EZPath: Expediting Container Network Traffic via Programmable Switches

Zili Zha*, An Wang†, Yang Guo‡, Qun Li§, Kun Sun*, Songqing Chen*
George Mason University*, Case Western Reserve University†, NIST‡, College of William and Mary§

***Abstract*—Containerization, while getting popular in data centers, faces practical challenges due to the sharing nature of cloud networks among tens of thousands containers simultaneously and dynamically. While the typical overlay approach enables network virtualization to facilitate multi-tenant isolation and container portability, this approach often suffers from degraded performance. Other proposed schemes addressing this performance bottleneck require either specialized hardware support, or customized software and extra maintenance. In this paper, we propose *EZPath*, a novel approach that can seamlessly expedite the container traffic by leveraging the programmable Top-of-Rack (ToR) switches in clouds. By utilizing the underlying programmable switch's data plane capabilities, *EZPath* can offload traffic directly from the container to the ToR switches, thus creating a fast and easy path to mitigate the network bottleneck. Such a ToR switch based solution is transparent to user applications, and does not require the change of OS kernel or the support of additional hardware. Using typical container workloads, we evaluate *EZPath*, and the results show that *EZPath* can significantly improve the application performance over the default overlay networking, e.g., a 35% throughput increase and a 42% tail latency reduction for Memcached.**

I. Introduction

In recent years, containerization has been increasingly adopted to deploy large-scale distributed applications (e.g., content providers [1], [2], eCommerce [3], [4], and in-memory key-value stores [5]) in clouds, such as AWS Lambda [6], Google Compute Platform [7], and Microsoft Azure [8] [1]. However, container networking has been suffering in such a multi-tenant environment, particular with the increasing deployment scale due to the following reasons. First, the sharing nature of multi-tenant cloud networks requires tenant isolation and quality of service (QoS) through the enforcement of thousands of control plane policies (e.g., access control) and data plane policies (e.g., tunneling, QoS and rate limiting), resulting in significant host computing resource consumption. The sheer density of the container deployment and its short-livedness further exacerbate the problem [9], [10]. Second, the container networking should be able to provide portability and flexibility for container placement and migration, obviating the need for the application developers to coordinate the assignment of port and IP addresses.

Existing container orchestration solutions mostly employ virtual overlay networks to achieve portability and flexibility. Essentially, overlays employ various tunneling technologies, e.g., VXLAN, GRE, to implement virtual networking among the containers owned by a single tenant, providing tenant isolation and network virtualization. One example of such software entities is Open vSwitch [11], which is widely deployed in data center servers to enable network virtualization on end hosts [12]–[15]. However, as shown in previous works [16]–[18], implementing the tunneling and other network virtualization functionalities in the software switch causes significant networking performance degradation.

The poor performance of overlay networking solutions has remained to be a pressing issue ever since virtualization came into play. In VM-based virtualization environments, Single Root I/O Virtualization (SR-IOV) has shown effectiveness in improving the networking performance [16]. Nonetheless, the number of VMs that a commodity server can accommodate is far less than that of containers in contemporary containerized data centers. More recently, some other designs [9], [17], [19] have been proposed to improve container network performance, while aiming to preserve its portability. However, they are difficult to deploy due to the requirement of customized software and extra maintenance [17], or specialized hardware support [9], [16]. On the other hand, network programmability has become a key feature in modern networking. Broadcom, Cisco and Edgecore, are expanding their portfolios to offer programmable Top-of-Rack (ToR) switches for data centers [20]–[22].

In this work, we propose to develop an efficient and application transparent framework, called *EZPath*, to expedite the container network traffic, by leveraging the programmable data planes of the prevalent Software Defined Networking (SDN) switches in data centers. We achieve this by offloading selective network traffic to in-network programmable switches to relieve the system resource pressure on the servers. The performance of the containerized applications is significantly improved in benefit of the programmability and line rate processing speed of modern switches (e.g., P4).

Nonetheless, due to the specific requirements and constraints in the containerized environment, migrating network functions and traffic to the programmable hardware poses several challenges. First, due to the sheer scale and density of containers in deployment, the amount of memory for

[1]Certain commercial equipment, instruments, or materials are identified in this paper in order to specify the experimental procedure adequately. Such identification is not intended to imply recommendation or endorsement by the National Institute of Standards and Technology, nor is it intended to imply that the materials or equipment identified are necessarily the best available for the purpose.

ISBN 978-3-903176-48-5 ©2022 IFIP

accommodating the metadata (i.e., the tunnelling mappings) required by *EZPath* can be substantial. Therefore, simply offloading the tunnelling operations for all container traffic is not practical, given the resource constraints imposed by programmable switch ASICs.

Second, containers usually have much shorter lifespans than virtual machines in many application scenarios, such as microservice deployment and serverless computing. Blindly offloading all traffic may cause constant update of offloading selections and tunnel mappings, which could potentially degrade the overall network performance. Therefore, we need an optimal strategy that strikes the balance between network performance and resource consumption.

To address these challenges, we design *EZPath*, a holistic framework that optimizes the performance of container network virtualization through hardware-assisted acceleration. *EZPath* is featured with a software-hardware codesign that incorporates the control plane software and the dataplane hardware. The control plane determines the offloading strategies in real time. The offloading strategy is then translated into P4 instructions executed in the programmable switches. *EZPath* can flexibly and seamlessly migrate the network virtualization functionalities flows to the ToR switches. Note that not only inter-rack traffic benefits from *EZPath*, but also intra-rack traffic, because it also goes through the ToR switches when containers are hosted on separate physical servers within the same rack.

