OpenNetVM: A Platform for High Performance Network Service Chains

Wei Zhang* Guyue Liu* Wenhui Zhang* Neel Shah* Phil Lopreiato* Gregoire Todeschi‡
K.K. Ramakrishnan† Timothy Wood*
*The George Washington University ‡INP ENSEEIHT †University of California Riverside

Abstract

Just as Software Defined Networking (SDN) research and product development was greatly accelerated with the release of several open source SDN platforms, we believe that Network Function Virtualization (NFV) research can see similar growth with the development of a flexible and efficient platform enabling high performance NFV implementations. Existing NFV research prototypes provide insufficient performance, flexibility, or isolation. Furthermore, high performance I/O platforms such as Intel's DPDK lack higher level abstractions. We present OpenNetVM, a highly efficient packet processing framework that greatly simplifies the development of network functions, as well as their management and optimization. OpenNetVM, based on the NetVM architecture, runs network functions in lightweight Docker containers that start in less than a second. The OpenNetVM platform manager provides load balancing, flexible flow management, and service name abstractions. OpenNetVM uses DPDK for high performance I/O, and efficiently routes packets through dynamically created service chains. Our evaluation achieves throughputs of 68 Gbps when load balancing across two NF replicas, and 40 Gbps when traversing a chain of five NFs, realizing the potential of deploying software services in production networks.

1. INTRODUCTION

Network Function Virtualization (NFV) promises to enable a vast array of in-network software functions running efficiently in virtualized environments. Network and data center operators alike envisage new functionality that can be deployed as virtual middleboxes, removing the expense and inflexibility of past hardware-based approaches. These functions range from lightweight software switches, high performance proxy engines to complex intrusion detection systems (IDS). Providers and practitioners seek to deploy these as parts of complex service chains comprising multiple network functions (NFs).

Recently, several high performance I/O libraries such as netmap, DPDK, and PF_RING have emerged to allow developers and researchers to build efficient NF prototypes. These libraries typically enable packet processing rates of 10 Gbps or higher by avoiding the kernel's networking stack and allowing direct access to packet data from a user space application. While this has been a boon for accelerating individual applications, these libraries do not assist with the composition of NFs nor their management. E.g., SoftNIC [2] and DPDK pipeline [5] only support fixed, pre-defined service chains, and they can not dynamically steer packets to NFs. Further, since libraries such as DPDK assume complete control of NIC ports and dedicate them to a single process, it is impossible to run several accelerated functions on the same server unless they have been carefully crafted to coexist with each other. Thus, while the low-level tools to build network functions are becoming available, we lack a platform that provides the higher level abstractions needed to compose them into service chains, control the flow of packets among them, and manage their resources.

From these needs we derive three design principles that guides our architecture:

- An NF management framework must be lightweight and efficient, while providing flexible packet steering.
- Both the management framework and NFs must have control over how packets are steered through service chains.
- NFs must be isolated, quick to deploy, and easy to develop by different vendors.

OpenNetVM encompasses these principles by providing a flexible and high performance NFV framework to support a "smart" data plane. Our work is based on the architecture developed for our NetVM platform [3], but extended to support flexible management capabilities, lighter weight NFs, and improved deployment and interoperability.

Container-based NFs: Network functions run as standard user space processes inside Docker containers, making them lighter weight than virtual machines, but still allowing a versatile linux development environment. This greatly simplifies NF deployment by multiple vendors since containers are self-contained and modular. Containers can be

HotMIddlebox, August 22–26, 2016, Florianopolis, Brazil
 ISBN 978-1-4503-4424-1/16/08…$15.00
DOI: http://dx.doi.org/10.1145/2940147.2940155

quickly started on demand and incur minimal overhead.

NF and flow management: OpenNetVM's management framework tracks which NFs are running and provides a service class abstraction (e.g., firewall, IDS, etc.) to map to specific instances. A flow table directs packets between NFs in service chains; this can be configured dynamically by an SDN controller or by other NFs. By providing control capabilities within both the manager and NFs, OpenNetVM reduces the communication with the SDN controller, increasing flexibility and efficiency.

