**AINFV: Analysis of Isolation (memory/packet) in Network Function Virtualization**

***Abstract:***

***Introduction:***

The starting days of networks, it was used to designed to send packets between two nodes. As the size of network increased, technology evolved and many network services were introduced from time to time i.e. routing, forwarding, security etc. Traditionally these network services were deployed using hardware middleboxes i.e. firewall, routers etc. These traditional hardware middleboxes are in the market for very long time and still serving their purposes. But there are disadvantages of using this approach, such as mentioned in [**Noval approach**] (a)inflexibility: unable to modify the network services, proprietary issue; (b) Non-Scalability: one needs to buy the new middlebox if the load of the network increases for certain period of time and the load stabilizes after some time, then the new middlebox is useless; and (c) Cost: expensive in terms of upgrading the network components by replacing old middleboxes with latest middleboxes to get the maximum throughput. These disadvantages encouraged the ETSI[**ref**] (European Telecommunication Standards Institute) in 2012 and the idea of NFV (Network Function Virtualization) was proposed. The idea was to replace the hardware middleboxes with software defined network services and deploy these network services as VM (Virtual Machine) on commodity servers. ETSI proposed that NFV will help the service providers as (a) swift deployment of network services; (b) comparatively cheap, by using the commodity servers; (c) more flexibility, upgrading of network service is in software. As mentioned in [**panda thesis**]NFV provide the blueprint of developing the network’s dataplane, that allows the developer to program every packet forwarding in the network. Same in SDN (Software Defined Networking), that provides the blueprint of managing the controlplane, i.e. allows developer to define the custom routing, managing network failures etc. NFV framework provides the following features[**panda thesis**]:

**-Multiplexing:** NFV framework should ensure that the NF (Network Functions) should be hardware independent, this helps in scaling of NF without changing the hardware.

**-Isolation:** NF deployed in virtualized share the under the underlying hardware, NFV framework should ensure the memory and packet isolation without affecting the performance

**-High Performance:** NF connected in series working as NF chains should have maximum throughput or equal to as of hardware middleboxes. NFV framework should ensure this throughput, as there is a major overhead of copying packets from one NF to other.

**-Efficiency:** Framework should ensure the minimal hardware utilization as the aim of NFV is to utilize the commodity servers in effective way.

**-Simplify NF Development:** Framework should ensure the simplicity in development of NF, by separating the tasks into two categories i.e. user defined functionality and preprocessing tasks. All of this should be automated.

**-Rapid Deployment:** Framework should ensure the rapid deployment by production ready NFs (i.e. NF testing and deployment in production environment on the go, to improve the performance) . This safes a lot of time.

***Problem statement***

NFV framework have many advantages but these frameworks are still long way from perfection in terms of development and deployment. For development part as addressed in [**Noval approach**] main issue is the performance trade-off due to low-level programming and optimization issues. Isolation. No standard model is defined, thus every vendor has its own programming model making NF operation complex to work in multi-tanent network environment. For deployment, current idea is to deploy NFs as VMs or Contianers to give isolation as it is main security concern. But at the cost of performance loss. Main idea is to deploy the NFs as a process instead of VMs or Containers.

***Available Solutions***

***Background***

***Requirements***

*As discussed earlier, NFV purposes is to simplify the development and deployment of NFs without changing the functionality and performance offered by traditional middleboxes. As mentioned in [***panda thesis***], there are some requirements that must be fulfiled:*

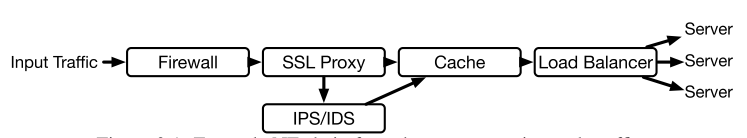
**-Performance:** Framework should not take more than 10 second of microsecond for processing packet. Single NF should be able to process 10-100Gbps of traffic. As the mentioned figures are equivalent to what we get with hardware middleboxes.

**-Efficiency:** Deployment should be done using single machine, because deployment across multiple machine will result in poor resource utilization and performance loss.