While details are to be presented later in the paper, the highlights of *EZPath* include:

- We quantitatively evaluate the overhead of the overlay approach for container networking, and show such overhead contributes significantly to the network bottleneck.
- Taking a software-hardware co-design approach, we design and implement *EZPath* that can offload network traffic of interest by leveraging the programmable data planes of modern SDN switches.
- *EZPath* is application transparent, preserving compatibility with legacy containerized applications. It does not require changes in host kernel, making it compatible with all existing network management tools.
- The evaluation results show that *EZPath* can expedite container traffic significantly, e.g., with a 35% improvement on throughput and a 42% tail latency reduction for Memcached.

II. Motivation

In this section, we first investigate the overhead of the overlay networking approach, and discuss other alternatives for improvement, which motivates the design of *EZPath*.

A. Overhead of Overlay Networking

To understand the performance issues of virtual switch based network virtualization, we perform an overhead breakdown analysis of a popular container network solution: Docker Overlay [23]. Docker overlay utilizes a VXLAN data plane that decouples the container network from the physical underlay network. A virtual Linux bridge is created per overlay along with its associated VXLAN interfaces. As depicted in Figure 1, our testbed consists of two KVM virtual machines (VMs) on a single physical host. The VMs are used to simulate two physical hosts. Each VM is configured with 2 vCPUs, 2 GB memory and a virtio NIC. The overhead remains the same in a physical machine environment as long as the underlay host network is not the bottleneck. We create a Docker container inside each VM, which runs network performance benchmarks Netperf [24] and iperf3 [25]. We then deploy a Swarm mode overlay network to connect the containers. To quantify the overhead, we compare the performance of the container overlay to the host mode network, where the container network stack is not isolated from the host stack and the host IP is directly allocated to the container. We measure the TCP/UDP throughput when sending data as fast as possible from the client to the server.

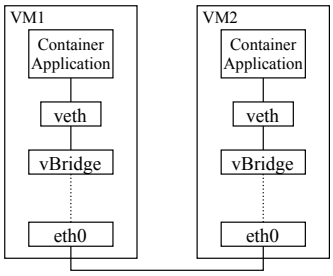


Fig. 1: Experiment setup

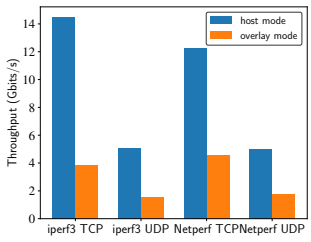


Fig. 2: Throughput overhead of container overlay

Figure 2 shows the results. The performance of iperf3 is shown on the left half while those of NetPerf is shown on the right half. Each experiment is repeated five times and the average results are reported here. As shown in Figure 2, the TCP throughput of the container overlay network is only 26.4% and 36.2%, respectively, when compared to the native host networking for iperf3 and Netperf. The trend of overhead for the UDP throughput is similar. UDP does not have congestion control, which results in lower throughput than TCP in iperf. We further use NPtcp [26] to measure the latency of TCP packets. The results show that the virtual switch based tunneling incurs 46.7% more latency on average (26.19 μs and 38.48 μs for the host and overlay modes, respectively).

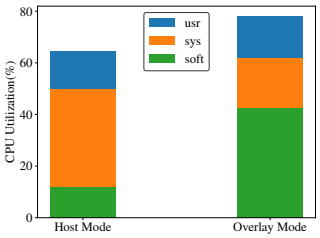


Fig. 3: CPU utilization: host mode vs. overlay mode

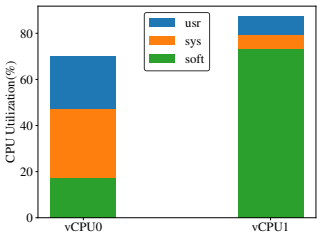


Fig. 4: Per-CPU utilization in overlay mode

To understand the bottleneck of the tunneling handling, we use mpstat [27] to measure the CPU utilization of the system when stress testing the overlay network using iperf3.

Specifically, we run iperf3 to generate traffic from the client container for 100 seconds. Figure 3 shows a breakdown of the average CPU utilization on the iperf3 server. For the host mode networking, 38.15% of CPU utilization is due to executing the kernel code (sys); while only 11.7% of CPU is spent on servicing softirqs (soft). For the container overlay networking, the softirq processing accounts for 42.48% of CPU utilization and the kernel code execution takes 19.14%. To quantify where CPU cycles are spent among the software interrupts, we use ebpf [28] to collect the CPU cycles (in μs) spent on each type of the softirq events. Results show that over 99.8% of the softirq processing is devoted to the networking softirqs NET_RX_SOFTIRQ and NET_TX_SOFTIRQ.

The experiment results show that the degradation of the overlay network performance is attributed to the execution of extra kernel code (for tunneling-related packet transformations) and the servicing of an increased number of softirqs. The overlay network entails the traversal of additional virtual network devices (i.e., the virtual bridge and the VXLAN interface), leading to an explosive growth of softirqs. We further look into the per-CPU utilization for the overlay networking. As shown in Figure 4, the majority of the softirqs is served by the ksoftirqd thread on vCPU1. This concentration of softirqs is determined by the IRQ affinity configuration of the system, which pins a type of interrupts to a particular set of CPU cores. This further confirms our analysis.