Efficient I/O: Using DPDK eliminates kernel overheads, allowing zero-copy access to DMA'd packets from a user space poll-mode driver. We extend this to support zero-copy I/O in service chains of multiple NFs using shared memory accessible to each Docker container within a common security domain.

Scalability and Optimizations: NFs can be easily replicated for scalability, and the NF Manager will automatically load balance packets across replicas to maximize performance. The framework is carefully optimized with techniques such as cached flow table lookups to avoid bottlenecks in the management layer.

While some of these techniques have been explored in the past, our contribution lies in combining them into an efficient and easy to use platform available for the community's use.[1] In this paper we describe the OpenNetVM architecture, discuss the optimizations and efficient data structures needed to guarantee its performance, and evaluate its effectiveness compared to existing approaches.

2. BACKGROUND & RELATED WORK

NFV provides network services through software appliances, rather than hardware devices [4]. The classical approach of running network functions such as routers and firewalls in hardware requires specialized devices for different network functions, and each needs to be individually deployed. NFV enables us to run NFs on commodity off the shelf servers, thus easing deployment of services from different vendors. NFV promises to greatly improve the flexibility with which services can be deployed and modified, while lowering costs.

Several high performance data plane platforms have recently been proposed and developed to support network applications. DPDK [5] bypasses overheads inherent in traditional interrupt based kernel network stack packet processing, and enables applications to directly access data from NICs. PF_Ring [1] achieves wire speed packet capture by utilizing ring buffers which allow both userspace and kernel space access. Netmap [9] preallocates packet buffers, reduces system call time by using large batches, and achieves zero-copy with shared memory buffers between userspace and kernel space. All of these platforms focus on rapidly delivering packets from the NIC to a userspace application.

NFV frameworks can be built on top of these I/O platforms. ClickOS [7] uses netmap [9] and the VALE switch [10] to efficiently move packets between lightweight virtual machines running Click software router elements. They focus on improving network performance of Xen by solving bottlenecks in the hypervisor's network I/O pipeline and using ClickOS, a light weight, fast booting mini-OS customized for network packet processing. This allows a dense and rapid deployment of NFs, which we achieve through our use of containers. ClickOS also supports interrupt-driven NFs that can share CPUs; OpenNetVM dedicates cores to NFs, which can consume more resources but provides higher performance as shown in our evaluation. In addition, ClickOS provides a fairly limited development environment for NFs since they must be designed within the Click framework's specifications and do not run within a standard Linux environment. The framework is also designed for static service chains of functions, and does not offer NFs flexible control over how packets are routed between VMs.

E2 [8], like OpenNetVM, is built using DPDK. E2 is a framework for end to end orchestration of middleboxes, including placement, resource and meta data management, and service chains. However, E2 has fixed service chains—impeding dynamic instantiation of new services, flexible packet routing, and deployment of NFs from competing vendors.

While these projects have illustrated the potential to process packets at line rates on commodity servers, they lack the flexibility and isolation needed in a full NFV platform. OpenNetVM seeks to fill this gap by adding higher level abstractions such as service chains, dynamic NF instantiation, flexible flow control, load balancing, and naming, while still retaining the high efficiency of the underlying DPDK platform.

Container technologies such as Docker and LXC have recently gained popularity as an alternative to server virtualization platforms. Containers use namespace and resource isolation provided by the kernel to encapsulate a process or set of processes within a lightweight capsule. Since containers run as processes within the same host operating system, sharing memory between them (critical for OpenNetVM's zero-copy I/O) is greatly simplified since existing memory mapping mechanisms can be used. Containers also consume fewer resources than virtual machines since they do not include their own operating system, instead relying on the host's kernel. While this prevents containers from running entirely different operating systems (e.g., a Windows container on a Linux host), it is still possible for the container to encapsulate completely different libraries and processes than the host (e.g., a container running the RedHat distribution with a customized version of libc can run on an Ubuntu host).

