High Speed User-Space Network I/O Solutions for Network Function Virtualization

A first look at how network function virtualization can achieve high speed north-south communication

Huimin She, Software Developer & Johan Sandström, RTOS Chief Designer

Virtualization of network appliances, using software implementations running in virtual machines, allows the usage of cost-efficient data center solutions as possible replacements for proprietary hardware. In order to maintain fulfilment of requirements on high speed network communication, efficient virtualization is required, affecting the virtual machine host as well as its guests. To speed up communication, the use of user space access to network I/O in host and guest is becoming more and more popular. This document presents performance benchmark results for a selection of recent solutions for high speed network I/O in user space host, and guest applications in a virtual machine. The paper compares the host and guest throughput performance, using state-of-the-art benchmarks and measurement methods.

1. Introduction

The operational and capital costs for the proprietary hardware appliances are substantial and it creates a complex network environment for managing network resources and for developing new services in the network. The network functions virtualization (NFV) [1] initiative from the ETSI industry specification group for NFV aims to virtualize the network functions that run on proprietary hardware appliances. The ETSI group defines requirements, use case examples, and an architecture for network function virtualization. The goal is to promote and evolve standard IT virtualization technology to meet the requirements of the telecommunication operators, and to allow network functions previously provided by proprietary hardware appliances to be implemented as virtualized network functions, running in a virtual machine (VM). The virtualized network functions are interconnected to implement an end-to-end service.

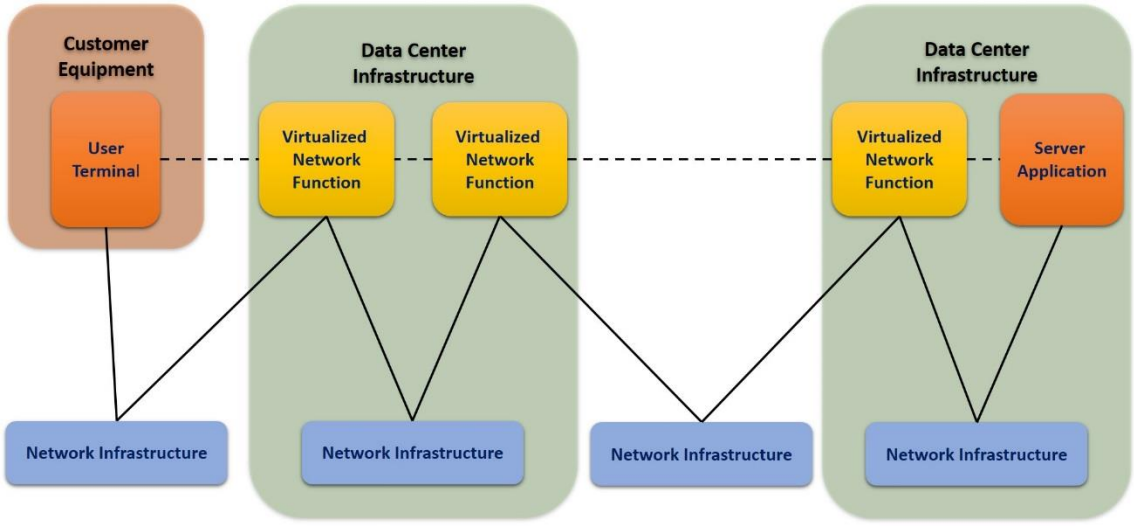

Figure 1: Example of end-to-end service implemented with virtualized network functions

Moreover, as the virtualized network functions are implemented in software, new services can be implemented and deployed quickly. The virtualized network functions are interconnected in a network function forwarding graph, to implement an end-to-end service for a user. The virtualized network functions require fast network I/O communication to provide fast service setup, high throughput, and low latency data communication.

The need for improved networking performance in virtual machines is partly met by providing fast user space I/O to network applications in the VM. The data center requirements for fast network I/O in virtual machines have driven the network I/O support, from fully virtualized network devices to PCI pass-through solutions of physical and virtual PCI devices (Single Root I/O Virtualization, SR-IOV [2]).

Direct network I/O access from user space applications has been used on the host, as it provides a solution for developing device drivers that do not need to be maintained with the kernel. It also makes it possible to short-cut kernel system calls when sending and receiving data. In the past, a big interest for direct user space access to fast network I/O has resulted in different solutions: first for packet capture, and later for implementation of network functions in user space. The solutions obtained can be divided into user space data path (i.e. kernel-bypass of data transmitting and receiving), and user space device driver.

