

# FlexMesh: Flexibly Chaining Network Functions on Programmable Data Planes at Runtime

Yu Zhou, Jun Bi, Cheng Zhang, Mingwei Xu, Jinaping Wu  
Institute for Network Sciences and Cyberspace, Tsinghua University  
Department of Computer Science, Tsinghua University

Beijing National Research Center for Information Science and Technology (BNRist)

**Abstract**—Programmable data planes (PDP) enable operators to implement various functions (e.g. routing and access control) on high-performance switches and define the chains of these functions with a *switch profile*. However, with the number of deployed functions increasing, the *switch profile* faces growing complexity during development and inflexibility to chain functions at runtime. This paper presents *FlexMesh*, an integrated platform which aims to introduce flexibility and simplicity to PDP while being compatible with existing programmable devices. *FlexMesh* designs (1) a set of chaining primitives, so operators can easily describe the function chain for each flow without facing the complexity of customizing the *switch profile* during development; and (2) a data plane model that can be reconfigured at runtime and can flexibly construct user-desired function chains. We implement *FlexMesh* based on P4 and evaluate it on various targets. Results indicate that with minor performance overheads, *FlexMesh* can be an efficient development-assistance tool for operators, as well as an automated platform to chain NFs flexibly while keeping conformance to complex policies.

## I. INTRODUCTION

Building high-performance network functions (NF) has long been a vigorous pursuit of operators. In early days, NFs running in proprietary middleboxes experience various problems such as complex management and inflexible chaining. To overcome the drawbacks of middleboxes, virtualization is introduced to grant chaining flexibility and scalability to NFs [1] but comes with performance compromise. The recent progress of programmable data planes (PDP) [2] opens new opportunities for offloading NFs with high performance and programmability. Many research proposals successfully offload a variety of NFs on PDP with remarkable performance benefits [3, 4]. Beyond the scope of traditional NFs, research proposals also explore implementing novel NFs on PDP to improve networked system performance, such as accelerating consensus protocols [5] and scaling distributed systems [6, 7].

To facilitate developing NFs, operators are supplied with the domain-specific languages including P4 [8] and POF [9]. At the configuration time, operators can incorporate multiple NFs into a PDP program and customize a *switch profile*, commonly represented as a Directed Acyclic Graph (DAG), to preset NF chains for different flows [10]. Then, operators deploy the program onto various network devices, such as programmable switches [11] and smart NICs [12], to enforce the NF chains. Furthermore, due to hardware constraints, changing the switch profile has to stop the running switching and brings performance degradation.

With more and more NFs running on PDP, chaining NFs with conformance to operators' policies becomes an essential

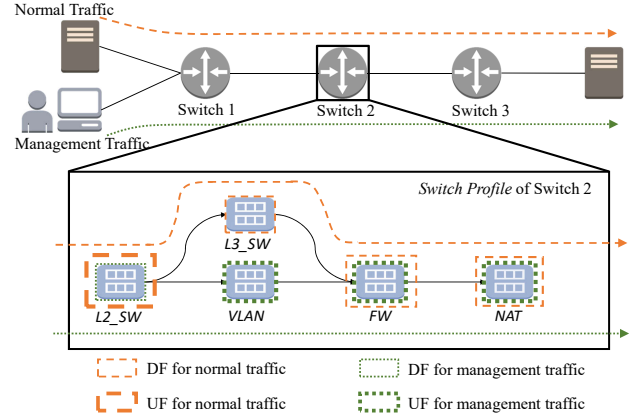


Figure 1. Flows traverse desired functions (DF) and undesired functions (UF) in the fixed switch profile.

but challenging network management task. Firstly, operators should make sure that every flow traverses the right sequence of NFs, just like service function chaining in the context of NF virtualization (NFV) [13]. Otherwise, incorrect NF chains can violate security and degrade the performance of certain NFs [14]. Secondly, a PDP program could comprise multiple NFs (e.g., `switch.p4` [15] has as many as 29 NFs) and policies of different flows require different NF chains [10, 16]. Thus, to satisfy policies of diverse flows, operators need to manually develop a sophisticated *switch profile* defining a satisfactory DAG composed of various NFs to support desired chains, which is cumbersome and time-consuming. Thus, a consolidated and automated platform is well needed for flexibly chaining NFs on PDP but faces the following problems derived from the inflexibility of PDP:

(1) *The rigid switch profile only provides immutable execution sequence of NFs at runtime.* As shown in Figure 1, *Switch 2* is deployed with an example *switch profile* including five functions (desired functions stand for the functions required by the flow policies, vice versa for the undesired functions). Traffic can traverse along the function path defined in the example profile's DAG. However, this example profile also has a restriction that any desired function chain should comply with the topological sequence of the DAG. For instance, *Switch 2* cannot support the chains of  $\{L3\_SW \Rightarrow NAT \Rightarrow FW\}$ , which could be the NF chain for reversed flows of the normal traffic in Figure 1.

(2) *The rigid switch profile is incapable of providing function chains with rigorous conformance to every policy.* The *switch profile* provides fixed function paths in regardless

of the unique demand for function chains from the traffic. However, the fact is that not all functions along the path are desired by the traffic [10], i.e., the function paths provide loose conformance to flow policies. For example, as shown in Figure 1, the normal traffic only requires a chain of  $\{L3\_SW \Rightarrow FW \Rightarrow NAT\}$  for assurance of security, while management traffic only needs  $\{L2\_SW\}$ . However, the example profile of *Switch 2* cannot provide the NF chain without undesired functions. As a result, the management traffic inevitably traverses undesired functions including VLAN and FW. The undesired functions in the function path could potentially lead to policy violation and performance degradation (see §II-A).