B. Alternatives for Container Networking

To optimize the container network performance, we first discuss pros and cons of existing solutions, ranging from hardware based acceleration to pure software solution, which sheds light on the design considerations adopted by *EZPath*.

1) Hardware Accelerations.

Since overlay networking requires packets to traverse both the guest and host network stacks, it introduces significant overhead to containerized applications. One natural solution is to assign physical network devices to selected containers and grant them exclusive access. Macvlan [29] and SR-IOV [30] are two such solutions.

With Macvlan, virtual interfaces are created, configured with host routable IP address, and assigned into the container namespaces. In contrast, SR-IOV requires hardware support that simulates a single PCI NIC as multiple virtual functions (VF), each with its own MAC address and functions as a physical NIC from the view of containers. However, the number of VFs supported by the hardware is quite limited (e.g., 64) and does not keep up with container network scales. Furthermore, to build applications that span across multiple hosts, both technologies require configuration of routable IP addresses on the host network. This may be a feasible solution in a VM-based virtualization environment, but it is not well suited for container networks. Different from VMs, containers are often short-lived and may be migrated in real time, making frequent network reconfiguration a nightmare. In addition, for intra-host communication where containers reside on the same physical host, SR-IOV imposes significantly higher overhead than Macvlan, since packets from one container must be sent through PCIe bus to the NIC before being forwarded to the other container [31]. Additionally, SR-IOV does not natively support live migration. For this purpose, popular container orchestration frameworks (e.g., Docker Overlay and Flannel for Kubernetes) or in-house built management solutions usually adopt overlay as their networking solution in real-world production management.

SmartNIC offloading. To cope with above concerns, hardware offloading is a viable solution with great potential to retain both the flexibilities of overlays and the performance of bare-metal. Intelligent or smart NICs have emerged recently to bridge the gap between the constantly increased network speed and the limited CPU processing power at the host machine. Nonetheless, hardware offloading in virtualized setups with OVS (e.g., VxLAN, connection tracking) is realized through SR-IOV or *virtio*. The aforementioned scalability limitation of SR-IOV remains in the container environment.

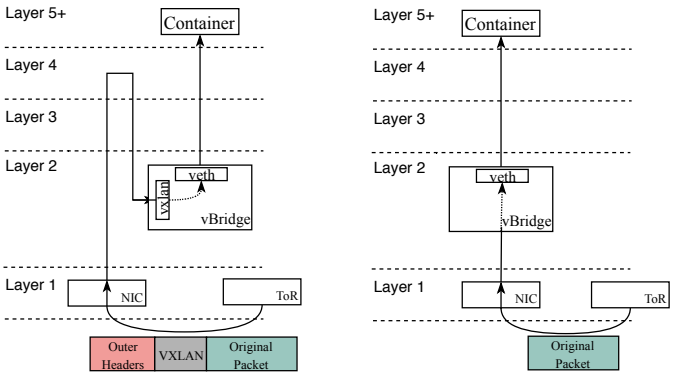
2) Software Accelerations.

Slim [17] proposed a pure software-based optimization approach to improve the container overlay network performance. It relies on extra pieces of software, including a shim layer to intercept socket related system calls, and a userspace router that creates connections on behalf of containers and maps host namespace file descriptors to the container namespace. This effectively shortens the packet path and improves the container network performance. However, Slim is not transparent to containers, which is a major limitation. Application binaries are required to be dynamically linked to the shim layer and extra care needs to be taken in the deployment and maintenance of the software components. As a result, application software would be less portable when using Slim. Moreover, after connections are established, Slim allows local containers to directly talk to the remote containers via the host namespace file descriptors, bypassing the container network interface and the virtual switch. As a consequence, it lacks support for conventional low-level network monitoring and debugging tools, such as *tcpdump*, as packets are not going through the virtual network interface and thus cannot be captured.

III. *EZPath* DESIGN

To overcome the aforementioned limitations of current container networking designs, in this section, we present *EZPath* to improve the performance of container networking by leveraging recent advances in programmable hardware.

At a high level, *EZPath* takes a software and hardware codesign approach. Specifically, it leverages the software for centralized controller while utilizing the programmable data planes in modern switches for traffic offloading. As an example, Figure 5b shows the change of ingress traffic path with *EZPath*, when compared to the default overlay approach 5a. From this figure, we can clearly see that an overlay network is realized by creating multiple virtual network devices that are connected through OVS. These include a VXLAN port, and a *veth* pair with one end assigned to the container namespace



(a) Overlay: vBridge tunneling (b) *EZPath*: ToR tunneling

Fig. 5: Network stack view: Overlay vs. *EZPath* offloading. Ingress data path of (a) host virtual bridge based overlay tunneling and (b) *EZPath* offloading tunneling to ToR switch. Through *EZPath* offloading, the in-host packet data path is shortened, the number of traversed network devices is reduced, the kernel processing for softirq is saved.

while the other end attached to OVS. Overlay packets follow this prolonged processing path, incurring much more software interrupts and corresponding context switches. Comparatively, in figure 5b, *EZPath* reduces the number of network devices that a packet has to traverse in the overlay. *EZPath* not only shortens the transmission path that a packet has to traverse, but also saves a lot of system processing overhead caused by explosive interrupt handling.

