Easy Path Programming: Elevate Abstraction Level for Network Functions

Fei Chen, Chunming Wu, Xiaoyan Hong, and Bin Wang

***Abstract*—As datacenter networks become increasingly programmable with proliferating network functions, network programming languages have emerged to simplify the program development of the network functions. While network functions exhibit high level abstraction over operations on the traffic flow and the interconnections among the operations, the existing languages usually require programming with detailed knowledge about the packet processing patterns at the switches. Such a mismatch between the program abstraction and development details makes developing network functions a nontrivial task. To solve the problem, this paper introduces the Easy Path Programming (EP2) framework. EP2 offers a high-level abstraction to simplify the program design process of the network functions. EP2 also provides a language that captures the common properties of network functions and uses predicates and primitives as basic language components. Specifically, predicates describe when to handle a flow with a global view of the flow dynamics; and primitives describe how to choose a path for a specific flow. Further, EP2 has its own runtime system to support the language and the abstraction model, especially to hide the low level packet-processing behavior at the data plane from the programmers. Throughout this paper, cases are given to illustrate the EP2 abstraction model, language details and benefits. The expressiveness of EP2, the potential overhead of the runtime system and the efficiency of the network functions generated by EP2 are evaluated. The results show that EP2 can achieve comparable performance while reducing programming efforts.**

***Index Terms*—Software-defined networking, network programming model, network function abstraction, data-plane programming, network functions, network programming language**

I. Introduction

Datacenter networks are increasingly programmable with network functions proliferating. Network functions range from load balancing [1]–[3], flow scheduling [4], [5] to congestion control [6], [7] and Quality of Service (QoS) [8], [9]. Today's datacenter workloads impose stringent requirements on the implementation of these network functions. First, datacenter traffic is very bursty and unpredictable [3], [10]–[13]. Given the high frequencies at which demands change and flows arrive, programmers should implement network functions that process packets in the data plane. Second, datacenter network comprises a large number of physical devices [9]. Programming a collection of distributed switches requires manual effort and is error-prone. This is because of the network-wide nature of network functions: they need to measure the current network state and carry out richer computation based on the measured network state. Hence, programmers are forced to manually manage the complex cooperation between the switches. Specifically, programmers usually decompose network function into multiple functionalities first, then design ways in which these functionalities interact with each other, and finally implement these functionalities on distributed switches. These steps often involve error-prone manual function decomposition and network-device configurations.

Software-Defined Networking (SDN) [14] could easily solve the problem as it enables programmers to specify network behaviors through centralized policies at a logically centralized controller. In particular, OpenFlow [15] offers a simple programming model, where switches are abstracted as match-action tables and controllers are responsible for the management of match-action tables. This model enables programmers to write a simple, centralized network function with a global view of the network state without involving the collection of distributed switches. In order to gain OpenFlow's full benefits, a number of domain-specific languages (DSL) have been proposed to further simplify programming of OpenFlow applications [16]–[22].

However, applying this approach to datacenter networks would result in limited scalability [3], [23]. This is because that a controller needs to frequently compute the bandwidth allocation and reconfigure the switches to match the current demand. Even the simple task of detecting elephant flows requires controller's reaction to flow arrivals and response to demand changes. The reaction and response involve frequent rule installations and network-wide statistics collections, leading to significant overhead on both the control- and data-plane.

A promising approach to avoid these problems is to handle the vast majority of packet at the data plane while leaving only a few packets to be handled by the controller [16], [23]–[29]. In particular, emerging data-plane languages [26]–[29] begin to feature advanced per-packet stateful processing at data plane, *i.e.,* programs that create and modify the states in the switches as part of packet processing. They help to program the stateful algorithms at a fine-grained, per-packet level. However, while they simplify the specification of the stateful packet-processing applications, building sophisticated network functions is still challenging in practice. The difficulty stems from the semantic mismatch between the characteristics of network functions and the offered programming model.

Manuscript received March 20, 2017; revised September 15, 2017; accepted October 27, 2017.

Fei Chen and Chunming Wu are with the College of Computer Science and Technology, Zhejiang University, China (e-mail: fei_chen@zju.edu.cn; wuchunming@zju.edu.cn).

Xiaoyan Hong is with the Department of Computer Science, University of Alabama, USA (e-mail: hxy@cs.ua.edu).

Bin Wang was with the Zhejiang University. He is now with Hangzhou Hikvision Digital Technology Co.,Ltd., China (e-mail: wbin2006@gmail.com).