The PCI pass-through for physical and virtual functions promises near line-speed and the user space network I/O solution in VM should provide an increased throughput when combined. The user space network I/O also provides the possibility to run a minimal protocol stack for the specific purpose of the user application, which can be especially interesting for virtualized network functions with requirements on limited function scope and on high throughput and low latency communication.

The remainder of this paper is structured as follows. In section 2, we describe different technologies we have tested on a host and in a VM. Section 3 contains the test setup, i.e. device under test, test instrument, chosen test methodology and software used (operating system and layer-2 forwarding applications). Section 4 provides the test results from layer-2 (L2) forwarding on host using a single core. The section continues to benchmark L2 forwarding in a Kernel-based Virtual Machine (KVM) [3] guest and an analysis of the results and profiling results of where the CPU time is spent on the tested core with the different solutions. Section 5 summarizes this paper.

2. Description of Tested Solutions

To test different solutions for user space applications to access network I/O, we use an Intel Xeon HW platform with Intel 82599 Ethernet Network Interface Cards (NIC). We use Linux, in the form of Enea Linux version 3.1. The HW platform is a commonly used server HW platform, and furthermore the open source community provides out of the box support for modified ixgbe drivers for PF_RING, netmap and Intel DPDK. Other suppliers provide NICs with similar function support, e.g. Broadcom or QLogic. For details regarding the HW platform, see section "3. Test Setup and Methodology". To get some comparisons with standard Linux solutions for user space network packet receive and transmit, AF_PACKET was also chosen to get a view of the kernel penalty. Network functions are preferably handled in the Linux kernel and as an in-kernel solution for network functions. All together the following five solutions (AF_PACKET, Open vSwitch, PF_RING, Netmap, DPDK) were chosen for host tests. Open vSwitch and DPDK were chosen for VM tests.

Linux kernel AF_PACKET

AF_PACKET [4] is a socket family implemented in the Linux kernel since version 2.2. It provides a packet interface for user applications to receive and transmit packets through NIC drivers. It also allows users to implement protocol modules in user-space on top of the physical layer.

AF_PACKET supports two socket types: SOCK_RAW for raw packets including the link level header, and SOCK_DGRAM for cooked packets with the link level header removed. SOCK_RAW packets are passed to and from the device driver without any changes in the packet data. For SOCK_DGRAM packets, the physical header is removed/added for received/transmitted packets. When the protocol is set, all incoming packets of that protocol type will be passed to the packet socket before they are passed to the kernel protocols.

The AF_PACKET solution can also use memory mapped buffers (shared memory) with mmap() to minimize the copy of frames between the application and kernel and also allow transmission of multiple frames at the same time. The memory mapped solution provides transmit (tx) [5] and receive (rx) [6] rings in shared memory. The solution minimizes the number of system calls and memory copies between the application and the kernel.

Open vSwitch

Open vSwitch (OVS) [7] is a multi-layer software switch under the open source Apache 2 license. It is designed to meet the need for automated and dynamic network control in large-scale multi-server virtualization environments, where the traditional hypervisor networking stacks (such as Linux kernel bridge) are not well suited. OVS can run on any Linux-based virtualization platforms including KVM, VirtualBox, Xen, XenServer, and Xen Cloud platform. OVS supports standard management interfaces (such as sFlow, NetFlow, IPFIX, RSPAN, and CLI), and is open to programmatic extension and control using OpenFlow and OVSDB management protocol. OVS can also operate entirely in user-space without assistance from a kernel module.

PF_RING

PF_RING [8] is a framework for high speed packet I/O that uses a kernel-bypass solution for data packets, while the control of the device is still handled by the Linux kernel. It can be used to build extremely fast traffic generators and monitors. The main components of PF_RING are: the PF_RING kernel module, user-space SDK that provides PF_RING support to user space applications, and several types of network device drivers. The high speed packet capturing performance of PF_RING mainly comes from these techniques: pre-allocating memory ring at creation time; avoiding copying by bypassing the Linux kernel and even the PF_RING kernel module.

PF_RING implements a new socket type (named PF_RING) on which user space applications can interact with the PF_RING kernel module. PF_RING can work both on top of standard NIC drivers and specialized drivers. There are two types of specialized drivers:

- PF_RING-aware driver: it can improve packet capture performance by copying packets from the network interface to the PF_RING module without going through the Linux kernel.
- PF_RING DNA (Direct NIC Access) driver: it allows packets to be read directly from the network interface by bypassing both the Linux kernel and the PF_RING kernel module.

In order to adapt to virtual environments, Virtual PF_RING [9] (vPF_RING) extends the kernel-bypass approach to hypervisor-bypass approach. By creating a mapping between the host kernel space and the guest user space, vPF_RING allows user applications running on VMs to read packets directly from the NIC.

