NFLOW and MVT Abstractions for NFV Scaling

Ziyan Wu*, Yang Zhang*, Wendi Feng†, Zhi-li Zhang*
*University of Minnesota – Twin Cities
†Beijing Infomation Science and Technology University

Abstract—**The ability to dynamically scale in/out network functions (NFs) on multiple cores/servers to meet traffic demands is a key benefit of network function virtualization (NFV). The stateful NF operations make NFV scaling a challenging task: if care is not taken, NFV scaling can lead to incorrect operations and poor performance. We advocate two general abstractions, NFLOW and Match-Value Table (MVT), for NFV packet processing pipelines. We present formal definitions of the abstractions and discuss how they can facilitate NFV scaling by minimizing or eliminating shared states. Using NFs implemented with the proposed abstractions, we conduct extensive experiments and demonstrate their efficacy in terms of correctness and performance of NFV scaling.**

I. INTRODUCTION

By running network functions as software on commodity servers instead of on expensive dedicated hardware middleboxes, network function virtualization (NFV) offers many benefits. It provides flexible control of network functions (NFs), and lowers the network management and deployment costs. In particular, NFV makes it easier and cheaper to *scale-out* to meet growing traffic demands by deploying more NF instances on more CPU cores and servers. These benefits notwithstanding, recent studies have shown that correctness, scalability and robustness of NFV hinge critically on NF state management [1]–[5]. This is because most (interesting) NFs are *stateful*, which introduces dependency among packet streams and how they are processed. For example, in the case of a "hole punching" NAT (network address translation) NF, the arrival of a packet from an outgoing "flow" in a network may lead to the modification/creation of a new state entry that allows packets from a reverse (incoming) "flow" to enter the network. Likewise, in the case of network monitoring/metering (NM) NF, once the arrival of a packet from a source host (specified by a source IP address) triggers the packet counter (the NF state) to reach a threshold, the subsequent packets from the same source may be throttled or dropped. Scaling out stateful NFs to multiple NF instances requires sharing/duplicating/sharding the NF state among the cores/servers and splitting/steering the traffic to the correct instances. If done improperly, shared states can lead to incorrect NF operations [5]; even if correctness is ensured, shared states can incur significantly performance penalty – in the worst case, completely negates the benefit of scaling out [6].

While there has been a flurry of academic research on NFV, relatively few attempts have been made to provide general abstractions for NFV. Most relevant work has focused on NFV frameworks and various state management issues (see §II). In contrast, software defined networking (SDN) provides several useful abstractions: i) a *flow table* abstraction whose entries define the ("informal") notion of *flows* and how packets of the flows are processed (*e.g.*, forwarded or dropped); and ii) a *match-action* abstraction [7] which defines the (generic) operations on incoming packets: packets are matched against the table; if an appropriate entry is found (we note that SDN flow table entries), the specified action is performed on the packet. The SDN match-action abstraction is inherently *stateless* in that processing of (*matched*) packets does not induce an *update* of the flow table that affects the processing of *subsequent* packets. These abstractions further enable a (logically) centralized control plane abstraction that maintains network-wide visibility and is responsible for generating flow tables, *e.g.*, generating and installing a flow table entry upon the arrival (of the first packet/packets) of a new flow. All these lead to a unified programmable network data (forwarding) plane and help simplify network management tasks, *e.g.*, via the development of higher-level languages for programming SDN switches and network OSes for running network control programs [8]–[12].

In this paper, we make the first attempt in developing general abstractions for NFV, *with an emphasis on scaling the NFV software processing pipeline on multi-core servers.* Clearly, NFV is far more complicated than SDN, as NFs (as virtualization of hardware "middleboxes") perform diverse and complex (*stateful*) operations on packets that go beyond packet forwarding. In other words, "actions" performed on packets are *NF-specific*; new actions will be often introduced when new NFs are developed or existing ones upgraded. Likewise, the actions (updates) performed on the NF state are also specific to each (type of) NF. We do not intend to standardize the "actions" (on packets and state) for NFV (in contrast to SDN). Nonetheless each NF follows a general packet processing pipeline: upon arrival of a packet, it is matched against a state table (or whatever its representation in software code) using certain header fields (and other metadata attributes); the matched state entry dictates what actions to be performed on the packet, and an update to the state may also be performed. Therefore we focus on developing useful (and formal) abstractions for the general NFV packet processing pipeline – the NF state representation and the induced traffic associated with each state entry. The goal is to help facilitate NFV scaling to tackle the aforementioned correctness and performance issues.

In particular, we propose two novel abstractions, **NFLOW** and *Match-Value Table* (MVT), as generalizations to the notions of flow and flow table in SDN (but also with several key differences). Intuitively, given an NF and its state (and actions), a sequence of packets that match a *single* NF state entry form an **NF**LOW. As a general NF state representation, each entry of MVT contains a *match* field (a predicate defined on packet headers and related metadata) and a *value* field[1] which represents the value of an NF state entry. Using these abstractions, the general packet processing pipeline is as follows: A packet to be processed is matched against the MVT, the matched entry determines the **NF**LOW it belongs to and the actions to be performed; the corresponding value field of the MVT entry may also be updated accordingly that affect the processing of subsequent packets in the same **NF**LOW. Clearly, different (types of) NFs define different kinds of **NF**LOWs. For example, an **NF**LOW defined by the stateful "hole-punching" NAT NF is bidirectional: it consists of packets from an outgoing UDP/TCP (5-tuple) flow and an incoming UDP/TCP flow with the same and reverse source/destination IP address and port number pairs. An **NF**LOW defined by the aforementioned NM NF consists of packets from the same source IP addresses (including all 5-tuple flows from the same source).

Unlike an SDN flow table which associate priorities to flow table entries – in other words, a packet may match multiple entries (we refer to these entries as "entangled"), with the one of highest priority determining the action on the packet, we impose a *uniqueness* condition on MVT (NF state) entries: *each packet (of an* **NF**LOW*) can match one and only one entry in* MVT. As such, we see that both the MVT entries and **NF**LOW's defined by each NF are independent of each other. MVT can be efficiently implemented as, for example, a key-value store, with one `get` operation and at one most one `put` operation per packet. These properties offer key benefits in terms of ensuring NFV scaling correctness and performance. When scaling an NF to multiple instances (whether across cores within a single server or across servers), we can arbitrarily partition a large MVT into smaller (sub-)tables (one per instance) while ensuring correctness of stateful packet processing. The **NF**LOW abstraction makes it easy to split and steer the corresponding **NF**LOW's to the right instances. As the partitioned MVT tables are independent, there are *no shared state* among the NF instances. Furthermore, we can shard the MVT in such a manner that each shard fits the L1/L2 caches dedicated to each core. These are crucial in ensuring high NFV scaling performance [6] especially when the line rate requirements of packet processing is 100 Gbps and beyond. The independence of MVT entries and **NF**LOW's enables dynamic NFV scaling without compromising correctness and performance that are otherwise not guaranteed with flow/state tables with entangled entries.