On one hand, network functions care about operations that refer to flow information, query path conditions, and carry out richer computations. On the other hand, existing languages force programmers to explicitly specify packet processing patterns involving per-packet actions and states maintenance. Consequently, programmers are required to manually describe the operations and their functionalities in terms of packet-processing primitives executed on distributed stateful data planes, while guaranteeing that the behaviors of the interactions between the packets and the states are compliant to network functions.

Our goal is to remove the above mismatch by introducing a new programming model and language system, so to further simplify the network function programming. Our work has the following two benefits:

- **Programmability:** It provides an intuitive programming language for expressing network functions while facilitating easy translation to an efficient implementation on the data plane.
- **Performance:** It provides performance guarantees under network dynamics. The implementation is robust to network dynamics such as a sudden spike in traffic demand and reacts quickly to environment changes such as link failures.

To meet the requirement on programmability, we leverage the observation that many network functions can be built upon three common operations in the data plane [30]. First, they rely on flow information to decide when to move a flow away (*i.e.,* semantic operation, called *semantics*). Second, they require path conditions to determine which path to follow (*i.e.,* conditional operation, called *condition*). Finally, they perform path computations over flows (*i.e.,* computational operation, called *computation*). Thus, the key idea behind our programming language is to allow the expression of network function in terms of path computations over traffic semantics and path conditions.

For the desired level of performance, we leverage the observation that today's stateful programmable switches can support these operations (*i.e.,* semantics, condition, and computation) to run on the data plane [31]. Thus, another key idea behind our design is to compile the programs to low-level stateful packet processing applications, and execute compiled programs efficiently on distributed programmable switches.

To this end, we propose Easy Path Programming (EP2), a framework that allows developers to program and execute network functions on the distributed stateful packet-processing data planes. EP2 provides a convenient *programming model* targeted specifically at network functions that build upon *semantics, condition,* and *computation*. It consists of the *EP2 language* for specifying network functions, and the *EP2 runtime system* for executing compiled programs efficiently on the stateful data planes.

- **EP2 Programming Model:** EP2 incorporates the operations underlying many network functions in the programming model, including semantics, condition, and computation. This makes it intuitive to represent a network function as a collection of interconnected operations, where the directed edges among those operations specifying the algorithmic logic of the network function. As a result, programming network functions becomes correspondingly simpler, namely, specifying the dependency between well-defined operations.
- **EP2 Language:** For the programming model, EP2 provides a novel high-level language. The language enables a programmer to express network functions according to the program model. It allows the programmers to describe when to handle a flow, and how to choose a path from a set of available paths for a specified flow. The programmers can focus on what is computed without worrying about the low-level details on how the computation is carried out.
- **EP2 Runtime System:** The EP2 runtime system is the complement of the EP2 language. It delegates the components of a network function to the stateful programmable switches. To guarantee the correctness of execution on the data plane, the runtime system handles the details on the decomposition of network function, placement of functionalities, and cooperation of distributed switches. As a result, the computation of flow paths and the response to network changes are executed directly in the data plane. By implementing these functionalities in the data plane, not only the overhead involving controller is reduced, but also the performance of network functions relying on these functionalities is improved.

Thus, EP2 requires no complex network function decomposition, nor error-prone network device configurations nor management of distributed switches from the programmers. It decouples the low-level details from the programming language while enabling programmers to directly capture the algorithmic part of the network function.

We have built a prototype implementation by using Antlr [32]. Given an EP2 program as input, our tool automatically generates a set of Domino [28] programs. Through case studies, we evaluate how well EP2 can be used to express network functions and also its performance in terms of implementation overhead. The results show that EP2 incurs reasonable overhead with negligible impact on performance. And the results are consistent with those reported in the papers where the same examples were studied and evaluated.

To summarize, this paper makes the following contributions:

- We highlight that a large fraction of network functions are rooted at three key operations: referring to traffic semantics, querying path conditions, and carrying out richer computations.
- We present a new programming abstraction based on the key operations and interactions between the operations. Our abstraction provides intuitive, yet critical features to describe network functions which are currently absent in the existing programming models.
- We present the design, implementation and evaluation of EP2, a framework that maintains the simplicity of centralized programmability and achieves the performance benefits of stateful data planes.
- We show that EP2 can achieve comparable performance

while reducing the programming effort.

The rest of the paper is organized as follows. Section II introduces the background and motivation of this work. An example is given as a problem statement. Section III gives an overview of EP2 architecture. Section IV describes EP2 programming model. Sections V and VI introduce the details of the EP2 language and the runtime system, respectively. We then present evaluations in Section VII, discuss related work in Section VIII, and conclude in Section IX.

