P4NFV: P4 Enabled NFV Systems with SmartNICs

Ali Mohammadkhan, Sourav Panda, Sameer G. Kulkarni, K. K. Ramakrishnan, Laxmi N. Bhuyan
University of California, Riverside, CA

Abstract—**Software Defined Networking (SDN) and Network Function Virtualization (NFV) are transforming Data Center (DC), Telecom, and enterprise networking. The programmability offered by P4 enables SDN to be more protocol-independent and flexible. Data Centers are increasingly adopting SmartNICs (sNICs) to accelerate packet processing that can be leveraged to support packet processing pipelines and custom Network Functions (NFs). However, there are several challenges in integrating and deploying P4 based SDN control as well as host and sNIC-based programmable NFs. These include configuration and management of the data plane components (Host and sNIC P4 switches) for the SDN control plane and effective utilization of data plane resources. P4NFV addresses these concerns and provides a unified P4 switch abstraction framework to simplify the SDN control plane, reducing management complexities, and leveraging a host-local SDN Agent to improve the overall resource utilization. The SDN agent considers the network-wide, host, and sNIC specific capabilities and constraints. Based on workload and traffic characteristics, P4NFV determines the partitioning of the P4 tables and optimal placement of NFs (P4 actions) to minimize the overall delay and maximize resource utilization. P4NFV uses Mixed Integer Linear Programming (MILP) based optimization formulation and achieves up to 2.5X increase in system capacity while minimizing the delay experienced by flows. P4NFV considers the number of packet exchanges, flow size, and state dependency to minimize the delay imposed by data transmission over PCI Express interface.**

I. Introduction

Software-Defined Networking (SDN) provides a logically centralized control plane for network service providers, enabling them to program and control the network forwarding plane (*e.g.,* SDN controller like ONOS [1]). OpenFlow [2], [3] is a standard for communication between the controller and switches to implement SDN. To overcome OpenFlow's limitations on being protocol specific and not being able to match on arbitrary packet fields [4], programmability using P4 [5] has been introduced. P4 enables simple networking devices to be programmed to support custom protocols and functionality [6]. This programmability provides flexibility to implement a custom networking stack and to quickly adapt (upgrade) to new protocols in the field. In addition, P4 being protocol-independent allows the implementation of custom switching pipelines and provides an API for the SDN controller to populate tables. Thus, P4 augments OpenFlow and extends SDN's capability to offer a new level of control and flexibility to dynamically configure and adapt the network forwarding plane to perform custom match-action processing.

Network Function Virtualization (NFV) is being widely adopted in large scale enterprise and data center networks to cater to more complex and diverse in-network processing. Although NFV supports dynamic instantiation of network functions (NFs) and scaling-out of the network's capabilities in response to demand, NFV creates new challenges concerning service management and orchestration. Since each NFV node is an integral part of the data plane, the SDN controller continues to have a critical role in directing packet flows appropriately through NFV nodes, and NFs within each node.

To ease the burden on the computing resources (CPU processing, data movement) as network link bandwidths increase (going from 10 to 100 Gbps), smart Network Interface Cards (sNICs) are becoming increasingly common. Modern sNICs provide programmable multi-core processors that enable the host to offload custom-built packet processing functions dynamically. These include OpenFlow and P4 match-action processing capabilities. They also support custom NF processing that can be offloaded from the host to the sNIC [7]. However, effectively leveraging the sNIC capabilities and integrating the sNIC into the SDN control domain poses several challenges, namely i) Configurability: The hypervisor or the host software switch needs to be aware of the sNIC capabilities and provide the necessary flow routing configuration to ensure the traffic is routed appropriately between the host and sNIC through specific virtual (SR-IOV) ports. ii) Manageability: An SDN controller now has to control and program both the switch instances (in the host hypervisor) and the attached sNICs to configure the data plane operations. This increases the complexity of the SDN controllers. iii) Utilization: Since sNICs have limited computation and memory capabilities, not all NFs and switching can be realized on sNICs. Complex NFs, including stateful NFs, still need to be realized in the host. Further, chaining different NFs across the host and sNIC may result in a packet making multiple traversals across the host PCIe bus. This can result in high latency and excessive overhead [8]. To overcome these challenges, we have designed and implemented P4NFV, which offers a unified host and sNIC data plane environment to the SDN controller. Our framework takes into account both host and sNIC capabilities (viz., compute and memory resources), communication considerations (sNIC to host virtual port mappings, PCIe bandwidth, and delay) and SDN policies (service chain configuration, service level objectives) to provide a unified view of a P4 capable NFV processing engine. The key contributions of P4NFV are:

- We create a DPDK [9] based P4 NFV framework to facilitate and configure SR-IOV based virtual ports for accessing the sNIC [10].
- We present a unified SDN-agent for the host and sNIC P4 switch. The SDN controller does not need to be aware of the differences between the host and the sNIC P4 capabilities.