**-Chaining:** Framework should be able to combine multiple NF called chaining i.e. *NF1→NF2→....→NFn*. Packet processing starts from *NF1 to Nfn.* ***Fig1*** show the NF chain for processing web traffic.

**Multi-vendor:** NFV framework should support the multi-vendor NF to exists in a network, with security measures i.e. isolation.

**Multi-tenant:** In cloud environment , multiple tenants exists sharing the virtual resource provided by the service provider. NF should be deployed in such a way that the deployment for one tenant should not affect the operation of other tenant.

******

***fig1***

Mentioned requirements help in building well structured NFs and deployment ensuring the isolation. To get the further insights of NFV framework, NFV is divided into two parts, one part deals with the development model and second part deals with the execution model.

**Development model:**

Throughput and latency are two major metrics affecting the network performance. Throughput is packet processing in a given time where as latency is time between sending and receiving of packet. These two metrics depend on number of things [**noval approach**]i.e. context switching and copying, network card to cache copying, TLB (Translation Lookaside Buffer) misses and memory allocation. There are many libraries available for fast packet processing namely DPDK[**ref**] and netmap[**ref**] etc. As mentioned in [**panda thesis**] DPDK (Data Plane Development Kit) libraries provide fast packet processing mechanism by: (a) Using PMD (Poll-mode Driver) instead of depending on the CPU interrupts for acknowledgment of received packet; (b) assigning NIC (Network Interface Card) to single process instead of relying on kernel for NIC multiplexing; (c) provides the interface for connecting NIC directly to NF, instead of using intermediate elements (i.e. vSwitch) that required additional computation for packet movement. These libraries helps in improving performance and developers to focus on optimization (i.e. how the packets should be batched). The use of vectorization, as proposed in [**ref],** VPP (Vector Packet Processing) allows the processing on vectors of packets (i.e. up to 256 packets can be read at once.)

As dicussed in [**panda thesis**] The Click modular router [**ref**] based on DPDK libraries, provides the abstraction to develop new NF in such a way by combining multiple packet processing elements. But does not define the how packets flow between different elements. Click provides the limited functionality for customization. Hence for every new NF, developers have to re-write the those elements from scratch, a lot of time is spent on optimizing the elements. Development model should be modular, some module with fixed functionality and common for all NFs, whereas other modules should be user-defined for specific functionality. Developer is responsible for optimizing the user-defined modules only.

**Execution Model:**

Current practice is to deploy NFs in VMs or Containers and for communication vSwitch is used. VMs and Containers ensures the memory isolation (i.e. operation on one NF will not affect the other NF in network). vSwitch allows the NFs to periodically use the NIC for sending and receiving of packets in networks or between NFs. But all this processing of packets is just copying of packets in network and every NF has its own copy of packets that violates the packet isolation(i.e. at any point in time, only one NF should have access to that particular packet) and considerable hard to achieve. Above mentioned technologies have greater influence on performance degradation. As mentioned in [**model paper**], comparing the single process with dedicated NIC, per-core throughput drops by 3x when processing 64B packets using Containers and up to 7x while using VMs. This performance degraded further more when NFs are chained, Containers are 7x slower compared to NF chained in single process and VMs are up to 11x slower. Furthermore, NF chained single process is 6x faster than NF chained Containers or Vms, where each NF having its dedicated core.

***NF Chains***

***Programming VNF***

***Isolation***

Both packet and memory Isolation is the major challenge to achieve, as it directly affects the performance (i.e. latency and throughput). Main reason of this performance gap is that. Firstly during packet processing, packets tends to cross the memory isolation barrier. Secondly the use of context switch that ensures that packet should cross core boundries [**model paper**]. These isolation issues can be catered as mentioned in [**model paper**]:for memory isolation, instead of using VMs and Containers. Checks can be introduced both compile-time and run-time. For packet isolation, “ZCSI (Zero-Copy Software Isolation)” is proposed by author in [**model paper**] that ensures the “safe 0-copy” packet I/O between network functions. This is implemented using unique types[**ref**].

In order to address the issues of development and execution model, author in [**model paper**] proposed a solution called NetBricks. This solution showcased a different way of developing and executing network functions that contradicts with the traditional approach.