Unlike virtual machines where a VM is defined by a disk image potentially gigabytes in size, containers are defined by a configuration list of packages and files to be installed within. This greatly simplifies the deployment and sharing of containers since their configuration files can be easily copied, modified, and versioned. OpenNetVM uses Docker containers, meaning that NFs can be trivially shared on the Docker Hub image repository. Using containers to run network functions provides a multitude of benefits: NFs by dif-

[1]Source code and NSF CloudLab images at http://sdnfv.github.io/

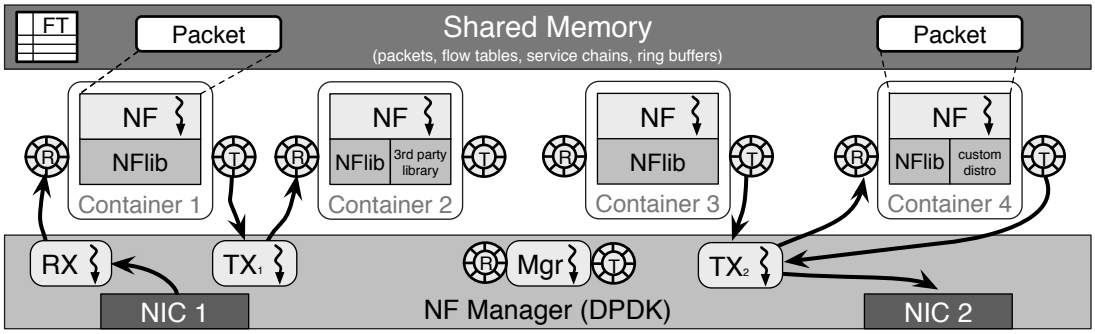

Figure 1: The NF Manager creates a shared memory region to store packets and meta data such as the flow table and service chain lists. Packets are moved between NFs by RX and TX threads that copy packet descriptors into an NF's receive (R) and transmit (T) ring buffers. NFs run in isolated containers that encapsulate all dependencies.

ferent vendors can easily co-exist while retaining isolation, each container can include its own libraries and dependencies, and resources can be precisely allocated.

3. OpenNetVM ARCHITECTURE

As shown in Figure 1, OpenNetVM is split between the NF Manager, which interfaces with the NIC, maintains flow tables, and facilitates communication between NFs; NFLib, which provides the API within an NF to interact with the manager; and user-written NFs that make use of that API. The NF Manager typically runs on the host system, although it can be run inside a container if desired. The NFs, compiled together with NFlib and any desired 3rd party libraries, run as Docker containers.

3.1 NF Manager

The NF Manager maintains network state by managing NFs and routing packets. The manager is composed of three types of components: a Manager thread that configures shared memory and tracks active NFs, RX threads that read packets from the NIC, and TX threads that move packets between different NFs.

Memory Management: When OpenNetVM first starts, the Manager thread uses the DPDK library to allocate memory pools in huge pages to store incoming packets. These pools are divided across all available CPU sockets so that packets will only be processed by cores in the socket local to the memory DIMMs holding them. When NFs start, they find the shared memory address in a configuration file and then map the memory regions using the same base virtual address as the manager. This ensures that a packet descriptor containing a virtual address of where the data is stored can be correctly interpreted by both the manager and the NFs without any translation. This vastly simplifies memory sharing compared to our prior approach in NetVM [3], where addresses needed to be carefully translated to allow access by KVM-based NFs over an emulated PCI device.

NF Management: After performing its initialization routines, the Manager thread begins checking for new NFs by polling a message queue in the shared memory region. New NFs use this message channel to notify the manager so they can be added to a registry of active NFs; similarly, when NFs shutdown they inform the Manager so it can clean up any relevant data structures. Message queues (implemented as ring buffers) are used throughout the OpenNetVM architecture because they allow efficient, asynchronous communication. In addition to the Manager's message ring, there are also two descriptor ring buffers (Receive and Transmit) created in shared memory for each NF.