Netmap

Netmap [10] is a framework for enabling user space applications to fast access to network packets. It is independent of the network interface hardware, and not designed for any specific network device or hardware features. Netmap is released under the BSD license.

One key feature of netmap is that it separates the application from the network device, and the application does not directly access the NIC registers. Since these operations are redirected though the kernel, memory protection between application and the network device will be enforced. Netmap improves throughput by

- Avoiding per-packet dynamic memory allocations (usually used in the kernel for sk_buf) by using pre-allocated memory.
- Avoiding system call overheads by batching frame processing heavily.
- Avoiding memory copies by sharing buffers and metadata between kernel and user-space.

The solution can be categorized as a kernel-bypass solution. For packet buffers, it uses a poll of memory mapped fixed size packets. The NIC is responsible for moving data between the network and the packet buffers in memory. The kernel/driver passes these buffers by reference (zero copy) to a software ring buffer from where the application can read the data. Netmap uses the standard poll/select system calls for synchronization with the application. Separate software RX/TX rings can be allocated for different applications, and these can be mapped to hardware queues if supported by the hardware. Flow classification capabilities of the NIC can be used to redirect packets to a specific application on a specific core.

When using netmap, traffic to and from the NIC will be disconnected from the host network stack. However, standard applications such as ifconfig and ethtool can be used to configure the NIC. To receive and send traffic using the host stack, a bridge configuration is needed. Netmap can also be used as a gateway between applications and the host network stack.

DPDK

The Intel Data Plane Development Kit (DPDK) [6] is a Linux framework created to make development of high speed network applications development easier. It consists of optimized network interface drivers and libraries, which provide a hardware abstraction layer, and specific libraries for hardware features. User applications are linked against the EAL (Environment Abstraction Layer) libraries. DPDK is intended for network applications designed for a run-to-completion model. DPDK is originally developed by Intel for their x86 platforms, however DPDK is now released as an open source project under the BSD license.

DPDK consists of several parts, making it faster than the general Linux networking:

- *Polled mode drivers* - By using optimized polled mode drivers instead of the default interrupt driven Ethernet drivers, one can avoid the overhead of dealing with interrupts, context switching and system call invocations. This helps to save CPU resources and reduce latency.

- *Hugepages for frame buffers, ring and other buffers* - Using hugepages is fast since it will reduce time spent in page lookups and TLB (translation lookaside buffer) misses. The DPDK memory is also allocated optimally over different memory channels to allow parallel access to memory.

- *Zero-copy* – DPDK receives packets in a pre-allocated memory, located in a user space buffer. It is never processed through the kernel, avoiding overhead of Linux sk_buf and system calls.

- *Run to completion* - DPDK supports the run-to-completion model, where the application is allowed to process the packet from start to finish without being interrupted.

- *Lockless implementations* - Queues and other features of the API libraries provided by DPDK are implemented in a lockless fashion. Data structures are also optimized to be cache-line aligned.

- *Environment Abstraction Layer (EAL)* - The EAL provides a standard programming interface to libraries that have been optimized for available hardware accelerators and other hardware and OS elements. It also provides additional services like boot support, time references, PCIe bus access, and debug functions.

3. Test Setup and Methodology

To create a test for a simple network function that could be virtualized, the host user-space network I/O performance on the host and in the virtual machine have been tested using L2 forwarding applications. The L2 forwarding (FWD) applications for the tested solutions differ in the network I/O parts but the actual L2 FWD is kept as simple as possible, i.e. receive Ethernet frame on one 10 Gigabit Ethernet (GE) interface and send out on another 10 GE interface. To be able to compare the results, the applications are pinned to a single physical CPU or logical CPU in the VM. The CPU is isolated to avoid OS noise that would interrupt the core. The test is chosen to emulate a user-space NFV network function (NF) that could execute on host and in a virtual machine guest, and that would only use the CPU resources required by the application. The setup might not give the most optimized results for all solutions, but the main purpose is to limit the CPU resources so that we can find the bottleneck, and at the same time trying to limit the effect of the multi-core environment on the application. At some level, resources (such as caches, memory and network devices) are shared, but the tests try to subdivide the network function application as much as possible, using Linux kernel configuration and Linux affinity support.