***Memory Isolation: Vms/Conatainers***

***Packet Isolation: High performance I/0 libraries for packet processing zero copy Isolation***

***Proposed Framework***

***Overview***

NetBricks, a framework for developing and executing network functions on a single machine. It requires the re-writing of network function compatible with NetBrick’s development model. As mentioned in [**model paper**]- it is not a limitation because of two reasons : (a) not enough development progress has been done for network functions; (b) NetBricks can also co-exists with the traditional network functions, at the cost of performance.

Proposed framework provides both development and execution environment. For development model, it helps developers to work on the high level abstraction of packet processing tasks and allows user defined programmability. Execution model uses safe language and runtimes to ensure memory isolation, whereas current approach uses scheduling for performance isolation [**model paper**]. Another important aspect to consider in execution model is communication between network functions. Message passing [inter-process communication] must not be modified by network function to ensure packet isolation. To achieve this functionality, NetBricks uses the “static checks” to avoid packet copying. Author in [**model paper**] named this functionality as ZCSI (Zero-Copy Software Isolation). ZCSI allows to achieve the memory and packet isolation as compared to VMs and Containers with no performance degradation.

***Main Components***

***Framework***

As discussed above, NetBricks is a complete package. In the section below, development architecture is explained in detailed, later section describes the execution model to deploy the network functions developed using development model

***Development Model***

As described in [**model paper**], NetBricks allow the developer to focus on the high-level programmability of network function. Network function programmability is divided into five sections: packet processing, bytestream processing, control flow, state management and event scheduling.

***Packet Processing***

In NetBricks, packets structure consist of (a) stack of header; (b) the payload; (c) reference to any per-packet metadata [**model paper**]. Header contains structure that defines the length of a packet based on the functional computation of its contents. Payload is the actual data carried by the packet. Metadata defines the internal communication within network function and it is customizable by the developer using user-defined functions. These user-defined functions are passed along with header structure and are able to access last deciphered header along with payload and related metadata. At the start, header stack contains a “null” value, occupying zero byte space. Author provided the four packet operations as follow [**model paper**] : ***Parse:*** This operation takes the header type and structure as an input. Later analyzes the payload accordingly by using header type and update the header stack. At the end header bytes are removed from the payload;  ***Deparse:*** This operation is applied on header stack, it remove the bottom header from the stack and returns it to the payload; ***Transform:***  This operation implement the user-defined functions on header and payload, allows developer to modify the packet size (i.e. by adding or removing bytes to payload as mentioned in “parse”). It also allows to add and modify the metadata of the packet; ***Filter:*** This operation is used to remove packets to be dropped at a specific node. It is a boolean operation return either True or False. Filter operation is based on user-defined and it drops all the packets at the specific node when the user-defined function returns the false value.

***Bytestream Processing***

Main function of bytestream processing is to convert the bytes arrays into packets. User-defined functions are applied on the bytes arrays, In [**model paper**], author provides the two bytestream operations as follow: ***Window:*** This operation takes four parameters as input i.e. window size, sliding increment, timeout and a stream user-defined function. This operation is responsible for receiving and re-arranging the cached packets and create a stream. User-defined function is called whenever there is enough data received to form a window of appropriate size or connection is closed or the timeout expires. Window operation can also forward all received packets without modifying them or it can drop all the packets and generate the modified byte array using *packetize node;* ***Packetize:*** This operation allows the conversion of byte arrays into packets. Providing the byte arrays and header stack, packtize converts and the data into packets and assign the relevant header. For Implementation of above mentioned operations author use the TCP (i.e. TCP sequence number for re-arranging, FIN packets to check connection closing and *packtize* operation on header by modifying the relevant header fields) [**model paper**].