Packet Flow: The manager's RX and TX threads handle the routing of packets between NFs and the NIC ports. Packets initially arrive in a queue on a NIC port based on the Receive Side Scaling (RSS) hash of the packet. The DPDK Poll Mode Driver ensures that packet data is DMA'd directly into the shared memory pool and that a packet descriptor (i.e., address and meta data) is copied into a queue associated with an RX thread pinned to a core on the same socket as the memory pool. The RX thread will remove a batch of packets from its queue and then examine them individually to decide how they should be routed to the Receive ring of the destination NF. TX threads perform a similar function, but instead of reading from a NIC queue they read from the Transmit ring buffers of NFs and either send the packets out a NIC port or to a different NF Receive ring. Packets may be directed using a flow table or a default service chain, as described in subsequent sections.

OpenNetVM uses TX threads to provide an abstraction layer between NFs. In other systems such as Click [6] and E2 [8], it is possible to compile a service chain of NFs together, procedures, into a single process; moving between the service chain in their case is thus simply a new procedure call. While that approach can be more efficient, it limits flexibility since the service chain order is typically hard coded and cannot be adjusted dynamically by a management entity, nor can an NF be dynamically moved because of resource availability constraints or workload demands. In contrast, OpenNetVM's TX threads can determine how to route a packet based on an action requested by the last NF or by a flow table lookup. The TX thread also provides load balancing functionality across NF replicas, without requiring NFs to be aware of what other containers are currently running.

3.2 Network Functions

OpenNetVM Network Functions interact with the manager using our NFLib API. When the NF process starts (either natively or in a container), it calls an initialization function that registers its service type with the manager. The NFlib run function then polls its Receive ring for packets arriving from the NIC or other NFs. For each packet received, NFlib will execute a callback function provided by the NF developer to process the packet. NFlib handles all interactions with the shared memory rings, as well as batching packets for transfer. Our current release only supports "polling" NFs that have a dedicated CPU core, although we are investigating support for interrupt driven NFs that can share a core.

When an NF runs its callback to handle an incoming packet it is given a packet descriptor that contains the address of the packet body and some meta data. The NF will perform the desired functionality, such as routing, Deep Packet Inspection (DPI), intrusion prevention, etc. Then the NF can indicate what action should be performed on the packet subsequently: Lookup in Flow Table, Send to another NF, Send to NIC, or Drop. By default, the NF will issue the Lookup Flow Table action, causing the next step to be determined based on flow table rules as described in Section 4. On the other hand, if a flow table is not being used, or some exception has occurred, the NF can specify an NF service type to send to, a NIC port to send out of, or can simply request that the packet be dropped. In all cases, the packet descriptor (containing the requested action) will be placed in the NF's Transmit queue, to be handled by one of the manager's TX threads. The TX thread then has the ability to perform or overrule the action.

4. FLOW MANAGEMENT

OpenNetVM facilitates movement of packets through service chains of NFs in several ways. First, it provides a service type abstraction so that new NFs announce what type of function they provide, allowing the manager to direct packets to the desired function type and load balance across replicas of a type. OpenNetVM defines a service chain as a list of actions, typically a series of send actions to different service types. Finally, the Flow Director provides a convenient and flexible way to match individual packet flows to the service chains responsible for them. These rules can either be provided by NFs, or via an SDN controller.

4.1 Service Types

When an NF starts, it declares its "Service ID" to the manager. A Service ID is a numeric identifier that represents one NF type. There can be many NFs with the same Service ID running at once, but each will be assigned a unique instance ID. NFs are always addressed by their Service ID. Service-based addressing means that one NF doesn't require knowledge of what other instance IDs are in use (as this may change over time). Instead, an NF simply sends its packets to the desired Service, and the Manager determines which instance to use. Also, if an NF exits unexpectedly, then the packet flow will continue unbroken, since the manager routes packets to other NFs of the same type and Service ID.

Addressing packets to NFs based on the Service ID instead of instances also allows the manager to easily load-balance packets. When the manager performs the Service ID lookup, it determines a list of the relevant instance IDs. The Manager then takes the packet's symmetric RSS hash (derived from a tuple consisting of the packet's IP protocol and source/destination address and port) modulo the number of available NFs of the desired type to select which instance to route the packet to. This approach ensures that, as long as NFs aren't starting and stopping, packets in the same (bidirectional) flow always get routed to the same instances of each service type they pass through, since the RSS hash does not change.