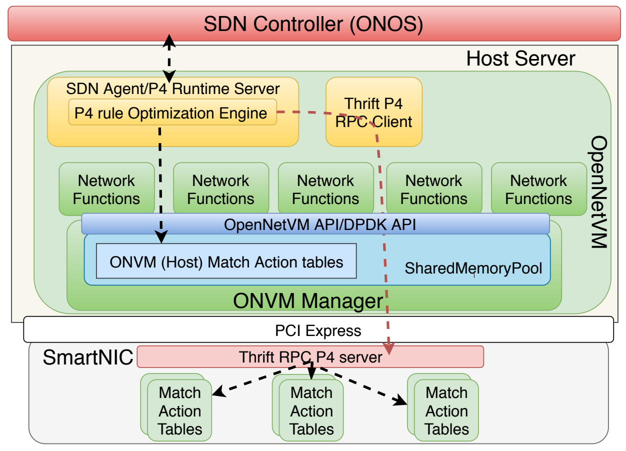

Fig. 1: P4NFV Architecture

- We provide an optimization framework for partitioning the P4 pipeline across the host and sNIC, accounting for the sNIC compute and memory constraints and the NF chain processing requirements.

II. Architecture

P4NFV is a framework that abstracts a server that benefits from a SmartNIC as a single entity to the SDN controller. In this framework, different Network Functions (NFs) that can be executed by the OpenNetVM NFV host [11] or the SmartNIC are sent to the SDN controller as possible external P4 actions which enables the central controller to shape the desired service chain within a P4 pipeline (hereon we use NFs and actions interchangeably) . When SDN Agent of this framework receives the pipelines configurations from the controller, it optimizes the placement of network functions and P4 tables between the host and sNIC. The decision is made locally, thus avoiding the overheads and complexity on the SDN controller. The design also minimizes communication between the host and sNIC, thus judiciously managing PCIe bus overheads. The overall architecture is depicted in Fig. 1.

A. *OpenNetVM*

We build P4NFV on top of OpenNetVM [11], which is a scalable and efficient NFV framework that supports dynamic steering of packets through NF service chains. Routing packets to appropriate NFs is performed by an NF Manager in OpenNetVM, while packet data stays in shared memory. The NF Manager mediates packet access by NFs through packet descriptors. First, we extend OpenNetVM to support sNICs that require the host device to interface with Single Root-I/O Virtualization (SR-IOV) based virtual ports (vport). OpenNetVM was initially designed with simpler DPDK-enabled NICs (that support multiple, 64-128 queues per port), where it maintains a fixed mapping between Ethernet port queues and the host Rx/Tx threads that poll one-or-more of the port queues. However, this design cannot support SR-IOV based vports, especially the Netronome sNIC [12] which limits to only one queue per vport, entailing a strict restriction of 1 Tx and 1 Rx thread per vport. This severely limits the throughput

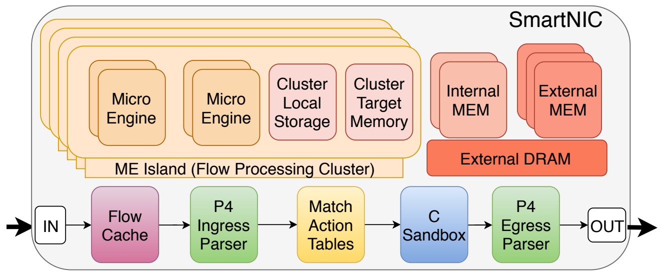

Fig. 2: SmartNIC - Typical Blocks and Processing pipeline (based on Netronome NFP-Agilio4000) design [12]

of the NF processing pipeline. To overcome this limitation, we design a scalable Tx Gateway component (thread with dedicated ring buffer) based on the Facade architectural pattern [13]. Tx Gateway thread can serve packets from different NFs and forward to dedicated vport queues.

Second, we extend OpenNetVM to provide the P4 switch capabilities conforming to the $P4_{16}$ switch specifications [14], [15], *i.e.,* include the P4 based tables (match-action processing pipeline) to process and route the packets. The p4 switch reads the input JSON file and generates the respective pipeline and tables. The progress through the tables is recorded by storing the pointer to the next table in the metadata of each packet. When a packet needs to be processed by an NF, the packet is sent to that NF's queue, receives the service, and then is processed in the switch by using the next table pointer. Hence, the p4 switch can keep processing other packets while a packet is processed in an NF. The p4 forwarding tables are stored in the shared memory and SDN agent can update the rules directly to lower update delay in the tables.

B. *Overview of SmartNIC architecture and capabilities*

sNICs accelerate network processing by offloading some or all of the network protocol processing functions usually performed by the server CPU. Beyond the on-chip network acceleration capabilities of traditional NICs, *e.g.,* checksum, segmentation and reassembly, sNICs provide several programmable processing cores that can be used to implement complex network functions usually part of server processing.

