vNetTracer: Efficient and Programmable Packet Tracing in Virtualized Networks

Kun Suo*, Yong Zhao*, Wei Chen† and Jia Rao*
*Department of Computer Science and Engineering, the University of Texas at Arlington
†Department of Computer Science, the University of Colorado, Colorado Springs
Email: {kun.suo, yong.zhao, jia.rao}@uta.edu, cwei@uccs.edu

***Abstract*—As the scale of cloud systems continues to grow, virtualized networks that provide connectivity between services within and across data centers, are becoming increasingly important to the performance and reliability of the cloud. Despite many advantages, including fast deployment, ease of management, and programmability, virtualized networks require additional layers of abstraction and complicate monitoring and diagnosis of performance issues compared to traditional networks on physical hardware. Virtualized networks usually connect components in multiple protection domains, such as a guest OS, the hypervisor, network bridges, and separate virtualized network functions. There is no efficient means to trace packet transmission across the boundaries. Furthermore, it is challenging to reason about the performance of dynamic virtualized networks. Therefore, fine-grained, user customizable, and reconfigurable network tracing becomes a great need. To address these challenges, we built *vNetTracer*, an efficient and programmable packet profiler for virtualized networks. vNetTracer relies on the extended Berkeley Packet Filter (eBPF) to dynamically insert user-defined trace programs into a live virtualized network without any changes to the applications or restarts of the monitored network. Through three case studies, we demonstrate the effectiveness of vNetTracer in diagnosing various virtualized networking problems.**

I. Introduction

The adoption of virtualization in enterprise systems and data centers has given rise to on-demand, elastic, and cost-effective cloud services. Virtualized networks, which provide connectivity to physically or virtually isolated virtual machines (VMs) or containers, are critical to horizontally scaling the cloud services. Studies have shown that network virtualization techniques, such as software defined network (SDN) and network function virtualization (NFV), can improve network utilization while offering better quality-of-service (QoS) guarantees [35], [36], [40]. However, the additional layers of abstraction, high resource consolidation and complexity in the virtualized networks make it difficult to understand, diagnose, and optimize networking performance in the cloud.

Unlike the conventional networks, virtualized networks present unique challenges to performance tracing. First, virtualized networks usually span multiple protected domains, such as the host OS or hypervisor, virtual devices and the virtual OS. Tracing end-to-end performance requires that events within each domain can be correlated. However, no tools can efficiently cross the boundaries of the protected domains to associate the tracing information. Second, performance issues of virtualized networks usually occur with high load, in which the networking performance is sensitive to the tracing overhead. This requires a lightweight tracing tool which can monitor virtualized networks with negligible expenditure. Third, the complexity and volatility of virtualized networks require that the tracing tool is reconfigurable in real time and provides a rich set of metrics for performance diagnosis.

There are a plethora of works focusing on tracing network and distributed systems. However, they fall short of addressing the challenges in tracing virtualized networks. To make sense the performance of virtualized networks, such as understanding the causes of long tail latency under high load, it is necessary to trace the network applications at the packet level. Existing studies have shown that recording the tracing data per packet, which often requires significant data copy and context switching between kernel space and user space, incurs prohibitive overhead during the system monitoring [20], [37]. Furthermore, virtualized networks often comprise multiple layers of abstraction to attain isolation and allow for reconfiguration. Thus, it is essential to instrument the virtualized system to provide the needed trace for performance diagnosis. Manual or static instrumentations [23], [27], [28], [31], [32], [50], [52], [59] often require intrusive changes to the system and cannot be generalized to tracing different applications. Machine learning-based log analysis [38], [43], [57] relies on comprehensive instrumentation of the system, from which meaningful information can be mined. Existing dynamic instrumentation tools, such as *DTrace* [2], *SystemTap* [13], and *DARC* [55], cannot trace across the boundary of protected domains in virtualized systems. Distributed tracing systems, e.g., Pivot tracing [41], are usually implemented at application or middleware level, thereby unable to trace packet transmission inside OS kernels.

In this paper, we leverage the extended Berkeley Packet Filter (eBPF), a dynamic tracing mechanism in modern Linux kernels, to enable lightweight and programmable tracing for virtualized networks on multiple nodes. Although eBPF has been increasingly adopted for traffic control [11], network security [24] or accelerating network infrastructure [6], [16], [21], there are no prior explorations of system performance tracing based on eBPF. To this end, we present the design and implementation of *vNetTracer*, an eBPF-based tracing framework, which enables efficient, flexible and end-to-end network performance monitoring for applications in virtualized networks. Different from the traditional tracing tools, vNetTracer has the following features:

- **Tracing across boundaries**: vNetTracer enables end-to-end tracing across boundaries and can correlate distributed events in separate, protected domains.
- **Efficiency**: vNetTracer incurs marginal runtime overhead and is efficient for performance monitoring and troubleshooting in the highly consolidated and optimized virtualized networks.
- **Programmability**: vNetTracer provides rich performance monitoring metrics, supports customized network packet tracing, and can be configured based on different requirements. Users can modify tracepoints, tracing rules or actions in vNetTracer at runtime.

To achieve the above goals, we make three contributions in designing vNetTracer. First, to enable the end-to-end tracing across software or hardware boundaries, vNetTracer generates a unique trace ID for each packet and embeds the ID into the network packet header of the target application. The trace ID is used to differentiate individual network packet and construct the tracing log for further analysis. Second, we develop a set of performance metrics based on the tracing data, which characterize the performance of virtualized networks, including per-flow throughput, the decomposition of end-to-end latency, per-flow packet drop rate, per-device network processing time, etc. Last, we make several optimizations to minimize the runtime overhead during the network tracing.

Another contribution of this paper is the use of vNetTracer to trace network performance in various virtualized systems and case studies show that vNetTracer can effectively monitor virtualized networks and satisfy different scenarios. Specially, vNetTracer helps us find that 1) the throughput-intensive flow in the ingress port of Open vSwitch (OVS) might cause the network congestion and delay latency-intensive flow through OVS; 2) the default configuration of the Xen's credit2 scheduler incurs long tail network latency when executing CPU-intensive VMs and latency-intensive VMs on the same physical CPU; 3) the inefficient processing of a large number of *softirqs* in multicore systems imposes a significant performance bottleneck for container overlay networks.

The rest of this paper is organized as follows. Section II introduces the background and Section III describes the system design and implementation of vNetTracer. Section IV presents the evaluation and case studies results. Section V discusses the related work and Section VI concludes this paper.

II. Background

eBPF based tracing. The classic Berkeley Packet Filter (BPF) [42] is a kernel architecture for packet capture, which permits sending and receiving network packets at data link layers. However, due to its limited instruction set and difficulty in programming, the classic BPF is only used in few applications, e.g., `tcpdump`. Extended BPF (eBPF) is an extension of classic BPF, which introduces lots of new features and improves the performance. For instance, eBPF introduces new in-kernel Just-In-Time (JIT) machine, more register support and many new data structures for generating more complex and advanced eBPF programs. As shown in Figure 1, eBPF

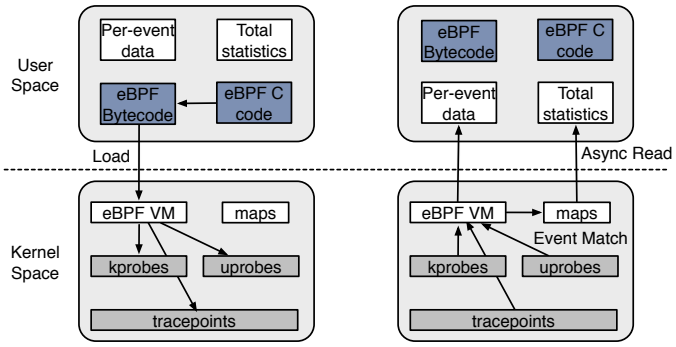

Fig. 1: The mechanism of eBPF in Linux. The left one shows the eBPF code insertion from user space into kernel space. The right one shows the trace data is collected from kernel space to user space when eBPF programs execute.

allows programmers to attach user-defined programs into the kernel and the compiled eBPF bytecode can be executed on a live in-kernel VM, which performs insignificant negative impact to the kernel. Once the tracing events are triggered, the monitoring data can be either temporarily stored in the eBPF data structures inside kernel or collected asynchronously to the user space. Linux started to support eBPF since kernel 3.15 and introduced more BPF enhancements in the later versions.

