InKeV: In-Kernel Distributed Network Virtualization for DCN

Zaafar Ahmed
FAST National University
zaafar.ahmed@sysnet.org.pk

Muhammad Hamad Alizai
CS Department, LUMS
hamad.alizai@lums.edu.pk

Affan A. Syed
PLUMgrid Inc.
asyed@plumgrid.com

ABSTRACT

InKeV is a network virtualization platform based on eBPF, an in-kernel execution engine recently upstreamed into linux kernel. InKeV's key contribution is that it enables in-kernel programmability and configuration of virtualized network functions, allowing to create a distributed virtual network across all edges hosting tenant workloads.

Despite high performance demands of production environments, existing virtualization solutions have largely static in-kernel components due to the difficulty of developing and maintaining kernel modules and their years-long feature delivery time. The resulting compromise is either in programmability of network functions that rely on the data plane, such as payload processing, or in performance, due to expensive user-/kernel-space context switching.

InKeV addresses these concerns: The use of eBPF allows it to dynamically insert programmable network functions into a running kernel, requiring neither to package a custom-kernel nor to hope for acceptance in mainline kernel. Its novel *stitching* feature allows to flexibly configure complete virtual networks by creating a graph of network functions inside the kernel. Our evaluation reports on the flexibility of InKeV, and in-kernel implementation benefits such as low-latency and impressive flow creation rate.

1. INTRODUCTION

Network Virtualization (NV) allows the creation of multiple virtual networks over the same physical network. This technology is fundamental in multi-tenant datacenters, where each tenant expects a logically isolated networking infrastructure between its workloads (i.e., tenant owned virtual machines). Since SDN provides the required underpinning of a programmable control abstraction for rapid provisioning, instead of box-by-box configuration, it has been globally adopted as the preferred approach to NV in multi-tenant datacenters. While the idea of NV is not new, the unique requirements of DCNs pose new challenges: for example, each tenant wants to overlay an isolated, potentially unmodified enterprise-network topology on a single physical network. A preferred approach to this is to connect tenant VMs to a *distributed virtual network* which is independent of the underlying physical topology of DCN [15, 5]. A distributed virtual network can configure a complete tenant-specific logical topology, by chaining virtual network functions (VNF), inside each edge[1] hosting tenant workloads. Outbound packets thus traverse the whole logical topology inside the source edge before being tunneled to a remote workload; the *overlay-model* for SDN. This is like shifting the network virtualization to compute domain (edges), requiring nothing more than an IP-tunneling facility from the underlying DCN.

While a promising concept for NV in datacenters, distributed implementation of virtual networks is challenging as production-level performance prefers in-kernel implementations [21]. However, current approaches to adding kernel-level functionality either require packaging a custom kernel or upstreaming to mainline kernel, both having drawbacks. The packaged, but non-standard, kernel approach has resulted in NV solution providers experiencing hesitancy from customers for their production systems, along with concerns for packaging and upgrading kernel modules. Upstreaming functionality into mainline kernel introduces the Linux kernel maintainers as an agnostic-to-business-concerns third party, typically leading to years long innovation cycle. These arguments are echoed by the architects of Open vSwitch (OVS): *"When Open vSwitch started on Linux, only in-kernel packet forwarding could realistically achieve good performance, so the initial implementation put all OpenFlow processing into a kernel module... This approach soon became impractical because of the relative difficulty of developing in the kernel and distributing and updating kernel modules. It also became clear that an in-kernel OpenFlow implementation would not be acceptable as a contribution to upstream Linux, which is an important requirement for mainstream acceptance for software with kernel components" [21].*