In this paper, we are centered on PDP itself and make efforts on solving the chaining inflexibility issue. To this end, we propose a novel platform, *FlexMesh*, which enables flexible NF chaining on PDP. *FlexMesh* introduces a unique data plane model, which logically converts *switch profile* of a PDP program to a full mesh connecting every NF. This model makes the fixed *switch profile* reconfigurable at runtime and dynamically chain NFs while keeping the benefit of high performance. Besides the chaining flexibility improvement, *FlexMesh* introduces a suite of chaining primitives to describe function chains at runtime and to simplify operators' responsibility of constructing NF DAG at configuration time. We made the following contributions in this paper:

- We propose *FlexMesh* to support NF chaining without modifying existing hardware implementations. *FlexMesh* can serve as an efficient development-assistance tool and an automated policy-enforcement platform for the emerging PDP architectures. We present motivations and challenges of *FlexMesh*.
- We design a set of chaining primitives that enable operators to describe flow-level NF chains at runtime.
- We develop a data plane model to enforce on-demand construction of NF chains.
- We devise an algorithm to optimize *FlexMesh* in the worst case of desired NF chains.
- We implement the *FlexMesh* prototype and evaluate the prototype on BMv2 [17], the programmable ASIC, and the SmartNIC [12] regarding performance overheads, performance improvements, and chaining flexibility. We illustrate the practicality and simplicity of *FlexMesh* by building a Fat-Tree topology constructing different desired function chains for diverse flows.

## II. MOTIVATIONS

### A. Motivations

1) *Enhancing chaining flexibility of PDP*: As NFs grow in number and variety, we need a more general and flexible way to chain NFs while keeping compatibility with existing PDP architectures. It seems that operators can use some tricky programming techniques to provide such a flexible composition of NFs during development. For example, operators could use a wide-ternary-match policy table to select and tag the specified flow, then place a predication expression (*if-else* statement) before the NF as a gatekeeper to determine

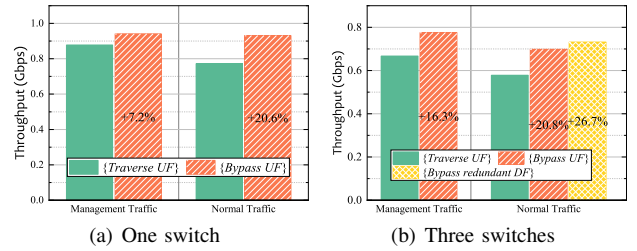


Figure 2. Improve performance of NF chains through bypassing UFs and redundant DFs.

whether the function should be executed or not. However, this technique cannot tackle the problem systematically. Because both the tagging logic and predication expression are parts of the *switch profile* and are fixed after deployment. Different flows may utilize different predication logic, while operators cannot pre-plan all possible cases into the *switch profile*. Intuitively, like middleboxes which use the physical connectivity to construct NF chains, native PDP utilizes the hard-coded DAG to provision NF chains and lacks essential flexibility.

In *FlexMesh*, we design innovative techniques to provide chaining flexibility with no modification to the PDP implementation. *FlexMesh* uses match-action tables (MAT) and metadata to equivalently express the predication logic, so the originally fixed predication logic in *switch profile* can be dynamically configured without compromising the expressiveness or flexibility. Furthermore, we utilize the recirculation mechanism cooperated with a dedicated state transition table to support arbitrary chains in the NF mesh. Similar to NFV, *FlexMesh* realizes para-virtualization of chaining-related logic on PDP through above techniques.

On-demand construction of NF chains can enable bypassing UFs and redundant DFs to achieve performance improvement. To prove this claim, we implement the example shown in Figure 1 on BMv2 [17] and compare the throughput in terms of bypassing and traversing UFs ( $\{Traverse UF\}$  and  $\{Bypass UF\}$  in the figure). As shown in Figure 2(a), flows that bypass UFs acquire a maximum performance increase by 20.6%. Similarly, in Figure 2(b), we measure the throughput based on a linear topology of three switches. Moreover, in the test of  $\{Bypass redundant DF\}$ , we also explore the possible performance degradation caused by redundantly executing the same NF across multiple switches. Not all functions need to be repeatedly executed along the forwarding path. Some unique functions, such as FW, monitor and heavy hitter detector, can be invoked just once or few times across the whole path. By executing FW once instead of three times, the normal traffic acquires another 5.9% increase in throughput.

2) *Simplifying development of PDP programs*: Currently, developing data plane programs is more like the early-binding method in object-oriented programming. If viewing the *switch profile* as an object and the DAG as the definition of the object, we can observe that the current method requires operators to declare a *switch profile* with a deterministic definition for all traffic during development without enough high-level policy information. However, the knowledge of which flow requires which functions is usually acquired as part of the high-level

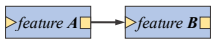

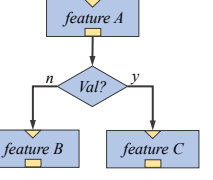
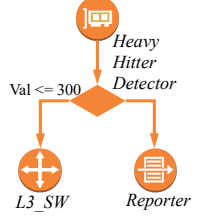
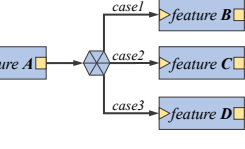
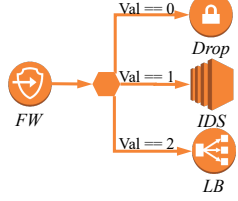
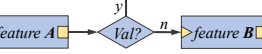
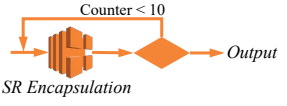
Type	Control Flow Chart	Syntax	Use Case
Sequence		<b>BNF:</b> sequence_declaration ::= feature_name => feature_name  <b>Note:</b> ✓ feature_name denotes the name of deployed features. <b>Example:</b> A => B	 <b>Command:</b> L2_SW => FW <b>Description:</b> The feature of Firewall follows the feature of L2_SW. Sequence structure is the most widely adopted logic.
Selection	<b>if-else:</b> 	<b>BNF:</b> if_else_declaration ::= feature_name => (feature_name : feature_name ? validator_declaration) validator_declaration ::= metadata_declaration op constant metadata_declaration ::= feature_name.global_metadata op ::= >   >=   ==   <=   <   !=  <b>Note:</b> ✓ The left value of the validator is the global metadata field assigned for each packet passing among features. <b>Example:</b> A => (C : B ? A.global_metadata == 1)	 <b>Command:</b> HITTER => (L3_SW : REPORTER ? HEHITTER.global_metadata <= 300) <b>Description:</b> The feature of Heavy-flow Hitter identifies the heavy flow and sends the flow to a Reporter, otherwise to L3_SW.
	<b>multibranch:</b> 	<b>BNF:</b> multibranch_declaration ::= feature_name => {(case_declaration :} case_declaration # metadata_declaration) case_declaration ::= const @ feature_name  <b>Note:</b> ✓ The const number in case_declaration can be set dynamically at runtime. <b>Example:</b> A => (n1@B : n2@C : n3@D # A.global_metadata)	 <b>Command:</b> FW => (0@DROP:1@IDS:2@LB # FW.global_md) <b>Description:</b> FW drops the packet; sends the packet to an IDS; sends the packet to a LB.
Loop		<b>BNF:</b> loop_declaration ::= feature_name => (self : feature_name ? validator_declaration) validator_declaration ::= metadata_declaration op constant metadata_declaration ::= feature_name.global_counter  <b>Note:</b> The orchestrator compiler guarantees the number of loops is determinate. The constant number can be set at runtime. <b>Example:</b> A => (self : B ? A.global_counter > 0)	 <b>Command:</b> SEGMENT_ROUTING_ENCAP=>(SEGMENT_ROUTING_ENCAP : Output ? SEG_ROUTING_ENCAP.global_counter < 10) <b>Description:</b> By the loop structure, operators can readily implement the feature with one time of encapsulation, and dynamically set the counter for each packet to precisely control the number of times for header encapsulation/de-capsulation. The SR Encapsulation will execute encapsulation for 10 times, then output the packet.