We used the Netronome SmartNIC [12], as part of the P4NFV framework. The sNIC comprises of 60 flow processing cores also known as Microengines, running at 633MHz that is shared by eight threads running the same program. The sNIC also has a hierarchical memory subsystem, providing memory for a FlowCache, and custom user-defined data structures, and match action tables through memory transactions. Different memory subsystem components in the sNIC vary in capacity and also result in different access delays.

The sNIC Microengines can support MicroC programs that are written as actions in the packet processing pipeline as well as separate functions that can be executed asynchronously, invoked *e.g.,* by timer expiration. Typically stateful actions are carried out in MicroC. This allows the programmer to select the appropriate memory subsystem on the sNIC for custom data structures. To fully utilize the sNIC, the host-resident

SDN agent must be aware of the capabilities of the attached sNIC, including partitioning tasks between P4 packet pipeline vs. MicroC programs, and then place data structures on on-chip (of the order of MBytes) memory vs. DRAM.

We focus on providing SDN support for sNICs, paying attention to the modification of rules in the P4 match action table, and handling the effects of caching those rules in a FlowCache. While the logical P4 pipeline is mapped to the sNIC and the table entries drive the selected action set, a FlowCache, similar to the Unified Flow Table fast path [16] and OVS megaflow [17], is employed to achieve high throughput. Since the sNIC supports P4, it allows for programmable parsing, including the definition of new headers. The packet processing cores conduct parsing (or header extraction) along with checksum verification. Subsequent packet processing is load balanced across available flow processing cores (Micro-engines) for match action processing. Based on the actions selected, packets may be DMA'd to the host using the PCIe DMA or sent to the egress packet processing cores that deliver the packets to the MAC buffer.

Typical flow processing pipelines involve multiple table lookups (hash key and index computation) for each packet at each stage of the table pipeline and executing the specified actions. These activities are generally repeated for subsequent packets. To accelerate the data-path processing, sNICs can also include a dedicated cache (called a FlowCache) to store the flow lookup information. However, using such a FlowCache comes with its limitations. The caching of packet processing actions is not suitable for the cases where the action set for different packets of the same flow can differ (such as sampling, when a timer or a metric of interest reaches a certain threshold) because only the first packet's action-set is applied to all the subsequent packets.

C. SDN Agent on the host

We construct an SDN agent in the host to communicate with the central SDN controller, presenting a unified view of the host-based P4 switch and the P4 switch in the sNIC. As shown in Fig 1, SDN agent is a module in the OpenNetVM P4 switch and maintains the host P4 tables in shared memory which is accessible by P4NFV P4 switch.

Upon startup, the SDN Agent gathers information on the capabilities of the sNIC and the software-based OpenNetVM P4 switch and provides it to the SDN controller. The controller uses the P4 Runtime interface to send the unified packet pipeline to the SDN agent. The SDN Agent also receives the set of tables and actions proactively for expected flows. When the SDN agent receives this information, it uses its optimization engine to decide the placement of the NFs (P4 actions) and division of tables between the host and sNIC. The optimization engine (see Section III) seeks to minimize overall packet latency by accounting for the data dependencies, whether the functions best execute at the sNIC or the host and provides the SDN agent the information on the tables to populate on the host and the sNIC (some tables of a pipeline may be on both the host P4 switch and the sNIC). While the SDN controller views the host and sNIC as a unified packet processing pipeline (and associated tables), the SDN agent will need to update the rules appropriately on the host and the sNIC based on its initial partitioning. This may require inserting or updating additional rules in the intermediate table entries, or tagging additional state information through the chaining of multiple actions for same rules across the sNIC and host P4 tables. *E.g.*, suppose we have the rule in Table 3 (on both host and sNIC) to send all the TCP packets with destination port 80 to a firewall (FW) NF (*i.e.,* FW is run on both sNIC and host). The optimization engine may have partitioned the FW functionality so that those packets that need to undergo deep packet inspection (rule in Table 4 of host) - a compute intensive function run only on the host, are sent to the host through an entry in Table 2 of sNIC, while rest of the packets are processed within the sNIC. Thus, in this case, the SDN agent will have to add new rules for sending the subset of the flows to the host by adding an entry into Table 2, with a corresponding action to DMA the packets to the host. However, these rules need some changes before being applied on the tables of the OpenNetVM P4 switch or the tables on the SmartNIC because the rules are sent based on the original pipeline and tables. The SDN Agent finds proper tables based on the assignment it has done before and update the rules accordingly. The act of sending packets to the other device can be done by using intermediate tables between original tables and add an action to send to the other device or by defining a specific action in p4 architecture.

Also, when the SDN controller seeks statistics from different tables, the SDN Agent does the aggregation of the results obtained from both devices (host and sNIC) before sending to the controller. Note: it is possible to infer such required state dependency based on the P4 code provided by the SDN controller. Overall, the SDN Agent ensures the different switching components (host P4 switch and sNIC) are opaque to the SDN controller and optimizes resource usage to reduce the packet latency by intelligent assignment of NFs and tables to different P4 switches within the node.

