A Scalable VPN Gateway for Multi-Tenant Cloud Services

Mina Tahmasbi Arashloo
Princeton University
arashloo@cs.princeton.edu

Pavel Shirshov
Microsoft
pavelsh@microsoft.com

Rohan Gandhi
Carnegie Mellon University
rgandhi2@andrew.cmu.edu

Guohan Lu
Microsoft
gulv@microsoft.com

Lihua Yuan
Microsoft
lyuan@microsoft.com

Jennifer Rexford
Princeton University
jrex@cs.princeton.edu

ABSTRACT

Major cloud providers offer networks of virtual machines with private IP addresses as a service on the cloud. To isolate the address space of different customers, customers are required to tunnel their traffic to a Virtual Private Network (VPN) gateway, which is typically a middlebox *inside the cloud* that internally tunnels each packet to the correct destination. To improve performance, an increasing number of enterprises connect *directly* to the cloud provider's network *at the edge*, to a device we call the provider's edge (PE). PE is a chokepoint for customer's traffic to the cloud, and therefore a natural candidate for implementing network functions concerning customers' virtual networks, including the VPN gateway, to avoid a detour to middleboxes inside the cloud.

At the scale of today's cloud providers, VPN gateways need to maintain information for around *a million* internal tunnels. We argue that no single commodity device can handle these many tunnels while providing a high enough port density to connect to hundreds of cloud customers at the edge. Thus, in this paper, we propose a hybrid architecture for the PE, consisting of a commodity switch, connected to a commodity server which uses Data-Plane Development Kit (DPDK) for fast packet processing. This architecture enables a variety of network functions at the edge by offering the benefits of both hardware and software data planes. We implement a scalable VPN gateway on our proposed PE and show that it matches the scale requirements of today's cloud providers while processing packets close to line rate.

CCS Concepts

•Networks → Middle boxes / network appliances; Cloud computing;

Keywords

Virtual Private Network Gateway, Cloud Provider Edge, Middleboxes

1. INTRODUCTION

Virtual networking is an increasingly popular infrastructure-as-a-service offered by major cloud providers such as Microsoft, Amazon, and Google. It enables customers to create networks of virtual machines (VMs) with private IP addresses on the cloud as if they were part of their own on-premise networks. The relatively low infrastructure and management cost of hosting virtual networks on the cloud has motivated an increasing number of enterprises to move *thousands* of VMs to public clouds [1, 8, 7, 3].

In such a virtual environment, multiple VMs with the same private IP address, but owned by different customers, can coexist. To isolate the address space of different customers, cloud providers typically require customers to tunnel their traffic to a *virtual private network (VPN)* gateway to reach their VMs. A VPN gateway receives traffic from each customer's on-premise network over a separate *external* tunnel. It uses the external tunnel ID and destination VM's IP address to lookup the *internal* tunnel to the physical server hosting that VM, and tunnels the packet to that server. The server then decapsulates the packet and delivers it to the destination VM. The processing for the reverse direction is similar. Given that a large-scale cloud provider can host hundreds of customers, each with thousands of VMs, the lookup table for internal tunnels can grow as large as *a few million* entries. Therefore, cloud providers typically run VPN gateways as bump-in-the-wire middleboxes, using either commodity servers or specialized blackboxes.

Until recently, the only way for customers to connect to their cloud services, including the VPN gateways and therefore their virtual networks, was over the Internet. However, to interconnect their on-premise and on-cloud networks, large enterprises require Service Level Agreements (SLAs) on the bandwidth of their connection to the cloud, which is not practically feasible to provide over the Internet. Thus, major cloud providers have started to offer direct and dedicated network connections to the cloud [4, 10, 5]. Customers can peer with the cloud provider by connecting their own border router, the *Customer's Edge* (CE), to what we call the *Provider's Edge* (PE) (Figure 1). The PE is operated by the cloud provider and sits at the edge of the cloud. This enables cloud providers to offer SLAs on bandwidth and latency of customers' access to their cloud resources since they have full control over the end-to-end path.