The applications we tested are:

- AF_PACKET – user space L2 forwarding application traffic through the kernel
- Open vSwitch – as an in-kernel L2 forwarding reference
- PF_RING – pfdnabounce provided within PF_RING
- Netmap – bridge example netmap API client
- DPDK – l2fwd provided within DPDK

The device under test (DUT) is an Emerson ATCA 7370 board with dual 1800 MHz Intel Xeon E5-2648 processors with hyperthreading disabled and with a RTM-ATCA-736X-10G NIC (Intel 82599 Ethernet controllers) running an Enea Linux with kernel version 3.10 on host and as guest. The Linux kernel is configured to isolate (isolcpus, rcu_nocbs, cpuset, nohz_full) the CPUs used for L2 FWD applications and also configured to minimize timer ticks. In the standard OVS setup and AF_PACKET tests, any interrupt from the NIC is bound to an isolated CPU.

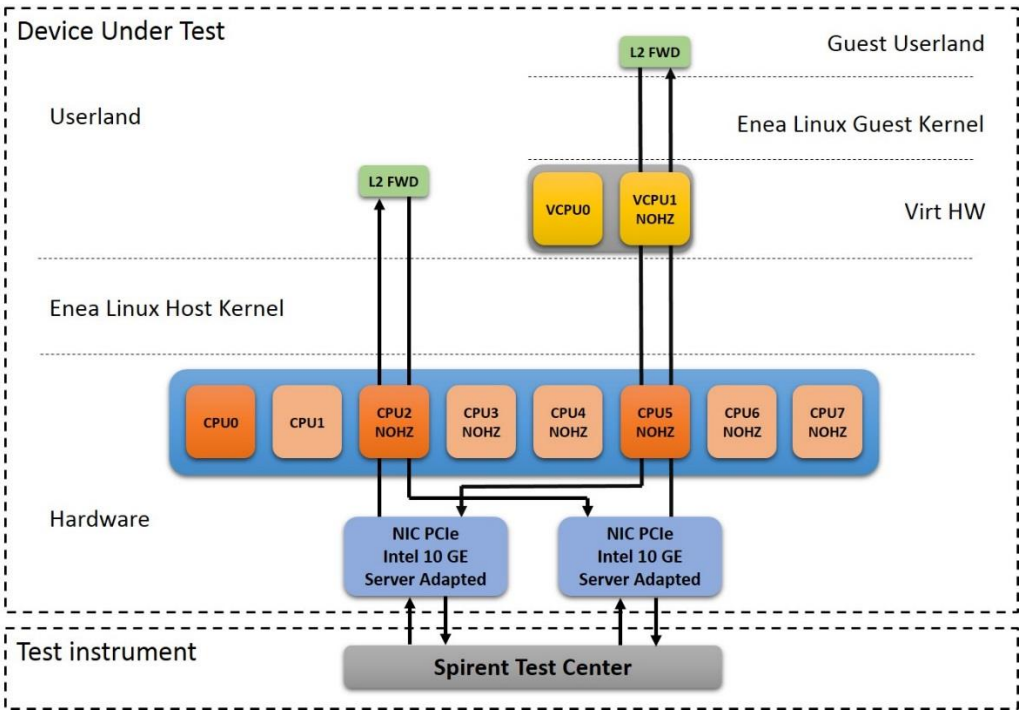

Figure 2: Test Setup

For traffic generation and measurement of throughput, we use a Spirent Test Center (STC)[1] to get good measurement results. The DUT and the STC are connected back to back, i.e. no Ethernet switch between DUT and STC. The STC is setup to generate and receive a single unidirectional traffic flow with Ethernet/IP/UDP frames.

The implementation of the L2 FWD applications for the five tested solutions differ slightly, as the network I/O programming interface differs, but the applications performs the simplest kind of forwarding, i.e. traffic on interface 1 is sent out on interface 2 (and vice versa) and the Ethernet frame is unmodified. The generated traffic from STC is received on one 10 GE NIC port in the DUT, forwarded in user space or in kernel, and sent out on another 10 GE NIC port in the DUT back to the STC. The STC RFC 2544 test suite is reused to get benchmark numbers according to RFC measurement.

4. Benchmark Results and Analysis

Physical interfaces and test application on the host

Figure 3 illustrates the forwarding rate of different L2 forwarding applications running on a single core. The AF_PACKET forwarding solution requires a copy of the received network frames (from rx ring buffer to tx ring buffer) before sending the frames out, and this decreases the maximum frames per seconds when the network frames size and the maximum bit rate levels out. In order to find out how time is allocated between the kernel and application activities, we use perf as a profiling tool. From the profiling of the AF_PACKET L2 forwarding and offloading rx and tx interrupts from the application CPU, we can see that a lot of time (more than 90%) is spent in the kernel for sending traffic on the AF_PACKET socket and a very small amount of time (about 7%) is spent on actually forwarding traffic.