***Control flow***

Control flow deals with the branching required in network function chains. Branching is used to define the conditions i.e. re-routing the packets to specific port etc. Other purpose of branching is to move packets across cores for processing. In order to get the maximum performance, there should be minimum caching of data between cores. Control flow provides the developer the abstraction for re-routing the packets as desired i.e. by user-defined functions, port, address etc. As mentioned above, control flow branching is useful while implementing the network function chains, it allows the developer to select which packet should be routed to next network function. Author provides the three operations for control flow [**model paper**] as follow: ***Group By:*** This operations allows the branching with in network function and branching across network function chains. It takes two input: number of groups for packet re-routing and user-defined function returning the packets with ID of group to which it belongs. Author also provided some per-defined grouping functions based on criterion i.e. TCP flow; ***Shuffle:*** This operation add additional functionality to “Group By” operation i.e. branching is done on the basis of cores. At “*Runtime*”, Group ID generated by shuffle is used to decide the which core to be used for packet processing. Shuffle allows both user-defined and per-defined grouping. Main point to consider is group id generated by the shuffle is not known at the “*Compile time*”; ***Merge:*** This operation provides a junction, where all the different branches can be merged together i.e. all packets from different branches entering a junction and exists as a single group.

***State Management***

When data is processed across multiple core, performance degradation can be observed. Due to communication between core i.e. cache coherence etc. Typically Developer program the network function to partition state and avoid cross-core access or allow minimal access when required without using partition state. NetBrick’s state management allows access across multiple cores. Within core accesses are synchronized but for cross-core accesses author proposed following options [**model paper**] : (a) no-external-access i.e. one core for each partition; (b) bounded inconsistency i.e. where one core has write access to partition and other cores only have read access; (c) strict consistency i.e. allows multi-read and multi-write access.

***Event Scheduling***

Event Scheduling allows the developer to create user-defined functions, that can be run repeatedly i.e. in order to monitor the NF and get the performance logs periodically etc.

***Execution Model***

NetBricks provides the execution environment to run the network function. This model ensures the isolation and also deals with network function placement and scheduling [**model paper**].

***Isolation***

***Memory isolation***

Traditionally isolation is obtained by using VMs and Containers, at the cost of performance loss for simple network function. Considering the complex network function, this performance loss dominates the other factors. To tackle these performance degradation, NetBricks used different approach to achieve isolation. It makes use of RUST [**ref**] safe language that ensure the type checks and LLVM [**ref**] as a runtime. This combination of safe language and runtime achieves the memory isolation similar to that is obtained using the hardware MMU (Memory management Unit). As mentioned in [**model paper**], safe language and runtime ensures following: (a) disallow pointer arithmetic; (b) bound checking on array accesses i.e. preventing random memory accesses; (c) disallow accesses to null object i.e. preventing undefined behavior to ensure memory isolation; (d) type casts are safe. These above features can be achieved using high-level programming languages i.e. Java, C# etc, but these languages are not system friendly.

***Packet Isolation***

Traditional mechanism is to send packets in physical network, NFV follows the same foot step. Network function sent packets in network by copying. This copying mechanism results in performance degradation of packet processing. NetBricks uses the unique types [**ref**] instead of using the copying mechanism. As mentioned in [**model paper**], unique types is used to cater the data races i.e. disallowing the simultaneous accesses to same data. This approach is applied while implementing the network functions, when network function sends a packet, sender function losses the accesses to that packet and only relevant network function should have accesses to that packet. This ensure the packet isolation without any copying of packets.

Above mentioned techniques used in NetBricks for isolation are referred to as ZCSI (Zero-Copy Soft Isolation) [**model paper**]. NetBricks runs as a single process, that can be assigned to one or more cores for processing and use one or more NICs for packets I/O. Packets are transferred between network functions using “function calls”, In case of network function chains, queue is maintained at the receiving end.

***Placement and Scheduling***

As mentioned before NetBricks operates as a single process in which many network functions can be run. Consider the network function chains running as a directed graph having access to multiple available NIC interfaces. Beforeio implementing, NetBricks have to decide at which core the network chains should be run. Based on this NetBricks make the scheduling decisions for packet processing. As author mentioned in [**model paper**] currently NetBricks places all the network functions on a single core to get the maximum performance. For more complex placements, NetBricks make use of *shuffle* operation to process packets across multiple cores.

In [**model paper**] author uses the “run-to-completion” scheduling for NetBricks i.e. packets entering network function, it starts processing till it exists. Scheduling is needed when dealing with more than one packet, considering this all packets are added to “*window*” operation i.e. receiving and re-arranging till enough packets has been collected and “*group by*” operation i.e. stack up packets to be processed by branch. NetBricks uses “*round-robin*” scheduling these operations.