A resulting compromise has been to implement data plane (DP) of any VNF by splitting functionality between a user-space component (e.g. *ovswitchd* in OVS) and a kernel component (e.g. *datapath* in OVS) to allow network acceleration by, for example, caching exact flow-rules inside the kernel. This split, resulting from the inability to safely make the kernel programmable, demands that user-space of the DP implementation is *thick*, with only a *thin* (and configurable) implementation in the kernel (cf. Figure 1). However, this skewed split results in additional latency, due to packet copies and context switching, in every scenario requiring packet processing in user space. Such a penalty is most significant for VNFs that implement per-packet processing (e.g., middleboxes for encryption and IPS) where every packet has to traverse into the userspace (VNF_2 to VNF_n in Figure 2), as we show in Section 4. However, DP for even simple VNFs can incur this penalty for the first packet of a flow (VNF_1 in Figure 2), affecting the flow-

[1] *simply*, servers inside DCN

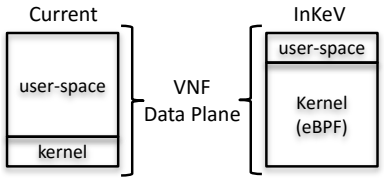


Figure 1: Logical difference between VNF data plane implementations, current and InKeV.

creation rate. Thus, for example, with a wildcarded rule in OVS, every packet for a new flow exits the kernel as its datapath module only caches an exact match. We, therefore, argue that an in-kernel *programmable* DP is essential for network acceleration, as it removes the above two penalties found in the thin-kernel approach and allows us to construct distributed implementations of a VNF data-plane.

An alternate approach to add functionality inside the kernel has been recently introduced in the form of eBPF, mainlined into Linux kernel 3.18 and above [14]. eBPF is an in-kernel engine that allows dynamic insertion of code via its own syscall. The ability of eBPF to hook code to any kernel event — packet capture or system calls — provides users with a generic ability to respond to events at nano-second granularity. While we already see widespread use of eBPF in tracing and seccomp [9], more importantly its in-kernel programmability allows us to *revisit* the DP implementation choices for NV. Thus, we can now dynamically insert a programmable implementation of a data-plane, inside the kernel of commodity machines, without any third-party bottleneck to feature delivery time.

InKeV is the *data plane* for a NV platform built around eBPF to retain the benefits of an in-kernel DP without having to compromise its flexibility and programmability. Figure 2 illustrates the difference between a thin-kernel implementation of a DP (Figure 2a) necessitating frequent user-/kernel-space context switching, and InKeV's thicker and programmable DP (Figure 2b). InKeV's key contributions, utilizing the in-kernel programmability of a network function's DP inherited from eBPF, therefore are:

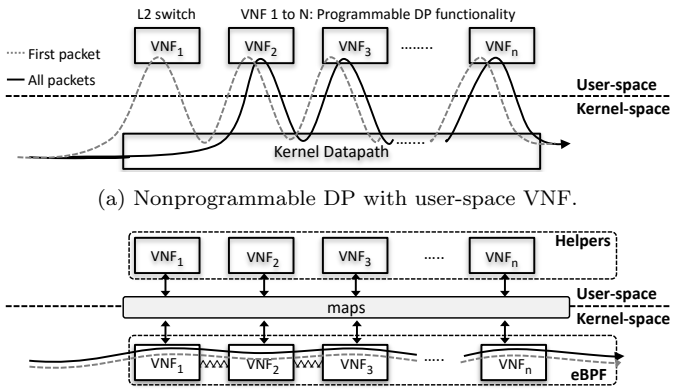
- An **Open-Source orchestration architecture** [2] to dynamically manage the insertion, stitching, and deletion of programmable DP elements at any edge, facilitating any overlay-based NV solution (Section 3).
- **Network acceleration** due to in-kernel VNFs as demonstrated by a virtual network deployment and comparison with state-of-the-art NV solution based on OVS (Section 4).

For brevity, we limit our target environment for InKeV to distributed virtual networks in multi-tenant datacenters. We shed light on its general feasibility across the NV research domain in Section 5.

2. BACKGROUND: INTRODUCING eBPF

We first briefly introduce the key features of eBPF which is the core technology used in InKeV, to facilitate understanding of its architecture in Section 3.

eBPF (extended Berkeley Packet Filter) is an extension of BPF [16], which introduced the concept of an in-kernel interpreter to translate bytecode for a fictional machine architecture. The instruction set for this machine however was



(a) Nonprogrammable DP with user-space VNF.