[1] STC SPT-9000 version 4.33.0086, Spirent Test Center application version 4.33.0086, and the 10G test module MX-10G-S2 HYPERMETRICS MX 10G SFP+ with firmware version 4.33.0086.

A comparison of AF_PACKET forwarding with the standard OVS L2 forwarding shows a substantial improvement when doing the L2 forwarding in kernel space with OVS. Both AF_PACKET and OVS forwarding process receive and transmit path on the same core. The performance improvement for OVS is mainly due to the zero copy in the kernel and the fact that the L2 forwarding in kernel space does not use rings/queues in the AF_PACKET socket layer and do not need to traverse through the kernel calls from user space. The profiling results also show that the AF_PACKET cost for passing the packets through the kernel decreases the maximum communication capacity.

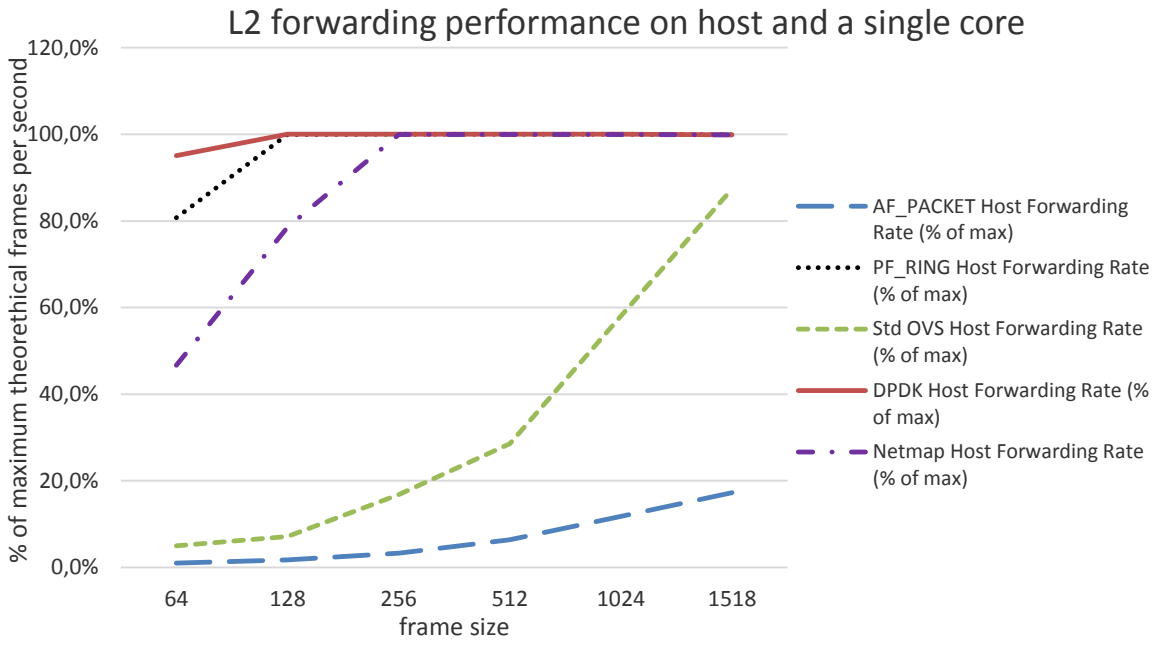

Figure 3: Single core L2 forwarding performance on host

Netmap shows a substantial performance improvement compared with AF_PACKET and OVS, as the solution does not involve the kernel code and only synchronize memory mapped rx and tx rings with poll/select. The memory mapped rx/tx rings and buffers are also allocated in a large consecutive memory area and the rx/tx ring buffers do not need to be allocated and freed per sk_buf. The netmap solution shows a lower performance gain than the DPDK solution, and the profiling shows that the netmap forwarding spends about 90% of the time executing in kernel space, mainly in the netmap ixgbe driver (ixgbe.ko module) and netmap kernel module (netmap_lin.ko). The kernel modules handle the network interrupts and move the data pointers from the hardware tx and rx rings to the netmap software tx and rx rings. The netmap_lin.ko kernel module is also responsible for waking up applications waiting (e.g. using "poll()") on the netmap rx notification file descriptors.

The PF_RING solution shows very good forwarding performance (80% of line rate for 64byte frame). Our results were obtained with out-of-the-box settings without tweaking. That is why the forwarding rate is slightly lower than what has been tested by NTOP [18] [19]. The profiling results show that only a very small amount of CPU is spent by the kernel, and about 96% of the CPU time is consumed by the user space forwarding application (pfdnabounce) that polls the PF_RING memory mapped tx and rx ring.