Although direct peering with cloud providers through PEs has emerged as a response to the need for more predictable and higher-quality connections to the cloud, it creates an opportunity for performance improvements inside the cloud as well: the PE is a chokepoint for all the traffic between the customer and the cloud, and therefore is a natural candidate for placing the VPN gateway functionality. By terminating the VPN tunnels on the PE devices, traffic does not need to take a detour to a separate, off-path VPN gateway.

However, this makes the design of the PE quite challenging. PE needs *high port density* with *high switching capacity* to connect hundreds of customers to the cloud's internal

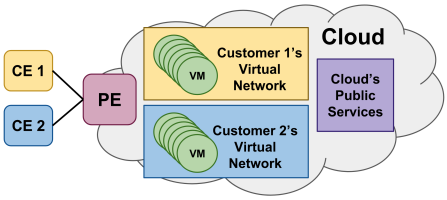

Figure 1: Overview of dedicated connections to the cloud at the edge

network with high-speed links. To act as a VPN gateway, it should also handle tunnel lookup tables with around *a few million entries*. Moreover, there are networking functions other than VPN that can benefit from running at the edge as opposed to a middlebox inside the cloud. These include, but are not limited to, per-customer traffic shaping and policing, NATing for access to public cloud services, and ACLs. Thus, it needs to be *easy to extend* the PE to support new networking functions.

We argue that no single existing box can satisfy all the above requirements. Many commodity switches offer tens of 40G ports, which can provide the required port density and switching capacity. However, they typically have only a few thousand tunnel lookup entries on the data plane. On the other hand, we cannot simply reuse the same commodity servers or special blackboxes used as VPN gateways since they cannot satisfy the required port density and switching capacity. Specialized hardware solutions might be able to satisfy the requirements for connectivity and VPN functionality, but they are not cost-effective and are harder to modify for new features than commodity devices.

In this paper, we propose a hybrid architecture, built out of a commodity switch, connected to a commodity server with multiple high-speed ports. The rest of the switch ports are used to connect to the customers as well as the routers in the cloud's internal network (Figure 2). This architecture allows us to flexibly distribute the network functions required at the edge between software and hardware, placing each on the platform that fits it best, and therefore, achieve all the PE requirements. The switch provides high port density and forwards packets at high speed between CEs, the server, and the cloud. The VPN functionality is distributed between the switch and the server. The switch sends packets from customers' on-premise network to a program on the server, which uses Data-Plane Development Kit (DPDK) [2] libraries to store the large internal tunnel lookup table, encapsulate the packet accordingly, and direct it back to the switch for forwarding to its destination. Packets from VMs back to customers directly go through the switch since looking up external tunnels is independent of the endpoint servers and can be implemented on the switch's data plane. Moreover, having both the DPDK-enabled server and the commodity switch as the data plane makes the PE easily extensible. Similar to how we implemented the VPN functionality on this hybrid architecture, developers can start off by adding and testing new features in the DPDK program, and gradually offload all or parts of it to the switch to improve performance.

We have built a prototype of the PE using a commodity server connected to a commodity switch with a single 40G NIC. Using 10 cores on a single CPU on the server, we show that our VPN gateway implemented on PE's hybrid data plane can process packets at 40G for packets larger than

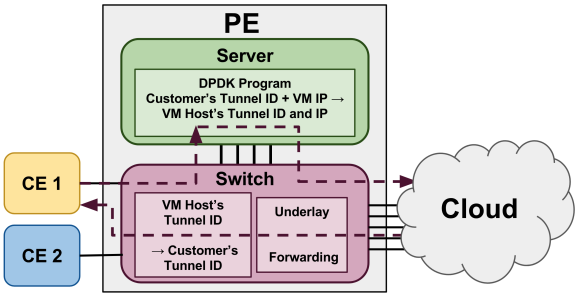

Figure 2: Proposed hybrid architecture for the PE

128 bytes, and, due to NIC constraints, close to 40G for smaller packets. These preliminary results encourage us that with more NICs, cores, and another CPU on the server, our hybrid data plane could process packets at $\sim$ 100G.