Figure 3. Illustration of chaining primitives.

policy information at runtime and cannot be entirely predicted or planned during development.

*FlexMesh* eliminates this contradiction by delaying the construction of the *switch profile* with the policy information to runtime, which is analogous to the late-binding method and changes the way of developing and managing PDP programs. At the development stage, *FlexMesh* frees operators of manually developing complex *switch profiles* and enables them to focus on core logic of NFs, i.e., operators only need to give *FlexMesh* a half-completed PDP program (i.e., a P4 program only with headers, parsers, and NFs composed of MATs). At runtime, operators can use intuitive primitives to describe function chain policies for each flow. Then the orchestrator in *FlexMesh* compiles the descriptions to the data plane model dynamically, i.e., *FlexMesh* auto-generates the control flow of the PDP program that guarantees compliance with given policies at runtime. Consequently, *FlexMesh* eliminates the complexity of developing PDP programs and reduces oper-

ational expenditures for directly manipulating lots of match rules in MATs at runtime.

### III. CHAINING PRIMITIVES

*FlexMesh* simplifies developing PDP programs and provisions a suite of high-level chaining primitives allowing operators to specify function chains conveniently. At the development stage, operators can create a data plane program by either importing NFs provided in *FlexMesh* default function library or developing new functions from scratch. At runtime, operators can flexibly define the desired function chains for each flow through the chaining primitives.

As shown in Figure 3, the syntax of the chaining primitives, specified with the Backus Normal Form (BNF), is rather easy to understand and use. Each statement of a primitive defines the relationship between two functions, and operators can customize the whole function chains by describing the relationships of multiple function pairs. After acquiring the

descriptions, the *FlexMesh* orchestrator assembles the relationships of function pairs, translates the function chains into data plane rules, and dynamically configures the data plane as operators required.

### A. Control Structure

*FlexMesh* provides three types of control structures for describing NF chains including the sequence structure, the selection structure, and the loop structure.

1) *Sequence structure*: The sequence structure determinately connects two functions. This structure requires that the processing sequence between two NFs should be deterministic regardless of the results from the predecessors. As shown in Figure 3, packets should sequentially traverse FW after being processed by L2\_SW. Most service function chaining frameworks in NFV only support the sequence structure [18]. However, chaining NFs on PDP requires more complex logic between NFs. Thus we introduce following structures.

2) *Selection structure*: *FlexMesh* provides two selection structures: *if-else* and *multi-branching*, whose syntax is similar to the ternary operator in C programming language. The selection structure can dynamically change processing behaviors at runtime. With the selection structure, operators can flexibly compose various networks functions, such as the *stateful firewall* and the *heavy hitter detector* [19].

3) *Loop structure*: The loop structure allows operators to repeatedly execute a NF according to the conditional expression. The runtime orchestrator guarantees that the loop structure is correctly invoked, and the number of loops should be deterministic at runtime. As shown in Figure 3, when implementing the Segment Routing [20] in a regular data plane program, it is hardly possible for operators to precisely predict the number of the header encapsulation/decapsulation at the development stage. In the current implementation, operators usually estimate the most likely maximum numbers for the encapsulation/decapsulation and statically program the corresponding actions in the MATs. With the loop structure, operators merely need to program the function and set the counter for each flow at runtime to make PDP correctly process the packets.

### B. Parameter Passing Mechanism

Apart from chaining NFs, *FlexMesh* also specifies how to pass parameters among NFs. *FlexMesh* allocates two parameters to each packet. One is *global\_metadata* used for the selection structure. The other is *global\_counter* utilized for the loop structure. Both *global\_metadata* and *global\_counter* are valid through the life of the packet and can be modified by particular NFs. Taking the heavy hitter detector in Figure 3 as an example, operators can dynamically set the threshold value at the time of describing the chain. Flows with *global\_metadata* above 300 will be forwarded to the reporter to upload the flow information to the control plane. The logic for setting the value of *global\_metadata* is implemented inside the heavy hitter detector and is correctly maintained.

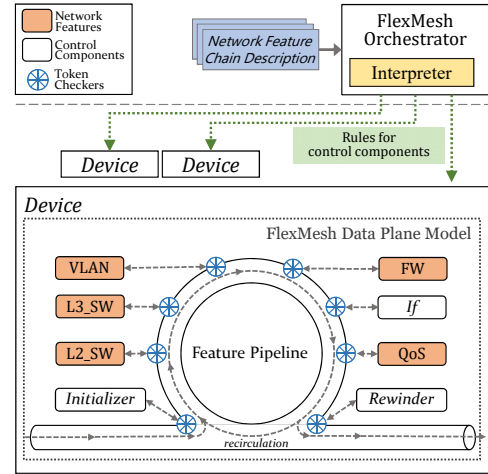


Figure 4. Architecture overview of *FlexMesh*.