The DPDK user-space network I/O solution shows the best performance compared with other forwarding solutions. The bypass of the kernel for receiving and sending Ethernet frames and avoiding the overhead of software interrupts, context switching and system call invocations provide the substantial improvement. The support for zero copy with user-space buffers in the DPDK user network I/O solution together with the use of hugepages, that reduce time spent in page lookups and TLB cache misses, further increase performance. The profiling results show that 99% of the time is spent by the DPDK forwarding application and there is very little kernel activity involved.

Virtual interface in VMs (north south VM traffic) with SR-IOV VF (Virtual Function) devices

When network traffic arrives on an interrupt driven virtual or physical IO device, the VM and virtual machine manager (VMM) interacts heavily. The general interaction steps between the host and VM are as follows:

- VM exits and host enters, as the VMM must handle the interrupt
- VMM handles the external interrupt and prepares the virtual interrupt request for the guest
- VM enters and host exits, and returns execution to the guest for handling the virtual interrupt request
- VM handles the virtual interrupt request and executes the ISR and performs end-of-interrupt (EOI) that causes an additional VM exit
- VM exits and host enters, VMM executes host part of EOI
- VM finally enters and host exits and the guest can process the received traffic

This is costly for heavy network traffic as it causes many VM exit and VM enter. For the same reason, a minimal number of timer interrupts are desired and core isolation together with the use of Linux NOHZ is required to optimize performance. The guests benefit from limiting the number of interrupts from the virtual or physical device. More advanced drivers and Ethernet controllers allow the user to limit the number of interrupts or optimize when interrupts should be triggered. This allows performance tuning to optimize VM network traffic processing. The configuration of the interrupt coalescing can be used to control the number of interrupts generated in the system and some Ethernet controllers also allow configuration of the maximum interrupt rate, e.g. the Intel ixgbe and ixgbevf drivers [13] allow configuration of the maximum interrupt rate (InterruptThrottleRate) for the physical or virtual device. All configurations that cause less interrupts can also cause increased latency. A pure polled driver in the guest with PCI pass-through solutions of physical and virtual PCI device (SR-IOV) can avoid the VM exit and VM enter penalty entirely.

The netmap solution was left out from the VM tests as it, at the time of the tests, required an external netmap bridge solution. The AF_PACKET solution was skipped from the VM tests due to its poor performance compared with other solutions. The PF_RING solution was left out from the VM test as the virtual PF_RING (vPF_RING) is defined as end of life (EOL) [9]. Therefore, we choose OVS and DPDK for the VM tests.