Compared to the traditional monitoring, tracing based on the eBPF provides several advantages. First, it enables users to trace high frequency modules, such as context switches or packet processing, with little runtime overhead. Second, instead of adding inflexible and dull log inside the systems, the eBPF tracing is highly programmable and can be designed for different purposes. Last, as the tracing logics can be loaded or unloaded dynamically, it does not involve too many modifications to the existing systems.

eBPF versus SystemTap. SystemTap [13] is a tracing platform which is used for dynamically instrumenting processes and Linux kernel activities. Many previous efforts have analyzed the SystemTap runtime overhead [10], [14], [30], [33]. In general, the overhead of SystemTap comes from two aspects. First, the frequency of traces and the actions that SystemTap script performs have a significant impact on the instrumentation overhead [33]. For instance, tracing high performance network I/O, which processes tens of thousands interrupts each second, might have non-negligible overhead to the monitored systems. Second, the compilation of the SystemTap script during the start stage or tracing data collection between kernel and user space during the finish stage might also incur some overhead. In comparison, eBPF programs execute through an efficient virtual machine inside the kernel and the JIT compiling minimizes the execution overhead of the eBPF code. We will further compare and discuss the overhead of SystemTap with eBPF in Section IV.

Limitation. vNetTracer relies on recent eBPF features, and consequently requires a Linux kernel 4.9 and above. Also, the eBPF program is limited by its size, which allows at

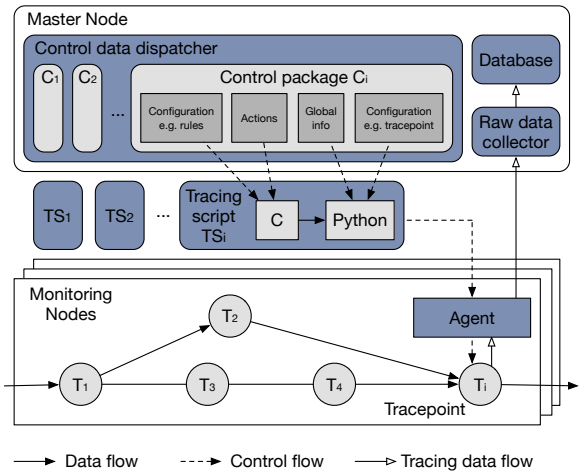

Fig. 2: vNetTracer architecture.

most 4k instructions. In addition, as the eBPF programs are only attached at tracepoints such as network sockets, *kprobes*, etc., vNetTracer is also limited by the tracepoint position. In comparison, many other techniques, such as Time Capsule [52], can add tracepoints arbitrarily inside the virtualized network. Furthermore, although vNetTracer is transparent to the network applications, we still need to modify the kernel in order to accurately trace each packet. Thus, vNetTracer is not completely transparent to the entire system. However, as the discussed in Section III, vNetTracer only involves tens of lines of code modification inside the kernel.

III. vNetTracer Design

A. Overview

Figure 2 illustrates the architecture of vNetTracer. The key components of vNetTracer include a *control data dispatcher*, an *agent* on each monitoring machine and a *raw data collector*.

The control data dispatcher executes on the master node. It reads the user input and generates formatted configuration files in control packages and tracing scripts. Then the dispatcher sends the files to agents on remote monitoring machines.
The agent receives the configured files from the dispatcher and executes eBPF programs at defined locations of the configuration files on the monitoring nodes. Agents collect the tracing data based on the rules in the configuration files and then send the collected data to a centralized raw data collector.
The raw data collector also executes on the master node. It collects the raw tracing data from the agents and performs offline analysis based on the tracing data.

Next, we used a concrete example to describe how vNetTracer works. Suppose we need to measure the network latency between two Virtual eXtensible LAN (VXLAN) layers in the multiple host container network. We use $flannel_i$ to represent the VXLAN network device on the i_{th} node. First, we input the following information into the control data dispatcher to generate formatted control package: (1) the filter rules, such as the containerized application source IP, destination IP, source port, destination port, etc; (2) the tracepoint information, including device name $flannel_i$, device ID, etc; (3) the action that records the current system time in nanosecond; (4) the global information like the database configuration, table names. Next the control data dispatcher sends the customized tracing scripts to remote tracing agents on the i_{th} node. All eBPF scripts are attached to device $flannel_i$ and execute the time record action when the targeted network packets pass through. Network packets which do not match the tracing rules will not be traced. Once the raw tracing data is fetched, it is stored locally and then gathered to the database on the master node. After the data is collected, further operations such as data cleaning or calculation can be done for analysis. In this example, we calculate the time from $flannel_i$ to $flannel_j$ to get network latency between two VXLAN devices.

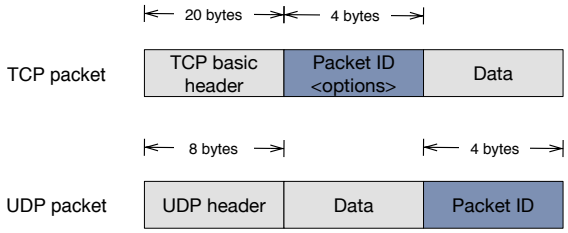

Fig. 3: Add unique packet ID into network packets.

B. Tracing Across Boundaries

Tracepoints provide the system level entry point for vNetTracer to attach customized source code to instrument the system. In the current design, vNetTracer supports instrumenting kernel functions, return of kernel functions, kernel tracepoints and raw sockets through *kprobe*, *kretprobe*, tracepoints and network devices. Application monitoring could be traced through user level tracepoints such as *uprobe* and *uretprobe*. The location of a tracepoint is defined and enabled through user configuration files. Whenever execution of the system reaches an enabled tracepoint, a tracing script configured for that tracepoint is triggered and executed, generating the corresponding tracing data.

In order to trace across boundaries, vNetTracer distinguishes the network applications through their IP address and port number in the packet header, and identifies individual network packets by adding a unique ID. The packet ID is embedded into the header of the packet such that it is carried over the boundaries of domains. As shown in Figure 3, for the TCP packets, we use a 4-byte space in the options of the TCP header. For UDP packets, we use `__skb_put()` to allocate a 4-byte additional space to the original packet at the sender side. We generate a 32-bit random number as the packet ID and store it in the space when the packet is copied from user space to kernel space. The UDP packet ID is then removed from the packet payload through `pskb_trim_rcsum()` before it was copied to the application buffer in the receiver side to guarantee the application transparency. As the above additional operations only involve tens of nanoseconds overhead, they do not harm the microsecond level application latency.