(b) InKeV's programmable DP.

Figure 2: A logical comparison of packet path between user-space VNFs in a nonprogrammable DP and InKeV enabled programmable DP with *stitched* VNFs. *Maps* provide an asynchronous medium for in-kernel implementations to communicate with companion userspace programs (*helpers*). Details in Section 2 and 3.

purposefully limited, enough to enable packet filtering and allow line-rate network monitoring but not more to allow easy verification of code stability. eBPF extends this machine architecture to allow implementing greater programming constructs inside the kernel rather than just packet header matching. eBPF introduced an enhanced 64-bit machine architecture along with support for typical programming constructs such as *call*, *load*, *store* and *conditional jumps*. eBPF also includes several helper functions, a library of routines that eBPF programs can utilize to interact with the host machine kernel. These helpers[2] provide a diverse set of functionality including, among many others, getting time and generating random numbers. Similar to BPF, eBPF also provides kernel-safe code execution by first verifying an inserted code through an exhaustive search of code-paths, as well as providing native performance guarantees through JIT compilation to x86 and ARM machine architectures.

One of the most important feature with regard to InKeV implementation is the support for maps: a $< key, value >$ data structure. The $< key, value >$ pairs in the maps are kept as binary blobs, allowing their program-specific interpretation. Thus, the map construct provides a generic data structure facility for in-kernel algorithms allowing, importantly, to keep state between packet arrivals. These maps are accessible (all basic primitives of `create, lookup, update, and delete`) from user-space using `bpf()` syscall and via helper functions from within eBPF code. Thus, they also enable asynchronous communication between eBPF modules and user-space programs allowing configuration and contextualization of the eBPF program execution. These maps are also accessible in parallel from any eBPF module, if provided with the corresponding file descriptor.

eBPF also provides the ability to *chain* together independent programs by using a special type of map that only contains mappings from one eBPF module descriptor to another (`BPF_MAP_TYPE_PROG_ARRAY`). This map is used to *tail*

[2]For an exhaustive list, see the man page at http://man7.org/linux/man-pages/man2/bpf.2.html

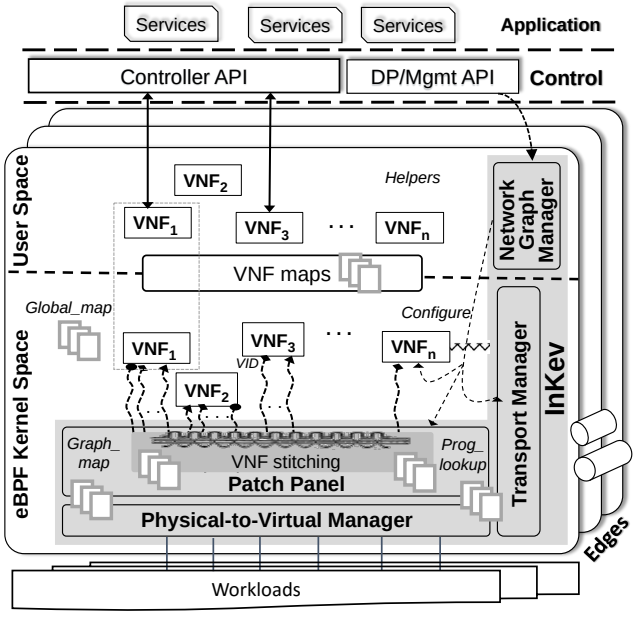


Figure 3: InKeV architecture: The VNF modules are inserted, deleted, and managed by Network Graph Manager. Patch-panel, Physical-to-Virtual Manager, and Transport Manager implement the stitching between modules, interaction with physical workloads, and transport to remote edges, respectively. *Maps* serve as an asynchronous communication point between modules.

call from one eBPF program to another, passing along the context (packet) directly and also avoiding the switching overhead of recreating a call stack. This ability is important as we use it to create programmable DPs and configuring logical virtual network topologies inside the kernel, as we describe next.