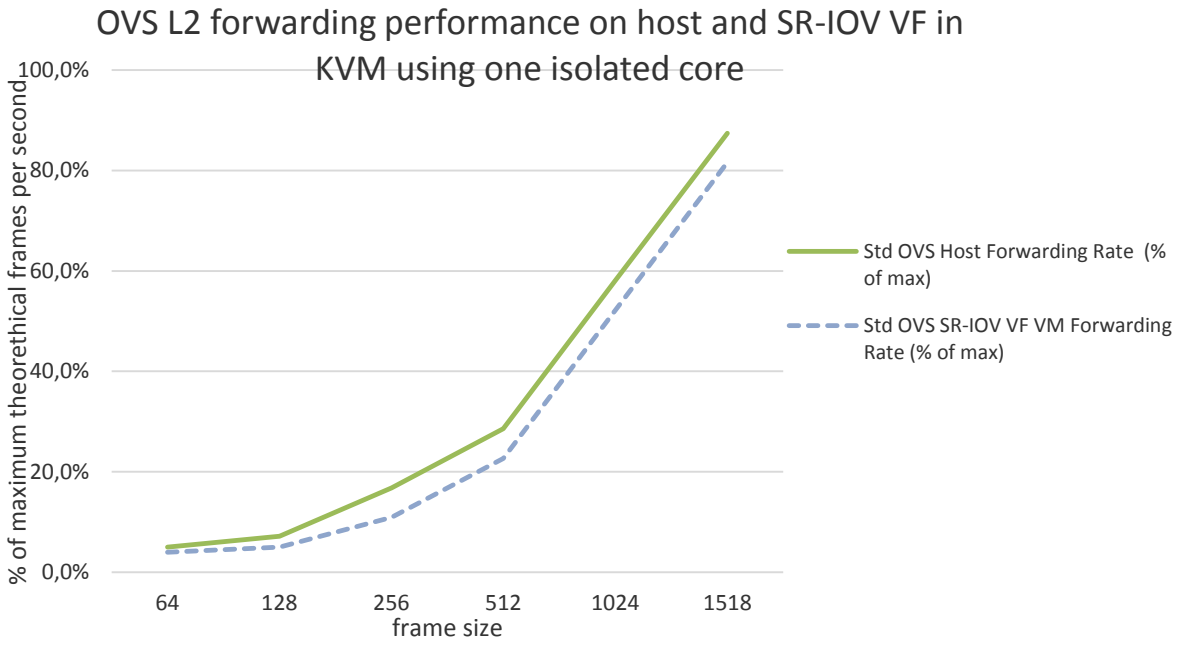

Figure 4: Single core OVS L2 forwarding performance on host and in VM using VF

Figure 4 shows that the forwarding rate of OVS in VM is only slightly lower than that on host. For 64 byte frames, the forwarding rate in VM and on host are 4% and 7% of the maximum theoretical rate, respectively. For 1518 byte frames, the numbers are 82% and 87%, respectively. The decreased performance is expected, as the additional cost for VM exit and VM enter for the interrupts adds to the penalty. Furthermore, the using of VF may also slightly bring down the forwarding performance in the VM. On the other hand, the performance decrease in VM is not significant thanks to the PCI pass-through technique.

Figure 5 illustrates the comparison of forwarding rate of DPDK L2 forwarding solution on host and in VM. The DPDK user space network I/O solution on the host shows a forwarding performance almost at line rate, 95% for 64 byte frames. The forwarding rate in the VM is 70% of line rate for 64 byte frames. There are several possible reasons for the performance decreasing in the VM. The statistics of the memory controller traffic counter show that there is much higher memory controller traffic load in the VM than that on the host. In addition, the system and device configurations in the VM might not be optimal. It is worthwhile to perform deeper dive into system and device tuning. On the other hand, compared with interrupt driven driver, the polled driver without interrupts shows a substantial improvement when receiving and sending Ethernet frames. Moreover, the polled driver avoids the VM exit and VM enter penalty. In summary, for a single threaded run-to-completion packet processing on one virtual CPU, the performance hit due to virtualization is limited.