D. Interfacing with SmartNIC

The SDN Agent administers rules for the sNIC's P4 Match action tables via a Thrift Remote Procedure Call (RPC) interface. As Apache Thrift [18] is a cross-language code generation engine, we generate a Thrift client in C and integrate it with the Host's SDN Agent to add, delete, edit, and retrieve table entries. To add a table entry to the sNIC, the Thrift client first creates an instance of a table entry object. The client then populates the instance's match attribute with a JSON string that indicates which packets should match this rule and the action attribute with a JSON string that indicates the action to undertake and the corresponding arguments to pass to the action subroutine. The client must also specify whether this is the default rule and the rule name. Once the object is constructed, the runtime API is invoked with the table ID and entry instance as arguments. The API runs over Thrift's binary protocol with a blocking socket I/O protocol.

III. Optimization of task assignment between Host and SmartNIC

The optimization engine is a subcomponent of SDN Agent, responsible to efficiently partition and assign packet processing tasks across the host and sNIC P4 switches. It accounts for the forwarding latency, packet processing cost, PCIe bus bandwidth, and packet data dependency constraints. We formulate a Mixed Integer Linear Programming Problem (MILP) to guide the partitioning of the pipeline and functions between the host P4 switch and the sNIC. Overall the optimizer has the following features:

- Considers the delay caused by the number of PCIe traversal
- Considers the update rate of the rules governing different flows (sNIC update rate is limited)
- Considers the data dependency and delay caused by the amount of data exchanged over PCIe.
- Satisfying the maximum tolerable delay of each flow and minimizes the overall delay of the flows.
- Decides the place of different actions and path of the flows.
- If possible, runs NF clones on both (Host and sNIC) devices.

We assume N flows are being served. Our objective is to minimize the total delay observed by all the flows:

$$Obj : \sum_{f=1}^{N} TotalDelay_f = \sum_{f=1}^{N} (ProcessingDelay_f + PCIFixedDelay_f + PCITransDelay_f)$$

Each packet is being processed through a number of tables (t) of the pipeline, where L denotes the length of the pipeline. C_a^H & C_a^S denote the time needed to perform action (a) on the packet by the host and sNIC respectively. Note that the time for performing different actions vary and the time for the same action when performed on host differs from the time it takes to perform the same action on sNIC. *E.g.*, large state manipulation is faster on the host, while simpler actions like forward to next table are faster on sNIC. If only primitive actions are being performed on the packet, then it is one of the special cases of the actions in which the forwarding delay on the tables for that device is the major part of the delay. The delay for forwarding on each table is represented as T^H and T^S for the host and sNIC respectively. If the device does not support a specific type of action, then the delay for this action can be defined as infinity. Finally, Si, j, k shows the action k is applied for flow i in table j and A shows the total number of actions. We define the Processing delay as:

$$ProcessingDelay_f = \sum_{t=1}^{L} \sum_{a=1}^{A} D_{f,t} * S_{f,t,a} * (C_a^H + T^H) + (1 - D_{f,t}) * S_{f,t,a} * (C_a^S + T^S)$$

$D_{f,t}$ is a decision variable that determines that what device serves table t for flow f. If $D_{f,t}$ is one, it is served by host, otherwise it is served by sNIC $((1 - Df, t) = 1)$.

$$D_{f,1} = D_{f,L} = 0$$

Each traversal over the PCIe imposes additional delay on the packets. This delay is reflected in $PCIFixedDelay_f$.

$$\forall f \in 1..N, \forall t \in 1..(L-1) :$$
$$D'_{f,t} = D_{f,t+1} - D_{f,t}$$
$$D''_{f,t} \geq D'_{f,t} \ , \ D''_{f,t} \geq -1 * D'_{f,t}$$
$$U_f = \sum_{t=1}^{L-1} D''_{f,t}$$
$$PCIFixedDelay_f = U_f * E$$

To calculate the delay caused by $PCIFixedDelay_f$, the number of PCIe traversal is needed. As a first step, we define D'. This helping variable shows the changes in the serving place of the flow, from sNIC to host (it will be equal to 1) and from host to sNIC (with value -1) or showing no change with value 0. To count the number of traversals, we need to the absolute value of this variable and a summation over the number of ones in this variable. However, using absolute value function makes the problem nonlinear, which is not desired. To avoid using absolute value function, we used D'' binary variable. By having the introduced constraints, whenever D''s index is one or minus one, D'' will be equal to one for those elements. The only other problem is that other elements can be either zero or one, so an over reporting on the number of PCIe traversal could be expected. However, this miscalculation will not happen, as we are minimizing the overall delay and optimizer will choose zeros instead of ones for non-constrained elements to keeps the number of PCIe traversal minimum.