3. IN-KERNEL VIRTUAL NETWORKS

InKeV allows creating a distributed virtual network inside edges hosting tenant workloads by stitching together eBPF-based data-planes; an approach similar to NVP [15], Each packet traverses the complete logical topology of the virtual network on source edge before being tunneled via the underlay to the destination edge.

InKeV architecture (cf. Figure 3) is inspired from the generic framework of the IO Visor Project [8], which extends the notion of eBPF to have a general abstraction for managing and programmatically responding to IO events. In InKeV, we use the concept of a Network Graph Manager responsible for creating a connected graph of VNF *data-planes* thus creating the virtual network at each edge. We first elaborate the definition and interfaces for a VNF DP and then describe the InKeV architecture responsible for stitching these VNFs together.

3.1 VNF Modules

VNFs represent software-based implementation of any network functions: from switches, routers to firewalls and flow classifiers. We define the DP of these VNF, henceforth referred to as *VNF modules*, implementation split between a thin user-space helper and an eBPF-based kernel component. This functional separation corresponds to the explanation in Section 2, where a user-space program can interact with eBPF modules using maps.

The user-space component of a VNF module can have two interfaces; an optional north-facing interface for interaction with the control plane of the corresponding network function. The south-facing interface enables interaction with its in-kernel eBPF module by polling the associated maps (e.g., for communicating statistics or configuration). However, to punt entire packets to the helper (e.g., for flow creation through an SDN controller), we can have the eBPF code send packet to a virtual interface on which the helper will be listening. The in-kernel module implements the DP of a network function as an eBPF code[3]. Each VNF has Virtual Interfaces (VIs) for packet ingress and egress. It is configured for its number of VIs and MAC addresses via maps. Optionally, function-specific maps can be used to communicate statistics.

3.2 InKeV Architectural Components

InKeV architecture acts as the "operating system" which can accept VNF modules (think "programs"). It consists of the Network Graph Manager, a user-space component, which is responsible for interacting with the DP configuration component of a larger, overlay-based, NV solution. The other three InKeV components (cf. Figure 3) are in-kernel eBPF modules implementing following functions: (i) interfacing packets between workloads and the virtual network (physical-to-virtual manager), (ii) stitching VNF modules (patch panel), and (iii) tunnel packets between workloads on different edges (transport manager).

3.2.1 Network Graph Manager (NGM)

The NGM creates the DP of a virtual network by importing, validating, and installing a graph of VNF Modules. For this purpose, it needs a virtual network configuration in terms of the constituent VNF modules and their connectivity. This configuration can be used to, for example, migrate tenant specific topologies to data centers.

With a virtual network configuration in place, NGM installs the relevant VNF DPs using the IO Visor compiler system for eBPF, the BPF compiler collection (BCC) [1], to appropriately package user/kernel space modules. As we explain below, each module's VI(s) needs a unique identifier (VID), assigned by NGM, within an edge. The NGM finally pushes the connectivity matrix to the *patch panel*, via the *Graph_map*, to stitch eBPF modules forming a virtual network inside the kernel. As an example, a simple virtual network consisting of two subnets can comprise two bridge VNF modules and a router VNF module. The VI of each bridge is stitched to a different VI of the router; thus forming a basic B-R-B topology.

3.2.2 Physical-to-Virtual Manager (PVM)

PVM represents the logic to interface physical endpoint (workload VM/container) and the in-kernel virtual network. The network interface of an endpoint, typically a tap or veth interface, has an `ifindex` registered with PVM. Packets coming out of an endpoint will transit PVM, which is an eBPF program hooked to the *tc* ingress queue, capturing packets as the `sk_buff` structure. PVM will also generate a unique VID for this `ifindex` to enable phy-to-virtual

[3]A future possibility is to extract this from a P4 like description of DP into eBFP code, an effort currently underway [4].

mapping. This VID is appended to the packet and is then delivered to the patch panel (as described next). Oppositely, when a packet from the virtual network has to be delivered to a workload, a similar mapping between a VID and the destination `ifindex` is maintained inside the PVM. The PVM uses the `bpf_clone_redirect()` helper function to deliver this packet to the appropriate workload interface. Overall, PVM acts as a map between network stack of the workloads and the InKeV's virtual network.