***Testing***

In order to test and validate the NetBricks, two network functions were re-programmed [**model paper**]. Firstly. the network function that decrements the IP TTL (Time To Live) and drops the packet with TTL zero. Secondly, google’s Magvel [**ref**]

(load balancer).

For the first example network function is build using NetBricks as shown in fig.2. Network functions are described as public function in RUST module. *Line 1* of fig.2 shows how the new instance of network function is created using “*ttl\_nf*” function. *D*uring the network function processing, at *Line 3* ethernet (MAC) header is parsed from packet and later at *Line 4* IP header is parsed from the packet. “*Transform*” operation is applied on the parsed IP header decrementing the packet’s TTL (*Line 5*). After the computation of packet’s TTL, “*Filter*” operation is performed that drop the packets having TTL zero (*Line9*). In the end “*Compose*” operation at *Line 12* indicate the end of description of network function and also allows the chaining of network function. As shown in fig.2, no *“Shuffle”* operation is applied while defining the network function. By default, NetBricks routes all the packet processing to single core. Fig.3 shows the code that is used to execute the user-defined functions. “*NetbricksContext*” is used to execute the user configuration. As mentioned in [**model paper**], at *Line 4* pipeline is created that : (a)receives packets from input queue; (b) these packets are forwarded to network function i.e. “*ttl\_nf*” for processing; (c) later processed packets are again to the same queue. Placement of pipeline is defined in user configuration.

For the second example, author used load balancer (Magvel) network function and tried to implement it using NetBricks. Magvel is responsible for dividing the user requests among the back-end servers. Magvel ensures that : it can be deployed in replicated cluster for scalability and fault tolerance; splits traffic; and handles failures. Magvel uses hashing algorithm (i.e. based on lookup table) to achieve above mentioned aims [**model paper**]. Fig.4 shows the packet processing and forwarding part of Magvel. First the lookup table is created (*Line 8*) and then a cache for recording the backend (*Line12*) that is used to start the instance of Magvel network function. Starting at *Line 15,* network function is declared. First the “*Shuffle*” operation (*Line 16*) uses the built-in functionality of using single core. *Line 17* ethernet headers are being parsed. Later *“Group by” operation* (*Line18*) is performed, it uses the “*ipv4\_flow\_hash*” built-in function to extract the flow hash i.e. consist of IP header and TCP or UDP header. This hash is used for two purposes: (a) ensuring that the received packet is TCP or UDP ; (b) to find the already assigned backend to flow (*Line 24*) or to assign new backend to flow using lookup table (*Line 25*). In the end, network functions generates the vector of nodes relevant to the backend, specified by the operation.

***Analysis Tool***

NetBricks network function chains are build using RUST language and mechanism proposed by the author. That is in contradiction with current available frameworks i.e. OpenMANO etc. These frameworks provide the developer with interface to manage without knowing the underlying programmability used for building the network functions. Author in [**model paper**] also proposed to use the same interface, as it provide following advantages: provides many optimization opportunities, currently using the RUST compiler’s optimization for optimizing the chaining code, later LLVM’s link-time optimization [**ref**] can be used to optimize the whole program, improving performance of packet processing; (b) it can be used to execute the complex network function chains and branches.