The remainder of this paper is organized as follows. In §II, we further motivate the two proposed abstractions and discuss related work. In §III, we provide a formal formulation of the **NF**LOW and MVT abstractions and present how to composite MVT. In §IV, we present the MVT partitioning problem. It

[1]The value field is a general object that may correspond to a single variable, *e.g.*, packet counter, a list or array of variables, or contain other more complex data structures.

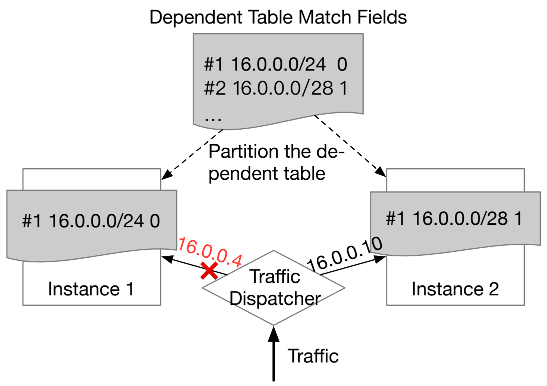


Fig. 1: Dependent table splitting without entry dependency considerations result in incorrect packet processing.

studies how to partition the MVT to multiple shards which can be placed across NF instances. In §V we demonstrate the performance benefits and correctness guarantees provided by **NF**LOW and MVT abstractions. Finally, we conclude the paper by summarizing its contributions.

II. Motivation and Related Work

NFs are programs that receive packets and process them based on *state*. This process can be formally represented as $(states_{new}, pkt_{out}) = NF(state_{cur}, pkt_{in})$. In other words, an NF transforms ("actions") an incoming packet to an outgoing packet based on the information carried on the incoming packet as well as the (current) state. A key distinction between a *stateful* NF from a *stateless* NF lies in that the former may also modify the state – the state is used by a stateful NF to track the changes and guide how subsequent packets are processed. Hence how to manage the state for efficient processing and scaling is a critical part of the design of NFs.

A. Need for New Abstractions for NFV Scaling

In order to attain near line-speed NFV packet processing to meet growing traffic demands, we need better state and traffic abstractions to effectively leverage multi-core servers for NFV scaling. We argue that the "flow table" abstraction used for SDN (which is implicitly adopted by many existing NFV platforms and NF implementation) is not best suited for NFs running on multi-core servers. As the "state" for (mostly) stateless data plane operations, the flow table abstraction (with "entangled" flow entries or rules with priority) is designed for SDN switches and can be viewed as an abstraction for hardware TCAM[2], which can simultaneously match a packet against multiple entries and outputs the best-matched entry (with the highest priority) with one operation. In other words, looking up a packet that matches multiple entangled entries ("rules") and deciding on the best entry using rule priorities incurs no additional costs in hardware switches (or NICs). Performing the same function in software requires multiple

[2]The generalizations of the openflow protocol to include multiple flow tables and those of P4 to support more general programmable switches with limited stateful operations again reflect switch hardware which often includes multiple TCAMs in the *hardware* packet processing pipeline.

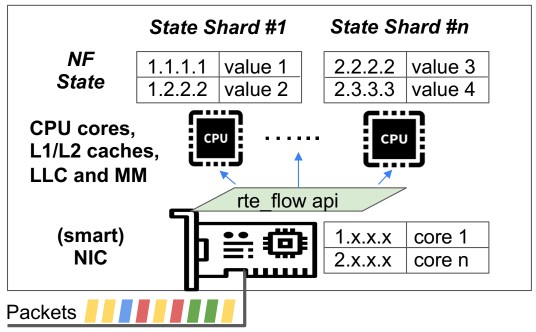


Fig. 2: Problem setting for scaling general NFs

memory accesses, which can significantly slow down the software NFV packet processing.

The *entangled* flow table can create more severe problems when scaling NF to multiple cores. As illustrated in Fig. 1, we scale an NF by running two NF instances, each with its own flow table. Two "entangled" flow entries, $16.0.0.0/24$ and $16.0.0.0/28$, are split into separate tables, which can cause incorrect operations if packets are steered to the wrong instance (by mechanism such as RSS [13]). For stateless NFs such as layer-3 forwarding, one might simply replicates the *read-only* flow table; nonetheless, if it is too large, it may slow down the packet processing – ensuring the state table is relatively small (*e.g.*, to fit with the caches) is a key reason for NFV scaling. For stateful NFs, replicating the read/write state creates *shared* states that requires consistency checks [5], leading poor scaling performance [6]. Hence we need better abstractions to facilitate NFV scaling in a correct and scalable manner, namely, to guide the NF state sharding and traffic splitting/steering. By studying a number of existing NFs [14]–[21], we find that "entangled" flow tables with entangled rules are common; some even use multiple dependent tables. This is likely a legacy of "virtualizing" (by emulating) hardware middleboxes in software. This motivates us to develop new *abstractions* for effectively scaling the NFV *software* packet processing pipeline on multi-core servers by eliminating unnecessary state dependency and minimizing shared state. In §III-B we will present methods to convert dependent/entangled flow/state tables in an NF to a single MVT with *disjoint* entries.

B. Problem Setting

Fig. 2 illustrates the problem setting for scaling general NFs. To scale out the NFs, we partition the NF state into different shards for dedicated cores to process. When NIC receives the packets, it distributes incoming packets into different queues owned by dedicated CPU cores based on the rules installed in the NICs. Then the problem is to design an algorithm that efficiently partitions the NF state, and a dispatching mechanism in the NIC. More specifically, given NF state, the algorithm is to calculate the tables (NF state tables and NIC dispatching table) in Fig. 2, and the dispatching mechanism is to distribute incoming packets accordingly.

C. NF State Management and Related Work

Several NFV frameworks [21]–[29] have been developed in recent years, with an emphasis on speeding NFV packet processing using multi-core and multiple servers for elastic scaling. Most do not explicitly address the state management issues, implicitly assuming that NFV scaling can be performed using 5-tuple flows (as in the conventional cloud computing systems). The authors in [1], [4] were perhaps the first to study the state management issues in NFV and their impact on NFV scaling, where they classified various NF state into "partitioned" state (related to per 5-tuple flow) and "shared" state and developed a framework for managing and replicated these two types of state. OpenNF and related efforts [2], [5], [30] aim to build a SDN-like centralized plane for NFV management and orchestration, with a focus on robustness issues via state replication, migration and recovery. A rollback mechanism is developed in [3] for NFV state recovery. The authors in [31] propose a "stateless" framework for maintaining and performing all NF state operations in a central server, whereas the authors [32] employ distributed hashing table (DHT) for managing the NF state as a distributed key-value store.