4.2 Service Chains

While OpenNetVM seeks to give significant control to NFs about how packets flow through the data plane, network administrators will typically provide a set of default rules defined for each service chain. A Service Chain is a list of actions along a path a flow has to follow. For example, a system may want all packets to travel to Service ID 3, then 5, and then be sent out port 3 on the NIC. An NF can set up this default service chain, and then NFs in Service 3 and 5 simply have to use the Flow Table action for the packet to continue through the service chain. If desired (and permitted by the manager's configuration), NFs can still send packets to specific alternate services if they prefer that packets not follow the default route.

4.3 Flow Director and Flow Tables

OpenNetVM's Flow Director component in the NF Manager contains a flow table that maps packet flows to service chains. Doing this in the management layer gives greater flexibility in how packets are steered compared to hardware techniques like Intel's Flow Director which does not support service chains. When a TX thread must move a packet through a service chain, it first needs to look up which chain to use (or the default Service Chain, if none exist for the given flow). The Flow Director also exposes an API to NFs so that they can dynamically assign what service chains are mapped to each flow.

OpenNetVM includes an SDN NF which will contact an SDN controller via the OpenFlow protocol to determine the desired service chain for each flow. The SDN NF then uses the Flow Director API to configure the manager's flow table. This approach keeps the manager as simple as possible; it merely enforces the flow table rules, while allowing other applications to provide the policy to set them.

Flow table lookups can be expensive when performed in the critical path of the data plane–our results in Section 5.2 show that a naïve implementation that hashes the packet header on each lookup lowers throughput by over 50%. Since flow table-like data structures—i.e., hash tables that map from a flow to data about that flow—are common to many NFs, OpenNetVM provides a flow table library optimized for packet lookups based on DPDK's Cuckoo Hash implementation.

The NF Manager's Flow Director uses the library to create a table mapping flows to service chains, but other NFs such as an IDS might use a table to store state about each flow. Rather than recalculate the packet hash for each lookup, OpenNetVM repurposes the RSS hash calculated in hardware by the NIC to simplify packet lookups for flow tables both within the NF Manager and within NFs. This optimization can provide a significant performance improvement, especially for service chains that contain multiple NFs that use hash tables to store state about flows.

5. EVALUATION

In this section, we evaluate OpenNetVM in terms of service chain performance, flow table overhead, and multi-port scalability.

Our experiment setup is comprised of two classes of servers. For traffic generation (via Pktgen-DPDK) and experiments requiring a single NIC port, we use HP servers with an Intel Xeon CPU X5650 @ 2.67GHz (6 cores), 64GB memory, and an Intel 82599ES 10G NIC. We also use a Dell machine with an Intel Xeon CPU E5-2697 v3 @ 2.60GHz (14 cores), 164GB memory, and four Dual Port NICs[2]. HP servers run Ubuntu 14.04.3 (kernel 3.19.0) and the Dell runs Ubuntu 12.04.5 (kernel 3.2.0) with Docker v1.9.1.

5.1 Service Chain Performance

We first evaluate the service chain performance of OpenNetVM running as processes and Docker containers, compared to ClickOS which uses Xen virtual machines. To avoid overhead differences caused by NIC processing, we have the first NF in the chain create a set of packets which are then repeatedly sent through the chain on a single host, each NF simply forwards the packet to the next. Figure 2 shows how throughput for 64 byte packets changes as we adjust the chain length. With ClickOS, we were unable to start more than three NFs, but see a trend comparable to the ClickOS paper. OpenNetVM's manager runs one TX thread per NF, and each is dedicated a core; we allocate the same number of cores when using ClickOS.

The performance difference for OpenNetVM when using processes or containers is negligible; for simplicity our remaining experiments use processes instead of containers. OpenNetVM achieves higher performance than ClickOS since it is based on DPDK's poll-mode driver (as opposed to interrupts) and has cheaper packet transfer costs between NFs since it avoids kernel overheads. Even with a six NF chain, OpenNetVM sees only a 4% drop in throughput, while ClickOS falls by 39% with a chain of three NFs.