A. Key Principles and Challenges

Resource Constraints While *EZPath* can offload all traffic through the hardware in an ideal situation and thus completely eliminate the network bottleneck in the container network, this is not feasible in practice, mostly due to the constraint of limited hardware resources. Despite the high-speed forwarding performance, switching hardware has highly constrained on-chip memory. Since the overlay operations require the mapping information for the tunnel endpoints, the naive offloading strategy that offloads all container traffic is clearly impractical considering the extensive scale and short lifespans of containers.

Transparency Requirement *EZPath* is designed for multi-tenant environments, aiming to be application and kernel transparent so that it can work with existing systems and applications. *EZPath* does not require any modifications to user applications or the host kernel, including the virtual switches. This brings additional challenges to the system design. For example, in a containerized cloud shared by multiple tenants, the IP addresses of containers in different tenant networks are assigned locally and independently and thus can be overlapped or even the same. If *EZPath* decides to offload flows from different container networks with overlapped IP addresses, the ToR switch must be able to distinguish them.

Seamless and Transparent Offloading As discussed earlier, the amount of entries needed for tunneling depends upon the total number of tunnels used by containers in the rack. If the total number is less than the number of available entries at ToR switch, all the flows can be offloaded and the performance can be maximized. In *EZPath*, we aim to minimize the occupation of SRAM in P4 switches in order to leave sufficient space for accommodating other networking functions. Therefore, we choose to selectively offload the tunneling operations of performance critical flows that can be determined by system operators. That is, we design *EZPath* to seamlessly offload resource intensive overlay network operations to the programmable switches at the DC network edge, so that we can effectively relieve the performance bottleneck in the host network stack.

B. Overview of EZPath Design

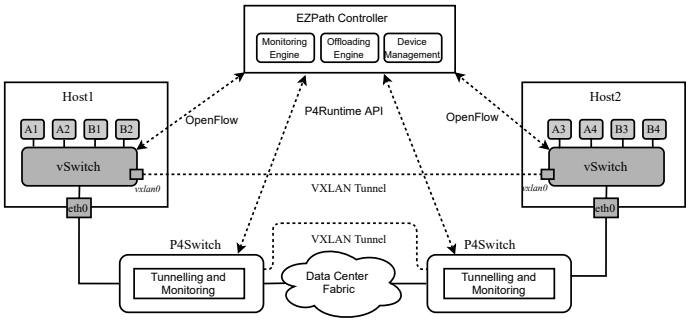


Fig. 6: *EZPath* design: major components.

To this end, Figure 6 depicts the major components of *EZPath* we design. As shown in the figure, it mainly consists of the centralized controller, the in-host software switch and the programmable ToR switch. The centralized controller monitors the traffic communications between containers and make offloading decisions in the ToR switches in real time. Moreover, it is also responsible for coordinating the offloading operations between the software switches, i.e., OVS on the end host, and the ToR switch. Unoffloaded flows follow the traditional path and are tunneled through the host network stack while ToR switches perform the heavy-lifting tunneling operations for offloaded flows. In *EZPath*, ToR switches are P4 switches, which provides programmable control for packet forwarding.

To maintain application and kernel transparency, ToR switches must be able to distinguish the flows from different tenants. For this purpose, in our design, OVSes leverages the VLAN headers to carry tenant information to ToR switches. For an outgoing packet, the directly attached ToR switch would first strip off the VLAN header, look up tunnelling related information in match-action tables, and perform packet encapsulations. On the other hand, for an incoming packet, the ToR switch would examine if it is the end of the tunnel. If it is, it would decapsulate the packet, re-tag the packet with VLAN header and forward it to the connected OVS. Otherwise, the packet is processed similarly as in normal case. VLAN only involves L2 processing and the logic is much simpler by following a shortened processing path, which greatly reduces the CPU cycles on the end hosts. The detailed steps and data structures involved will be discussed in Section IV.

Real-time migration is another key component of *EZPath*, which aims to seamlessly offload the flows from the host to P4 switch without disrupting existing connections. The control plane makes real-time offloading decisions. Once the candidate flows for offloading are determined, the controller updates tables on both OVSes and P4 switches according to the following steps: (1) It installs the tunnel mapping for the offloaded containers into the P4 tables on both ToR switches; (2) It modifies OVS flow rules on both end hosts to bypass the host network stack. This is a critical step to guarantee that there is no interruption to the existing connections. Reversing the steps could cause packet drop since there is no tunnel mapping entries in P4 to handle the offloaded packets. On the contrary, flows that have been identified as inactive would be migrated back to host.

IV. Implementation

We have implemented a prototype of *EZPath*. The implementation is about 1000 LOC of P4 in the dataplane and 350 LOC in the control plane. Since the packet processing in the dataplane involves multiple entities along the path between each pair of containers, in the section, we present their implementation details.