3.2.3 Patch Panel

The Patch Panel (PP) implements the actual stitching to create a virtual graph between VNF modules. Each VNFs thus forward its packets, with their VID as metadata, to the patch panel. This is achieved by using PP's well-known index in a *Global_map* of type *PROG_ARRAY* (see Section 2), allowing a tail call into the PP's code. Furthermore, the metadata containing the VID, is appended to the `sk_buff.cb` field as convention. However, for the case requiring packet traversal to another edge (e.g., for a distributed bridge), the VNF module use the interface provided by the transport manager (described next).

PP uses two maps, *Graph_map* and *Prog_lookup*, to achieve packet traversal inside a virtual network. The *Graph_map* uses the connectivity matrix to stitch VIDs. Upon receiving a packet, a simple lookup on its source VID metadata provides the VID of packet destination. The *Prog_lookup*, then, provides a mapping between the destination VID and the corresponding eBPF module.

With this implementation, we can now connect VNFs in an arbitrary graph to build a distributed virtual network within an edge.

3.2.4 Transport Manager (TM)

The TM is responsible for tunneling packets between tenant VMs on different edges, an essential characteristic of multi-tenant datacenters. For this purpose, underlay tunnels are created (using e.g., GRE, VXLAN, or Geneve) enabling overlay networking between workloads on different edges [15]. It is important to note that the eBPF module implementing TM keeps track of tenant-to-tunnel information, thus ensuring isolation and security. However, the actual tunnel creation can use native-to-kernel facilities (like VXLAN tunneling), thereby allowing us to best utilize kernel software as well as hardware offload capabilities. TM provides an API, which is also registered in the *Global_map*, to VNFs. We require, as part of the API, that a packet to TM should contain the appropriate tunnel ID. The TM will use this identification to either forward the packet to a workload on the same edge, or over the tunnel, or drop it if the corresponding tunnel does not exist. The information about the correct tunnel and their creation remains a function of the NV platform implementation; since this paper focuses on the programmable DP we do not further discuss these details.

The accumulated functionality of the four architectural components delineates our **first contribution**, i.e., an orchestration architecture to create distributed virtual networks inside the kernel of all edges hosting tenant workloads.

4. InKeV NETWORK ACCELERATION

We now empirically demonstrate our second contribution, i.e., InKeV's network acceleration. With regard to NV in datacenters, NVP [15] and OpenStack's Neutron[4] [19] represent the state-of-the-art, and hence candidates to be compared with InKeV. Since both these platforms use OVS as their software data-plane, their in-kernel implementations are limited to exact match flow-rules. Thus, any VNF in the virtual topology (L2-above for Neutron and middleboxes in NVP) results in packet exiting in-kernel DP for VNF processing in a namespace/process/container, which may or may-not be hosted on the same edge. InKeV by the virtue of its programmable DP allows to chain the DP of VNFs inside the kernel, avoiding packet exits from the kernel. In this evaluation we compare InKeV with Neutron because it is open source and a representative of state-of-the-art for the *data plane* of NV platforms that are based on Open vSwitch.

4.1 Latency

Latency is an important performance parameter in multi-tenant DCN, where even a fraction of a second can impact user experience and lower operating revenues [12]. With our programmable in-kernel DP of VNFs, we can push code to exponentially increase the functionality of DP, without requiring any user-space context switch. Whereas, for OVS based platforms, any programmable VNF in the data path will result in user-space switching.

Setup: The latency experiments are performed on a single, i7/32GB, 3.4Ghz, ubuntu-14 desktop machine. We use devStack install of OpenStack, using the stable liberty release [18]. We also upgrade the Kernel to version 4.3, with the BPF Compiler Collection (BCC) tools to implement InKeV's components [1].