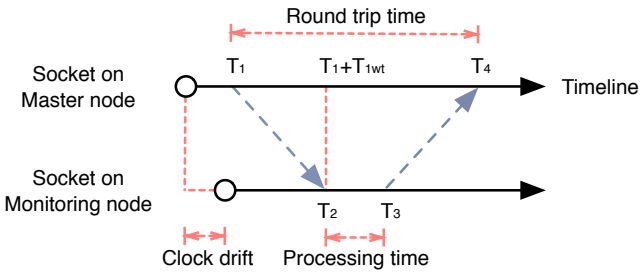

Fig. 4: Calculating the clock drift between two machines under the Cristian's algorithm [29].

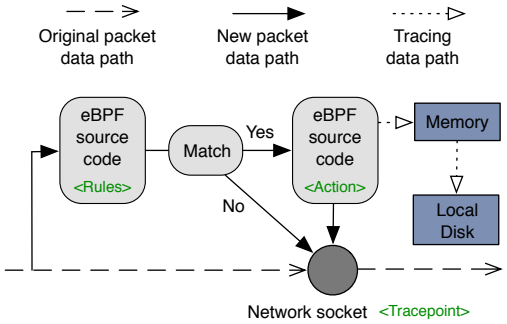

Fig. 5: How eBPF code works for packet filtering and tracing.

Besides the unique packet ID, vNetTracer also records the packet number, packet length and current system time for the detailed network measurement when a packet goes through the tracepoints, which is further discussed in Section III-D.

In order to measure network performance metrics such as throughput, latency, etc., we need to get timestamps from a high resolution clock source. In each tracing script, we obtain the nanosecond-level granularity time record from the function `bpf_ktime_get_ns()`. This function reads the clock source `CLOCK_MONOTONIC` inside Linux kernel, which cannot be set by users and represents monotonic time since the kernel is booted. The nanosecond resolution of `CLOCK_MONOTONIC` is adequate for both network throughput measurement at second granularity and latency measurement at microsecond granularity. In addition, as the function `bpf_ktime_get_ns()` is executed inside the tracing script as backend and such a process always runs in the kernel space, there is almost no overhead to read time from the clock source, and no kernel and user space context switches happen during the above process.

The clocks on different physical or virtual nodes may inevitably have time skew for cross-machine tracing in distributed systems. To mitigate this issue, we adopt Cristian's algorithm [29] and measure the relative clock skew between the master node and the monitoring nodes. As depicted in Figure 4, we attach two tracing scripts at the NIC interfaces of master node and monitoring node. We record the timestamps once the packets were sent or received from the interfaces. On the client side, the round trip time T_{RTT} is measured as $T_4 - T_1$. On the server side, the processing time T_{Pro} is measured as $T_3 - T_2$. Thus, the one way transmission time T_{1wt} can be calculated as $(T_{RTT} - T_{Pro})/2$. To mitigate the network interference, we sample 100 packet records and chose the minimum one as the one way transmission time. Then the clock drift ΔT_{skew} between the master node and monitoring node can be treated as $|T_1 + T_{1wt} - T_2|$ and this value is used for tracing data offline calculation and analysis.

C. Efficiency

As the position of tracepoints, rules and actions are defined by users through configuration files, the tracing scripts are normally attached to those tracepoints and execute the corresponding actions when monitored events happen. Figure 5 illustrates the workflow of tracing on a network device. When a packet goes across a network interface, the original process is just to pass it to the next layer or network device. However, when an tracing script is attached to the interface, the program will be executed and check whether the packet matches the user defined rules. If it matches, the user-defined actions, such as recording the system time, updating the counters, etc., are executed in the tracing script. Afterward the raw tracing data is copied into the local memory associated with this tracing script. If the above actions finish or the rules are not matched, normal packet processing proceeds.

As the network monitoring might generate lots of intermediate tracing data, the overhead of vNetTracer will be extremely high if the storage of that temporary tracing data involves too many disk operations like traditional logs. Such overhead is unacceptable and might hurt the application performance, especially for high speed network services or highly consolidated virtualized networks. To mitigate this issue, we load a kernel module on each monitoring machine to temporarily store the intermediate tracing data. We used `mmap()` to map a kernel buffer to the `/proc` file system in user space. When the tracing scripts generate some intermediate data, it is first copied to the memory buffer. Then we periodically dump the tracing data from the buffer onto the disk, clear the buffer and then collect the raw tracing data on each monitoring machine to a centralized data processing node. As we can adjust the memory buffer size [1] to make the data be stored and collected infrequently, the above steps will not incur so much overhead as to affect the application performance in virtualized networks.

The tracing data is collected by the raw data collector from monitoring nodes to the master node. The collection can be processed either online or offline. For applications which require realtime monitoring, tracing data could be sent from agents to the collector directly. However, such a process could consume additional CPU and network bandwidth. For applications whose QoS is sensitive to the network or tracing overhead, the tracing data can be collected offline. After tracing is completed, all the tracing records at different tracepoints are dumped into the trace database, where records are indexed

[1]Due to the limitation of `malloc` in Linux, the buffer size range is from 32 bytes to 128k-16 bytes.

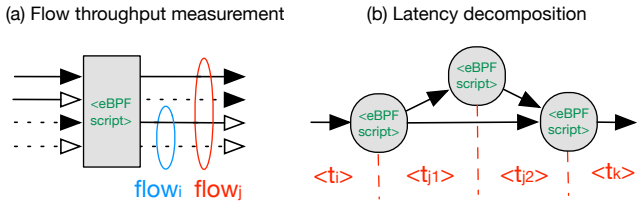

Fig. 6: Advanced network metrics illustration.

by their packet IDs. After the data cleaning and recomputation, such as identifying incomplete records, timestamp alignment for the clock skew, etc., one then can query the database to perform customized analysis of network performance. As the raw data collector periodically receives tracing data from the agents, it also acts as a heartbeat monitor to guarantee that the agents work properly.

D. Programmability

After the agents trace the network activities and the data is collected by the raw data collector, additional calculation is required based on those raw tracing data. In this section, we briefly introduce the network performance metrics that vNetTracer is focusing on.

Throughput. The network throughput measures the amount of network packets transmitted from one side to another during a certain period of time. To quantify the throughput at a specific place, e.g., one network socket interface or a kernel function, we track the packet size S_i and the arrival time T_i during the data transmission, and calculated the network throughput as $\sum_{i=1}^{N}(S_i - S_{ID})/(T_N - T_1)$, where the i refers to the order of the network packets during the transmission and S_{ID} is the 4 bytes packet unique ID.

Latency. The network latency measures the time that one packet is transferred from one designated point to another. Based on the packet ID mentioned in Section III-B, we track two packets for the same packet ID at two tracepoints and record the system time through tracing scripts. Suppose the time we record at the two tracepoints are t_1 and t_2. If these two tracepoints are within the same monitoring node, the latency between the two tracepoints is treated as $\Delta T = t_2 - t_1$. If the two tracepoints are located on two different nodes, the latency can be calculated as $\Delta T = t_2 - t_1 + \Delta T_{skew}$.

Jitter. Besides the absolute latency value, the variability of packet latency over a period of time, named jitter, is also important, especially for realtime applications such as video services and live broadcast. This item reflects whether the transmission across the network is smooth or not. Based on the latency measurement, we calculate the network jitter as $\Delta T_{i+1} - \Delta T_i$, where the ΔT_i refers to the i_{th} network latency of traced packet.