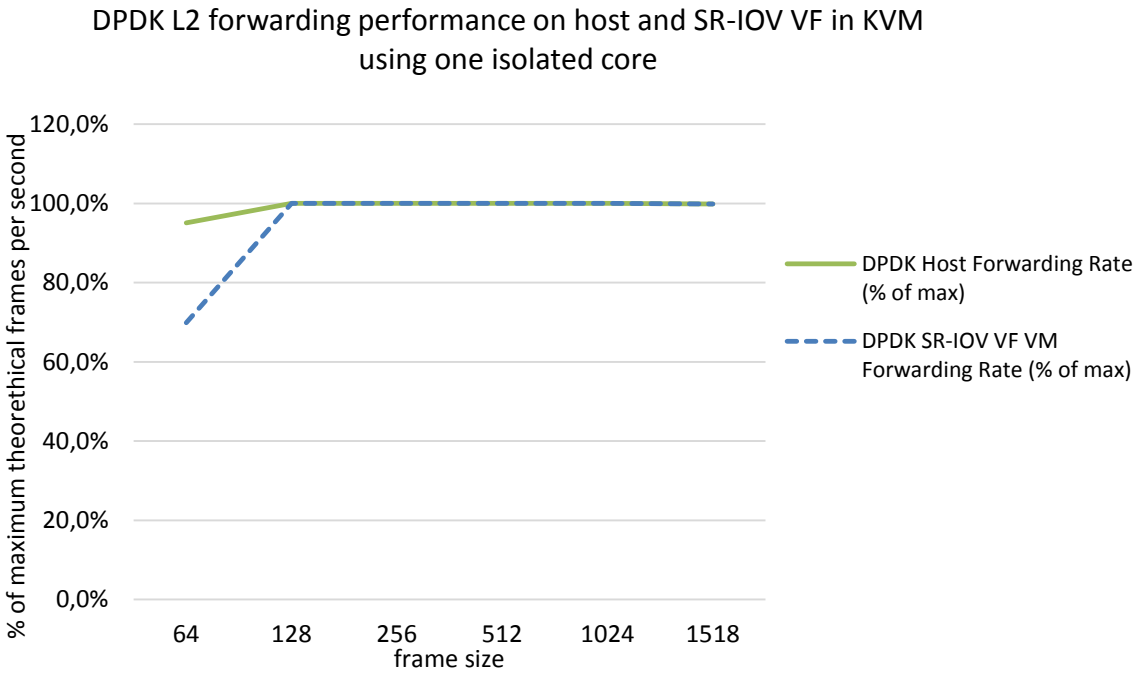

Figure 5: Single core L2 forwarding performance on host and in VM using VF

5. Conclusions

The solutions with user-space direct access to network I/O data provide a substantial improvement in performance compared to running through the kernel. The gain of bypassing the kernel system calls and utilizing user space memory for zero copy of the data traffic to the user application is substantial compared to using AF_PACKET or in-kernel forwarding with OVS. The performance depends on many factors such as system configuration, kernel and driver settings. The tweaks need to be done for the HW setup (BIOS configuration), Linux kernel and network device settings, load balance and packet processing configurations, and also the application settings. Our tests with the ATCA hardware platform revealed that BIOS tuning (such as disabling c-state and frequency scaling) could improve the throughput 10%-20%. Core isolation also helps to improve throughput up to 5%. A lot of configuration options exist for system configuration [12]. Different Ethernet network controllers, e.g. ixgbe [13], support different features to improve performance. These need to be explored and tested to achieve best performance.

The DPDK solution is the fastest, but it is a polled solution that will not allow the processor to idle and thus save power. The polled mode driver together with the run-to-completion application model causes a heavy CPU load. The DPDK solution provides a vendor specific solution optimized for the Intel architecture and Intel NICs that forces the application developers to implement support for yet another vendor specific data plane SDK. PF_RING and netmap provide open application programming interfaces to support development of easily portable applications. PF_RING and netmap allow the devices to be managed and configured through the kernel. With the DPDK polled mode driver, the devices must first be removed from the kernel and the devices cannot be managed through the kernel.

There exist solutions similar to DPDK, provided by other hardware vendors that also provide an environment for data plane applications to use direct network I/O in user space. Similar to DPDK, they usually provide hardware support for

classification and other types of work offloading, saving CPU time for higher layer protocol processing. Such solutions are dependent on vendor specific software development kits. Examples are

- TI Keystone data path acceleration SDK TransportNetLib [14]
- Freescale Netcomm or QorIQ SDK for User Space Data Path Acceleration Architecture (USDPAA) [15]
- Cavium Networks Octeon Simple Exec [16]

The task of migrating applications between different vendor specific solutions is cumbersome and the need for open solutions becomes more and more important. Enea invests time in the Linaro initiative to provide an open SDK solution called OpenDataPlane (ODP) [17], which aims to provide a well-defined data plane application programming interface for networking applications that shall support vendor specific hardware features and allow creative software solutions. In many cases, ODP can be implemented on top of vendor specific SDKs like DPDK. ODP would allow developers to migrate their applications between different HW platforms and solutions with a minimum effort and ensure their development investment in data plane applications.

References

[1] Network Function Virtualization, http://www.etsi.org/technologies-clusters/technologies/nfv

[2] Single Root Input/Output Virtualization, http://www.pcisig.com/specifications/iov/single_root/

[3] Kernel-based Virtual Machine KVM, http://www.linux-kvm.org/page/Main_Page

[4] AF_PACKET, http://manpages.ubuntu.com/manpages/lucid/man7/packet.7.html

[5] AF_PACKET tx, http://wiki.ipxwarzone.com/index.php5?title=Linux_packet_mmap (AF_PACKET transmit example)

[6] AF_PACKET rx, http://www.scaramanga.co.uk/code-fu/lincap.c (AF_PACKET receive example)

[7] Open vSwitch, http://openvswitch.org

[8] ntop.org PF_RING, http://www.ntop.org/products/pf_ring

[9] ntop.org vPF_RING, http://www.ntop.org/products/pf_ring/vpf_ring/

[10] netmap, http://info.iet.unipi.it/~luigi/netmap/

[11] Intel DPDK, http://www.dpdk.org/

[12] http://lib.tkk.fi/Dipl/2012/urn100605.pdf

[13] ixgbe driver tuning, ftp://ftp.supermicro.com/CDR-NIC_1.30_for_Add-on_NIC_Cards/Intel/LAN/PROXGB/DOCS/LINUX/ixgb.htm#Performance%20Tuning

[14] Texas Instrument TransportNetLib, http://processors.wiki.ti.com/index.php/TransportNetLib_UsersGuide

[15] Freescale USDPAA, http://www.freescale.com/infocenter/index.jsp?topic=%2FQORIQSDK%2F2918191.html

[16] Cavium Networks Octeon Simple Executive, http://www.cavium.com/octeon_software_develop_kit.html

[17] Open Data Plane, http://www.opendataplane.org/

[18] http://www.ntop.org/pf_ring/benchmarking-pf_ring-dna/

[19] http://www.ntop.org/pf_ring/pf_ring-dna-rfc-2544-benchmark/