Our control plane is built upon Barefoot Runtime Interface (BRI) that comes with Barefoot SDE (9.1.1). It provides APIs for the control plane to configure and manipulate the dataplane pipeline and objects. On each P4 target, a gRPC server runs and listens for the requests from the control plane, which will be further parsed into target-specific actions. The controller periodically sends a register read request to ToR P4 switches every T seconds, defined as a polling interval. The collected flows in each interval are constructed into time-series and are analyzed. Through the control plane, container information can be retrieved from the orchestration systems (e.g., Kubernetes, Amazon ECS, Mesos and Marathon). For example, Kubernetes allow Pods to indicate the importance of a Pod relative to other Pods [32]. Orchestration platforms also have access to the types of the applications during deployment, such as batch processing jobs, machine learning training workloads, and user-facing services. The control plane can make offloading decisions based on these factors.

V. Evaluation

In this section, we present our evaluation of *EZPath* following the experiment setup. To quantify the performance improvement, we first use microbenchmarks to study the performance of *EZPath* for tunnel offloading with respect to the normal container overlay networking. Then we evaluate the performance of some typical real-world containerized applications, including an in-memory key value store Memcached [33], ZeroMQ [34] for large scale distributed message library, Nginx [35] for web servers, when adopting *EZPath*.

A. Experiment Setup

Our testbed consists of two STORDIS BF2556X-1T tofino switches with P4 programmability support and two host servers. The setup of this testbed is similar to that shown in Figure 6. The two P4 switches serve as the ToR switches and the two host servers are placed on two server racks. Each server is equipped with an Intel Xeon Silver 4110 2.10 GHz CPU, 32 GB DDR4 RAM and NetXtreme-E RDMA 25 Gbps NICs. They are both running Ubuntu Bionic 18.04 LTS OS with Linux kernel 4.15.0. On each host, we leverage Docker 19.03 to create/manage containers and deploy containerized applications. The switches and servers are directly connected through 25 Gbps cables. Since our experimental setup is different from the ones used in previous works, we compare EZPath's performance with the host networking mode so that we can make a relative comparison with prior works, such as Slim [17], that are also compared with the host mode.

B. Network Throughput and Latency

First, we compare the performance of different networking modes, i.e., with and without offloading. The former means the default overlay networking, while the latter represents *EZPath*. Specifically, we measure the network performance in terms of both throughput and latency. As we discussed earlier, in offloading mode in *EZPath* the traffic egressing from the containers is directly forwarded to the physical NIC on the host server. The tunnelling-related operations (e.g., packet encapsulation and decapsulation) are performed by the P4 switches. In contrast, in non-offloading mode, the container traffic follows the normal path by traversing both the container and host kernel network stacks.

We use iperf3 to measure the network throughput of a TCP flow in both modes. Each run takes 90 seconds with the messages sizes varied across 128B, 256B, 512B, 1024B and 1440B. The results averaged out of 5 runs under each setting are shown in Figure 7.

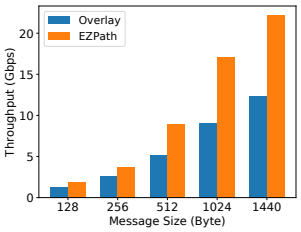


Fig. 7: Network throughput comparison

TABLE I: Packet Latency

Networking Mode	RTT (μs)
Overlay	45.0
EZPath	32.8

As we can see, by offloading to the hardware P4 switches, we can significantly improve the throughput performance of the container overlay networking. On average, we see 68% throughput improvement. When the message size is 1440B, the throughput is improved by 80%. In contrast, Slim achieves the same throughput as the throughput on the host mode network, thus resulting in about 48% throughput improvement compared with the overlay networking.

We use sockperf-3.6 to measure the packet latency. The tests are performed in its ping-pong mode, where the latency of single packet is measured without waiting for the reply before sending the subsequent packet on time. As shown in Table I, Without offloading, the measured round trip latency, averaged on 5 runs, is approximately 45 μs for a packet of moderate size 350B. In contrast, with *EZPath* offloading the

latency is significantly reduced to 32.8 μs, an improvement of 27%. While for Slim, its latency is also the same as using the host mode network, which leads to an 85% improvement compared to the overlay networking. Note that the sender and receiver are running on two physical machines that are directly connected with a 40 Gbps cable in Slim. Similar results are observed in other tests as well.

C. Application Performance

In this section, we evaluate the improvement to application performance brought by *EZPath*. For this purpose, we evaluate the performance with popular containerized applications. Considering that there is a wide range of overlay networking solutions, we choose the host networking mode as the baseline in our evaluation. In the host mode, all containers on the same host share the host network namespace. Therefore, they have direct access to all the host's network interfaces. In real-world applications, the host mode is rarely employed due to its inflexibility in supporting multi-tenant cloud applications. In particular, the ports cannot be reused by the same type of applications co-located on the same physical host. For example, once port 80 is assigned to one containerized web server, the other containers would have to use different ports for their web services to avoid conflicts. Nonetheless, compared to the other networking solutions, the host mode achieves the highest performance despite its inflexibility. Our evaluation adopts similar comparison approaches as used in the existing work by comparing *EZPath* to the baseline host mode and the legacy overlay mode. The relative numbers of the performance improvements can demonstrate how *EZPath* compares with other approaches that are evaluated in different hardware setups.

1) Memcached Benchmarking

Memcached has been widely used to deploy distributed key-value services in commodity data centers (e.g., Facebook, Google, AWS, Netflix) to improve web service performance.