To summarize, we make the following contributions:

- A hybrid data plane enabling scalable implementation of network functions at the edge of a large-scale cloud provider network by offering the benefits of both hardware and software data planes.
- A scalable VPN gateway, implemented on the hybrid data plane, that can connect hundreds of enterprises to their thousands of VMs on the cloud.
- A prototype of the hybrid data plane with a single 40G NIC between the server and the switch, and preliminary results showing that a VPN gateway running on it can keep up with 40G input traffic.

2. THE PROVIDER'S EDGE

Our proposed Provider's Edge (PE) has both a commodity switch and a commodity server on its data plane (Figure 2). The hardware data plane on the switch is a *high-throughput pipeline* that is capable of common networking functionalities, such as standard L2 and L3 forwarding, and tunneling. However, it has limited memory and supports a limited set of functions compared to that of a software data plane. The software data plane on the server, on the other hand, is a C program that is executed on multiple cores. Thus, it can implement *arbitrary packet processing logic*, and has *memory in abundance* compared to the switch. Using DPDK libraries to send and receive packets at high speed, each core is capable of processing gigabits of traffic per second. However, the software data plane has a run-to-completion, as opposed to pipeline, execution model. So, it cannot sustain its throughput as the complexity of the C program increases.

Network functions have a wide range of requirements in terms of functionality, memory, and throughput. Thus, we believe that this hybrid architecture provides a suitable platform for implementing a variety of them at the edge. More specifically, the rest of this section describes the functional requirements of a VPN gateway at the edge and how it can be implemented on top of PE to satisfy those requirements. We also discuss several other network functions whose implementation at the edge is facilitated by PE's hybrid data plane to demonstrate PE's extensibility.

2.1 VPN Gateway's Functional Requirements

Overlay Routing. To act as a VPN gateway, the PE should participate in multiple *overlay* networks, each interconnecting a customer's on-premise network to its virtual

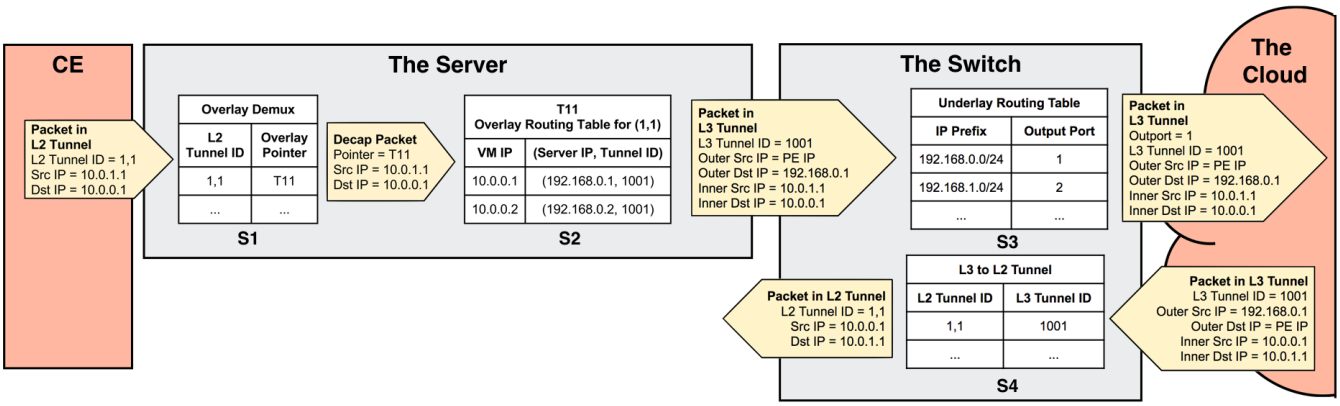

Figure 3: Packet processing for traffic between endpoints in Customer 1's Overlay 1 in Figure 4.