Packet loss. Packet loss occurs when the packets are transferred across the network but fail to reach their destination. Packet loss is usually caused by network congestion, network disconnection, device failure, etc. To measure packet loss, we track the number of packet N_i at each tracepoint and calculate the packet loss between two tracepoints as $N_{loss} = N_i - N_j$ and the packet loss rate as $R_{loss} = N_{loss}/N_i$.

Additional metrics. Beside the above basic metrics, more information could also be dug from the raw data for certain scenarios, such as packet arrival time. In addition, combined with the tracing rules, advanced tracing information, like per-flow throughput and the decomposition of end-to-end latency, which are illustrated in Figure 6, can be obtained based on the user needs.

The programmability of vNetTracer also allows the user to control the tracing at runtime. Unlike traditional tools which couple the monitoring logic with system execution or need to stop the system for new tracing logic, vNetTracer strips the tracing from the monitored system. We encapsulate the network tracing into highly configured eBPF scripts and execute them at certain tracepoints based on monitoring purposes. As shown in Figure 2, we separate the vNetTracer control plane from the application network flow and tracing data flow. To realize that, we created highly modularized control package, which includes the tracing rules, tracepoint locations, actions and global configurations, for each tracing script. During the execution, the vNetTracer control data dispatcher formatted the user requirements into tracing configuration files. For instance, users provide information such as ethernet type, source IP, destination port, etc. to generate the filter rules, or file names, function, device ID, etc. to generate tracepoint locations. Once the tracing configuration files are defined, customized tracing scripts are sent to tracing agents across the system and collect the data. When the networks or tracing requirements change, all the above control information can be modified or reconfigured, and then resent to the monitoring nodes during the system runtime. Such a process provides high programmability and flexibility for monitoring the dynamic virtualized networks.

E. Implementation

To enable tracing across boundaries, we added 4-byte variable `option` in `tcp_out_options` for the TCP header and allocated 4-byte space for the UDP header. The unique ID is written when packets are sent through `tcp_options_write` or `udp_send_skb`. To mitigate the local storage overhead, we implemented a kernel module on each monitoring node to temporarily store tracing data and used `/proc` file system to avoid kernel and user space data copies. The prototype of vNetTracer is implemented in C and Python. Specifically, the agents are daemon processes, which are woken up once receiving new tracing scripts. The backend of each tracing script is implemented in C, which executes actions inside the kernel and collects the tracing data. The frontend is implemented in Python, which completes the initialization, stores local data, and periodically sends the tracing data to the collector. The control data dispatcher consists of a frontend, which reads the user input from terminal and generates the formatted configuration files, and a client side that sends the configured tracing scripts to the remote agents. The raw data collector consists of a daemon,

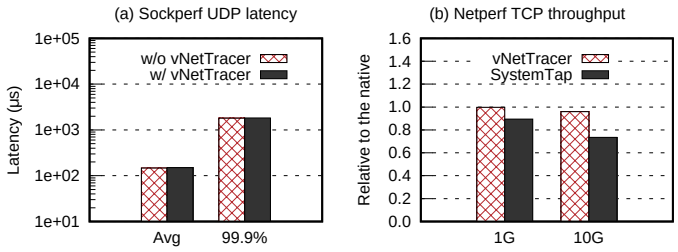

Fig. 7: vNetTracer incurs negligible overhead to (a) application average latency, tail latency, and (b) throughput. Compared to SystemTap, the overhead is marginal due to the eBPF benefits and the optimizations to vNetTracer.

which receives tracing data from the agents, and the database operation functions to calculate the network metrics. Both the control data dispatcher and the raw data collector are implemented in Python. We adopt *InfluxDB* for the offline storage and create tables for each tracepoint.

IV. Evaluation

In this section, we first evaluate the overhead of vNetTracer to show its high performance. Next, we demonstrate the utility of vNetTracer with three case studies. The first case study is to show critical path analysis in Open vSwitch with vNetTracer. The second case study is using vNetTracer to tune the Xen hypervisor scheduler for a long tail latency issue in a highly consolidated environment. The third case study shows how we identify the network bottlenecks inside a container architecture using vNetTracer.

A. Evaluation Settings

Our experiments were performed on two DELL PowerEdge T430 servers, connected by a one-Gigabit Ethernet and a ten-Gigabit Ethernet. Each server was equipped with a dual ten-core Intel Xeon E5-2640 2.6GHz processor, 64GB memory and a 2TB 7200RPM SATA hard disk. The ten-Gigabit NIC is Intel x540. We used Ubuntu 16.10 and Linux kernel 4.10 as the host and guest OS. We used Open vSwitch 2.6.0 to connect various VMs on the same host. The hypervisor we adopted is KVM 2.6.1 or Xen 4.8.1, and the Docker version is 1.12.1. The evaluation setting details of individual case studies are slightly different and further discussed in the respective sections.

B. Overhead Analysis.

Overall overhead. We first analyzed the overhead of vNetTracer on application network performance. We created two VMs using KVM on two servers and configured each VM with 4 vCPUs and 4GB memory. We pinned the vCPU of the VMs to different physical CPU cores to avoid the interference. First, we executed *Sockperf* [12] client side on one VM and sent UDP requests to the Sockperf server side on another VM to measure the average and tail latency. Then we booted vNetTracer to trace the Sockperf performance. We executed four tracing scripts and attached them into the Open vSwitch port `ovs-br1` in the hypervisor and virtual ethernet port `ens3` in the VM on the two physical servers. Figure 7(a) plots the average and 99.9^{th} percentile latency of Sockperf UDP packets with and without vNetTracer execution. As shown in the Figure 7(a), both average and tail latency of Sockperf were not influenced significantly with vNetTracer. Compared to the default performance without vNetTracer, the average latency with vNetTracer increased less than 1%, and no traffic burst happened during the tail latency measurement. Meanwhile, our tracing also showed that vNetTracer did not introduce additional network packet loss for the applications.

Comparison with SystemTap. We also compared the performance of vNetTracer with SystemTap. We built a VM which had one vCPU and 4GB memory on Xen and executed the Netperf server inside the VM. A Netperf client was sending TCP packets on another physical server. We wrote a SystemTap script attached at `tcp_recvmsg` to get the network packet. We executed the SystemTap with option `STP_NO_OVERLOAD` to disable the tracing overhead threshold. In comparison, we used vNetTracer to attach the same kernel function and trace the network. As shown in Figure 7(b), due to the benefit of eBPF and our optimizations, the throughput of Netperf degraded insignificantly when tracing under vNetTracer. However, SystemTap tracing introduced around 10% performance loss. As explained in Section II, the frequency of traces and the continual data copies between the kernel space and user space introduced such overhead with SystemTap. vNetTracer traces the network inside the kernel and keeps the tracing data in memory. For I/O-bound applications with high load, such the overhead cannot be neglected. We also evaluated the Netperf performance on a 10G network and SystemTap introduced 26.5% performance loss due to high frequency of traces and inefficient data copies.

C. Case Study I: Network Delay in the Open vSwitch

Open vSwitch (OVS) is a virtualized network switch, which provides high quality packet switching for virtualized networks and is widely adopted in the current cloud. In this section, we first describe the network delay inside the OVS and discuss the challenges to diagnose the problem. Next, we talk about how vNetTracer helps us analyze and locate the issue. Lastly, based on the tracing information, we share a simple yet effective solution to mitigate the above issue of Open vSwitch.