To meet the increasing traffic demands, switches and servers with 100 Gbps and higher-speed ports and network interface cards (NICs) are becoming more common. This poses more challenges in performing NFV packet processing at near line-speeds. Several recent studies [6], [33], [34] have examined the impact of server microprocessor architecture on NFV scaling performance. For example, NFV performance profiling in [6] shows that it is critical to maintain the NF state within the L1/L2 caches that are dedicated to individual cores to reap the benefits of scaling NF instances on multiple cores. When the NF state grows too large and cannot fit into the L1/L2 caches, NFV scaling performance degrades quickly, as access speeds to the LLC/L3 cache (shared among all cores on the same NUMA node) and the main memory can be 10s to 100s slower. Worst, *shared state* with locking (for consistency and correctness [5]) completely negates the benefits of scaling NF instances to more than two cores. Clearly, with at least 1s and 10s millisecond latency, accesses to NF states maintained on another server via centralized or distributed state management frameworks [31], [32] will further degrade NFV scaling.

III. NF Model and NFV Abstractions for Scaling

In this section, we present a formal model for representing stateful NFs and introduce several important abstractions.

A. NF Model: MVT, NFlows and Atomic State Units

We model each network function abstractly as a mapping, NF: $\{$ In, $State$ $\}$ $\rightarrow$ Out, where In and Out represent the input and output packet streams, and $State$ denotes the NF state. We represent the NF state as a table of tuples ("entries"): $\langle pred, value\rangle$, where $pred$ is a *predicate* defined

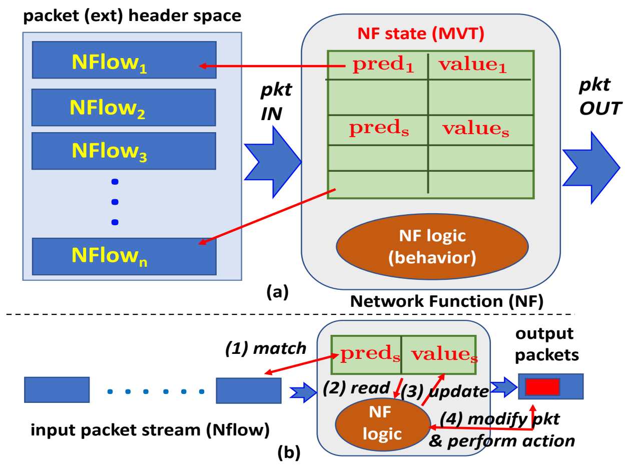


Fig. 3: NF model: MVT, NFlows and atomic state units

on the *extended*[3] packet header space, and $value$ can be any *struct* (e.g., integer, float, Boolean, list, finite state machine, etc.) that is used to keep track of the state or state transition that governs the actions on packets matching $pred$ (see Fig. 3).

Definition 1 (**Match Value Table** (MVT)). *Given a table of S tuples, $\{\langle pred_s, value_s\rangle : s \in S\}$, representing the state of an NF, we say it is a* **match value table** *(MVT) if it satisfies the following condition: for any input packet p of the NF, it matches* one and only one *predicate: namely, if $pred_s(p)$=***true***, then $pred_{s'}(p)$=***false*** *for any $s' \neq s$, and $\vee_s pred_s(p)$=***true***.*

We remark that the condition we pose on the predicates (on the *extended* packet header space) of the MVT corresponds to the notion of *atomic predicates* introduced in [35]. In other words, the MVT entry predicates form a set of atomic predicates. By abuse of notation, we will also refer to a data structure (e.g., a hash lookup table) implementing an NF MVT state representation, i.e., its entries satisfying the same atomic predicate condition, as an MVT.

Definition 2 (**NF Flows** & **Atomic State Units**). *We refer to a sequence of input packets matching each predicate $pred_s$ in an MVT defined above for an NF as an* **NFlow** *of the NF, and each tuple $\langle pred_s, value_s\rangle$ as an* **atomic state unit** *of the NF.*

As an example, consider a "hole-punching" network address translation (NAT) NF. An NFlow of this NF corresponds to a pair of *incoming* and *outgoing* TCP/UDP flows with matching (public) source and destination IP addresses and port numbers. In the case of a network monitor (NM) NF which tracks and counts the number of packets coming from outside hosts (e.g., for DDoS detection), an NFlow contains all packets with the same source IP address.

We can view an MVT as a (state) table or data structure that matches **NF**LOWs to their states uniquely (with one *lookup* operation): each entry in the MVT comprises of two fields: the *matching field* and the *value field*. The uniqueness of the MVT ensures that the matched **NF**LOWs for different entries are *disjoint*. This property guarantees that each packet can only match one MVT entry: this facilitates software-based packet table-lookup, as the one-to-one matching eliminates the need for comparing the priority of multiple "entangled" entries that one packet may match, as in existing SDN systems which require multiple lookup operations, slowing down the *software packet processing pipeline*. As each entry is an *atomic* state unit, the MVT can be *arbitrarily* partitioned/sharded for *NFV scaling on multi-core servers* such that each NF state shard can be, e.g., fitted into the *core-dedicated* L1/L2 caches without *shared state* to ensure high NFV scaling performance [6].

Formally, with the notions of NFlows and atomic state units, we can decompose or "slice" an NF into *disjoint* elements, $NF = \uplus_s NF_s$, by *simultaneously* partitioning the input (extended) packet header space and the state of NF: Here $\uplus$ represents *disjoint* union, and $NF_s = \langle pred_s, value_s\rangle$, with $\{pred_s\}$'s satisfying the atomic predicate condition above. We will use $pred_s$ as a short hand for an NFlow, namely, a sequence of packets p_i's for which $pred_s(p_i)$=**true** for all the packets. In a sense, we can regard our notions of NFlows and atomic state units as generalization and extension of SDN "flows" and "atomic predicates" introduced in [35] for *stateless* networks and operations to *stateful* NFV and network functions. By refactoring NF as $\uplus_s NF_s$, each disjoint element NF_s operates on its own NFlow (specified by $pred_s$) *independently* – performing transformation (e.g., re-writing headers) on packets of the NFlow and updating its state entry $value_s$ based on the NF logic (see the bottom part in Fig. 3).

B. MVT *Composition*

The disjoint nature of MVT brings benefits for software-based packet-entry matching in NFV systems. However, it shifts the complexity to the composition of an MVT compared to the composition of a traditional dependent table. Thus, we present the two methods for composing MVT in this section: a *pro-active* method and a *reactive* method.

1) Pro-active Composition: The pro-active method is applied for composing MVT when NF rules are pre-defined. For example, access control has pre-defined rules to block certain traffic. Such NF rules are defined based on users' intents, and thus they may be entangled with each other, as illustrated in the above L3FWD example. We rely on multi-dimensional traffic classification mechanism [36] to decouple them.

2) Reactive Composition: NF state is not only generated from pre-defined rules, but also created at the packet processing stage. In the case of a "hole punching" NAT, the arrival of a packet from an outgoing "flow" creates a new state entry that allows packets from a reverse (incoming) "flow" to enter the network. Thus, in reactive composition method, it creates an MVT entry for each **NF**LOW. This method is often used by NFs whose **NF**LOW predicates are defined by the same set of packet fields and/or metadata attributes that the **NF**LOWs are guaranteed to be disjoint.