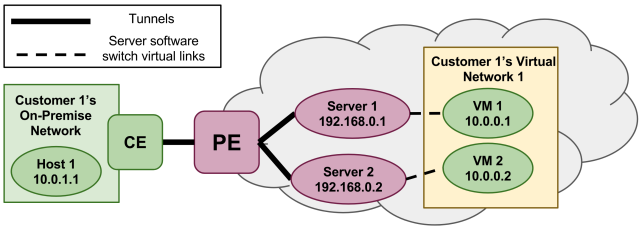

Figure 4: Example overlay network for customer 1's virtual network 1.

network on the cloud. An overlay network consists of the VMs in a customer's virtual network, and the hosts in its on-premise network who wish to communicate with them. The hosts in the on-premise network are connected to the customer's CE router, each VM is virtually connected to the software switch on the physical server hosting it, and the PE connects the CE and the servers, as shown in Figure 4. To deliver a packet from a customer's on-premise network to a VM in the cloud, the VPN gateway should find the IP address of the physical server hosting the destination VM by routing the packet on the overlay network. Note that a customer may own multiple virtual networks on the cloud, and therefore, participate in multiple overlay networks.

Tunneling. All the overlay networks share the underlying network infrastructure of the same cloud provider. Thus, to isolate the traffic of overlay networks from each other, the links between the PE and the servers and the customer's CE router in each overlay network should be implemented as tunnels. We assign a unique identifier C to each customer and a unique identifier O to each of its overlays, and use (C, O) as the globally unique identifier of an overlay network. This unique identifier can be used to compute tunnel ids for that overlay network. The links between the PE and the servers should be implemented as L3 tunnels (e.g., GRE, NVGRE, and VXLAN) on the cloud's internal network. All L3 tunnels of an overlay network have the PE as one endpoint, a server as the other endpoint, and the same tunnel identifier that is in one-to-one correspondence with the overlay identifier. PE and CEs, however, are connected at L2 using Ethernet. Thus, the link between the PE and the customer's CE can be implemented as an L2 "tunnel", e.g., using Q-in-Q tunneling or VLAN, where the Q-in-Q label or the VLAN tag is in one-to-one correspondence with the overlay identifier.

Underlay Routing. To deliver traffic from customers' on-premise networks to the physical servers hosting their VMs on the cloud, the PE should be able to route encapsulated packets on the cloud's internal network or the *underlay*.

Figure 3 shows, at a high level, how overlay routing, together with tunneling and underlay routing can enable VPN gateway functionality at the PE, in four stages labeled S1 through S4. Upon receiving a packet from a CE through a L2 "tunnel" in the underlay network, the PE should decapsulate the packet to terminate the tunnel, and use the tunnel identifier to find the corresponding overlay network (S1). Each overlay network has a routing table, which maps each VM IP address to the L3 tunnel that connects the PE to the server hosting that VM and gets populated as part of provisioning new VMs in that overlay. The PE should look up the packet's destination VM in the overlay routing table and encapsulate based on the corresponding tunnel information (S2). The encapsulated packet will have the IP address of the server as its destination and the tunnel id corresponding to the overlay identifier. Now, the packet should be routed on the cloud's internal network. By participating in routing protocols of the cloud's internal network, the PE should keep a Longest-Prefix-Match (LPM) forwarding table that maps IP prefixes to their output port in the underlay network, and use that to send the encapsulated packet to its destination server (S3).

Delivering packets from the cloud to customers' on-premise networks is less involved. Upon receiving a packet from a server through an L3 tunnel, the PE should decapsulate the packet to terminate the tunnel, and use the tunnel identifier to find the corresponding overlay network. Given that the customer's CE is connected to the PE through Ethernet, no L3 routing is needed. The PE can look up the L2 tag corresponding to the overlay identifier, tag the packet accordingly, and send it to the CE at L2 (S4). This look-up table is populated as part of provisioning for a new virtual network, after it has been assigned an overlay identifier.

2.2 Implementing the VPN Gateway on PE

To implement the abstract pipeline in Figure 3 on the PE, we can exploit PE's hybrid architecture to place each part of the pipeline on the data plane that fits it best: the hardware data plane on the switch or the software data plane on the server. Given the high throughput of the hardware data plane and the run-to-completion execution model of the software data plane, we want to place as much functionality as possible on the switch, and use the server for network

functions that are either too complex or have memory requirements that exceed the switch capacity.