We use a single machine setup to present the best-case latency comparison with InKeV, even though reference implementation of Neutron requires VNFs as Linux namespaces on a separate "network node". This single machine setup eliminates latency incurred due to any underlay/fabric traversals. We want to evaluate the traversal similar to in Figure 2, where a packet has to exit into user-space for packet processing. To emulate this behavior we use the basic router implemented as a network namespace, connecting two different subnets represented by two bridges (a VNF requiring user-space punt). To extend the chain of such network functions we repeat this B-R-B topology where each bridge connects a workload[5]. We thus create eleven bridges to host ten routers and calculate latency as a function of increasing size of this chain by pinging to hosts on different subnets. Each experiment, repeated thirty times, uses a ping test to calculate latency. We use a $60s$ interval to allow flows to timeout and necessitate user-space transits.

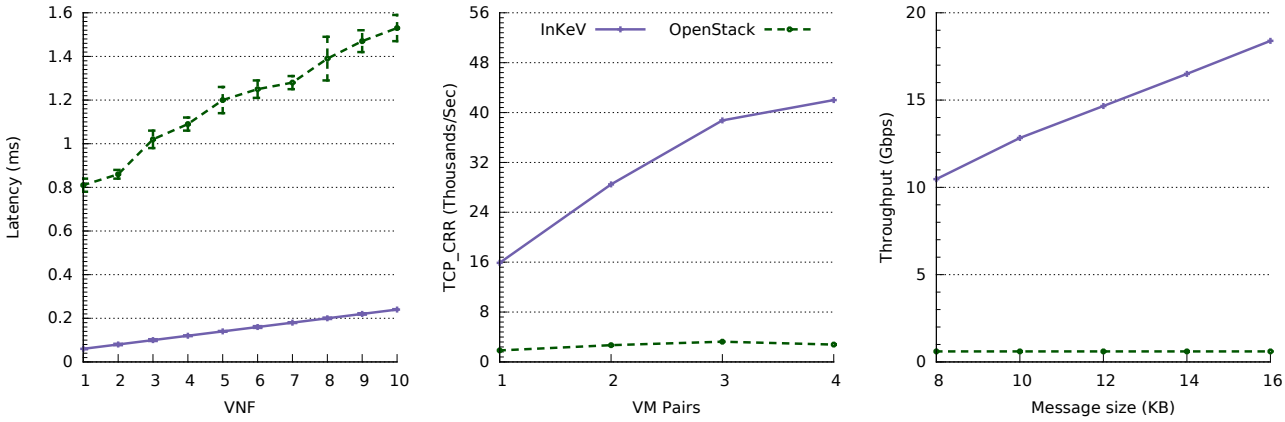
Results: Figure 4a summarizes an order of magnitude improvement in latency for experiments where each workload in the topology pings all other workloads. The x-axis depicts different experiments each with different number of routers in the topology. The error bars represent standard error.

We expect these results to largely apply when compared with any user-space, programmable DP solution as they mainly depict the latency associated with frequent user-/kernel-space transits.

4.2 CRR and Throughput

Next to latency, we evaluate the impact of InKeV on

[4]OpenStack's reference implementation for NV in DCN.
[5]We manually add routes to connect workloads.



(a) Latency (b) Flow Creation Rate (10 pkts per flow) (c) Throughput

Figure 4: InKeV vs. Neutron: Evaluating the benefit of traversing the DP of an in-kernel distributed virtual network.

connection-request-response CRR test [13] — a system level metric that is affected by latency to CP — and throughput.

Setup: We now deploy a multi-node OpenStack setup with the network node (where an OpenStack router namespace is hosted) on a different edge. This setup consists of two desktop machines (specs as before) connected via a 1Gbps switch. Thus, while the InKeV implementation of the B-R-B topology resides on the same physical edge requiring no transit on the underlay, a reference multi-node OpenStack implementation will require traversal over the underlay to the router name-space at flow-creation time. This example thus emulates any network function that requires hairpinning to outside-kernel, e.g., service insertion of functions into a NV solution.