[3]The *extended* header space includes not only the standard header fields defined by SDN or P4 programmable switches and used in, e.g., header space analysis-based *stateless* network verification, but also special tags, meta-data, application-layer protocol header fields, even certain payload that are specified and operated on by various NFs.

```
alert tcp $HTTP_SERVERS $HTTP_PORTS -> $EXTERNAL_NET any (pcre:"/^Command\s+?\b/sm";)
```

Fig. 4: A SNORT Rule Example.

A Real-world NF Example: We present a real-world example – SNORT [19], a rule-based Intrusion Detection System (IDS), to show how an MVT is constructed by combining both pro-active and reactive methods.

SNORT identifies intrusion by matching (suspicious) signatures against packet payload. As shown in Fig. 4, each rule comprises three parts, a predicate ("`tcp $HOME_NET 2589 -> $EXTERNAL_NET any`"), a signature ("`/^Command\s+?\b/sm`"), and an action (`alert`). SNORT filters incoming packets by predicates to retrieve signatures. SNORT keeps track of flow status (per predicate) by maintaining a finite state machine (FSM). For example, in Fig. 4, the signature is a regular expression containing the suspicious "Command" keyword. A real SNORT data pattern part may be very long depending on its internal signatures. One packet payload may not be long enough to match till the end of the signatures. Thus, the FSM is used to keep track of the status of matching (*e.g.*, where the matching has been reached) for the following packets to continue the matching. During the arrival of a packet, SNORT looks for two types of information based on packet predicate: one is signature in the user defined rule, while the other one is FSM which keeps track of the signature matched so far. Then, SNORT continues the pattern matching and updates FSM. If no corresponding FSM is found, SNORT creates a new FSM with initial state for the new predicate. When the FSM reaches the termination state (*i.e.*, matching till the end of a signature), the corresponding action (*e.g.*, `alert`) is triggered.

Hence, SNORT maintains two MVTs according to the proposed abstraction: a rule MVT and an FSM MVT. The rule MVT is constructed pro-actively and converted from predefined rules (*e.g.*, Fig. 4), while the FSM MVT is generated by a reactive method during packet processing. When we compose the rule MVT, the predicate of each user defined rule is used as the key in the rule MVT, and its corresponding signature-action pair is stored as the value. In contrast, when we compose the FSM MVT, the predicate of each incoming packet is taken as key, and its associated FSM state is stored as the value in the FSM MVT.

IV. Partitioning MVT for NFV Scaling

As the entries of the MVT are mutually exclusive, we can freely partition it to be processed in parallel in multiple instances running in different cores on multi-core systems for higher performance. Besides, we need to ensure that the packets are steered to the instance directly that contains the corresponding entry to avoid inter-core transfer of software dispatching overhead for optimal performance. Thus, the steering rules calculated from the partition should be installed in the NIC to facilitate hardware dispatching.

A. Problem Statement

The purpose of MVT is to make entries independent and enable them to be partitioned freely into multiple MVT shards, so that different shards can be placed on different instances for the benefits of scaling-out. MVT itself can be partitioned freely, but two requirements make it challenging to apply on real-world NFV platforms: (i) the traffic should be dispatched to the correct instance that has its MVT entry, and, (ii) the workload should be evenly distributed to each instance. We next describe why these two requirements are essential.

Consider the case of running an NFV system on a multi-core CPU server equipped with 100 Gbps DPDK-enabled NIC cards that support the Receiver-Side Scaling (RSS) functionality. Each NF instance is deployed on a dedicated CPU core, and each core has its dedicated L1/L2 caches. Processing packets at 100 Gbps or above line-rate speed requires fetching data (states and packets) directly from L1/L2 caches (whose access latency should be L1/L2 bound [33]) as the average packet processing latency for 64B Ethernet packets is only 5.7ns. However, the L1/L2 cache have limited capacity (*e.g.*, 32KB L1-D cache and 1MB L2 cache on our server with Xeon Platinum 8168 @2.7GHz). Because it is crucial to exploit L1/L2 for every core, we need to ensure that workload is distributed evenly among all the available cores.

Modern servers are often equipped with (DPDK) NICs that employ hardware mechanisms to dispatch packets to different NF instances running on different CPU cores. DPDK provides a flexible interface to manipulate these hardware mechanisms. DPDK rte_flow API enables installing rules for different items and actions on packets – it is more flexible compared to RSS that uses only 5-tuple TCP/UDP/IP header fields. Items here refer to the headers of common protocols or bit-strings starting from a certain offset; actions refer to operations such as dispatching packets to different queues, dropping some of them, etc. If the NIC has enough (hardware memory) capacity to install rules that map each **NFlow** to a physical core, it is then trivial to implement the dispatching mechanism. The challenge, however, lies in how to use the limited NIC rules for partitioning MVT to meet our design goal.

B. Problem Formulation

Terminology: Given a MVT, we can define an incompletely specified Boolean function $f_{mvt} : B^n \mapsto \{0, 1, -\}, B = \{0, 1\}$ where n is the number of bits to specify a predicate. For a vector $v \in B^n$, if there is a predicate $pred$ in the MVT such that $pred(v) = 1$, we assign 1 to it, otherwise we assign $-$ to it. Given the assignment, we can have two subsets of B^n: the on-set, $f^1 = \{x|f(x) = 1\}$, and don't-care-set $f^{DC} = \{x|f(x) = -\}$. A cube c is set of points in B^n defined by the function $f_c : \{0, 1, ..., n-1\} \mapsto \{0, 1, -\}$, which can be represented as a vector $v_c = (f_c(0), f_c(1), f_c(2), ..., f_c(n-1)) \in \{0, 1, -\}^n$.

Partitioning f^1: In order to create NIC rules for correctly dispatching traffic, we need to find a non-overlapping partition of the cover (or cube) for f^1. The result of the process should satisfy the following conditions:

1) The cardinality of the cover should be less than the maximum number of rules that the NIC supports
2) The cardinality of the partition should be equal to the number of instances and each partition is assigned to one NF instance only.
3) The workload corresponding to each partition should be as even as possible to efficiently utilize the L1/L2 cache.

To satisfy the constraints and requirements, we need a process to reduce the number of products in the cover because the initial number of products is equal to the number of atomic predicates, which is often much larger than the number of NIC rules. After the process, the dispatching policy as a collection of products should be able to be installed into the NIC. Next, we introduce the basic idea of the process.

1) MVT *as a BDD with CE:* We start by constructing a BDD (Binary Decision Diagram [37]) with CE (Complemented Edges) for the Boolean function. We use it because its ability to provide a canonical form for the Boolean function, its compactness compared to the BDD and it is trivial to obtain the cover for the Boolean function in the form of DSOP (Disjoint Sum of Products) by extracting the "1-path". Then we apply the techniques of [37] to minimize the number of products by minimizing the number of paths. After this step, we can get a cover equal to f^1. Each product in the cover can be installed as a rule in the NIC. However, the result may not be acceptable because the number of entries may still be larger than the available entries in the NIC.