## IV. FLEXMESH DATA PLANE MODEL

To overcome the chaining inflexibility issue of PDP, *FlexMesh* provides a general data plane model implemented by the dedicated *switch profile* to operators, which logically presents a NF mesh abstraction. The data plane model proposed in this section is not constrained by the underlying PDP architectures, such as RMT [2], dRMT [21], PISA [22], and Domino [23]. To be concise, we use RMT as the underlying PDP to support *FlexMesh* in this paper.

*FlexMesh* flexibly chains NFs in an on-demand way for various flows through following steps. (1) At the configuration stage, *FlexMesh* can be deployed with dedicated *switch profile* and all the registered NFs onto the data plane. (2) At runtime, the function orchestrator in the control plane can compile the NF chain description into rules of the control components in the *FlexMesh* data plane model. (3) Then, the data plane model could construct NF chains accordingly.

As shown in Figure 4, *FlexMesh* organizes all registered NFs into a ring-like *default function pipeline* with the dedicated *switch profile*. At the start of the pipeline, *FlexMesh* places an *initializer* to assign each packet a *token* according to the rules from the orchestrator. The *token* uses a bitmap to identify which NFs should be executed in one round of pipeline traversing. Then, while the packet is traversing the pipeline, the *token checker*, standing before each NF, checks the *token* to guarantee the packet is only processed by desired functions.

The *default function pipeline* may not fit for all desired function chains due to the immutable execution sequence problem. Thus, we design a *rewinder*, residing at the end of the pipeline, to recirculate packets for multiple rounds of pipeline traversing (*resubmit* action is used). For example, operators could use the *loop primitive* to implement multiple-header decapsulation. Accordingly, the orchestrator will arrange the packet to be decapsulated by the NF through multiple-round traversing. Although coming with performance compromise, *FlexMesh* can provide the *chaining flexibility equivalent to NFV* with the *rewinder*.

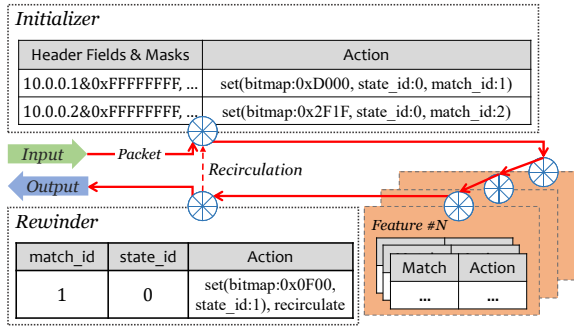


Figure 5. Packet-processing procedure in FlexMesh.

### A. Finite State Machine

FlexMesh takes the user-specified chains as the input and calculates a Finite State Machine (FSM) based on the existing *default function pipeline*. Each state in FSM identifies that the flow should be processed by a set of particular functions in a round of traversing. In case of single-round traversing, desired function chain can be satisfied by one state without recirculation. In case of multiple-round traversing, traffic will transit from one state to the next until completing the desired chain. The state transition table in the *rewinder* can be configured at runtime by the orchestrator.

### B. Control Components

1) *Token and token checker*: The *token* is a metadata allocated by FlexMesh to each packet and occupies 4 bytes, about 1% of the total memory space of RMT. It contains three parts. The first one is a *bitmap* that identifies which NFs should be invoked. The second one is a *state\_id* representing the current state in the FSM. So FlexMesh can utilize this *state\_id* to complete the desired function chain according to the state transition table. The last one is a *match\_id* which is used as a unified identifier of flows in all control components except for the *initializer*. Compared with matching various header fields, the *match\_id* could cut down resource usages of match fields.

The *token checker* verifies the bitmap in the *token* to determine whether the packet should apply the corresponding NF, which keeps rigorous conformance to chaining policies through bypassing UFs. To make the checking procedure lightweight, FlexMesh implements the *token checker* by using the conditional expression with the bitwise-and operator. Then, the *token checker* can be compiled into hard-coded logic and leads to small performance overheads.

2) *Initializer*: The *initializer* is responsible of specifying NF chains for every parsed packet, which is implemented by one MAT shown in Figure 5. The MAT assigns every ingress packet a *token*, whose value can be dynamically configured by the orchestrator. Packets should only be processed by the *initializer* for one time, so FlexMesh uses a flag stored in the metadata as an identifier for skipping the *initializer* in the case of multiple-round traversing.

3) *Rewinder*: The *rewinder* recirculates packets whose chains need multiple-round traversing for the purpose of overcoming immutable execution sequence problem. As shown in Figure 5, the *rewinder* contains one MAT that matches the *match\_id* and *state\_id*. The *rewinder* will (1) set the *state\_id* and *match\_id* for the next transition state, then using the *resubmit* action to recirculate the packet; or (2) output the packet if the FSM ends. The *rewinder* improves the chaining flexibility but introduces a performance overhead. To mitigate the performance overhead of the *rewinder*, we develop a default function pipeline construction algorithm in §V to minimize the number of recirculation for a set of chaining policies. Furthermore, operators could employ FlexMesh to flexibly chain NFs across devices to achieve the same effect of the construction algorithm.

4) *Other control components*: In addition to above components, FlexMesh also provides some powerful components for advantages of both rich expressiveness and dynamic reconfigurability at the same time. These control components including *if-else*, *multi-branching*, and *loop*, are implemented by MATs and intrinsic metadata to express the corresponding chaining logic. FlexMesh can place one or more control components in the *default function pipeline* and combines NFs with control components to implement complex chain policies.

## V. FUNCTION PIPELINE CONSTRUCTION

As aforementioned, when the *default function pipeline* is unable to satisfy the desired execution sequence of NFs in a chain, the orchestrator can utilize the *rewinder* with the help of an FSM to complete the NF chain through recirculating packets. Thus, constructing a *default function pipeline*, which is general enough to satisfy possible desired chains, becomes a challenge. To this end, FlexMesh designs a novel algorithm for constructing an optimal *default function pipeline* based on the known data sets of desired chains to avoid multiple traversing maximally.