Results: Figure 4b shows the results for the CRR test using *netperf* as we increase the number of VM pairs. We can clearly see that InKeV sustains an order-of-magnitude faster CRR due to its ability to create flows inside the kernel. This improved CRR eventually translates into better response for application flows requiring frequent connection establishment (handshakes), such as HTTP.

To further investigate the impact on raw throughput, we calculate the throughput over a single flow (cf. Figure 4c), and observe that the OpenStack network hits a throughput limit around 0.6Gbps, primarily due to the 1Gbps physical network which it has to traverse. Instead, InKeV's in-kernel traversal of the network allows delivery of packet, across different virtual subnets, to confront no such constraints, reaching 20Gbps with the default netperf message size, when the source and destination workloads are on the same physical machine.

Overall, this evaluation uses multiple metrics to demonstrates the benefits of InKeV's implementation of a distributed, in-kernel, virtual networks using programmable DPs.

5. DISCUSSION

Our main technical contribution is showing how a very nascent technology, i.e. eBPF, can be used to not only build in-kernel programmable VNFs but also (and in a novel way) how to interconnect them on a single edge and across different edges. The implementation required a significant engineering effort but is essential to demonstrate the full potential of a flexible and in-kernel alternative to NV. We consider the CP aspect, to orchestrate the creation of these VNFs and extending these VNFs to be distributed across multiple edges, as a complimentary work beyond the current scope. Nonetheless, the benefits of InKeV overlap and extend to two related fields.

5.1 NFV and InKeV

Network Function Virtualization (NFV) presents an intersection of SDN and cloud technologies specialized to satisfy the demands of the Telecom Operators [17]. The NFV use-case however demands carrier grade performance of network functions and 99.999% (or five nines) reliability, but with virtualized workloads over commodity hardware. This requires efforts to improve the packet processing performance of DP, especially in the context of virtualized workloads that share physical interface(s). Here, one concern has revolved around line-rate packet processing alternative to the slower standard Linux networking stack. For example, a zero-copy approach, i.e., sharing buffers between user & kernel space, in PF_RING [6] is employed to bypass the kernel stack for fast processing in the user-plane [7]. Similary, for VMs—commonly used in cloud environment—PCI passthrough allows direct access to physical NICs. To scale such an approach to multiple VMs sharing a NIC, techniques like SR-IOV are used [10]. However, above approaches require specialized drivers restricting them to specific hardware NICs [7, 10]. More importantly, with VM/container based workloads that share the kernel of a host machine, packet traversal includes kernel and then to any user-space program — incuring the overhead of a per-packet upcall (cf. Figure 2 in [11]). NetVM tackles this issue through an architecture that extends DPDK with support for "virtual bumps" to chain processing logic in user space [11], requiring modification in the hypervisor. Furthermore, the architectures for container based workloads, or containers *inside* VMs, are not immediately supported. With an eBPF engine, its associated helper functions, and maps, packets entering a kernel can be captured, processed, forwarded — all with native performance on any commodity machine.

Similarly, the stitching feature of InKeV can be extended, straight-forwardly, to chain together VNFs and implement

the service chaining concept. For such an extension the CP, orthogonal to our DP architecture here, will be different from a typical NV platform. It can, for example, focus more on orchestration and load-balancing to service chains rather than distribution of network functionality.

5.2 Programmable DP and InKeV

A second overlapping area to InKeV arises from work to make DP, along-with CP, programmable and thus enable SDN implementation to be truly vendor-agnostic and future proof. P4 provided a template for this approach, by arguing for a platform independent specification of DP functions [3]. OVS provides a flow-based data-plane API to configure network functionality, along with an in-kernel datapath module that ensures native performance *once* a flow is written [21]. However, its kernel module remains a bottleneck in implementing new features and protocols, requiring either a custom-kernel or longer delivery times. InKeV provides, through its eBPF-based VNF abstraction, an ability to build and stitch programmable network functions. Interestingly, this approach is validated by the recent use of eBPF-engine to augment OVS and support network functionality specified in the P4 format [20]. Similarly, efforts are already underway to write a P4 compiler that generates eBPF code, allowing P4 to integrate with any commodity machine [4]. Thus, it is possible that InKeV can, in the near future, incorporate VNF specified in P4. This possibility will enable InKeV to host third-party VNF, specified in the P4 language, in its virtual network graph.