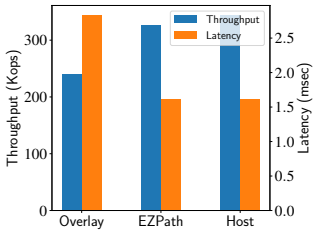


Fig. 8: Memcached: throughput & latency

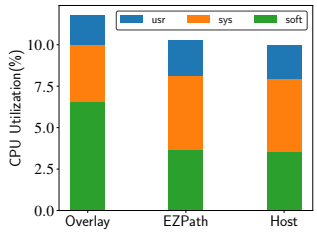


Fig. 9: Memcached: CPU utilization

In our experiments, we create one Docker container on each physical server. The Memcached server is deployed in one container, and the Memcached benchmarking client runs in the other container. We measure the throughput and latency of the Memcached service in the host mode (baseline), the overlay mode (common practice), and *EZPath*.

The benchmark tool we use is memtier_benchmark [36] developed by Redis Labs. In our experiments, we use the default settings with four client threads and 50 connections. Each experiment sends 100000 requests and the SET:GET request ratio is set to 10. We repeat the experiments under different modes. The results are averaged over five runs.

Figure 8 shows the throughput results in the number of total completed Memcached operations per-second. As we can see, the offloading mode by *EZPath* achieves the throughput comparable to the host mode, and outperforms the overlay mode by 35% on average. Slim's throughput is within 3% less of host mode with the same setting. *EZPath*'s throughput is within 5% less of the host mode and latency is within 0.7% of the host mode with the same setting.

EZPath also reduces Memcached request latency. Figure 8 also shows the 99.9th percentile latency to complete a Memcached request. We find the latency through the *EZPath*'s offloading mode is the same as the host mode, which is also the same with that of Slim. Compared to the overlay mode, *EZPath* reduces the latency by 42%.

2) ZeroMQ Benchmarking

ZeroMQ is an asynchronous network messaging library widely deployed in large scale distributed/concurrent systems. Different from brokered message queues (e.g., Apache Kafka [37], ActiveMQ [38], RabbitMQ [39]), ZeroMQ does not rely on brokers and thus achieves much higher throughput. In our experiments, we again measure two key performance metrics, throughput and latency, of ZeroMQ under different modes as before. Each measurement is performed across a wide range of message sizes. On each host, we create a Docker container with ZeroMQ-4.2.2 installed. One container serves as the sender and the other processes the requests as the receiver. The throughput is measured in terms of the number of messages per second; while the latency is measured as the average time it takes to transfer a single message between the two endpoints. The message size varies from 64B to 128KB. For each measurement, the experiment is repeated three times and the average is reported.

Figure 10 and Figure 11 depict the resulted throughput and latency, respectively. From the figures, we can observe that in all settings, *EZPath* offloading greatly outperforms the non-offloading counterpart in the overlay mode. Moreover, the improvement becomes more pronounced with the increase of the message size. In particular, for larger message sizes, e.g., 128KB, the throughput is increased by around 42% from 13K msg/sec to 18.5K msg/sec (which is not clearly visible due to the large scale of y-axis in Figure 10), while the latency is reduced by 30%.

When compared to the host mode, *EZPath* offloading achieves comparable performance with merely slight degradation in both throughput and latency. We believe that this result is acceptable considering the various advantages of offloading over the host mode.

In addition to the application performance, we also examine the variation in the composition of CPU consumption under various modes, mainly including sys, usr, and soft, which represent CPU utilization for executing at user level, system level and service software interrupts, respectively. In our testbed, each server has 16 physical cores with hyperthreading enabled (equally, 32 virtual cores). Therefore, when measuring

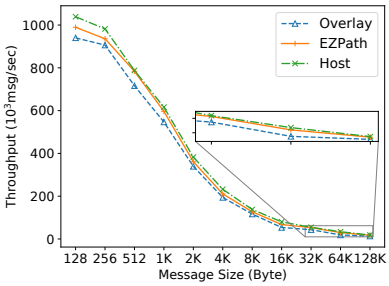


Fig. 10: ZeroMQ: throughput

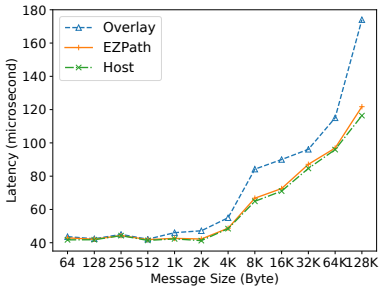


Fig. 11: ZeroMQ: latency

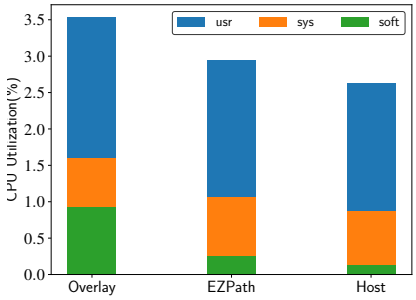


Fig. 12: ZeroMQ: CPU utilization

the CPU overhead breakdown, the CPU utilization can be conveniently converted to the amount of virtual cores. The result are shown in Figure 12. As clearly shown in the figure, *EZPath* offloading also reduces CPU cycles significantly. In particular, the CPU cycles spent on serving the software interrupts are reduced by 73%.