As shown in Figure 6, the pipeline construction algorithm contains three steps. The first step involves decoupling and sorting. This step collects as many operator-desired function chains as possible and decouples these chains into pairs of NFs. Then, it sorts pairs of NFs based on the dependency of protocol layers. The second step uses a mature algorithm, minimal acyclic finite-state automata (MA-FSA) [24], to generate a minimal DAG of NFs. MA-FSA is an algorithm used for constructing minimal, deterministic, acyclic finite-state automata from a set of words, by which a minimal DAG of alphabets can be built to guarantee that all words can be found in the DAG. We can use the standard MA-FSA to construct the minimal DAG of NFs where all function pairs can be found in this DAG. Notably, this minimal DAG fits all pairs of functions with the minimal number of redundant functions.

The third step uses the topology sorting algorithm to serialize the minimal DAG and construct the *default function pipeline* for FlexMesh. In this way, the control plane can periodically run the algorithm based on the policy updates and

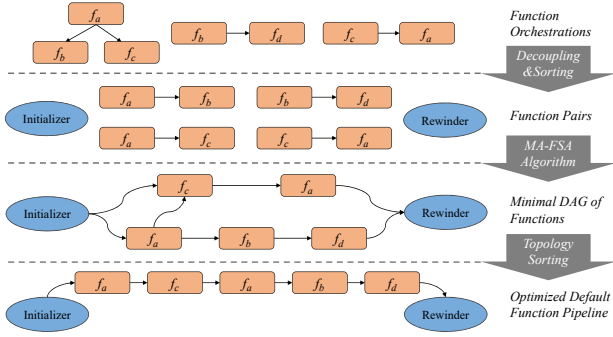


Figure 6. Pipeline construction algorithm.

deploy the optimized *default function pipeline* to maximally avoid recirculation. By the optimized construction algorithm, *FlexMesh* constructs the *default function pipeline* based on the perceived knowledge and makes a trade-off between flexibility and performance.

## VI. EVALUATION

To evaluate *FlexMesh*, we have conducted various experiments on one software target (BMv2) and two hardware targets (SmartNIC and ASIC). Our evaluation comprises three aspects. (1) §VI-A shows that the performance improvements brought by *FlexMesh* are target-dependent. For SmartNIC, *FlexMesh* brings remarkable *performance improvements* via bypassing UFs. (2) To manifest the *flexibility and simplicity* of *FlexMesh*, §VI-B employs the chaining primitives of *FlexMesh* to construct function chains for various flows in the Fat-Tree testbed. (3) §VI-C demonstrates and analyzes *performance overheads* of *FlexMesh*.

**Setup:** We run BMv2 and SmartNIC on DELL R730xd servers which are equipped with  $2 \times 6$  Intel Xeon E5-2620 cores and 64G RAM. The SmartNIC target has two 10 Gbps ports and the NFP-4000 chip [12]. The other hardware target is the programmable ASIC-based switch which can be programmed by P4 and equipped with  $32 \times 100$  Gbps ports. We use MoonGen [25] to test *FlexMesh* on BMv2 and SmartNIC by 10 Gbps traffic. We use the Spirent Packet Generator [26] to test *FlexMesh* on ASIC.

### A. Overall Performance Improvement

In this section, we will evaluate how much performance improvement *FlexMesh* can achieve on three targets. Instead of inserting UFs into the function pipeline, we use undesired MATs (UTs) to simulate the performance costs caused by traversing UFs. As different functions have different MAT compositions, using UTs can ensure better generality and produce more fine-grained results.

In the experiments, all UTs uniformly match five-tuples and apply the default action to the traffic. The *native baseline* is implemented as L3\_SW appended by a varied number of UTs. The traffic in the *native baseline* cannot match any MAT entry in UTs and will execute the default action. Comparatively, in *FlexMesh*, the traffic bypasses all UTs under the control of the control components.

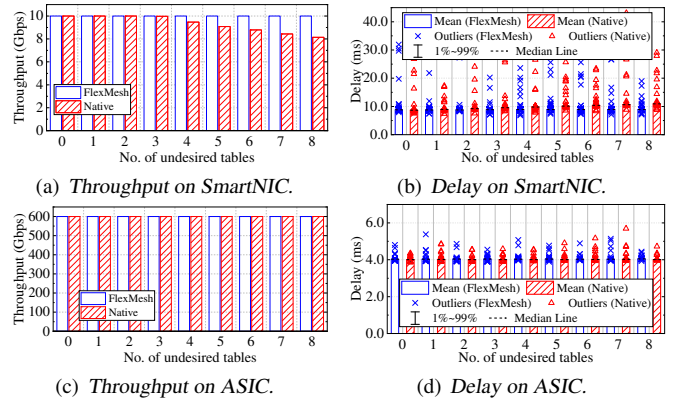


Figure 7. Performance improvements on the single hardware target.

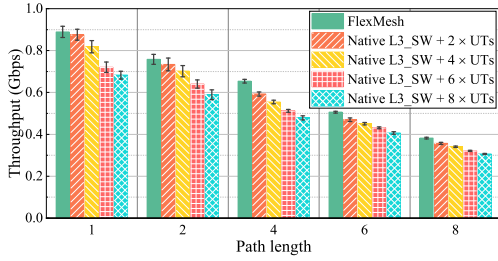
We employ two experiments to evaluate performance improvements. (1) *Performance improvements on a single hardware device:* We measure the throughput and delay on two hardware targets in terms of bypassing different numbers of UTs. (2) *Cumulative performance improvements across multiple devices:* To further understand how performance improvement accumulates across multiple devices, we evaluate the network performance when traffic traverses the paths with varied lengths. At each hop along the path, the device is deployed with different numbers of UTs.

1) *Improvement on a single device:* The performance improvement brought by bypassing UTs largely depends on the implementation of programmable devices. For SmartNIC, Figure 7(a) and Figure 7(b) show that bypassing UTs can provide an obvious performance improvement. When there are only 8 UTs, SmartNIC has a throughput penalty of 1.9 Gbps (22.7%) and a delay increase of over  $2 \mu\text{s}$  (19.3%). The performance degradation will aggravate when the undesired functions become more complicated. Comparatively, *FlexMesh* enables traffic flows to bypass UTs and is immune to the additional processing costs of UTs. For ASIC, as shown in Figure 7(c) and 7(d), bypassing UTs do not bring any performance improvement.