***Framework Evaluation***

***Test-bed***

For evaluating the NetBricks, as mentioned in [**model paper**] testbed with following configuration was used: dual-socket servers with Intel Xeon E5-2660 2.6GHz CPUs, each having 10 cores. Each having 120GB of RAM, divided between sockets. Each server was also equipped with Intel XL710 QDA2 40Gb NIC. Hyper-threading was disabled and hardware virtualization was enabled i.e. Intel VT. Servers running Linux kernel 4.6.0-1 and DPDK 16.04 and programming language RUST nightly version. Two virtual switches were used i.e. first Open Vswitch with DPDK (OVS DPDK) [**ref**] and second SoftNIC [**ref**]. VMs used were running on KVM connected to virtual switch with DPDK’s “*vhost-user*” driver. As per requirement of DPDK, docker containers were used with privileged mode and for their connection to virtual switch DPDK’s ring PMD driver were used. By default, PMD driver does not allow both virtual switch to copy packets and no packet isolation was observed as network function can modify the packets after they sent it. To achieve isolation author made few modification in virtual switches, allowing copying while connecting to containers. Even this copying using DPDK’s PMD driver have higher performance than other approaches i.e. “*veth*” pair. In order to test the traffic, DPDK-based packet generator was used running on a separate server having 40Gb NIC. This server was directly connected to test server without any intermediate switches. This packet generator server acted as source and sink. Performance (i.e. throughput and latency) results were collected at sink.

***Overheads and Results***

This section describes the overheads imposed by the NetBricks and the results obtained besides these overheads comparing to baseline network functions build using other framework. For comparison purpose, both NIC and DPDK were configured same for NetBricks and baseline network function. In [**model paper**], author mentioned few overheads that are as follow: for simple network function; (b) for checking array bounds.

In case of simple network function, packet TTL network function was used as shown in fig.2 and explained in ection above. Both NetBricks version and normal version of network function were executed using single core by send packets of 64 byte and observed the throughput. The result of both network functions were almost same as expected by the author. As mentioned in [**model paper**], after 10 experimental runs, the average throughput observed for basline network function was 23.3 MPPS (million packets per second) whereas for NetBricks network function it was observed 23.2 MPPS. For latency at 80%, the RTT (round trip time) for baseline observed was 16.15 micro seconds compared to NetBricks was 16.16 microsecond.

In case of array accesses [**model paper**], safe language was used that imposed some overheads i.e. array bound checking. These checks can be major source of overhead of safe language (i.e. Null-checks and other safety checks performed at runtime are difficult to seperate). In order to test these checks overhead, network function was used that updates several cells in 512KB array during packet processing. The update of these array cells depends on UDP source port number of packet under processing, making it difficult to ignore array bound checks. Author compared the NetBricks network function with baseline network function executed using single-core and used packets with random UDP source port. Fig.5 graph shows the throughput achieved by two network functions when memory accesses per packet increased. NetBrick’s throughput is 20% less as compared to baseline network function for 1 to 8 memory accesses. As the number of accesses increased i.e. 16 or more, the effect of checks overhead started to fade out. This is because of large memory accesses causing cache misses resulting in reduced throughput. These cache misses dominate the overhead imposed by the NetBricks.

***Performance analysis of framework based different Nfs***

To further validate the development model of NetBricks, author further implemented different categories of network functions using NetBricks framework. Some of them [**model paper**] are mentioned below:

- Firewall: based on Click [**ref**], performs the linear scan of an ACL (Access Control List) to find the relevant entry.

- NAT: based on MazuNAT [**ref**] built using Click.

- IDS: based on Snort [**ref**], for signature matching.

- Monitor: maintains per flow counter similar to monitor module of Click.

-LD: Load balancer – Magvel, already described in above section.

Testing of above mentioned network functions was done using both NetBricks framework and original network function. These network functions were executed on single core. Comparison results can be seen in Table.1, NetBricks performance is better as compared to other frameworks. For example as mentioned in [**model paper**] NetBricks NAT has 3x better performance than MazuNAT. Further experiments were performed on NetBricks version of Magvel to be tested on multi-core. Table.2 show the comparison of NetBricks version of Magvel with google’s Magvel. It was observed that NetBricks throughput is 2.9x to 3.5x better than the results mentioned in [**magvel ref paper**]. And average latency observed was 19.9 microsecond for NetBricks and 32 microsecond for original Magvel. These throughput and latency figures are better but not to relied on because these two experiments (i.e. NetBrick’s Magvel version and Google’s Magvel) were performed on two different test-bed.

***Throughput/MPPS***

***Isolation Analysis***

Isolation is ensured by NetBrick’s safe language and runtime checks to avoid costs associated with core boundaries and crossing process [**model paper**]. Author first checked the cost involved with the single network function, based on that evaluation was done to check the results when the length of packet’s network function chain increases. One point to consider here is that these costs are only applicable for simple network functions, but when the computational cost of network function is higher then NetBricks execution environment becomes irrelevant.