2) Merging products by leveraging don't-care-set: The intuition for the step is that we can leverage the points in the don't-care-set to expand the cover to reduce the number of products in the cover. We introduce following notations to analyze the cover.

Definition 3 (Connected Cubes)**.** *Two cubes c_1, c_2 are connected if $c_1 \cap c_2 \neq \emptyset$.*

Definition 4 (Path)**.** *There exists a path between two cubes c_1 and c_n if there is a list of cubes $(c_1, c_2, c_3, ..., c_n)$ such that $\forall k \in \{1, 2, ..., n-1\}$, c_k and c_{k+1} are connected.*

Definition 5 (Component)**.** *A set of cubes $\{c_1, c_2, c_3, ..., c_n\}$ form a component if $\forall k,j \in \{1, 2, ..., n-1\}$, there exists a path from c_k and c_j.*

To install a list of products as rules in the NIC, we need to ensure that the packets of one component are dispatched to one instance. Otherwise, a packet may be included in the predicates of two partitions which may result in the incorrect behaviour of dispatching traffic. Hence, an element of the resulting partition should be a union of a collection of components.

Definition 6 (Centroid of a Cube)**.** *Given a cube c_1, the centroid of a cube $Centroid(c_1)$ is a vector a^n where $a \in \{0, 0.5, 1\}$, the mean of the extreme value of each dimension.*

Definition 7 (Distance of Two Cubes)**.** *Given two cube c_1, c_2, the distance of two cubes is the distance of their centroid.*

Given a cube, when we want to select another cube to merge them together, we always choose the closest cube to merge. The result of the merging should be a supercube which is the smallest cube that contains both cubes.

Definition 8 (Supercube)**.** *Given two cube c_1, c_2 defined by f_{c_1} and f_{c_2}, the function of supercube c_s is defined as follows:*

$$f_{c_s}(i) := \begin{cases} 0 & f_{c_1}(i) = f_{c_2}(i) = 0 \\ 1 & f_{c_1}(i) = f_{c_2}(i) = 1 \\ - & \textit{otherwise} \end{cases}$$

3) Balancing the Workload among the Instances: As explained by the previous section, we need to balance the workload for different partitions for better utilization of the L1/L2 cache since each partition are assigned to one instance. To characterize the workload, we develop the concept of weight of the cube based on the number of packets it contains, the related traffic pattern, priority of the corresponding **NF**LOWS.

Definition 9 (Weight of the Cube)**.** *For each point t in the Boolean space we assign a value $v \in R$ such that $weight(t) = v$ based on whether it corresponds to an* **NF**LOW*, characteristics related to its traffic pattern (pps, throughput) and priority and any other metrics that influence its processing overhead. The weight of the cube $weight(c) = sum(\{weight(t) | t \in c\})$.*

Ideally, for a balanced partition, each element of the partition should have around the same weight. For the partitioning problem for multiple servers, we can create a partition according to the capabilities of different instances.

4) NIC Entry Constraint: The number of traffic steering entries should not exceed the number of NIC entries. This is written as: $k \leq m$.

5) Formulation: To summarize formally, we need to find a partition $\mathcal{P}$ for cover $C = \{c_1, c_2, c_3, ...c_k\}$, such that $f^1 \subseteq C \subseteq f^1 \cup f^{DC}$, with the goals and constraints as follows:

$$\begin{aligned} & \min \left(Variance(\{\sum_{c \in p} weight(c) | p \in \mathcal{P}\}) \right) \\ & \text{s.t.} \quad \forall p_1, p_2 \in \mathcal{P}, p_1 \neq p_2 \Rightarrow \bigcup p_1 \cap \bigcup p_2 = \emptyset, \\ & \qquad k \leq m \text{ and } |\mathcal{P}| = n_i \end{aligned} \tag{P}$$

C. Solution

We propose a heuristic algorithm called Roger to solve the problem. The algorithm is a three-stage approach: it first extracts DSOP form of the Boolean function by constructing BDD-CE (binary decision diagram with complemented edges), then it merges the cube with its closest cube to reduce the number of rules, the last step is to minimizing variance of the weight of the partitions. The Roger algorithm is not an optimal solution, but it solves Problem P in polynomial time complexity.

The Roger algorithm is detailed in 1. Line 1 constructs a BDD-CE for the Boolean function corresponding to the MVT. At Line 2, we extract the minimal sum of products using the algorithm in [37]. After this step, we obtain a cover for the f^1 , but it cannot satisfy either the constraint from the NIC or the goal of balancing the workload. In lines 4-6, we calculate

Algorithm 1: The Roger Algorithm.

```
Input: $T$
/* Constructing a binary decision diagram
   for the Boolean function                  */
1  $BDD_{ce} \leftarrow BDD_{CE}(f_{mvt})$
/* Use the algorithm in [37] to minimize
   the number of products                    */
2  $BDD^{'}_{ce} \leftarrow Minimize\_path(BDD_{ce})$
/* Extract the disjoint sum of products
   from the minimized BDD                    */
3  $S \leftarrow Extract\_DSOP(BDD^{'}_{ce})$
4  Initialize an array $W\_S$ to store the weight of each cube.
5  for $i \in S$ do
6      append $(i, weight(i))$ to $W\_S$
/* Merge to reduce the number of rules  */
7  while $|W_S| \geq m$ do
8      $c_1$ ←min(W$_S$)
9      $c_2$ ←ClosestCube(W$_S, c_1$)
10     $c_s$ ←Supercube(c$_1, c_2$)
11     Delete $c_1$, $c_2$ from $W\_S$
12     Insert $c_s$
13     Update the $W\_S$
14 Initialize an array $C$ to store the components.
15 Initialize an set $\mathcal{P}$ to store the partition.
16 $C = Get\_Component(W\_S)$
17 Using $C$, traverse the combinations of partitions to
     search the solution with the lowest variance.
18 Apply $\mathcal{P}$ to the NIC traffic steering rule.
```

the weight for each cube. In lines 7-14, we merge the cubes to reduce the number of products by eliminating the cube with the lowest weight. By calculating the supercube of two cubes, we can reduce one cube. The iteration ends until the constraint of the NIC is met. Then we traverse all the possible partitions for the cover to: 1. minimize the variance of the sum of the weight of cubes in each element of the partition. 2. ensure that each component is contained in one partition. At the last step, we install the NIC rules according to the solution we have found. As a result, the workloads among instance are well-balanced and no state-sharing exists among instances, which improves the scalability of the system.

Discussion: The goal of the algorithm is to balance the workload among the instances, but the problem could be more complex. Considering the different capabilities of different instances, we can limit how much weight of cubes that one instance can handle. When dealing with **NF**LOW with higher priority hence requiring more processing capability, we assign more weights to this **NF**LOW which leads to a deployment plan where fewer **NF**LOWs have exclusive usage of L1/L2 cache. If users have different requirements, the problem is transformed into different combinatorial optimization problems. For NICs that supports real-time update of the NIC rules, the above methods can be applied periodically to adjust for dynamic traffic similar to RSS++ [38].