In addition, to the fixed delay for crossing the PCIe interface, we also add a term for the delay caused by the transmission over this interface, which is dependent on the data size.

$$PCITransDelay_f = \sum_{t=1}^{L-1} (P_f + M_{f,t}) * D''_{f,t} * E^D$$

$M_{f,t}$ shows the amount of state needed in table $t + 1$ for the flow f from previous steps. For example, forwarding data in this step may be depending on the metadata stored in the previous table. P_f is the average packet size for this flow and E^D is the delay in microsecond per byte for the data transfer over PCIe.

So far, we introduced constraints which are related to the objective directly. Hereon, we cover the constraints used to provide the correctness of the solution. The first constraint in this group is to assure the PCIe bandwidth is not violated.

$$B^E \geq \sum_{f=1}^{N} \sum_{t=1}^{L-1} ((P_f + M_{f,t}) * D''_{f,t}) * B_f / P_f$$

In this constraint, we adjust the bit rate of the flow (B_i), base on the overhead of necessary state (M) and calculate the overall bandwidth usage of the traversal between sNIC and the host. The summation over all the bandwidths of the flows, should be less than the bandwidth of the PCIe (B^E).

In addition to the bandwidth limit of PCIe, each flow is required to meet its maximum delay requirement, which is received as input:

$$TotalDelay_f \leq DelayRequirement_f$$

Each action instance can serve a limited number of flows:

$$ActHCapacity_a \geq NumActH_a = \sum_{f=1}^{N} \sum_{t=1}^{L} D_{f,t} * S_{f,t,a}$$

$$ActSCapacity_a \geq NumActS_a = \sum_{f=1}^{N} \sum_{t=1}^{L} (1 - D_{f,t}) * S_{f,t,a}$$

Each device can accommodate several actions at the same time. For example, in OpenNetVM, it is suggested to dedicate a CPU core to each NF. Hence, the number of actions can be limited to the number of available cores. To count the number of actions running on each device, we need a binary representation of the above variable to be able to run summation over them. To convert these variables ($NumActH\&S$) to binary variables ($IsActH\&S$), we define the first four of the following constraints. The next two constraints are for applying the maximum number of actions that can be running on the host (R_H) and the SmartNIC (R_S). Finally, the last one does not allow to have multiple instances of the action on two devices if that action is not cloneable. The values for $NotCloneable_a$, a binary vector, is provided by the system admin. If shared state between different flows is used in an action, we cannot have multiple instances of that action. If an action is stateless or its state is only related to the flows currently using the action, it is cloneable.

$$NumActH_a \geq IsActH_a \text{ , } NumActS_a \geq IsActS_a$$

$$NumActH_a \leq IsActH_a * N \text{ , } NumActS_a \leq IsActS_a * N$$

$$R_H \geq \sum_{a=1}^{A} IsActH_a \text{ , } R_S \geq \sum_{a=1}^{A} IsActS_a$$

$$1 \leq (IsActH_a + IsActS_a) NotClonable_a$$

Another consideration for deployment of P4 tables on the host and sNIC is maximum rate of the updates that can be handled by each device. Especially in our experiment the maximum rate for the rule update on sNIC was limited. To ensure that device will not be overwhelmed by the amount of rule updates, following constraints need to be considered:

$$MaxHostUpdates \leq \sum_{f=1}^{N} \sum_{t=1}^{L} ExpUpdates_{f,t} * D_{f,t}$$

$$MaxSnicUpdates \leq \sum_{f=1}^{N} \sum_{t=1}^{L} ExpUpdates_{f,t} * (1 - D_{f,t})$$

$MaxHostUpdates$ and $MaxSnicUpdates$ show the capacity of the devices in rules updates. $ExpUpdates_{f,t}$ shows how many updates we expect for flow f on table t, and as we explained earlier, decision variable D shows if host or sNIC is selected for serving flow f on table t.

TABLE I: Table entry modification response time on sNIC and P4 OpenNetVM switch

Operation	sNIC (μs)	Host (μs)
Insert	221 ± 11	0.42 ± 0.69
Edit	220 ± 18	0.31 ± 0.61
Delete	169 ± 20	0.31 ± 0.61

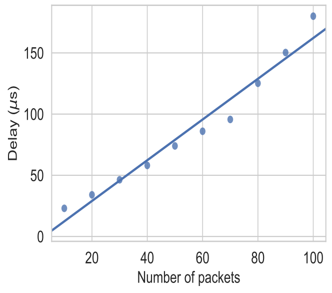

Fig. 3: PCIe RTT measured from sNIC (varying burst size)

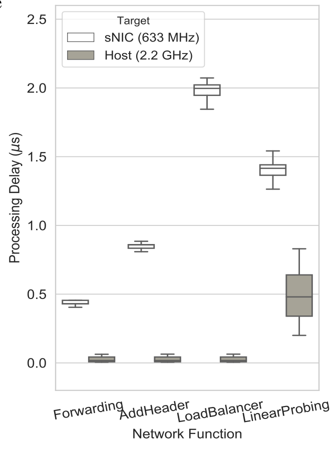

Fig. 4: NF Processing delay

IV. Evaluation

We evaluate the P4NFV framework on a server with 40 Intel Xeon 2.20GHz CPU cores and 256GB memory and Netronome Agilio4000 CX Dual-Port 10 Gigabit sNIC.