3) Web Server Benchmarking

To study how various networking modes could affect the performance of popular web applications, we run a Nginx container on one host and a client container on the other host. The benchmark software we used is wrk2 [40], which takes throughput as an input argument. The throughput is specified in terms of the total requests per second combined across all connections. Specifically, we create 2 threads in wrk2 and each thread establishes 100 HTTP connections concurrently to the Nginx server. For test purpose, we also randomly generate files with sizes 1KB and 1MB on the web server. In our experiment, the request throughput is set to 10000 and 2500 per second for 1KB and 1MB, respectively. In this way, we can keep the average latency within the order of milliseconds. We measure the average latency and report the result with its standard deviations computed across multiple runs. Figure 13

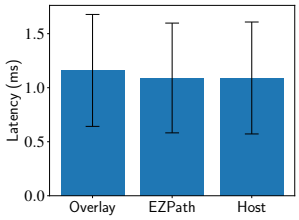


Fig. 13: Nginx: 1KB files with 10k requests/sec

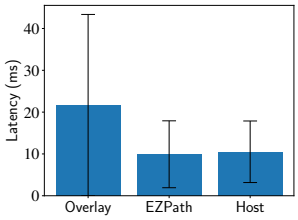


Fig. 14: Nginx: 1MB files with 2.5k requests/sec

and Figure 14 show the results. As shown in the figure, when requesting 1KB files with 10K requests/sec throughput, the average latency in *EZPath* offloading mode is 1.09ms, which is noticeably smaller than the latency in non-offloading mode (1.16ms). The difference is even larger when requesting 1MB files with 2.5K requests/sec throughput. The average latency in *EZPath* offloading mode is 9.92ms, while the latency in non-offloading mode is 21.68ms, an improvement more than 54%. In both cases, the request processing latency in the offloading mode roughly amounts to that in the host mode. With the same setting, Slim also achieves similar latencies with that of the host mode.

Figure 15 further shows that the CPU cycles resulted from serving software IRQs are significantly reduced by approximately 50% in the *EZPath* offloading mode. On the other hand, we observe negligible difference between the CPU cycles incurred in the *EZPath* offloading mode and the host mode. In contrast, there exists a 5%–6% gap between Slim and the host mode for the CPU utilization. Therefore, the offloading in *EZPath* to the programmable hardware achieves substantially better web performance, with low CPU overhead similar to that of the native host mode. This preserves server resources and opens up further opportunities for overall performance improvement.

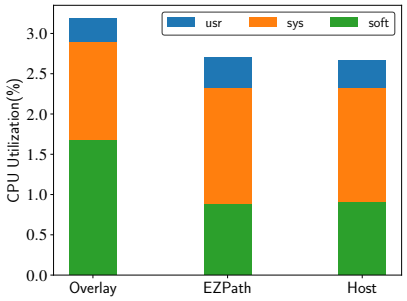


Fig. 15: Nginx CPU utilization under various modes.

D. Discussion

To achieve full optimization, *EZPath* requires a centralized control plane, which controls and manages the software/hardware network devices over the entire DC and across DCs. If containers are deployed and replicated across data centers (e.g., to enhance service reliability and availability), it will place burden on the centralized control plane to manage such large scaled networks. One workaround to mitigate this is to use local controllers that make offloading decisions on their own, without coordination between the endpoints.

On the other hand, the offloading function offered by *EZPath* does not have to be tied to any particular offloading strategy. Essentially, regardless of how the target flow is identified, *EZPath* can offload that seamlessly.

VI. Related Work

Efficiently and flexibly managing containerized clouds poses a myriad of challenges from different aspects [9], [16],

[17], [19], [41], [42]. Among them, the network performance degradation due to the software based overlay networking has attracted a lot of attention [16]–[18]. This is mainly due to the overhead when implementing the tunneling and other network virtualization functionalities in the software switch, such as Open vSwitch, a key component in Weave [12], one of the most widely used container network interface (CNI) plugins for production container orchestration platforms such as Kubernetes [13], Apache Mesos [14], and Amazon ECS [15]. Approaches like SR-IOV [16] and Macvlan [29] aim to assign physical network devices to selected containers, but they are not suitable for containerized clouds. For example, SR-IOV is limited by the simulated 64 virtual functions (VF) while container networks often have massive scales [43], [44], and is also incurs extra overhead [45] in networking among intra-host containers as discussed in section II-B.

Recent designs [9], [17], [19], on the other hand, either demand hardware support or fail to support legacy monitoring and debugging tools in data centers. For example, Slim [17] can achieve promising performance boosts by redesigning container overlay networks with a dedicated user-space router to reduce the packet traversal of OS-kernel network stack. But it is impractical for real world deployment due to the customized software demand. Compared to them, *EZPath* is both application and system kernel transparent, and is compatible with any existing tools, thus offering a transparent alternative.

VII. Conclusion

Cloud computing is increasingly adopting containers, evidenced by the popularity of microservices offered by all major cloud service providers. However, containerized applications often suffer from degraded networking performance in practice due to the extra processing overhead induced by the container overlay. Previous solutions addressing this problem are either not compatible with the legacy monitoring and debugging tools, or demanding customized hardware support. In this paper, we have instead designed and implemented *EZPath* to expedite the container traffic, by leveraging the high speed of the programmable switches in data centers. *EZPath* is transparent to applications or the underlying system kernel, compatible with all existing tools. Our evaluation shows it can significantly expedite container traffic by improving the throughput and reducing the latency of typical containerized applications.