Packets from customer's on-premise network to the cloud experience both overlay and underlay routing. Each overlay network can have on the order of thousands of VMs, and its routing table needs to keep an entry for mapping each VM to the tunnel to its host server. If the IP addresses of the VMs on the same server could be aggregated, we could keep fewer entries in the routing table. However, VMs are typically assigned to servers based on criteria other than the aggregation possibility of IP addresses. Thus, the overlay routing table typically has as many entries as the number of its VMs, on the order of thousands. Given that the PE can be part of hundreds of overlay networks and that switches can only keep a total of a few thousand tunnel mappings, overlay routing cannot be implemented entirely on the switch. Therefore, when packets from CEs enter the PE through the switch, they are redirected to the server, where the C program takes care of the decapsulation, overlay routing, and L3 tunnel encapsulation. Underlay routing, however, happens over a standard IP network with IP aggregation and is a standard functionality of switches. Thus, after overlay routing, the server sends the packets back to the switch, where they get forwarded based on the LPM forwarding table to their destination server.

The pipeline for packets from on-cloud VMs to on-premise hosts can be implemented entirely on the switch. For each overlay, there is only one destination for the packets coming from the servers: the customer's CE device. Thus, the switch terminates the L3 tunnels from the servers by decapsulating the packet, looks up the L2 tag of the corresponding overlay network, and tags and relays packets to their corresponding CE device. The L2 tag lookup table has one entry per overlay network, and therefore fits on the switch.

Note that by placing a stage of the pipeline on the server, the throughput of the PE for the portion of traffic passing through that stage will be limited by the throughput of the software data plane, which is lower than that of the switch. However, we only place those pipeline stages on the server that the switch cannot handle on its own. Moreover, the hybrid data plane is still able to achieve a higher throughput than a stand-alone server. This is because offloading other stages of the pipeline to the switch makes the software data plane much simpler, which results in a higher-throughput due its run-to-completion execution model.

2.3 More Network Functions

So far, we have mostly focused on how to use a hybrid of software and hardware data planes to implement a scalable VPN gateway at the edge of a large-scale cloud provider's network. However, we argue that, in general, this hybrid design enables implementing a variety of network functions with different requirements altogether at the edge. One just needs to specify the high-level flow of packet processing for a network function, similar to Figure 3, and estimate the requirements at each stage of packet processing (e.g., size of tables). Next, one should assign as many stages as possible to the hardware data plane, and the rest to the software data plane. The following paragraphs provide examples of network functions who can benefit from running at the cloud's edge and how they can be implemented on the PE.

Traffic Shaping and Policing. The cloud providers may need to do shaping and policing on the aggregate traffic of each of their hundreds of customers, which perfectly fits within the capabilities of a commodity switch. However, they may want to further allow customers to specify per-VM traffic shaping and policing rules for their virtual networks on the cloud. This can result in millions of traffic shaping and policing rules across all customers, which can no longer be supported by a commodity switch, and needs to be implemented in software using approaches similar to [22, 23].

Access Control Lists (ACLs). Cloud providers may want to enable customers to specify access control lists (ACLs) for traffic between their on-premise network and virtual networks on the cloud. A typical commodity switch can only handle a few thousand access control rules, which is not enough for the possibly hundreds of customers connecting to the cloud's edge. Thus, we can use the software data plane on the server to fit all the ACLs at the edge, and use the ACL tables on the switch to cache the most used entries using approaches similar to [16].

Network Address Translation (NAT). Many enterprises use public cloud services for storage, data analytics, etc. Traffic from a customer's on-premise private network to these public services needs to be NAT'd before entering the cloud's network. The cloud provider can offer NATing as a service to make access to public services transparent to the customer. Similar to the previous cases, support for NATing exists in commodity switches, however, with a limit on the number of concurrent connections they can track. Thus, to implement NAT on the PE, we can keep track of connections in software, and push NATing rules for a subset of active connections to the switch.

In all the above examples, as well as the VPN gateway, we manually distributed the functionality between hardware and software. However, given the benefits of a hybrid data plane for implementing network functions, an interesting avenue for future work is to automate this process. To do so, we need high-level abstractions for specifying i) the packet processing stages of a network function (e.g., a data-plane programming language similar to P4 [11]), ii) the requirements of each stage (e.g., sizes of the tables), and iii) capabilities of the target data planes (hardware and DPDK-based software data plane in PE's case). Next, we need a compiler to use these specifications to automatically distribute packet processing stages across the target data planes.