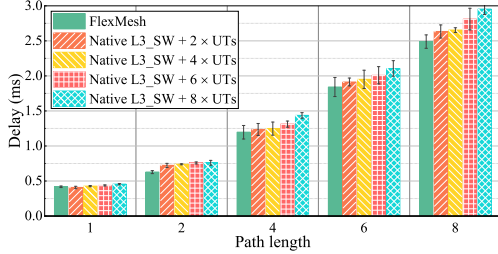
2) *Improvement across multiple devices:* To quantitatively evaluate cumulative performance improvements along the forwarding path, we build a software testbed in which traffic can traverse forwarding paths with different lengths. At each hop, the BMv2 switch is configured with L3\_SW appended with different numbers of UTs. Figure 8 shows that *FlexMesh* effectively improves the throughput and decreases the delay, compared with the *native baseline*, and surpasses the overheads of *FlexMesh* itself. Besides, the absolute value of delay reduction compared with the *native baseline* increases as the path length grows, which demonstrates that *FlexMesh* brings cumulative improvements. For a large-scale network with quantities of programmable switches, *FlexMesh* can provide a promising performance improvement.

### B. Flexibility and Simplicity

To demonstrate flexibility and simplicity of *FlexMesh*, we build a Fat-Tree (K=4) testbed. In the testbed, the software



(a) Throughput on software testbed.



(b) Delay on software testbed.

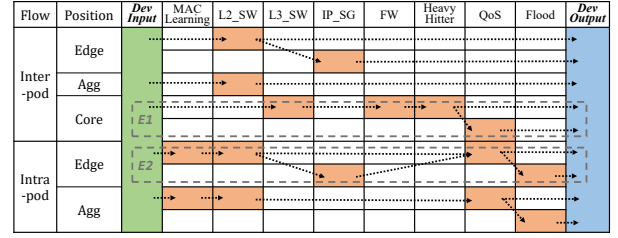
Figure 8. Performance improvements across multiple devices.

switches configured with *FlexMesh* running on the same server and are connected by virtual links to constitute the topology. Packet generators and sinks run on a separated server which connects to the server (the edge switches in the Fat-Tree testbed) by eight 1G cables.

We categorize traffic flows as inter-pod flows and intra-pod flows in the testbed. As shown in Figure 9(a), we design different policies to these two categories of flows at different positions of the Fat-Tree testbed. For example, inter-pod flows apply desired function chains differently at edge switches, aggregation switches, and core switches. Moreover, Figure 9(b) illustrates two examples of describing the complex function chains for flows by using the chaining primitives designed by *FlexMesh*. To the best of our knowledge, no existing control application can dynamically map such complex configurations onto PDP programs while only requiring a few lines of descriptions. For native PDP, operators have to sophisticatedly organize these NFs in the *switch profile* to satisfy these policies at the development stage, but *FlexMesh* could simplify operators' responsibility of coordinating NFs and making chaining NFs much easier.

We use two counterparts in this experiment. (1) For the *ideal baseline*, we develop dedicated PDP programs for each switch and let each switch statically enforce the designated policies for particular flows. The *ideal baseline* can produce optimal performance while statically satisfying the policies in Figure 9(a). (2) Besides comparing *FlexMesh* with the *ideal baseline*, we also conduct experiments on MPVisor which pursues the dynamic reconfigurability of PDP but introduces significant performance and resource overheads. The detailed comparison between *FlexMesh* and MPVisor is listed in §VIII. Through comparing *FlexMesh* with the *ideal baseline* and MPVisor, it shows that *FlexMesh* makes a good trade-off between the performance and the chaining flexibility.

The experiments conducted on the Fat-Tree testbed reveal



(a) Policies of the edge switches, aggregation switches and core switches in the Fat-Tree testbed.

E1: A policy example for inter-pod flows

```

Inter-pod Flows in Core Switches {
  Input => L3_SW;
  L3_SW => FW;
  FW => Heavy Hitter;
  Heavy Hitter => (Output : QoS ?
    hitter_value < 100);
  QoS => Output;
}

```

E2: A policy example for intra-pod flows

```

Intra-pod Flows in Edge Switches {
  Input => MAC Learning;
  MAC Learning => L2_SW;
  L2_SW => (p1@IP_SG : p2@QoS :
    p3@QoS # input_port);
  IP_SG => QoS;
  QoS => (Output : Flood ?
    flood_flag == 0);
  Flood => Output;
}

```

(b) Policy description examples for E1 and E2.

Figure 9. Policies and description examples for the Fat-Tree testbed.

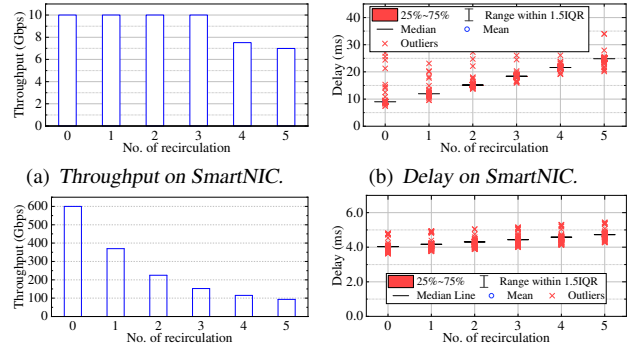


Figure 10. Performance overhead of the rewinder

the flexibility of *FlexMesh* through following two dimensions. (1) For the same position, *FlexMesh* can provide flow-specific function chains to enforce various policies without any conflicts. For example, *FlexMesh* could provide different function chains for inter-pod flows and intra-pod flows to satisfy different requirements at the edge switch. (2) For the same flow, *FlexMesh* can provide different function chains on distinct devices in the network. For example, inter-pod flows are forwarded by L2\_SW in the pod, and the core switches use L3\_SW to deliver the inter-pod flows.

### C. Performance Overheads in FlexMesh

We implement  $\{L3\_SW \Rightarrow FW\}$  on PDP without any other redundant MAT as the *ideal baseline* in this experiment. Then we compare the performance of *FlexMesh* enforcing  $\{L3\_SW \Rightarrow FW\}$  with the *ideal baseline*.