V. EVALUATION

Testbed Setup: Our testbed consists of two dual-socket Dell R740 servers. One is used for generating packets using TRex [39], and another is the Device Under Test (DUT) running the NFs. Each server is equipped with Intel Xeon Platinum 8168 @2.7GHz (24 cores) on each NUMA [40] socket, 384GB DRAM and a (DPDK-capable) dual-port Mellanox ConnectX-5 EN 100Gbps NIC. All the experiments are NUMA-aware to avoid NUMA penalties [6]. We connect both the servers back-to-back [41]. On the DUT, we run each NF instance on a dedicated CPU core isolated from kernel scheduler[4]. We evaluate the correctness and performance benefits of **NF**LOW and MVT abstractions.

TABLE I: NFs used in expressiveness evaluation.

Name	Description	State Entry Dependency
NM	Network monitor that counts the number of packets for each source host.	Independent state table
NAT	Translates addresses between internal/external addresses.	Independent state table + shared state
L3FWD	Forwards packets based on layer-3 address using Longest Prefix Match.	Dependent state table
SNORT	Rule-based IDS.	Dependent state table

A. Correctness

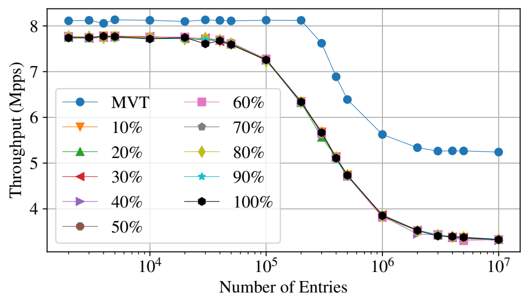
We show that dependent table can result in incorrect packet processing due to wrong state entry accesses in multi-instance NFV systems. We then demonstrate how MVT can avoid it. We implement the example shown in §II and Fig. 1 with DPDK-based L3FWD. We place the "16.0.0.0/24 0" on instance 1 and "16.0.0.0/28 1" on instance 2. We then use the traffic generator to generate flows from $16.0.0.0 - 16.0.0.15$ and send these to this DUT. The traffic steering side do not consider the entry dependency and arbitrarily "sprays" different flows across instances. We can observe that port 0 can receive packets from the traffic generator side. However, according to the LPM mechanism, all packets should match "16.0.0.0/28 1" and send the packets to port 1.

In contrast, our MVT-based L3FWD can ensure the correctness. We construct the MVT from the two rules and identify the finest-granularity **NF**LOW is 16.0.0.0/28, we spawn the 16.0.0.0/24 **NF**LOW to #1 16.0.0.0/28 1, #2 16.0.0.16/28 0, #3 16.0.0.32/27 0, #4 16.0.0.64/26 0, and #5 16.0.0.128/25 0, no dependency among them. The entries are arbitrarily placed on different instances.

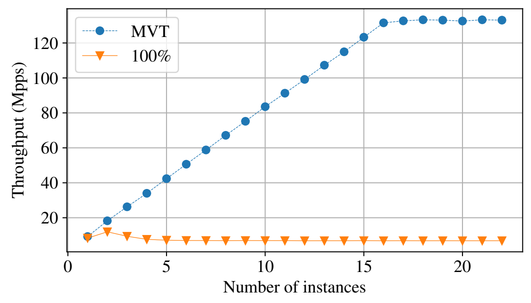
B. Performance

We evaluate the performance of the proposed **NF**LOW and MVT abstractions with the NFs shown in TABLE I on the following grounds: i) the performance improvement of MVT due to the elimination of entry comparison using a single NM instance; ii) the performance improvement of MVT because of the solve of entry dependencies when scaling-out to multiple NM instances; iii) the performance improvement of MVT-based NAT after eliminating the shared resource pool; and iv) the scalability evaluation of the MVT-based NFs.

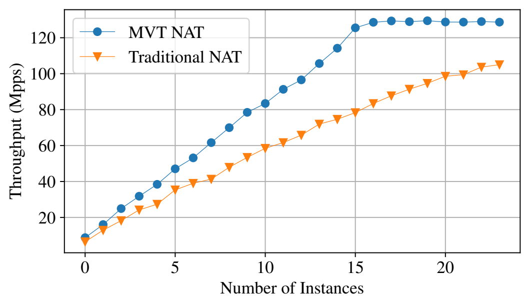
[4]Kernel parameters on the DUT server is set to the following: "isolcpus=1-47 intel_idle.max_cstate=0 processor.max_cstate=0 intel_pstate=disable nohz_full=1-47 rcu_nocbs=1-47 rcu_nocb_poll default_hugepagesz=1G hugepagesz=1G hugepages=256 vt.handoff=1"



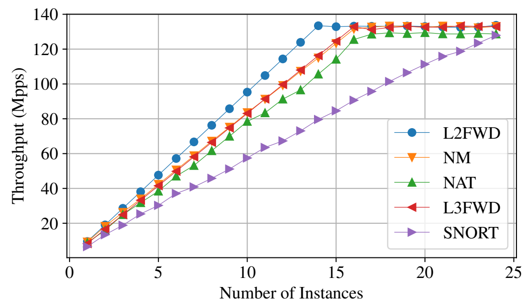
(a) Throughput of different ratio of “entangled” entries in the dependent table-based NM and MVT-based NM.



(b) Throughput of MVT-based NM and all entries are “entangled” (100%) dependent table-based NM.



(c) Throughput of shared resource pool NAT (Traditional NAT) and no shared states NAT (MVT).

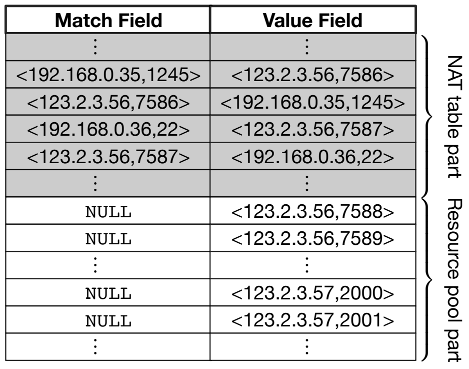


(d) Throughput of MVT-based NFs using different numbers of instances.

Fig. 5: Performance experimental results.