A. sNIC and Software P4 Switch Performance

In the first experiment, we evaluate the elapsed time between the table entry modification request and when it took effect. The delay incurred before observing the impact of the new rule on the traffic can be found in Table I. The updates are drastically slower on the sNIC, which limits the sNIC's ability to host flows with a higher rate of rule updates. In the next set of experiments, we demonstrate the processing delay for various NFs resident on the sNIC and host.

In Figure 4, we show that the processing delay experienced by packets in the host NFs are lower compared to that of the sNIC NFs. This can be attributed to the higher frequency of the CPU core that is 2.2 GHz vs. 633 MHz attained from sNIC flow processing cores. However, we observe the latency overhead of one round-trip over PCIe incurs approx. 15μs. Furthermore, the sNIC exploits parallelism by load balancing packets, for processing, on all of its 60 Microengines.

In Figure 3 we compare the total round trip time over the PCIe, observed from the sNIC, for varying burst sizes of 1500-byte packets. sNIC sends these batches of the packets and host returns the packets to the sNIC. We observe a linear relationship between the amount of sent data and the total round trip time. By assuming a linear trend equation and using the least square method, the slope of 1.66 and intercept of -4.27 is obtained. These values are used to minimize the overhead of PCIe traversal in the optimization engine.

B. Optimization Engine

We used the Gurobi MILP solver to implement and solve the optimization formulation. In the first experiment, two of the actions have a limit on the maximum number of flows that they can support on the sNIC. The limit for the first action is 100 flows (stateless) and 300 flows for the second (stateful). The first, stateless, action can be scaled out as needed by having instances on the host as well as the sNIC. Increasing

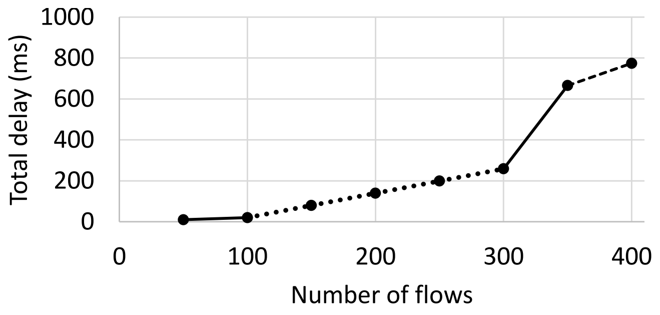

Fig. 5: Total delay based on the number of flows

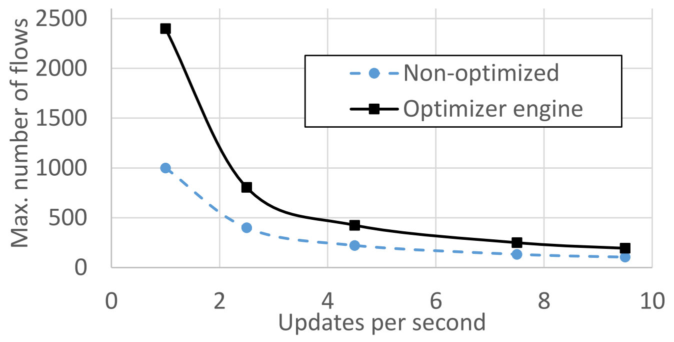

Fig. 6: Maximum supported flows based on the update rate

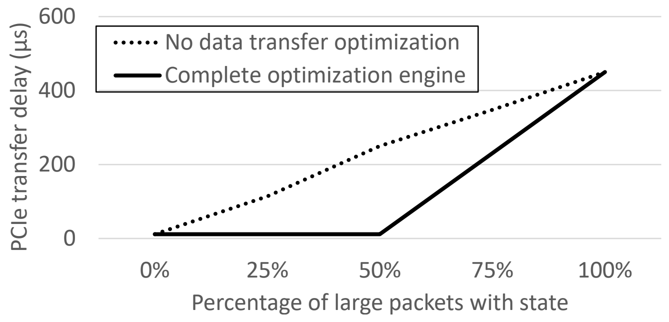

Fig. 7: PCI Express transfer delay based on packet composition

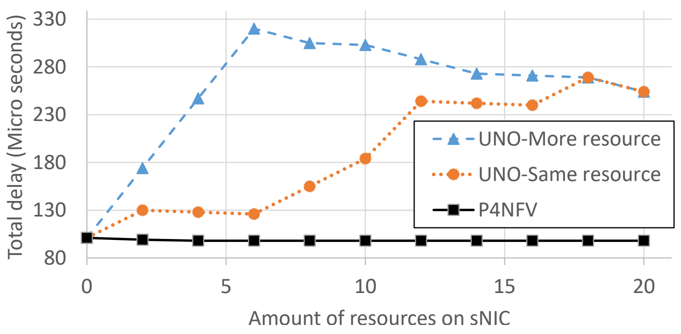

Fig. 8: Comparison of observed delay of P4NFV Vs. UNO [7]