Previous work in the literature has studied a similar automation problem, with high-level languages to describe the desired packet processing stages and programmable switching chips as targets [15, 24]. They specify target capabilities in terms of the capacity of their tables, the type of memory they use, and the headers they can match. A complier then uses this information to translate the high-level packet processing programs into target configurations. One could extend these abstractions to express capabilities of heterogenous data planes and design a compiler that distributes packet processing stages of network functions across these data planes considering their specified capabilities.

3. PROTOTYPE

We have built a prototype of the PE and implemented a VPN gateway on top of it to evaluate its scalability and performance. The prototype uses Q-in-Q for L2 tunnels, and VXLAN for L3 tunnels. It has a commodity switch with 32 40G interfaces and the Broadcom's Trident II chipset,

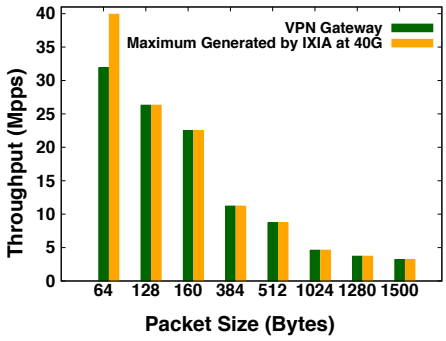

Figure 5: Maximum throughput (Mpps) of VPN gateway compared to the maximum generated by IXIA at 40G.

which is capable of Q-in-Q and VXLAN tunneling. The commodity server in the prototype has two Xeon(R) 2.3 GHz CPUs, each with a 45MB L3 cache and 18 Cores. We only use one CPU in our experiments. The server is connected to the switch with a 40G Intel XL710 NIC.

We run a C program on the server that is executed on multiple cores and uses DPDK libraries for fast packet processing. More specifically, we dedicate a set of *receive* cores to pull packets from receive queues on the NIC. Each receive core itself has a queue, implemented using DPDK's ring library with lock-free enqueue and dequeue, in which it places the packets it receives from the NIC. A set of *worker* cores pull packets from these queues, process them as follows, and direct them back to the switch through the NIC.

Each worker receives Q-in-Q packets from customers destined to their VMs on the cloud. Q-in-Q packets have two 802.1Q headers, each containing a 12-bit VLAN identifier. We use the outer identifier to differentiate customers (C), and the inner one for their overlay networks (O). We consolidate all the overlay routing tables into one by merging the Overlay Demux table with overlay routing tables in Figure 3. This table, implemented as a DPDK hash table, maps the overlay identifier (C, O) and the IP address of a VM in the overlay, to the IP address of the server hosting the VM and the VXLAN tunnel identifier for that overlay. Thus, after extracting C and O and removing the 802.1Q headers, the program matches the packet against the consolidated overlay routing table, wraps it in a VXLAN header, and sends it back to the switch.

4. EVALUATION

Given that the throughput bottleneck is the software data plane in DPDK and not the switch, we focus our evaluation on the performance of the DPDK program. We use an IXIA packet generator [6] to generate packets at 40G and measure the maximum throughput achievable for packets from customers to the cloud that need to go through the server. For a large-scale cloud provider, we expect the overlay routing table to have a few million entries. However, to explore the effect of CPU cache hit rate on performance, we run our experiments with one, one million, and 16 million entries in the overlay routing table in the DPDK program and configure IXIA to generate packets uniformly at random such that they hit all the overlay routing entires. In our prototype (Section 3), the CPU's L3 cache is large enough to store one million entires, but not 16 million.

One Entry. Figure 5 shows the throughput for different packet sizes in terms of Mpps, compared to maximum Mpps that IXIA generated at 40G, when the overlay routing table has a single entry. The XL710 controller for our 40G NIC is not designed to operate at line rate for packets smaller than 128 bytes, which explains the 20% gap between the maximum possible throughput with IXIA and what our VPN gateway achieves for 64-byte packets. For packets larger than 128 bytes, our VPN gateway can keep up with line-rate at 40G. We used two receive cores, as well as 6 worker cores for 64-byte packets, 4 cores for 128 and 160-byte packets, and two cores for the rest.