Overhead Due to Priority Comparison: We evaluate dependent table overheads introduced by entry priority comparisons, which are caused when a single packet matches multiple “entangled” entries. We vary the ratio of the number of “entangled” entries (over the total number of entries) in the dependent table and evaluate the throughput under different ratios. We use NM to conduct the experiments due to its simplicity. NM originally does not have dependent entries. We perform the priority comparison operations across the matched table entries to emulate the overheads in locating the entry with highest priority. Fig. 5a depicts the throughput performance comparisons between MVT and dependent-table-based NM using a single instance. We can see that the MVT-based NM outperforms dependent-table-based NM by a margin of 56.6% . We also observe the performance differences between different ratios of “entangled” entries in dependent tables are insignificant. This is because of the entry comparison procedure requires loading all the “entangled” entries to the L1 cache. But due to its limited capacity, loading of these entries leads to L1 cache misses, and the CPU core will have to swap entries with L2. This is the dominating reason behind the degraded performance. As shown in the figure, with less than 50,000 total entries, all the ratio configurations can fetch the entries from the L2 cache, and hence their throughput performances are roughly the same. As the number of entries increases, the number of “entangled” entries also increases, and the L2 and even L3 can not store the entries leading to DRAM-bound memory access, which results in further degrading the throughput performance. We confirm this behavior by using Intel VTune [42], a tool that exposes the L1/L2/L3 cache hits/misses and DRAM accesses. Overall, the “entangled” entries of dependent tables downgrade the performance of NFs, while MVT entries can reach the optimal performance due to the elimination of “entangled” entries.

“Entangled” Entries Correctness Overhead: We then consider the performance overhead by sharing the “entangled” entries of the table to guarantee the correctness of packet processing. Without careful consideration, separating the “entangled” entries and placing them on different instances could result in incorrect packet processing behavior (see §V-A). We now consider an extreme condition where all entries in the dependent table are “entangled”. We use NM NF to illustrate the impact of sharing the dependent state table in memory across all the NF instances. We compare the packet processing performance of this version against the MVT version. The traffic generator generates a workload of 100K synthesized **NF**LOWS at 100 Gbps line rate speed towards the DUT. We generate 64-byte packets to evaluate the packet processing performance. Fig. 5b shows MVT-based NM achieves a linear performance increase while the shared dependent-table-based NM peaks at 11.96Mpps with 2 instances. Further increasing the number of instances degrades the throughput by over 40% before stabilizing the throughput at about 6.83Mpps. This is because updates made by the instances to the NM counters associated with the “entangled” entries lead to gaining exclusive state access (using locks) thus compromising the benefits envisioned to be gained by scaling to multiple instances.



Match Field	Value Field
⋮	⋮
<192.168.0.35,1245>	<123.2.3.56,7586>
<123.2.3.56,7586>	<192.168.0.35,1245>
<192.168.0.36,22>	<123.2.3.56,7587>
<123.2.3.56,7587>	<192.168.0.36,22>
⋮	⋮
NULL	<123.2.3.56,7588>
NULL	<123.2.3.56,7589>
⋮	⋮
NULL	<123.2.3.57,2000>
NULL	<123.2.3.57,2001>
⋮	⋮

NAT table part
Resource pool part

Fig. 6: The MVT of NAT's states. This MVT consolidates the NAT table and the resource pool. The gray shaded part is the NAT table, and the white part is the resource pool.

Eliminating Shared States: We use a bidirectional dynamic Network Address Translation (NAT) to demonstrate how NAT constructs the MVT eliminates the shared states, and fuses the two types of states as a single MVT. The NAT translates the internal source network address[5] to the mapped external source network address. In the reverse direction, it translates the external destination network address to the mapped internal destination network address. When a new internal **NF**LOW is generated (*e.g.*, a new internal host is added), the NAT needs to allocate a new external network address for the translation. Hence, the NAT uses two types of states: the *NAT table*, and the *network address resource pool*.

Each NAT table entry comprises of two parts: a packet predicate (source IP address) and a translated IP address. Each NAT entry is an MVT entry in the reactive method. The **NF**LOW of each NAT entry is defined by the either (direction=inbound, destination network address) or (direction=outbound, source network address), and hence, **NF**LOWs of different NAT entries are disjoint. Therefore, the NAT table is an MVT. The network address resource pool (shorthanded as resource pool) can also be represented by MVT. Conceptually, the resource pool is accessed by all new **NF**LOWs, and thus, existing work [17], [21] shared it across **NF**LOWs. But the semantic of the resource pool is a set of network addresses. Where each element is an unmatched network address and will be dedicated to an **NF**LOW after being matched.

When implementing the MVT on NAT, we pre-generate the network addresses in the resource pool and initialize it as an empty MVT. In the empty MVT the match field is set as `NULL`, and value field is set as a particular network address. We construct the resource pool MVT at the initialization stage of NAT. At this stage, the MVT is a pure resource pool MVT, and each entry is an available network address (resource) to be mapped. When a new **NF**LOW come, the NAT matches the **NF**LOW and hashes it to locate its MVT entry. As the coming **NF**LOW is new, NAT cannot find the entry in the MVT (the match field of the indexed entry is `NULL`). Then, NAT modifies the match field of the entry and makes the entry as a NAT table entry (see Fig. 6). Hence, the MVT can not only represent the resource pool, but also can fuse the two state types as a single MVT. The number of available resources in the resource pool is determined by the user configuration. Using the MVT to represent the resource pool will only pre-generate all the resources that will be used based on the configuration. Hence, pre-generating all resources are reasonable.

[5]We use "network address" to represent the $\langle IP\ Address, Port\ Number\rangle$ pair.

The partitionable MVT eliminates the need for sharing the resource pool state across multiple instances. Next, we evaluate the impact of this elimination of shared state over the performance by using MVT. Fig. 5c shows the performance comparisons between MVT-based NAT and the range-structure-based NAT (traditional NAT) implementations. We can see that the MVT-based NAT is able to outperform traditional NAT by a factor of up to 62.5%. Increasing the number of instances shows a linear performance increase for both versions, with MVT-based NAT achieving line-rate with 16 instances. The performance degradation of the range-based NAT is mainly attributed to the shared resource pool which is completely eliminated by MVT-based NAT.

Scalability Evaluation: Fig. 5 shows the scalability performance of all the four NFs (L2FWD, NM, NAT, L3FWD and SNORT) using MVT. We vary the number of instances and evaluate the overall packet processing throughput. The traffic generator generates 100K synthesized (NF specific) **NF**LOWs at 100 Gbps (*i.e.*, 148Mpps) line rate speed. With the least packet processing overheads, we consider L2FWD as the baseline to represent the hardware platform's performance capacity. We find that the performance of all NFs increase linearly as they are scaled out to more NF instances. Undoubtedly, L2FWD is the fastest to reach line rate with just 14 instances. Next in line are NM, NAT and L3FWD reaching the line rate with 16 instances, while SNORT requiring almost 24 instances. These differences between the NFs are due to different packet processing complexities and state sizes. More in-depth studies on packet processing overheads [6], [34], [43] are out of the scope of this paper.

VI. Conclusion

Performance and scalability are key enablers for large adoption of NFV systems. We motivated and presented two novel abstractions, **NF**LOW and MVT. For each of the abstractions, we provided formal definitions, and discussed methods of constructing and partitioning MVT, and traffic steering based on **NF**LOW for NFV scaling. In order to demonstrate the efficacy of the proposed abstractions by effectively leveraging multi-core servers for NFV scaling, we implemented four common NFs using MVT and conducted extensive experiments. The results show that the performance of NFs implemented using our abstractions can scale with the number of cores, attaining the line speed while ensuring correctness.