5.2 Flow Director Overheads

We next consider a more flexible case where real packets are routed via the flow table instead of hard coded rules within the NF. Figure 3 measures the throughput of OpenNetVM when traffic arrives from a packet generator over a 10Gbps link and is directed to a single NF; the manager uses 1 RX thread and 1 TX thread. In the Default Chain case, the RX thread receives the packets and sends them to the NF using a globally defined default chain without any flow table lookups; this achieves close to the 10Gbps rate even when sending 64 byte packets. If the RX thread must perform a flow table lookup using our optimized Flow Director, the throughput drops since the RX thread becomes a bottleneck; this can be mitigated by using multiple RX threads, and the 10Gbps rate can easily be met for reasonably sized packets. However, if a naïve flow table implementation is used that must hash the header of each packet, the throughput drops even further, and is unable to meet the line rate even for 256 byte packets.

To precisely measure the latency and avoid other component interference (e.g., read / write to NIC, networking congestion), we test the latency of each Flow Director method using locally generated packets. We have an NF create a set of packets and send them to the TX thread, which sends the packets back to the NF either using a default service chain or a flow table lookup. The latency with the default service chain (4.4 us) and our Optimized Flow Table lookups (6.9 us) are similar. However, Naïve Flow Table lookups take 24.9 us, increasing latency 5.68 times compared to the default service chain.

5.3 Multi-port Scalability

To evaluate OpenNetVM's scale in a more realistic environment, we use eight ports on our Dell server and send packets to it from directly connected HP servers. The traffic generators replay a PCAP trace of HTTP packets: 44.3%

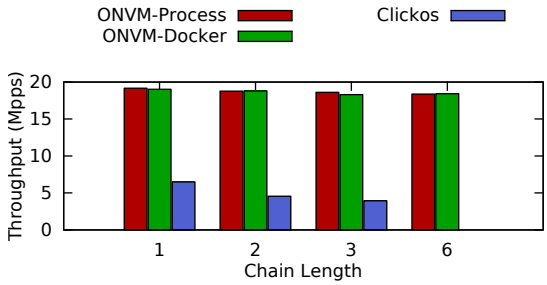

Figure 2: OpenNetVM achieves high throughput, even when running through long service chains, by avoiding expensive packet copies and system calls.

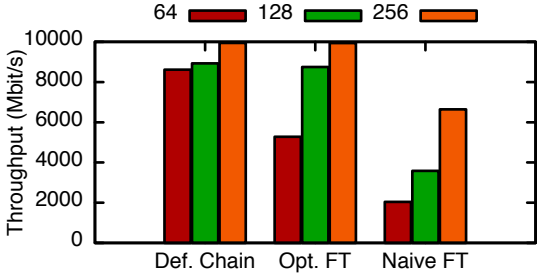

Figure 3: Flow table lookups add overhead to RX and TX threads, but our optimizations eliminate this effect except for 64 byte packets.

[2]The Dell machine has three Intel 10G Dual Port X520 NICs and the fourth is an Intel 82599EB 10G Dual Port NIC. None of our cards can meet a full 20 Gbps for 64 byte packets, which we believe to be a hardware limitation.

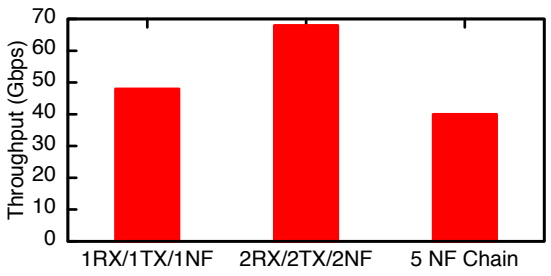

Figure 4: OpenNetVM achieves nearly 70 Gbps on real traffic using only six cores.

are 64 bytes, 18.5% are 65-1023 bytes, and 37.2% are 1024-1518 bytes. The packets are sent to a default NF, which immediately forwards the packets back out the NIC port.