1) *Performance overhead of the rewinder*: As for the *rewinder*, we evaluate the performance overhead with different numbers of recirculation in an extreme scenario where all packets need recirculation (except the baseline). As is shown in Figure 10(a) and 10(b), SmartNIC shows a moderate performance degradation. Throughput starts to decrease from

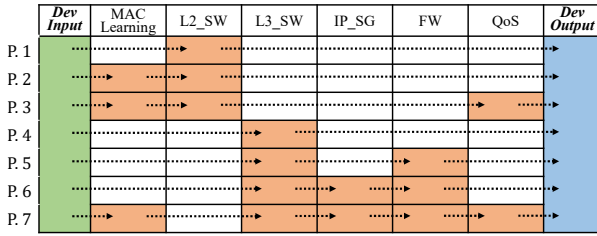


Figure 11. Policies. *IP\_SG* stands for IP source guard.

the four times of recirculation, while delay increases with the number of recirculation. Thus, SmartNIC can keep 10 Gbps throughput if the recirculation times are lower than 4.

For ASIC in Figure 10(c) and 10(d), the throughput degrades proportionally, while the processing delay shows a steady increase at about 130 nanoseconds for one time of recirculation. The *resubmit* action will cause the same packet being processed multiple times on the same port, and naturally divides the total bandwidth of a 100G port by the number of recirculation. If the recirculated packets could be processed on a separate port, the results may become better.

Based on the above analysis, the recirculation can cause performance overhead when multiple traversing occurs. Thus, *FlexMesh* proposes an algorithm, mentioned in §V, to construct the *default function pipeline* that can maximally avoid multiple times of recirculation. For traffic that is optimized by the algorithm, the evaluation can be referred to the results of no recirculation in Figure 10. For traffic that is inevitably recirculated, the performance also can be referred to above tests. Similar results will not be redundantly presented. Network policies update constantly. So the optimized algorithm cannot entirely avoid recirculation based on the existing limited training knowledge. Instead, operators can periodically run the optimization algorithm and update the *default function pipeline* to maximally avoid recirculation.

2) *Performance overhead with different policies*: Apart from above micro-benchmarks on individual control components, we also evaluate the performance overhead of *FlexMesh* as a consolidated system with various policies. Figure 11 shows the policies and their desired function chains, e.g., P.6 requires  $\{L3\_SW \Rightarrow IP\_SG \Rightarrow FW\}$ . For each policy, we deploy the same *default function pipeline* containing all six NFs onto the *FlexMesh* data plane model, chain NFs accordingly, and measure the throughput and delay respectively. We implement the NFs required by the policies for each case and use the corresponding results as the *ideal baseline*.

In Figure 12(a), there is no obvious performance overhead in throughput for most policies except for P.7. For P.1, P.2, P.3, and P.4 in Figure 12(b), the relative increase in delay is moderate while the absolute delay increase is trivial. The NF chains required by policies are simple, which inevitably leads to highlighting the performance overheads caused by *FlexMesh*. For ASIC, there is almost no throughput overhead, compared with the *ideal baseline*. Meanwhile, the delay increases a few tens of nanoseconds across all policies.

Overall, *FlexMesh* incurs a minor performance overhead

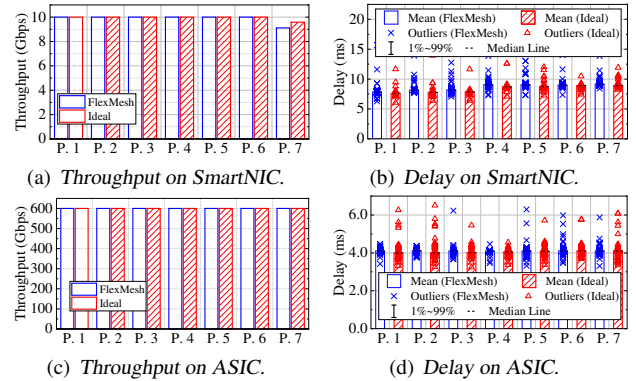


Figure 12. Performance overhead when supporting various policies.

to achieve on-demand construction of NF chains on PDP. In the next section, we will illustrate that the flexibility brought by *FlexMesh* can also improve the packet-processing performance of specific targets.

## VII. DISCUSSION

**Performance improvements on hardware.** We admit that the claimed performance improvements vary from the hardware implementation. For run-to-completion-based (RTC-based) hardware like dRMT [21], SmartNIC, and BMv2, performance improvements of bypassing undesired tables are remarkable, since these targets do not need to allocate processing cycles for the undesired tables. However, for RMT-based hardware like [11], bypassing undesired tables shows no performance improvement, because the physical stage of the pipeline is statically pre-allocated by the target-dependent compiler and will consume the ASIC processing cycles irrespective of whether the table is enforced or not. Thus, even if a table is logically bypassed, the traffic will still consume the same processing cycles in the pipeline, which is different from the RTC-based implementation.

**Network-wide function orchestration and management.** Besides chaining NFs, the NF orchestration on programmable data planes involves NF placement, NF scaling, and fast NF failover, which are also important and interesting topics. Although we only concentrate on the chaining flexibility and simplicity in this paper, *FlexMesh* can be an ideal platform to achieve goals of network-wide function orchestration. Besides, for NF management, programmable data planes expose new challenges, such as program-dependent APIs which make it hard for operators to develop general applications controlling PDP programs. Meanwhile, heterogeneity caused by enhanced data plane programmability also increases the difficulty of managing different PDP programs in a network. Thus, another gap which *FlexMesh* plans to fill is to help operators express network-wide packet processing intents.

## VIII. RELATED WORK

MPVisor [27], Hyper4 [28], and HyperV [29] are recently proposed hypervisors for P4. They are devoted to full virtualization techniques and virtualize most programmable elements including the parser, the ingress/egress pipeline, etc. However,



they suffer from remarkable performance overhead caused by the full virtualization. Comparatively, *FlexMesh* adopts the technique of virtualization in a more lightweight way. *FlexMesh* merely virtualizes the control flow between non-virtualized NFs so that the control flow can be configured at runtime to gain chaining flexibility and makes the performance overhead acceptable. *FlexMesh* represents a promising way to improve simplicity in developing PDP programs and enhance flexibility in running PDP programs.