***NF vs NF Chain vs Complex NF***

In this section analysis is presented for single network function, network chain and complex network function in term of isolation, and for that author compared the NetBricks with VMs or Containers [**model paper**]. First case, single network function (i.e. “that swaps the source and destination ethernet address for received packets and forward them out the same port”) built using NetBricks and other with C language were compared and runned in their execution environment i.e. VM, Container and NetBricks. Fig.6 show the test environment of VM and Container. A vSwitch is used that receives packets from NIC and froward them to network function running in either VM or Container. Network function process those packets and forward them back to vSwitch, which is them forwarded to NIC. Both vSwitch and network function runs on DPDK and depend upon polling. For better performance network function is assigned its own dedicated CPU and two cores are assigned to vSwitch. As mentioned in [**model paper**], with isolation comes two kinds of overheads: first due to cache and context switching cost (i.e. cost associated with cross process or crossing core boundaries) and second due to copying of packets. In order to analyze these isolation overheads, author used SoftNIC to send packets between Containers without copying (i.e. 0-copy SoftNIC Container) violating packet isolation, later compared the it with NetBricks. As shown in fig.7 NetBricks receives packets directly from NIC, process them using network function code and send them back to NIC, all this process run on a single-core. Fig.8 graph show the throughput achieved using different isolation techniques. As discussed in [**model paper**] Comparison of 0-copy SoftNIC Container and NetBricks with respect to throughput shows that 0-copy SoftNIC Container is 1.6x slower then NetBricks due to crossing core boundaries. Despite the fact that NetBricks was running on single-core and other was using three cores for processing. SoftNIC Container is 2.7x slower than NetBricks due to packet copying. This performance further degrades when using VM instead of Containers. Because VM use “*vhost\_user*” communication channel developed by DPDK from interacting with VM has more performance issues as compared to DPDK’s ring PMD driver used for Container.

Second Case, Network function chain (i.e. multiple instances of single network function, compute packet’s TTL (time to live) and drops the packet with TTL 0). Fig.9 shows the execution environment using VM or Container to run the network function chain (i.e. NF0 and NF1). Assigned two cores to vSwitch and two cores for network function chain, one for each network function. Fig.10 shows the execution environment of NetBricks to run the network function chain (i.e. NF0 and NF1). NetBricks was tested for two cases: (a) with single-core; and (b) with two-cores. Fig.11 shows the throughput comparison of NetBricks with single-core, multi-core, VM and Container. As mentioned in [**model paper**], NetBricks multi-core (NB-MC) is 7x better than container with SoftNIC and 11x better compared to VM with SoftNIC. NetBricks with single-core (NB-SC) is 4x better than container with SoftNIC and 6x better compared to VM with SoftNIC. Author also compared the 0-copy SoftNIC container and observed that with packet size of 64 Bytes, resulted in 3x times performance loss. As shown in fig.10, NetBricks multi-core throughput decreased when the size of chain increased from 4. Author explained this decrease in throughput is due to more cores tries to accesses the NIC (i.e. 4 parallel I/O threads). Above throughput figures are for packet size of 64 Bytes. Increasing the packet size degrades the performance by 15% [**model paper**]. Fig.12 shows the latency when using different isolation techniques.

Third case, complex network function (i.e. increases computation cycles for per packet processing) and used the same execution environment as first case (i.e. fig.6 and 7). Three core for VM and Container (i.e. one for network function and two for vSwitch) and in case NetBricks two cases: single-core and three-cores. Author modified the network function to use busy loops for the number of cycles after packet processing. Fig.12 shows the throughput comparison of per-packet processing by using different isolation techniques. As mentioned in [**model paper**], increasing complexity of network function results in increased computation time that overcome the improvements offered by NetBricks (i.e. 300 cycles per packet). As shown in fig.12, NetBricks isolation performs better as compared to other techniques (i.e. VM or Container).

***Comparison of Frameworks***

***Based on Packet/memory Isolation***

***Discuss new & old Framework introduced***

***Conclusion drawn***

***Future Work***