When configured with one RX thread, one TX thread, and one NF, OpenNetVM achieves 48 Gbps. If we start a second NF and add another RX and TX thread to the manager, OpenNetVM automatically load balances flows across the NFs to achieve 68 Gbps. This illustrates the potential of using OpenNetVM as a platform for software routers even deep in a data center network.

Finally, we reconfigure the system to create a linear, default service chain of five NFs, with 2 RX and 5 TX threads. This setup gets a throughput of 40 Gbps. Note that the chain of five NFs increases the load on the manager by 5X—the TX threads are steering packets at a total rate of 34 million packets per second. This configuration consumes all of the cores on our server, but we expect that with additional cores the entire service chain could be replicated to further increase the throughput.

5.4 Flexible Packet Steering

Finally, we demonstrate the different ways to steer packets with OpenNetVM: an SDN controller can provide flow rules, the manager can redirect packets to load balance, and NFs can divert flows based on packet data. The system begins with a default service chain that sends all new packets to the SDN NF if there is a Flow Director miss. The SDN NF contacts our SDN controller for each new flow that arrives. For this experiment, the controller returns *(send to IDS, send out port 1)* for all flows, but this could be adjusted dynamically. Once the rule has been installed into the Flow Director, subsequent packets in that flow are sent directly to a fake IDS NF instead of to the SDN NF.

When the workload on the IDS NF rises, we start an additional replica with the same service ID. Booting up this NF in a Docker container takes on average 0.526 seconds from when the container is started to when the NF receives its first packet (an insignificant difference from 0.524 sec with native processes). This is substantially faster than KVM VMs, which took at least 12 seconds to initialize in our NetVM platform. Once the NF is initialized, the manager will automatically load balance packets across the two replicas.

Later, we mimic a case where the IDS NF detects suspicious traffic. When this happens, the IDS diverts some packets from the original service chain based on their content. The IDS NF is able to directly steer packets (subject to any constraints imposed by the manager) without interacting with the SDN controller. OpenNetVM supports this type of smart data plane behavior in order to reduce the load on SDN controllers and increase the agility with which packets can be redirected.

6. CONCLUSION

OpenNetVM is a scalable and efficient packet processing framework that supports dynamic steering of packets through service chains. Unlike prior approaches with limited programming paradigms or restrictive runtime environments, OpenNetVM deploys network functions in Docker Containers, facilitating development of NFs by diverse service providers, while minimizing memory consumption and startup time. OpenNetVM leverages DPDK for high performance I/O, and achieves a throughput of 68 Gbps using only six CPU cores. This opens up the possibility for complex software based services to run deep within the network and data centers. We have released OpenNetVM, for the community's use, providing both source code and experiment templates on the NSF CloudLab platform. We believe OpenNetVM will provide an ideal basis for NFV/SDN experimentation and commercial prototype development.

Acknowledgements: This work was supported in part by NSF grants CNS-1422362 and CNS-1522546.

References

[1] L. Deri and others. Improving passive packet capture: Beyond device polling. In *Proceedings of SANE*, volume 2004, pages 85–93. Amsterdam, Netherlands, 2004.

[2] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy. Softnic: A software nic to augment hardware. Technical report, Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, 2015.

[3] J. Hwang, K. Ramakrishnan, and T. Wood. NetVM: High Performance and Flexible Networking using Virtualization on Commodity Platforms. In *Symposium on Networked System Design and Implementation*, NSDI 14, Apr. 2014.

[4] E. T. S. Institute. Network Functions Virtualisation. In *SDN and OpenFlow World Congress*, 2012.

[5] Intel. Data plane development kit.

[6] E. Kohler. The Click Modular Router. *PhD Thesis*, 2000.

[7] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. ClickOS and the Art of Network Function Virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 459–473, Seattle, WA, Apr. 2014. USENIX Association.

[8] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: A Framework for NFV Applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 121–136, New York, NY, USA, 2015. ACM.

[9] L. Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *USENIX Annual Technical Conference*, pages 101–112, Berkeley, CA, 2012. USENIX.

[10] L. Rizzo and G. Lettieri. VALE, a Switched Ethernet for Virtual Machines. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 61–72, New York, NY, USA, 2012. ACM.