In order to describe the network delay inside the OVS, we created three VMs on a single physical server. The hypervisor we used was KVM and all the VMs were connected through OVS. The VMs were configured with four vCPUs and 4GB memory. As shown in Figure 8(a), we executed the Sockperf and iPerf [7] clients on VM0, another iPerf client on VM1, and the Sockperf server as well as two iPerf servers on VM2. As a comparison baseline, we only run the Sockperf application to measure the latency in an uncongested network, which is denoted as Case I in Figure 8(b). Next, we run the iPerf client with Sockperf client simultaneously on the same VM and record the Sockperf latency as Case II. Finally, we add the second iPerf client on another VM based on Case II and denote such a scenario as Case III.

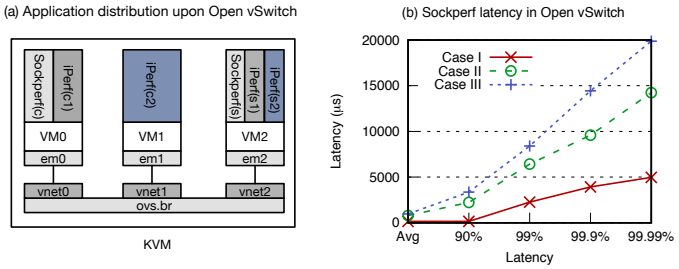

Fig. 8: Evaluation setting and Sockperf latency in OVS. The 'C' and 'S' denote the client and server side, respectively.

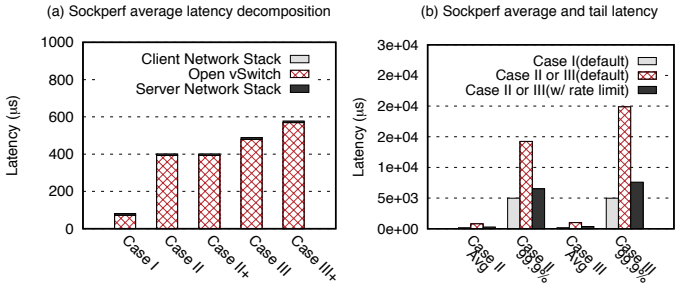

Fig. 9: Open vSwitch latency decomposition and analysis.

As illustrated in Figure 8(b), the tail latency of Sockperf in Case II and Case III increased significantly compared to the latency in the uncongested network. Similar problems can also be observed in the physical switches [34], [58]. The reasons behind the issues are extremely difficult to analyze and locate as many factors along the data path can introduce the additional overhead. For example, the latency-intensive and throughput-intensive applications in Case II share both the client and the server network stack, which might incur long tail latency for Sockperf. The existing tracing techniques either focus on a traditional environment, such as physical network links, or lack of ability to differentiate network flows and locate the congested part. To analyze the bottlenecks in the virtualized network path, we executed agents of vNetTracer on both the VMs and host machine, and bound the tracing scripts at application sockets `em` and OVS ports `vnet`. We used eBPF scripts to filter the Sockperf packet and decompose its latency into three parts: the time spent inside the sender network stack, the OVS and the receiver network stack. As the latency decomposition shown in Figure 9(a), the time spent inside the OVS dominated the total transmission time. As more applications occupied the network path, the network became increasingly congested and the time in the OVS increased.

To better understand the network delay inside the OVS, we add more iPerf clients on VM0 based on Case II as Case II+. Similarly, we add more iPerf clients on additional VMs and denote that as Case III+. As shown in Figure 9(a), the time gap between Case I and Case II is due to the queueing delay in the OVS. Multiple applications (e.g., Sockperf and iPerf) send network packets at the same ingress port of OVS and the delivery speed of OVS falls far behind the packet incoming speed. Such a gap does not increase when we added more the application clients on VM0 in Case II+ because the queue at ingress is highly saturated. In comparison, the time gap between Case II and Case III, which results from the processing delay that OVS needs to switch the network flows from different ingress ports, increased when more clients are sending packets through more OVS ingress ports in Case III+. Both the above two delays make the network flow load much larger than the OVS processing ability and introduce additional network latency inside OVS. Compared to the delay inside OVS, the time spent inside the client or server network stack did not increase significantly.

To mitigate the above issues, one potential solution is to limit the network flow rate at the OVS ingress. Rate limiting sets the maximum packet transmission rate at the virtual ingress port of OVS, which just simply drops the network packets above the rate and can thus limit the packet transmission number from VMs to the OVS. In our experiment, we set the `ingress_policing_rate` as 1×10^5kbps and `ingress_policing_burst` as 1×10^4kb at both `vnet0` and `vnet1` of the OVS. Then we repeated the above experiments in Case II and Case III. As shown in Figure 9(b), both the average and tail latency of Sockperf decreased significantly with rate limit in the OVS. As the default Sockperf packet size was just 56 bytes, the workflow of iPerf was mainly limited when its packets entered the OVS. Therefore, both the queueing delay and OVS processing delay were mitigated with the rate limit. The setting in the above experiments was not the optimal configuration and just used to show the effectiveness. In addition to the rate limit, we also tried setting QoS policy with Hierarchy Token Bucket (HTB) at the virtual port of OVS, which limited the clients saturating the network bandwidth. The effect was similar as the results using rate limit shown in Figure 9(b).

Summary. Existing tools either lack of ability to differentiate the network with complex flows or cannot decompose the long tail latency into different components along the data path. In comparison, vNetTracer can filter and monitor the target network flow, and locate the potential congested component in the virtualized networks efficiently.

D. Case Study II: Tuning the Scheduler in Hypervisors

Credit2 [18] is a new generation of general purpose scheduler for the hypervisor Xen, which is designed with focus on fairness, responsiveness as well as scalability. In this section, we first describe the issue we found in the current credit2 scheduler. Then, we talk how we located the issue with vNetTracer and solved it through tuning the scheduler.

We created two VMs on a single physical server. The hypervisor we used was Xen 4.8.1 and the VMs were configured with one vCPU and 4GB memory. The hypervisor scheduler was set as credit2 inside of Xen and the client side was executed on another physical server. All the applications were running within containers on the VMs. First, we executed the Sockperf server side on one VM and sent requests to measure the latency as the baseline. Next, we executed a

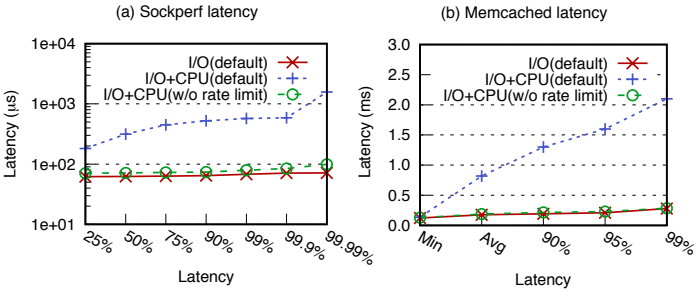

Fig. 10: Network latency in the unconsolidated and highly consolidated virtualized environments.