VIII. Acknowledgment

We appreciate the constructive comments from the reviewers. This work was supported in part by the NSF grants CNS-2007153, CNS-2008468, an ONR grant N00014-18-2893, a Commonwealth Cyber Initiative grant and a Google Faculty Research Award.

References

[1] Building products at soundcloud — part I: Dealing with the monolith. https://tinyurl.com/2p88dk75.
[2] Building netflix's distributed tracing infrastructure. https://tinyurl.com/yzwtr3fa.
[3] What led amazon to its own microservices architecture – the new stack. https://tinyurl.com/ytc5vtse.
[4] Monolith to microservices: Transforming a web-scale, real-world e-commerce platform using the strangler pattern. https://tinyurl.com/2p2we3ej.
[5] Apache ignite as an inter-microservice transactional in-memory data store. https://tinyurl.com/5exs7dhp.
[6] Aws lambda - serverless compute - amazon web services. https://aws.amazon.com/lambda/.
[7] Cloud computing services - google cloud. https://cloud.google.com/.
[8] Azure service fabric. https://tinyurl.com/yckmvd35.
[9] Tianlong Yu et al. Freeflow: High performance container networking. In *Proceedings of ACM HotNets*, 2016.
[10] Shelby Thomas, Lixiang Ao, et al. Particle: ephemeral endpoints for serverless networking. In *Proceedings of ACM SoCC*, 2020.
[11] Ben Pfaff, Justin Pettit, et al. The design and implementation of open vswitch. In *Proceedings of USENIX NSDI*, 2015.
[12] Introducing weave net. https://tinyurl.com/5n888rsp.
[13] Kubernetes. https://kubernetes.io/.
[14] Apache Mesos. http://mesos.apache.org/.
[15] Amazon elastic container service. https://tinyurl.com/49x8zbjz.
[16] Radhika Niranjan Mysore et al. Fastrak: enabling express lanes in multi-tenant data centers. In *Proceedings of ACM CoNEXT*, 2013.
[17] Danyang Zhuo et al. Slim:{OS} kernel support for a low-overhead container overlay network. In *Proceedings of USENIX NSDI*, 2019.
[18] Jiaxin Lei et al. Tackling parallelization challenges of kernel network stack for container overlay networks. In *Proceedings of USENIX HotCloud*, 2019.
[19] Ryo Nakamura, Yuji Sekiya, et al. Grafting sockets for fast container networking. In *Proceedings of ACM/IEEE ANCS*, 2018.
[20] Trident4 / BCM56880 Series, 2021.
[21] Edgecore Networks.
[22] Cisco Nexus 3000 Series Switches, 2020.
[23] Docker container networking. https://tinyurl.com/5y46h56c, 2019.
[24] Rick Jones et al. Netperf: a network performance benchmark. *Information Networks Division, Hewlett-Packard Company*, 1996.
[25] Jon Dugan et al. iperf3, tool for active measurements of the maximum achievable bandwidth on ip networks.
[26] Quinn O Snell et al. Netpipe: A network protocol independent performance evaluator. In *Proceedings of IASTED IIS*, 1996.
[27] mpstat: Report processors related statistics. https://tinyurl.com/3u97ef47.
[28] Linux Extended BPF (eBPF) Tracing Tools. https://tinyurl.com/49yrx8u7, 2009.
[29] Use macvlan networks. https://tinyurl.com/ye24cp53.
[30] Yaozu Dong, Xiaowei Yang, et al. High performance network virtualization with sr-iov. *Proceedings of Elsevier JPDC*, 2012.
[31] Jason Anderson, Hongxin Hu, et al. Performance considerations of network functions virtualization using containers. In *Proceedings of IEEE ICNC*, 2016.
[32] Pod priority and preemption. https://tinyurl.com/4z7xyz24.
[33] Memcached. http://memcached.org/, 2019.
[34] ZeroMQ: An open-source universal messaging library. https://zeromq.org/.
[35] Nginx. https://www.nginx.com/.
[36] RedisLabs. memtier_benchmark. https://tinyurl.com/56xbxvfz, 2019.
[37] Apache Kafka. https://kafka.apache.org/.
[38] Apache ActiveMQ. http://activemq.apache.org/.
[39] RabbitMQ. https://www.rabbitmq.com/.
[40] wrk2. https://github.com/giltene/wrk2, 2019.
[41] Junaid Khalid, Eric Rozner, et al. Iron: Isolating network-based {CPU} in container environments. In *Proceedings of USENIX NSDI*, 2018.
[42] Cong Xu, Karthick Rajamani, et al. Nbwguard: Realizing network qos for kubernetes. In *Proceedings of ACM/IFIP Middleware*, 2018.
[43] Docker at insane scale on IBM Power Systems. https://tinyurl.com/2p9etvdx, 2015.
[44] Wei Zhang, Jinho Hwang, et al. Flurries: Countless fine-grained nfs for flexible per-flow customization. In *Proceedings of ACM CoNEXT*. ACM, 2016.
[45] Chapter8. performance and optimization reference architectures 2017 - red hat customer portal. https://tinyurl.com/3zm83asn.