In NFV, operators can flexibly compose service chains of Virtual Network Functions (VNFs) [30], which is similar to the on-demand chaining in *FlexMesh*. However, due to the various constraints of the hardware implementation, such as limited programmability, implementing NF chains in hardware devices is more challenging than software. Moreover, VNFs suffer from the lower performance of software when comparing with the programmable hardware devices.

## IX. CONCLUSION

In this paper, we propose *FlexMesh* to provide on-demand construction of NF chains on PDP. We provide a suite of chaining primitives for operators to describe the desired function chain for each flow. Moreover, we devise a novel data plane model that is reconfigurable at runtime and can enforce user-specified function chains flexibly. Besides, we present an algorithm to optimize the default function pipeline and to minimize performance overheads incurred by the data plane model. Through above techniques, *FlexMesh* serves as an efficient tool to facilitate PDP program development as well as an automated platform to flexibly chain NFs with rigorous conformance to complex policies.

## ACKNOWLEDGEMENT

We thank all anonymous reviewers for their constructive comments. We thank Chen Sun, Zhilong Zheng, Heng Yu, Yunsenxiao Lin, Yiran Zhang, and Zili Meng, Yimin Jiang, Weibin Meng, and Ya Su for their important suggestions on this work. Prof. Mingwei Xu is the corresponding author. This research is supported by National Key R&D Program of China (2017YFB0801701) and the National Science Foundation of China (No. 61872426, No. 61625203, and No. 61832013).

## REFERENCES

- [1] ETSI. Network functions virtualization. Website. <http://www.etsi.org/technologies-clusters/technologies/nfv>.
- [2] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *Proceedings of SIGCOMM*, pages 99–110, 2013.
- [3] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of SIGCOMM*, 2017.
- [4] Jiamin Cao, Ying Liu, Yu Zhou, Chen Sun, Yangyang Wang, and Jun Bi. Cofilter: A high-performance switch-accelerated stateful packet filter for bare-metal servers. In *Proceedings of ICCCN*, pages 1–9, 2019.
- [5] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just say no to paxos overhead: Replacing consensus with network ordering. In *Proceedings of OSDI*, pages 467–483, 2016.
- [6] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *Proceedings of NSDI*, 2018.
- [7] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM CCR*, 44(3):87–95, July 2014.
- [8] Haoyu Song. Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane. In *Proceedings of HotSDN*, pages 127–132, 2013.
- [9] Anubhavnidhi Abhashkumar, Jeongkeun Lee, Jean Tourrilhes, Sujata Banerjee, Wenfei Wu, Joon-Myung Kang, and Aditya Akella. P5: Policy-driven optimization of p4 pipeline. In *Proceedings of SOSR*, pages 136–142, 2017.
- [10] Barefoot Networks. Barefoot tofino. Website. <https://barefootnetworks.com/technology/>.
- [11] Netronome Company. Agilio cx smartnics. Website. <https://www.netronome.com/products/agilio-cx/>.
- [12] Dilip A. Joseph, Arsalan Tavakoli, and Ion Stoica. A policy-aware switching layer for data centers. In *Proceedings of SIGCOMM*, pages 51–62, 2008.
- [13] Brendan Tschaen, Ying Zhang, Theo Benson, Sujata Banerjee, Jeongkeun Lee, and Joon-Myung Kang. Sfc-checker: Checking the correct forwarding behavior of service function chaining. In *Proceedings of SDN-NFV*, pages 134–140, 2016.
- [14] P4 Language Consortium. P4 switch. Website. <https://github.com/p4lang/switch>.
- [15] W Liu, H Li, O Huang, M Boucadair, N Leymann, Z Cao, Q Sun, and C Pham. Service function chaining (sfc) general use cases. *Work in progress*, 2014.
- [16] P4 Language Consortium. P4-bmv2. Website. <https://github.com/p4lang/behavioral-model>.
- [17] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: A framework for nfv applications. In *Proceedings of SOSR*, pages 121–136, 2015.
- [18] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *Proceedings of SOSR*, pages 164–176, 2017.
- [19] Clarence Filisfilis, Stefano Previdi, Bruno Decraene, Stephane Litkowski, and Rob Shakir. Segment Routing Architecture. Internet-Draft draft-ietf-spring-segment-routing-11, Internet Engineering Task Force, February 2017. Work in Progress.
- [20] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, Ariel Orda, and Tom Edsall. drmt: Disaggregated programmable switching. In *Proceedings of SIGCOMM*, pages 1–14, 2017.
- [21] Barefoot Networks. Protocol independent switch architecture. Website. <https://barefootnetworks.com/technology/>.
- [22] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *Proceedings of SIGCOMM*, pages 15–28, 2016.
- [23] Jan Daciuk, Bruce W. Watson, Stoyan Mihov, and Richard E. Watson. Incremental construction of minimal aperiodic finite-state automata. *Comput. Linguist.*, 26(1):3–16, March 2000.
- [24] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. Moongen: A scriptable high-speed packet generator. In *Proceedings of IMC*, 2015.
- [25] Spirent Communications. Spirent testcenter. Website. <https://www.spirent.com/Products/TestCenter>.
- [26] Cheng Zhang, Jun Bi, Yu Zhou, Abdul Basit Dogar, and Jianping Wu. Mpvisor: A modular programmable data plane hypervisor. In *Proceedings of SOSR*, pages 179–180, 2017.
- [27] David Hancock and Jacobus van der Merwe. Hyper4: Using p4 to virtualize the programmable data plane. In *Proceedings of CoNEXT*, pages 35–49, 2016.
- [28] Cheng Zhang, Jun Bi, Yu Zhou, A. B. Dogar, and Jianping Wu. Hyperv: A high performance hypervisor for virtualization of the programmable data plane. In *Proceedings of ICCCN*, pages 1–9, 2017.
- [29] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. Netbricks: Taking the v out of nfv. In *Proceedings of OSDI*, 2016.