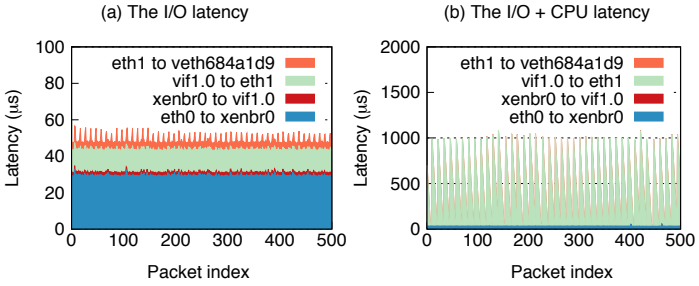

Fig. 11: Latency decomposition when the Sockperf VM runs alone or shares the physical CPU with a CPU-bound VM.

loop on another VM and pinned the vCPU of the two VMs on the same physical CPU core. As shown in Figure 10(a), the latency of Sockperf increased dramatically when the I/O-bound VM shared the CPU resources with the CPU-bound VM. For instance, the 99.9^{th} percentile latency increased 22x compared to the baseline. Besides, we also chose Data Caching from Cloudsuite benchmark suites [1] to evaluate this issue. The server side of Data Caching executed *Memcached* which simulated the behavior of a Twitter caching server using the Twitter dataset. On the client side, we set up 4 worker threads executing 20 connections to send the requests and the ratio of GET/SET requests was configured as 4:1. We set a fixed request rate as 5000 rps to measure the request latency. As depicted in Figure 10(b), similar to the results of Sockperf, the average and tail latency of memcached increased 4.7x and 7.5x respectively compared to the baseline. The greatest challenge to analyze this issue is the multi-layer virtualization, including the hypervisor, the guest OS and the containerized applications, and the complicated virtualized network along the software stack. These virtualized boundaries make many traditional tools, such as Xentrace [15], DTrace [2], ineffective.

In order to analyze this problem, we executed the agents of vNetTracer at the client, Dom0 and server side VM to trace the network. We bound the tracing scripts at the following network interfaces: ethernet port `eth0` in the client side, network bridge `xenbr0` and backend `vif1.0` in Dom0, ethernet port `eth1` and container virtual ethernet port `veth684a1d9` in the server VM. We decomposed the packet latency based on the above setting and repeated the experiments. As shown in Figure 11(a), when the I/O-bound VM executed alone, the client-to-server transmission delay dominated the one way latency. In comparison, when the I/O-bound VM shared the same CPU core with the CPU-bound VM, the time spent between the backend `vif1.0` in Dom0 and frontend `eth1` in the server VM took more than 90% of the one way latency. As the data path was the same as in Figure 11(a), this indicated that scheduling delays inside of Xen caused the issue. We checked the source code in Xen credit2 scheduler and found the vCPU priorities used in credit1, such as *OVER*, *UNDER* and *BOOST*, were all removed and all the vCPUs were just ordered by their credit. Then we traced vCPU credit and the data showed that the credit of the I/O-bound VM vCPU was always larger than the credit of the CPU-bound VM vCPU, which indicated that the scheduling order of vCPUs had no problem. We further analyzed the tracing data and found that the scheduling delay first increased up to $1000\mu s$. Then the scheduling delay descended for the next few packets and such process repeated periodically as shown in Figure 11(b). That reminded us of the scheduling rate limit inside the Xen credit2 scheduler, which is set as $1000\mu s$ by default.

The rate limit [2] was introduced into the hypervisor credit scheduler since Xen 4.2 [17]. In order to avoid too many schedulings and context switches, the scheduler sets the minimum amount of time which a VM is allowed to run without being preempted, even through a woken VM has higher priority. This mechanism performs well and does not harm the throughput of most network applications. However, the average latency as well as the tail latency of many online applications is highly interfered with such a mechanism, especially in the highly consolidated virtualized environments. In addition, the jitter of the I/O application also increased significantly. For instance, the range of jitter in Figure 11(a) was only (-7.2μs, 9.2μs) while the value grew to (-117.8μs, 1041.4μs) in Figure 11(b). To mitigate such issues, we tried to tune the rate limit as 0 in Xen credit2 scheduler. As shown in Figure 10(a) and (b), the network latency with rate limit disabled is close to the baseline even though the I/O-bound VM runs simultaneously with the CPU-bound VM on the same physical core. Such a solution also works for the same issue in credit1 scheduler inside Xen. We reported our findings to the Xen open source community and the above issues were confirmed by engineers from Citrix [9].

Summary. Unlike many tools limited in monitoring within certain ranges, vNetTracer can efficiently trace the application end-to-end performance in the virtualized networks and associate the network activities across hardware and software boundaries in isolated domains.

E. Case Study III: Bottlenecks of the Container Architecture

Containers are widely used in the cloud nowadays and the overlay network is one of the extensively adopted infrastructures to support the container communication across multiple

[2]The rate limit in Case Study I refers to limiting the application packet sending rate while the rate limit here refers to the minimum scheduling time slice inside the Xen hypervisor.

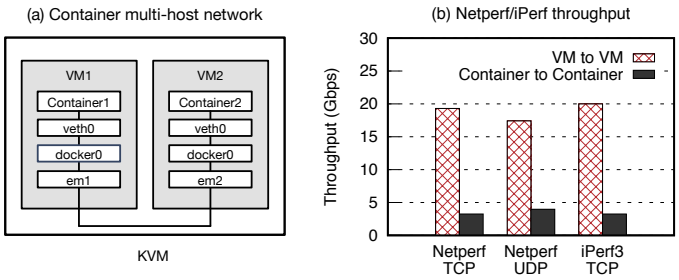

Fig. 12: Container multi-host networking setting and the Netperf/iPerf throughput of VMs versus containers.

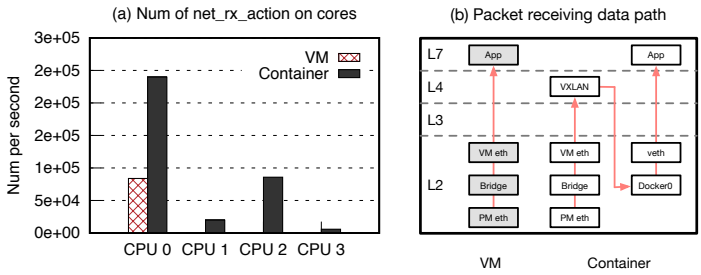

Fig. 13: Software interrupt distribution among multi cores and network data path in VMs or containers.

hosts. Although overlay networks bring lots of benefits, such as ease of use, independence from the underlying architecture, etc., they also introduce many new issues. For instance, the performance of the overlay network is usually much worse than the other kinds of container networks [53]. Moreover, it is also difficult to monitor the overlay network as the packets are encapsulated with the underlying network information. In this section, we describe how vNetTracer helps us analyze the bottleneck in a container overlay network.

As depicted in Figure 12(a), we created two VMs using KVM on a single physical server. Each VM was configured with 4 vCPUs, 4GB memory and a virtio NIC. Inside the VMs, we used Docker to create some containers executing the network applications. In order to connect the containers on the two VMs, we built a default Docker overlay network and used *etcd* [3] 2.2.5 as the distributed key value store. The overlay network used VXLAN to encapsulate the original network packets. We used Netperf and iPerf to measure the throughput among VMs or containers. As depicted in Figure 12(b), the TCP throughput between containers decreased significantly compared to the VM throughput. For instance, the Netperf TCP and UDP throughput between containers were just 16.8% and 22.9% of that between VMs, which indicated significant overhead in the container overlay network.

