feamster, and M. Ammar, "Answering what-if deployment and configuration questions with wise," in *SIGCOMM*, 2008, pp. 99–110.

[34] M. Gebai and M. R. Dagenais, "Survey and analysis of kernel and userspace tracers on linux: Design, implementation, and overhead," *ACM Computing Surveys*, vol. 51, no. 2, pp. 1–33, 2018.

[35] Z. Guo, X. Wang, J. Tang *et al.*, "R2: An application-level kernel for record and replay." in *OSDI*, 2008, p. 193208.

[36] tcptrace. http://tcptrace.org/.

[37] R. R. Sambasivan, A. X. Zheng, M. De Rosa *et al.*, "Diagnosing performance changes by comparing request flows." in *NSDI*, 2011, p. 4356.

[38] Y. Gan, Y. Zhang, K. Hu *et al.*, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in *ASPLOS*, 2019, pp. 19–33.

[39] Q. Huang, H. Sun, P. P. Lee *et al.*, "Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy," in *SIGCOMM*, 2020, pp. 404–421.

[40] C. Kim, A. Sivaraman, N. Katta *et al.*, "In-band network telemetry via programmable dataplanes," in *SIGCOMM Poster*, 2015.

[41] S. Luo, H. Yu, and L. Vanbever, "Swing state: Consistent updates for stateful and programmable data planes," in *SOSR*, 2017, pp. 115–121.

[42] J. Xing, A. Chen, and T. E. Ng, "Secure state migration in the data plane," in *SPIN*, 2020, pp. 28–34.

[43] J. Rasley, B. Stephens, C. Dixon *et al.*, "Planck: millisecond-scale monitoring and control for commodity networks," in *SIGCOMM*, 2014, pp. 407–418.

[44] D. Kim, Z. Liu, Y. Zhu *et al.*, "Tea: Enabling state-intensive network functions on programmable switches," in *SIGCOMM*, 2020, pp. 90–106.

[45] gRPC. https://www.grpc.io/.

[46] Thrift. http://thrift.apache.org/.

[47] J. Ousterhout, A. Gopalan, A. Gupta *et al.*, "The ramcloud storage system," *ACM Transactions on Computer Systems*, vol. 33, no. 3, pp. 1–55, 2015.

[48] Redis. http://redis.io/.

[49] switch.p4. https://github.com/p4lang/switch.

[50] NetFilter. https://www.netfilter.org/.

[51] tcpdump and libpcap. https://www.tcpdump.org/.

[52] mmap. https://man7.org/linux/man-pages/man2/mmap.2.html.

[53] P. Emmerich, S. Gallenmüller, D. Raumer *et al.*, "MoonGen: A scriptable high-speed packet generator," in *IMC*, 2015, pp. 275–287.

[54] J. Leverich and C. Kozyrakis, "Reconciling high server utilization and sub-millisecond quality-of-service," in *EuroSys*, 2014, pp. 1–14.

[55] Caida internet traces. http://www.caida.org/data/overview/.

[56] Pktgen. https://pktgen-dpdk.readthedocs.io/.

[57] Q. Huang, X. Jin, P. P. Lee *et al.*, "Sketchvisor: Robust network measurement for software packet processing," in *ACM SIGCOMM*, 2017, pp. 113–126.

[58] M. Ghasemi, T. Benson, and J. Rexford, "Dapper: Data plane performance diagnosis of tcp," in *SOSR*, 2017, pp. 61–74.

[59] S. Palkar, C. Lan, S. Han *et al.*, "E2: a framework for nfv applications," in *SOSP*, 2015, pp. 121–136.

[60] M. Gallo and R. Laufer, "Clicknf: a modular stack for custom network functions," in *ATC*, 2018, pp. 745–757.

[61] G. P. Katsikas, T. Barbette, D. Kostic, R. Steinert, and G. Q. Maguire Jr, "Metron: Nfv service chains at the true speed of the underlying hardware," in *NSDI*, 2018, pp. 171–186.

[62] M. Kogias, M. Weber, and E. Bugnion, "Slog: Your switch is also your load-generator," in *NSDI Poster*, 2020.

[63] X. Jin, X. Li, H. Zhang *et al.*, "Netcache: Balancing key-value stores with fast in-network caching," in *SOSP*, 2017, pp. 121–136.

[64] T. Yang, J. Jiang, P. Liu *et al.*, "Elastic sketch: Adaptive and fast network-wide measurements," in *SIGCOMM*, 2018, pp. 561–575.

[65] X. Chen, Q. Huang, D. Zhang *et al.*, "Approsync: approximate state synchronization for programmable networks," in *ICNP*, 2020, pp. 1–12.

[66] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica, "Netchain: Scale-free sub-rtt coordination," in *NSDI*, 2018, pp. 35–49.

[67] X. Chen, H. Kim, J. M. Aman *et al.*, "Measuring tcp round-trip time in the data plane," in *SPIN*, 2020, pp. 35–41.

[68] B. Arzani, S. Ciraci, L. Chamon, Y. Zhu, H. H. Liu, J. Padhye, B. T. Loo, and G. Outhred, "007: Democratically finding the cause of packet drops," in *NSDI*, 2018, pp. 419–435.

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

[70] Y. Wu, A. Chen, and L. T. X. Phan, "Zeno: diagnosing performance problems with temporal provenance," in *NSDI*, 2019, pp. 395–420.

[71] F. Uyeda, L. Foschini, F. Baker, S. Suri, and G. Varghese, "Efficiently measuring bandwidth at all time scales." in *NSDI*, 2011, p. 7184.

[72] D. Zhuo, M. Ghobadi, R. Mahajan, K.-T. Förster, A. Krishnamurthy, and T. Anderson, "Understanding and mitigating packet corruption in data center networks," in *SIGCOMM*, 2017, pp. 362–375.