the number of flows from 50 to 400, we see that the delay for all the flows goes up, as seen in Fig. 5. With 50 to 100 flows, all flows are served by the sNIC. Above 100 flows, the instance for the first action reaches the sNIC limit. The overflow of flows is sent to the host. Thus, the delay increases more rapidly. Once we reach 300 flows, the limit for the second action on the sNIC is reached. Since this action cannot be split across the host and sNIC, we observe a sharp increase in the total delay. Beyond this, the delay increases depending on the service time for both actions to be executed on the host. Thus, the optimizer decides to forward packets between the sNIC and host intelligently as needed, leveraging on the ability to instantiate multiple instances of stateless NFs. It initially chooses to instantiate an NF on faster sNIC, but will switch to the host because of its higher capacity, while sharing the execution of actions across both when possible.

Since the rule update rate is lower on the sNIC compared to the host, we then measured the maximum number of flows that can be served in the network. For the update workload, we vary the number of updates for each flow using a uniform distribution varying from 1 to 10 updates per second for each table. Based on our measurement, the maximum update rate of the sNIC is set to 10000 updates per second based on the order of 100 micro seconds to update the rules in tables of sNIC (Table I). The result for this experiment is illustrated in Fig. 6. It is possible to serve up to 2.5 times more flows if update rate is one of the considerations for the placement of the NFs and routing of the flows.

As we observed in Fig. I, the delay over PCIe is depending on the amount of data transmission. Fig. 7 shows if a number of the flows with different average packet sizes and data dependency need to be exchanged between the host and sNIC, the optimizer will select flows with smaller packet size and state dependency to traverse the PCIe. In this experiment, some of the NFs in the sNIC can handle half of the flows and packets of the rest of flows are sent to the host. In addition, packets are separated into two groups. In one group, packets are larger (1500 bytes) and they need 1kB of state from the previous tables. In the other group, packets are smaller (64 bytes) and they do not need any state from previous tables. By selecting the smaller packets with smaller data dependency, the optimizer reduces the delay imposed by PCIe transmission.

Finally, we compared the performance of the proposed optimization engine with one of the related work, UNO [7]. We implemented UNO's optimization formulation for the same solver that we used to solve our problem. We considered a scenario that 20 NFs serve a single flow. Six of these NFs are complex NFs which are faster on the host (10μs processing delay on the host and 40 μs on the sNIC, 4x difference is based on our observation in Fig. 4). Other actions are faster on sNIC (1μs on the sNIC and 2μs on the host). The result of this comparison is depicted in Fig. 8. We have considered two different scenarios for UNO. In *UNO-More resources*, complex actions need more resources in comparison to other actions and in *UNO-Same resources*, they use a similar amount of resources. UNO focuses on minimizing resource usage on the host, while our engine considers all the available resources on both host and sNIC, and uses resources on the host if they are available for different NFs. This difference allows P4NFV engine to minimize the delay caused by the processing of the packets and PCIe traversal.

V. Related work

sNIC as VNF accelerators and Optimization frameworks for offloading: Several works [7], [8], [19], [20] address NF acceleration by offloading NFs from the CPU to sNIC. Uno [7] presents an ILP based NF placement algorithm. They proposed an intuitive NF migration solution to alleviate overloads by identifying the bottleneck NF with maximum processing capacity and migrating it to CPU. However, this approach may adversely impact the latency of the service chain due to increased cross PCIe bus data transfers and not considering the differences in processing delays. In [8], authors propose the NF selection scheme, PAM, that tries to reduce the service chain latency by alleviating the hot spots on the SmartNIC by pushing only the border NFs (*i.e.,* either the start or end of chain NFs on sNIC) or their immediate neighbor NFs. They again seek to minimize the cross-PCIe transfers, but do not account for the difference in the processing delay of NFs, data dependency, or delay constraints of individual flows that we account for in our work. Also, the details on supporting the SDN controller to set up rules on sNIC and host are not considered. Unlike offloading an entire NF, authors of [19] identify a set of candidate operations *e.g.,* TCP connection setup and teardown, connection splicing and packet filtering and coalescing operations for stateful middleboxes that can be dynamically offloaded to programmable NICs, and gauge the expected benefits in terms of CPU load reduction, throughput, and latency improvements.

Virtualization on P4 platforms: The focus of another body of the work, such as [21], [6], [22] , and [23] is on providing the virtualization on P4 platforms. They facilitate serving multiple pipelines on a single P4 switch. While because of the virtualization, they are close to the area of this paper and our framework can benefit from these approaches to support multiple pipelines in parallel, the focus of this paper is on designing an efficient sNIC enabled P4 switch which differentiates the focus point of this paper. Another contribution of the studies, such as [22] and [23], is providing uninterrupted reconfigurability of data plane in P4 switches. Similar techniques are applicable on P4NFV as well.