To understand the potential issue inside the container overlay network, we first analyzed the network rate in the virtualized network stack. We attached tracing scripts on the kernel function `net_rx_action`, which is the default *softirq* handlers when receiving network packets. As Figure 13(a) shows, although the throughput of containers is far less than that of VMs, the execution rate of `net_rx_action` in containers is 4.54 times of that in VMs. This indicated that more *softirqs* happened when receiving network packet in the container network. Why do additional interrupts introduce such significant overhead? First, additional interrupts incur lots of context switches. As revealed by Peter et al. [44], the scheduling overhead differs by up to 14x difference depending on whether the receiving process is currently running. The time to context-switch to the server process from the idle process has more than 10x impact on a receiving process. Second, too many interrupts might cause lots of sleep and wakeup operations to the *ksoftirqd*, which is a daemon thread that executes on each CPU core to handle the software IRQs. The interrupt processing of *ksoftirqd* will be significantly interfered if too many sleeps and wakeups consume lots of CPU cycles [39].

Besides the network rate, we also used vNetTracer to analyze the *softirq* distribution and track packet data path inside container networks. We attached tracing script at the kernel function `get_rps_cpu` to get which CPU core the *softirq* is processed on. As Figure 13(a) shows, most *softirqs* are concentrated to a few CPU cores. For instance, 99.7% and 62.9% of the `net_rx_action` is executed on CPU 0 in VMs and containers, respectively. However, as mentioned above, the number of *softirqs* in containers is far more than that in VMs. Therefore, such concentrated *softirq* distribution and lack of multi-core usage heavily limited the network speed in containers. The reason is due to the fact that those software interrupts come from the same hardware interrupt on the NIC. To keep the cache hot and take full advantage of data locality, OSes process the *softirqs* from the same source on certain cores, which significantly degrades container network performance with large number of software interrupts. Receive Packet Steering (RPS), which helps balance network packets on different CPU queues, is also limited for accelerating the container networks. As RPS only balances the packets based on its IP addresses and port numbers and packets of one containerized application connection has the same IP and port combination, all the software interrupts of one application are processed in the same CPU and the container network performance will not benefit with RPS enabled.

To analyze the packet data path, we attached multiple eBPF scripts at different layers in the virtualized network stack to track the source and destination of the network packets. The device ID and name are acquired by the `ifindex` and `name` in `net_device` of the skb. Note that the tracing scripts need to strip the VXLAN header off to read the skb information in container networks. As shown in Figure 13(b), the data path in container networks is far more complex than that in VMs, which also implicated the enormous amount of software interrupts in Figure 13(a). Different from the normal packet processing inside VMs, the packets travel across different layers repeatedly in the container networks. This is due to the overlay network architecture which abstracts additional virtualized network layers on top of the VM network. During traveling in such much deeper and more complex container

networks, additional efforts, such as security checks, header operation, network forwarding, etc., are needed for the packets [44]. All efforts consume additional resources and slow down the network processing. In addition, vNetTracer also revealed many other details such as the bottlenecks at the Docker bridge `docker0`, the latency at the VXLAN device, the additional acknowledgment overhead for TCP transmission, etc. Although optimizing container networks is a new and complex topic, the valuable statistics provided by vNetTracer can still reveal the potential reasons behind the poor container network performance in Figure 12(b) and shed light on solving this problem in our future work.

Summary. Characterizing the Linux kernel, especially for analyzing network performance, is challenging. With vNetTracer, we can attach user-defined eBPF programs into the systems and instrument the runtime environment in a highly efficient and customized manner.

V. Related Work

Monitoring Based on System Logs. Many tools or monitoring systems [4], [8], [19], [22], [48], [49] provide non-intrusive tracing, which leverages existing application logs and performance counters, to diagnose performance issues and detect bugs in complex systems. However, tracing based on existing system logs or static tracepoints cannot guarantee providing what the users are interested in. In addition, too many logs might also introduce lots of overhead and overwhelm the valuable information. Statistical approaches, e.g., data mining, machine learning [28], [43], [51], [57], [56], are promising directions towards automated identification of the issues inside the systems based on the massive tracing logs, the execution environment as well as the application performance. For example, Chow et al. proposed the Mystery Machine [28], which analyzed traces of over 1.3 million requests collected over 30 days in Facebook and generated the system dependency model and calculated the critical path. However, such solutions are still based on existing system logs or tracing information, which lack of the flexibility for satisfying the dynamic user requirements and service changes, especially in virtualized networks. Different from the above works, vNetTracer adopts user-defined eBPF programs to trace the system, which not only is highly efficient, but also provides customized system information of the virtualized networks.

Dynamic Instrumentation. Annotation based monitoring [23], [27], [32], [50] allows users to selectively trace the systems based on their purposes. For example, Pip [47] provided application level annotation monitoring while Pinpoint [27] and X-Trace [32] added annotations in the libraries and middleware software. However, such approaches not only are still lack of the flexibility in the dynamic monitoring, but also face the challenges of balancing the tradeoff between the tracing efficiency and its overhead. Dynamic instrumentation allows logging and tracing to be installed dynamically and flexibly. Pivot Tracing [41] leveraged aspect-oriented programming to export variables for dynamic tracing and designed a query languages to selectively invoke user-defined tracepoints. However, Pivot Tracing is designed specifically for Hadoop and is used to trace distributed applications only in user space. Instead, vNetTracer focuses on dynamic tracing through the entire virtualized network stack. Many other system tools, such as *SystemTap* [13], *DTrace* [25], also support dynamic instrumentation. However, SystemTap introduces much overhead for high frequency tracing, which is not suitable for virtualized network instrumentation. DTrace is a troubleshooting tool in the OSes like Solaris, FreeBSD, etc., and it does not support system tracing in the Linux.

Tracing in Distributed Systems. In order to diagnose performance problems, numerous tracing tools are designed and used in today's complex systems. However, many tools, such as gperf [5], Perf [8], SystemTap, DTrace, Xentrace and Xenalyze [15], can only trace within certain boundaries. To address such limitations and enable monitoring in distributed systems, many efforts [23], [26], [27], [32], [50], [54] have been proposed to associate events across the hardware and software boundaries. For instance, Whodunit [26] uses a compact representation of a transaction context named synopsis to profile transactions across distributed machines. Some other papers [41], [45], [46] proposed to add the tracing information along with the requests or packets to break the isolation. For example, Pivot Tracing uses a per-request container called baggage to correlate logging information with a particular request context. Appinsight [45] and Timecard [46] try to locate the latency bottleneck for mobile networks and their idea is to add time information into the packet during the transmission. Unlike these efforts, vNetTracer does not require to add predefined tracepoints and the monitoring processes can be programmed at runtime to handle the complex and dynamic tracing in virtualized networks.

VI. Conclusion

It is important to trace network performance in order to guarantee application performance and analyze potential issues. However, virtualized networks make that increasingly challenging as they introduce additional complex infrastructure and changes dynamically. This paper presents *vNetTracer*, a highly efficient and programmable profiler that traces network performance in the virtualized systems. vNetTracer incurs negligible overhead to network performance and does not require any changes to the user level applications for end-to-end network tracing. In addition, it can also be programmed and configured at runtime to satisfy various tracing requirements. Our evaluation and case studies demonstrated that vNetTracer shed light on the virtualized network monitoring and can help users analyze, identify and localize potential issues inside virtualized networks.

Acknowledgement

We are grateful to the anonymous reviewers for their comments on this paper and our shepherd Malte Schwarzkopf for his suggestions. This work was supported in part by U.S. NSF grants IIS-1633753.

REFERENCES

[1] *Cloud suite benchmarks*. http://cloudsuite.ch/.

[2] *DTrace*. https://dtrace.org/.