6. CONCLUSIONS

We presented an eBPF based DP architecture that brings the benefits of in-kernel implementations to the network virtualization domain while removing flexibility and delivery time concerns for kernel level implementations. InKeV's key contribution is an architecture to flexibly stitch together programmable and in-kernel DP of VNF to build a network graph that can be used to construct a distributed and accelerated NV solution. The performance evaluation highlights network acceleration of InKeV when compared with OpenStack's reference NV implementation using the OVS-based DP implementation, typical of most NV solutions. Although this paper maintains its focus on distributed virtual networks, the general feasibility of InKeV spreads across the whole NV domain including NFV, service chaining, and programmable DPs in SDN. Hence, exploring this general feasibility, and a complete CP architecture for NV delineates our primary future work.

7. REFERENCES

[1] Bpf compiler collection. `https://github.com/iovisor/bcc`, 2015.

[2] Z. Ahmed. ebpf patch panel. `https://github.com/zaafar/ebpf_turtle/tree/master/InKeV/core`, 2015.

[3] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[4] M. Budiu. Compiling p4 to ebpf. `https://github.com/iovisor/bcc/tree/master/src/cc/frontends/p4`, 2015.

[5] M. Casado, T. Koponen, R. Ramanathan, and S. Shenker. Virtualizing the network forwarding plane. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*, page 8. ACM, 2010.

[6] L. Deri et al. Improving passive packet capture: Beyond device polling. In *Proceedings of SANE*, volume 2004, pages 85–93. NLUUG Association, 2004.

[7] DPDK. List of supported nics. `http://dpdk.org/doc/nics`, 2015.

[8] L. Foundation. IO Visor Project. `https://www.iovisor.org/`, 2015.

[9] B. Gregg. ebpf: One small step. `http://www.brendangregg.com/blog/2015-05-15/ebpf-one-small-step.html`, May 2015.

[10] H. Guan, Y. Dong, K. Tian, and J. Li. Sr-iov based network interrupt-free virtualization with event based polling. *Selected Areas in Communications, IEEE Journal on*, 31(12):2596–2609, 2013.

[11] J. Hwang, K. Ramakrishnan, and T. Wood. Netvm: high performance and flexible networking using virtualization on commodity platforms. *Network and Service Management, IEEE Transactions on*, 12(1):34–47, 2015.

[12] K. Jang, J. Sherry, H. Ballani, and T. Moncaster. Silo: Predictable message latency in the cloud. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 435–448. ACM, 2015.

[13] R. Jones. Care and feeding of netperf. *Hewlett Packard Company*, 2007. `http://www.netperf.org/svn/netperf2/trunk/doc/netperf.pdf`.

[14] Kernelnewbies. Linux 3.18. `http://kernelnewbies.org/Linux_3.18`, 2015.

[15] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, N. Gude, P. Ingram, et al. Network virtualization in multi-tenant datacenters. In *USENIX NSDI*, 2014.

[16] S. McCanne and V. Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference*, pages 2–2. USENIX Association, 1993.

[17] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba. Network function virtualization: State-of-the-art and research challenges. 2015.

[18] OpenStack. Devstack liberty release. `https://github.com/openstack-dev/devstack/tree/stable/liberty`, 2015.

[19] OpenStack. Neutron. `"https://wiki.openstack.org/wiki/Neutron"`, 2015.

[20] B. Pfaff. P4 and open vswitch. `http://p4.org/p4/p4-and-open-vswitch/`, 2015.

[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 *12th USENIX Symposium on Networked Systems Design and Implementation*, 2015.