VI. Conclusion

We identified key challenges (configuration, management, and utilization), in incorporating P4 capability into NFV platforms, including both the host and sNICs. We propose P4NFV as a framework to provide a unified P4 data plane component, including both the host and sNIC P4 switches that can be managed by an SDN controller. P4NFV incorporates local intelligence to efficiently partition the packet processing tables and NF functionality on both the host and sNIC. P4NFV accounts for both networking *i.e.,* bandwidth, workload and packet forwarding latency of the unified host device (sNIC and the host P4 switches) specific constraints in a MILP formulation. We consider the packet processing cost, PCIe bus bandwidth, and latency for crossing the PCIe boundary as well as packet data dependencies. We demonstrate that P4NFV can achieve 2.5x improvement in system utilization and throughput, and minimize overall latency by up to 4x compared to the placement engine of [7].

Acknowledgement

This work was supported by US NSF grants CNS-1522546, CRI-1823270 and CNS-1763929.

References

[1] P. Berde et al., "Onos: Towards an open, distributed sdn os," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '14. New York, NY, USA: ACM, 2014, pp. 1–6. [Online]. Available: http://doi.acm.org/10.1145/2620728.2620744

[2] N. McKeown et al., "Openflow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[3] "Openflow switch specifications," https://www.opennetworking.org/software-defined-standards/specifications, accessed:2019-05-01.

[4] S. Jouet, R. Cziva, and D. P. Pezaros, "Arbitrary packet matching in openflow," in *2015 IEEE 16th International Conference on High Performance Switching and Routing (HPSR)*, July 2015, pp. 1–6.

[5] P. Bosshart et al., "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[6] D. Hancock and J. Van der Merwe, "Hyper4: Using p4 to virtualize the programmable data plane," in *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*. ACM, 2016, pp. 35–49.

[7] Y. Le, H. Chang, S. Mukherjee, L. Wang, A. Akella, M. M. Swift, and T. Lakshman, "Uno: uniflying host and smart nic offload for flexible packet processing," in *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 2017, pp. 506–519.

[8] Z. Meng, J. Bi, C. Sun, S. Wang, M. Wang, and H. Hu, "Pam: When overloaded, push your neighbor aside!" *arXiv preprint arXiv:1805.10434*, 2018.

[9] "Data plane development kit," http://dpdk.org/, 2014.

[10] P. Kutch and B. Johnson, "Sr-iov for nfv solutions," *Practical Considerations and Thoughts (Intel, Networking Division)*, 2017.

[11] W. Zhang et al., "Opennetvm: A platform for high performance network service chains," in *Proceedings of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, 2016, pp. 26–31.

[12] Netronome, "Netronome nfp-4000 flow processor," https://www.netronome.com/m/documents/PB-NFP-4000.pdf.

[13] M. Fowler, D. Rice, M. Foemmel, E. Hieatt, R. Mee, and R. Stafford, "Catalog of patterns of enterprise application architecture," *URL: http://martinfowler. com/eaaCatalog/index. html*, pp. 25–26, 2003.

[14] M. Budiu and C. Dodd, "The $p4_{16}$ programming language." *Operating Systems Review*, vol. 51, no. 1, pp. 5–14, 2017.

[15] "$p4_{16}$ portable switch architecture(psa)," https://p4.org/p4-spec/docs/PSA.pdf, 2019, [ONLINE].

[16] D. Firestone, "VFP: A virtual switch platform for host SDN in the public cloud," in *14th USENIX Symposium on Networked Systems Design and Implementation*, 2017, pp. 315–328.

[17] B. Pfaff et al., "The design and implementation of open vswitch," in *12th USENIX Symposium on Networked Systems Design and Implementation*, 2015, pp. 117–130.

[18] A. A. Mark Slee and M. Kwiatkowski, "Abstract thrift: Scalable cross-language services implementation."

[19] Y. G. Moon, I. Park, S. Lee, and K. S. Park, "Accelerating flow processing middleboxes with programmable nics," in *Proceedings of the 9th Asia-Pacific Workshop on Systems*. ACM, 2018, p. 14.

[20] R. Durner, A. Varasteh, M. Stephan, C. M. Machuca, and W. Kellerer, "Hnlb: Utilizing hardware matching capabilities of nics for offloading stateful load balancers," *arXiv preprint arXiv:1902.03430*, 2019.

[21] M. He, A. Basta, A. Blenk, N. Deric, and W. Kellerer, "P4NFV: An NFV Architecture with Flexible Data Plane Reconfiguration," in *14th International Conference on Network and Service Management*, 2018.

[22] C. Zhang, J. Bi, Y. Zhou, A. B. Dogar, and J. Wu, "Hyperv: A high performance hypervisor for virtualization of the programmable data plane," in *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, July 2017, pp. 1–9.

[23] C. Zhang, J. Bi, Y. Zhou, and J. Wu, "Hypervdp: High-performance virtualization of the programmable data plane," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 3, pp. 556–569, March.