[3] *Etcd*. https://github.com/coreos/etcd/.

[4] *Ftrace*. http://elinux.org/Ftrace.

[5] *GNU gperf*. https://sourceware.org/binutils/docs/gprof/.

[6] *IO Visor*. https://www.iovisor.org/.

[7] *iPerf*. https://iperf.fr/.

[8] *Linux Perf*. https://perf.wiki.kernel.org/.

[9] *Long tail latency caused by rate-limit in Xen credit2*. https://lists.xenproject.org/archives/html/xen-devel/2017-06/msg01410.html.

[10] *Observe your system with perf, ftrace, eBPF and systemtap*. https://bit.ly/2JlY1kR.

[11] *Offloading OVS Flow Processing Using eBPF*. https://bit.ly/2F8DJIP.

[12] Sockperf. https://github.com/Mellanox/sockperf.

[13] *SystemTap*. https://sourceware.org/systemtap/.

[14] *SystemTap Performance measurement*. https://elinux.org/System_Tap.

[15] *Tracing with Xentrace and Xenalyze*. https://goo.gl/6Ep9fz.

[16] *Using eBPF to Accelerate OVS Datapath*. https://bit.ly/2JeocK7.

[17] *Xen Context-Switch Rate Limiting*. https://goo.gl/GHnNss.

[18] *Xen Credit2 Scheduler*. https://goo.gl/nzvS5v.

[19] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of ACM SOSP*, 2003.

[20] Z. Ahmed, M. H. Alizai, and A. A. Syed. Inkev: In-kernel distributed network virtualization for dcn.

[21] Z. Ahmed, M. H. Alizai, and A. A. Syed. Inkev: In-kernel distributed network virtualization for dcn. In *ACM SIGCOMM Computer Communication Review*, 2016.

[22] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *Proceedings of SIGCOMM*, 2007.

[23] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *Proceedings of OSDI*, 2004.

[24] G. Bertin. XDP in practice: integrating XDP into our DDoS mitigation pipeline. In *Technical Conference on Linux Networking, Netdev*, 2017.

[25] B. Cantrill, M. W. Shapiro, A. H. Leventhal, et al. Dynamic instrumentation of production systems. In *Proceedings of USENIX ATC*, 2004.

[26] A. Chanda, A. L. Cox, and W. Zwaenepoel. Whodunit: Transactional profiling for multi-tier applications. In *Proceedings of EuroSys*, 2007.

[27] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of ACM DSN*, 2002.

[28] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch. The mystery machine: End-to-end performance analysis of large-scale internet services. In *Proceedings of USENIX OSDI*, 2014.

[29] F. Cristian. Probabilistic clock synchronization. *Distributed computing*, 1989.

[30] F. C. Eigler and R. Hat. Problem solving with systemtap. In *Proc. of the Ottawa Linux Symposium*, 2006.

[31] Ú. Erlingsson, M. Peinado, S. Peter, M. Budiu, and G. Mainar-Ruiz. Fay: extensible distributed tracing from kernels to clusters. *Proceedings of TOCS*, 2012.

[32] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of USENIX NSDI*, 2007.

[33] B. Gregg. *Systems performance: enterprise and the cloud*. 2013.

[34] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. Watson, A. W. Moore, S. Hand, and J. Crowcroft. Queues don't matter when you can jump them! In *Proceedings of USENIX NSDI*, 2015.

[35] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. Secondnet: a data center network virtualization architecture with bandwidth guarantees. In *Proceedings of CoNEXT*, 2010.

[36] F. Hao, T. Lakshman, S. Mukherjee, and H. Song. Secure cloud computing with a virtualized network infrastructure. In *Proceedings of USENIX HotCloud*, 2010.

[37] J. Hwang, K. Ramakrishnan, and T. Wood. Netvm: high performance and flexible networking using virtualization on commodity platforms. In *Proceedings of USENIX NSDI*, 2014.

[38] S. P. Kavulya, S. Daniels, K. Joshi, M. Hiltunen, R. Gandhi, and P. Narasimhan. Draco: Statistical diagnosis of chronic problems in large distributed systems. In *Proceedings of ACM DSN*, 2012.

[39] J. Khalid, E. Rozner, W. Felter, C. Xu, K. Rajamani, A. Ferreira, and A. Akella. Iron: Isolating network-based CPU in container environments. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[40] A. Kumar, S. Jain, U. Naik, A. Raghuraman, N. Kasinadhuni, E. C. Zermeno, C. S. Gunn, J. Ai, B. Carlin, M. Amarandei-Stavila, et al. Bwe: Flexible, hierarchical bandwidth allocation for wan distributed computing. In *Proceedings of ACM SIGCOMM*, 2015.

[41] J. Mace, R. Roelke, and R. Fonseca. Pivot tracing: dynamic causal monitoring for distributed systems. In *Proceedings of ACM SOSP*, 2015.

[42] S. McCanne and V. Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *USENIX Winter*, 1993.

[43] K. Nagaraj, C. Killian, and J. Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of USENIX NSDI*, 2012.

[44] S. Peter, J. Li, I. Zhang, D. R. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *Proceedings of USENIX OSDI*, 2014.

[45] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. Appinsight: Mobile app performance monitoring in the wild. In *Proceedings of OSDI*, 2012.

[46] L. Ravindranath, J. Padhye, R. Mahajan, and H. Balakrishnan. Timecard: Controlling user-perceived delays in server-based mobile applications. In *Proceedings of ACM SOSP*, 2013.

[47] P. Reynolds, C. E. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the unexpected in distributed systems. In *Proceedings of USENIX NSDI*, 2006.

[48] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat. Wap5: black-box performance debugging for wide-area systems. In *Proceedings of WWW*, 2006.

[49] R. R. Sambasivan, A. X. Zheng, M. De Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger. Diagnosing performance changes by comparing request flows. In *Proceedings of USENIX NSDI*, 2011.

[50] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. *Google research*, 2010.

[51] J. Snee, L. Carata, O. R. Chick, R. Sohan, R. M. Faragher, A. Rice, and A. Hopper. Soroban: attributing latency in virtualized environments. In *Proceedings of USENIX HotCloud*, 2015.

[52] K. Suo, J. Rao, L. Cheng, and F. C. M. Lau. Time capsule: Tracing packet latency across different layers in virtualized systems. In *Proceedings of the 7th ACM APSys*, 2016.

[53] K. Suo, Y. Zhao, W. Chen, and J. Rao. An analysis and empirical study of container networks. In *Proceedings of IEEE Conference on Computer Communications (INFOCOM)*, 2018.

[54] E. Thereska, B. Salmon, J. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez, and G. R. Ganger. Stardust: tracking activity in a distributed storage system. In *Proceedings of ACM SIGMETRICS*, 2006.

[55] A. Traeger, I. Deras, and E. Zadok. Darc: Dynamic analysis of root causes of latency distributions. In *Proceedings of ACM SIGMETRICS*, 2008.

[56] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of ACM SIGMOD*, 2017.

[57] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of ACM SOSP*, 2009.

[58] Y. Xu, M. Bailey, B. Noble, and F. Jahanian. Small is better: Avoiding latency traps in virtualized data centers. In *Proceedings of the ACM SoCC*, 2013.

[59] X. Zhao, Y. Zhang, D. Lion, M. F. Ullah, Y. Luo, D. Yuan, and M. Stumm. lprof: A non-intrusive request flow profiler for distributed systems. In *Proceedings of USENIX OSDI*, 2014.