Acknowledgement

The research was supported in part by NSF under Grants CNS-1814322, CNS-1831140, CNS-1836772, CNS-1901103, CNS-2106771 and CCF-2123987.C

References

[1] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/merge: System support for elastic execution in virtual middleboxes," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013, pp. 227–240.

[2] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "OpenNF: Enabling innovation in network function control," in *ACM SIGCOMM Computer Communication Review*. ACM, 2014.

[3] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. a. Martins, S. Ratnasamy, L. Rizzo, and S. Shenker, "Rollback-recovery for middleboxes," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. London, United Kingdom: ACM, 2015.

[4] S. Rajagopalan, D. Williams, and H. Jamjoom, "Pico replication: A high availability framework for middleboxes," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM.

[5] J. Khalid and A. Akella, "Correctness and performance for stateful chained network functions," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, 2019, pp. 501–516.

[6] P. Zheng, W. Feng, A. Narayanan, and Z.-L. Zhang, "Nfv performance profiling on multi-core servers," in *2020 IFIP Networking Conference (Networking)*. IEEE, 2020, pp. 91–99.

[7] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 99–110, 2013.

[8] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker, "Modular sdn programming with pyretic," *Technical Reprot of USENIX*, 2013.

[9] F. Németh, M. Chiesa, and G. Rétvári, "Normal forms for match-action programs," in *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, 2019.

[10] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "Nox: towards an operating system for networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, pp. 105–110, 2008.

[11] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow *et al.*, "Onos: towards an open, distributed sdn os," in *Proceedings of the third workshop on Hot topics in software defined networking*, 2014, pp. 1–6.

[12] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker, "Snap: Stateful network-wide abstractions for packet processing," in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016, pp. 29–43.

[13] T. Barbette, G. P. Katsikas, G. Q. Maguire Jr, and D. Kostić, "Rss++ load and state-aware receive side scaling," in *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, 2019, pp. 318–333.

[14] Inlab Networks, "Balance," 2019, https://www.inlab.net/balance/. Accessed: 2019-11-11.

[15] W. Tarreau *et al.*, "Haproxy-the reliable, high-performance tcp/http load balancer," https://www.haproxy.org. Accessed: 2020-07-11.

[16] "squid : Optimising web delivery," https://www.squid-cache.org/, accessed: 2020-07-11.

[17] Linux Community, "Netfilter - Firewalling, NAT, and Packet Mangling for Linux," https://www.netfilter.org. Accessed: 2020-07-11.

[18] Mazu Networks, Inc., "Mazu NAT," https://github.com/kohler/click/blob/master/conf/mazu-nat.click. Accessed: 2020-07-11.

[19] M. Roesch *et al.*, "Snort: Lightweight intrusion detection for networks." in *Lisa*, 1999, pp. 229–238.

[20] gamelinux, "PRADS - Passive Real-time Asset Detection System," http://gamelinux.github.io/prads/. Accessed: 2020-07-11.

[21] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy, "Softnic: A software nic to augment hardware," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-155*, 2015.

[22] J. Hwang, K. K. Ramakrishnan, and T. Wood, "Netvm: High performance and flexible networking using virtualization on commodity platforms," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, 2014, pp. 445–458.

[23] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "Netbricks: Taking the v out of NFV," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, 2016, pp. 203–216.

[24] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker, "E2: a framework for nfv applications," in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015, pp. 121–136.

[25] A. Bremler-Barr, Y. Harchol, and D. Hay, "Openbox: A software-defined framework for developing, deploying, and managing network functions," in *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 2016, p. 511–524.

[26] Y. Zhang, B. Anwer, V. Gopalakrishnan, B. Han, J. Reich, A. Shaikh, and Z.-L. Zhang, "Parabox: Exploiting parallelism for virtual network functions in service chaining," in *Proceedings of the Symposium on SDN Research*, 2017, pp. 143–149.

[27] G. P. Katsikas, T. Barbette, D. Kostić, R. Steinert, and G. Q. M. Jr., "Metron: NFV service chains at the true speed of the underlying hardware," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018, pp. 171–186.

[28] X. Fei, F. Liu, H. Xu, and H. Jin, "Adaptive vnf scaling and flow routing with proactive demand prediction," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 486–494.

[29] X. Fei, F. Liu, H. Jin, and B. Li, "Flexnfv: Flexible network service chaining with dynamic scaling," *IEEE Network*, vol. 34, no. 4, pp. 203–209, 2020.

[30] J. Khalid, A. Gember-Jacobson, R. Michael, A. Abhashkumar, and A. Akella, "Paving the way for NFV: Simplifying middlebox modifications using statealyzr," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016, pp. 239–253.

[31] M. Kablan, A. Alsudais, E. Keller, and F. Le, "Stateless network functions: Breaking the tight coupling of state and processing," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 97–112.

[32] S. Woo, J. Sherry, S. Han, S. Moon, S. Ratnasamy, and S. Shenker, "Elastic scaling of stateful network functions," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018, pp. 299–312.

[33] P. Zheng, A. Narayanan, and Z.-L. Zhang, "A closer look at nfv execution models," in *Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019*, ser. APNet '19. Beijing, China: ACM, 2019, pp. 85–91.

[34] A. Manousis, R. A. Sharma, V. Sekar, and J. Sherry, "Contention-aware performance prediction for virtualized network functions," in *Proc. of SIGCOMM'20*. ACM, 2020, p. 270–282.

[35] H. Yang and S. S. Lam, "Real-time verification of network properties using atomic predicates," *IEEE/ACM Transactions on Networking*, vol. 24, no. 2, pp. 887–900, 2015.

[36] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 213–224.

[37] G. Fey and R. Drechsler, "Minimizing the number of paths in bdds: Theory and algorithm," *IEEE Transactions on Computer-Aided Design of Integrated circuits and systems*, vol. 25, no. 1, pp. 4–11, 2005.

[38] T. Barbette, G. P. Katsikas, G. Q. Maguire Jr, and D. Kostić, "RSS++ load and state-aware receive side scaling," in *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, 2019, pp. 318–333.

[39] The TRex Comunity, "TRex," 2019, https://trex-tgn.cisco.com. Accessed: 2019-10-11.

[40] C. Lameter, "Numa (non-uniform memory access): An overview," *Queue*, vol. 11, no. 7, pp. 40:40–40:51, Jul. 2013.

[41] Wikipedia, "Back-to-back Connection," 2019, https://en.wikipedia.org/wiki/Back-to-back_connection. Accessed: 2020-07-11.

[42] "Intel VTune Profiler," https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html, accessed: 2020-08-11.

[43] C. Sieber, R. Durner, M. Ehm, W. Kellerer, and P. Sharma, "Towards optimal adaptation of nfv packet processing to modern cpu memory architectures," in *Proceedings of the 2nd Workshop on Cloud-Assisted Networking*, 2017, pp. 7–12.