16 Million Entries. With 16 million entries, the overlay routing table does not fit on the CPU's L3 cache. Our VPN gateway achieves the same throughput as the case with one entry (Figure 5); however, we need 10 worker cores for packet sizes of 64 and 128 bytes, 8 cores for 160-byte packets, 4 cores for for 384 and 512-byte packets, and two for the rest.

One Million Entries. To verify that the performance difference in the two previous experiemnts is largely due to the 100% hit rate on the L3 cache, we repeat the experiment with one million entries in the overlay routing table, so that it completely fits in the L3 cache. We achieve the same throughput with the same number of cores as the case with a single entry in the overlay routing table.

We also measured PE's average latency for these packets, which is between 16 to 20 microseconds in all three experiments, and includes the time spent in both the server for overlay routing and the switch for underlay routing. For packets from the cloud to the customers, which only go through the switch, the throughput is 40G for all packet sizes and latency is between 2 to 3 microseconds.

We plan to do more extensive experiments to quantify the trade-offs between the PE's software and hardware data plane. More specifically, we plan to explore the implications of DPDK's run-to-completion processing model on its performance by increasing the complexity of the DPDK program in terms of the number of table lookups and branches. Moreover, we are going to study, in more detail, the effect of CPU caches on the server as well as caching DPDK's table entries on the switch on performance. However, these preliminary results encourage us that with more NICs, cores, and another CPU on the server, and extensive caching, our VPN gateway on PE can achieve $\sim$ 100G throughput.

5. RELATED WORK

Previous works have looked into implementing software [18] and hardware [13] VPN gateways to connect a customer's on-premise network to its virtual network in a data center. However, they only scale to a few thousand tunnels, which is reasonable for larger-than-average enterprise deployments, but not large-scale clouds [13]. PE combines software and hardware to scale up to a million tunnel entries.

Several previous works [16, 20, 21] have explored co-designing data planes with a slow processing path and a fast one serving as its cache. PE's hybrid data plane offers similar benefits, but also allows developers to use each of slow path (software) and fast path (hardware) separately for different packet processing stages of a network function. Duet [14], a cloud-scale load balancer, uses the ECMP and tunneling features of commodity switches for assigning flows destined to virtual IP addresses (VIPs) to dynamic IP address (DIPs), and tunneling them to their assigned DIP.

Similar to our PE, Duet also faces the problem of limited number of entries for tunneling and ECMP in switches and solves that by distributing the load balancer across multiple switches. PE needs to run at the edge, so it cannot scale using switches in the data center, and therefore solves the problem by connecting a commodity server to the switch to take advantage of its larger memory and the possibility of doing arbitrary packet processing in a software data plane. Duet also uses commodity servers in its design, but only as backups in case of switch failures.

Moreover, NBA [17] and ClickNP [19] explore offloading Click modules to GPUs and FPGAs, respectively. However, these platforms cannot support the required port density at *the cloud's edge*. PE is similar to these works as it explores offloading parts of network functions to hardware, a commodity switch. Unlike these works, we have not focused on high-level programming languages for PE's hybrid data plane. Moreover, we could use NBA and ClickNP for the commodity server in PE's design to improve performance. Additionally, OpenBox [12] proposes packet processing graphs to describe network functions, and develops algorithms for composing them. Its data plane consists of a set of heterogenous elements, each capable of implementing part of the packet processing graphs. However, the paper assumes that the assignment of sub-graphs to data-plane elements is given by the programmer, and does not explore the trade-offs between different assignment strategies based on the capabilities of the target elements.

Finally, programmable switches (e.g., Tofino [9]) enable network operators to specify and run a custom packet processing pipeline, and have the switch allocate resources among the pipeline stages accordingly. As a result, a programmable switch can potentially fit more tunnel lookup entries than a fixed-function one, which can waste part of its resources to implement data-plane functionalities not used in a VPN gateway. However, a VPN gateway is only one of the several applications that benefit from running at the edge (Section 2.3). Therefore, the problem of limited switch resources remains valid as we do not want to dedicate all the tens of megabytes of the switch's on-chip memory to a single application. Exploring how to assign packet processing stages to a hybrid of software and programmable hardware data plane is an interesting avenue for future research.

Acknowledgements

This work is supported by NSF grant CNS-1409056. We thank the anonymous reviewers for helpful feedback.

6. REFERENCES

[1] Cloud Computing Trends: 2016 State of the Cloud Survey. http://www.rightscale.com/blog/cloud-industry-insights/cloud-computing-trends-2016-state-cloud-survey#enterpriseworkloads. Accessed: January 2018.

[2] DPDK. http://dpdk.org/. Accessed: January 2018.

[3] Enterprise Adoption Driving Strong Growth of Public Cloud Infrastructure as a Service, According to IDC. https://www.idc.com/getdoc.jsp?containerId=prUS41599716. Accessed: January 2018.

[4] ExpressRoute. https://azure.microsoft.com/en-us/services/expressroute. Accessed: January 2018.

[5] Google Cloud Interconnect. https://cloud.google.com/interconnect/. Accessed: January 2018.

[6] IxNetwork. https://www.ixiacom.com/products/ixnetwork. Accessed: January 2018.

[7] Next-Generation Enterprise Branch Network Communications in a Cloud-Connect Environment. https://www.globalservices.bt.com/static/assets/pdf/campaign/Network\%20like\%20never\%20before/IDC_Analyst_Connections_Briefing_Document.pdf. Accessed: January 2018.

[8] Roundup of Cloud Computing Forecasts and Market Estimates, 2016. http://www.forbes.com/sites/louiscolumbus/2016/03/13/roundup-of-cloud-computing-forecasts-and-market-estimates-2016/#1c86a8c774b0. Accessed: January 2018.

[9] The World's Fastest & Most Programmable Networks. https://barefootnetworks.com/resources/worlds-fastest-most-programmable-networks/. Accessed: January 2018.

[10] AWS Direct Connect. https://aws.amazon.com/directconnect. Accessed: January 2018.

[11] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *ACM SIGCOMM CCR*, 2014.

[12] A. Bremler-Barr, Y. Harchol, and D. Hay. OpenBox: a software-defined framework for developing, deploying, and managing network functions. In *SIGCOMM*, 2016.

[13] B. Davie, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. Gude, A. Padmanabhan, T. Petty, K. Duda, and A. Chanda. A Database Approach to SDN Control Plane Design. *ACM SIGCOMM Computer Communication Review*, 2017.

[14] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang. Duet: Cloud scale Load Balancing with Hardware and Software. In *SIGCOMM*, 2014.

[15] L. Jose, L. Yan, G. Varghese, and N. McKeown. Compiling Packet Programs to Reconfigurable Switches. In *NSDI*, 2015.

[16] N. Katta, O. Alipourfard, J. Rexford, and D. Walker. Cacheflow: Dependency-aware rule-caching for software-defined networks. In *SOSR*, 2016.

[17] J. Kim, K. Jang, K. Lee, S. Ma, J. Shim, and S. Moon. NBA (network balancing act): A high-performance packet processing framework for heterogeneous processors. In *EuroSys*, 2015.

[18] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, P. Ingram, E. J. Jackson, et al. Network Virtualization in Multi-tenant Datacenters. In *NSDI*, 2014.

[19] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, and P. Cheng. ClickNP: Highly flexible and high-performance network processing with reconfigurable hardware. In *SIGCOMM*, 2016.

[20] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim. Language-Directed Hardware Design for Network Performance Monitoring. In *SIGCOMM*, 2017.

[21] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, et al. The Design and Implementation of Open vSwitch. In *NSDI*, 2015.

[22] S. Radhakrishnan, Y. Geng, V. Jeyakumar, A. Kabbani, G. Porter, and A. Vahdat. SENIC: Scalable NIC for End-Host Rate Limiting. In *NSDI*, 2014.

[23] A. Saeed, N. Dukkipati, V. Valancius, C. Contavalli, A. Vahdat, et al. Carousel: Scalable Traffic Shaping at End Hosts. In *SIGCOMM*, 2017.

[24] C. Schlesinger, M. Greenberg, and D. Walker. Concurrent NetCore: From policies to pipelines. In *ICFP*, 2014.