II. Background and Motivation

SDN Programming. SDN [14] decouples network applications from the underlying devices and provides an appropriate abstraction for expressing application logic. As a result, control functionalities are centralized logically in the SDN controllers, and network devices are abstracted out as a simple packet-processing device that can be programmed via an open interface (e.g., OpenFlow [15], P4 [26], etc). This design enables the expression of network functions with an appropriate abstraction and using packet-processing primitives.

Specifically, OpenFlow [15] serves as an abstraction for describing the forwarding behavior desired by the network application. Inside an OpenFlow switch, packets are handled through a sequence of flow tables. Flow table entries consist of matching rules used to match incoming packets and a set of actions to be applied upon a match. Upon a new packet arrival, packet header fields are extracted and matched against the matching fields. If a matching entry is found, the entry's actions are performed. Thus, an OpenFlow program is expressed in terms of forwarding rules. This makes building sophisticated applications a complex task in practice. Programmers must manually handle the low-level details such as the priority ordering of rules, network-wide traffic statistics, and composition of multiple programs. Hence, a number of domain-specific languages (DSL) have been proposed to address this challenge [16]–[22]. Rather than explicitly handling the complex low-level forwarding rules, programmers can use a DSL that has a compiler responsible for translating the program to OpenFlow.

However, adaptively changing the resource allocation to meet the network traffic demand can become a major bottleneck for a centralized controller [3], [23]. This challenge stems from the fact that packets in an OpenFlow network may be processed by the centralized controller. Given the high frequencies at which demands change and flows arrive [3], [10]–[13], even the simple task of detecting elephant flows can become a major bottleneck for a central controller managing thousands of switches [9]. Hence, a number of schemes have been proposed to process packets in the data plane [16], [23]–[29]. Specifically, they prefer to handle the vast majority of packets at the data plane while leaving only a few packets to be handled by the controller.

In particular, emerging data-plane languages [26]–[29] begin to feature advanced per-packet stateful processing at the data plane. They help to program the stateful process algorithms at a fine-grained, per-packet level. Inside a stateful switch's pipeline, programmers should specify parsers that pass packets through the pipeline, as well as ways to create, access, and modify states from various stages in the pipeline. Specifically, a P4 program can operate on per-packet states that travel with the packet in the pipeline, and operate on persistent states that are accessible from any stage in the pipeline. This design enables the program to make packet forwarding decisions based on the states, and thus supports the network functionalities that create and modify states as part of packet processing procedure.

Difficulties. Although languages like Domino [28] make it simple to implement stateful packet-processing programs at the data plane, building sophisticated network functions that advocate per-packet stateful processing in the data plane is still challenging in practice. Specifically, the programmers must tackle with several difficulties:

1) Programmers must manually map network functions into the underlying programming model using packet-processing primitives;
2) Programmers have to decompose the high-level policies used in network functions into distributed packet-processing programs installed on each switch;
3) Programmers are required to carefully design and program such that the results from the interactions of packets and states are compliant to network functions, especially in the presence of network dynamics.

These difficulties stem from the semantic mismatch between the requirements of network functions and the offered programming models. Most of the network functions involve computation over traffic semantics and network states, but the existing models represent only the packet-processing behaviors. As a result, programmers must carefully design and program with existing programming models.

A Motivation Example. To highlight the challenges of building sophisticated network functions using an existing language, we consider the *elephant rerouting* application, *i.e., forwarding elephant flows to paths with minimal load.* This function takes the form of a control loop that estimates the current traffic demand, measures the total number of bytes of outgoing path for each packet, and selects the minimal loaded path for each elephant flow. To implement this function, a programmer may build packet processing programs as shown in Figure 1.

To detect the elephant flow, a packet processing program may take the form of *elephant* as shown in snippet ❷. It is invoked at the incoming of packets (snippet ❶). The *elephant* function takes packet p as input and maintains the demand of flows in *flow_size* variable. To get information about the flow size, the program directly expresses a query on packet size as *p.byte*. A programmer then has to maintain a state on flow size on the switch manually, using packet as an index to access and modify the state. Specifically, the programmer explicitly associates the packet p with the flow demand *flow_size*; the linkage is accomplished in the *elephant* function body by using a numeric offset *p.id* in the data object *flow_size*. Given the flow demand, one can detect an elephant flow by checking (*flow_size[p.id]* $>$ *threshold*).

To monitor the total number of bytes of each packet on the outgoing path, the programmer could write a program counting

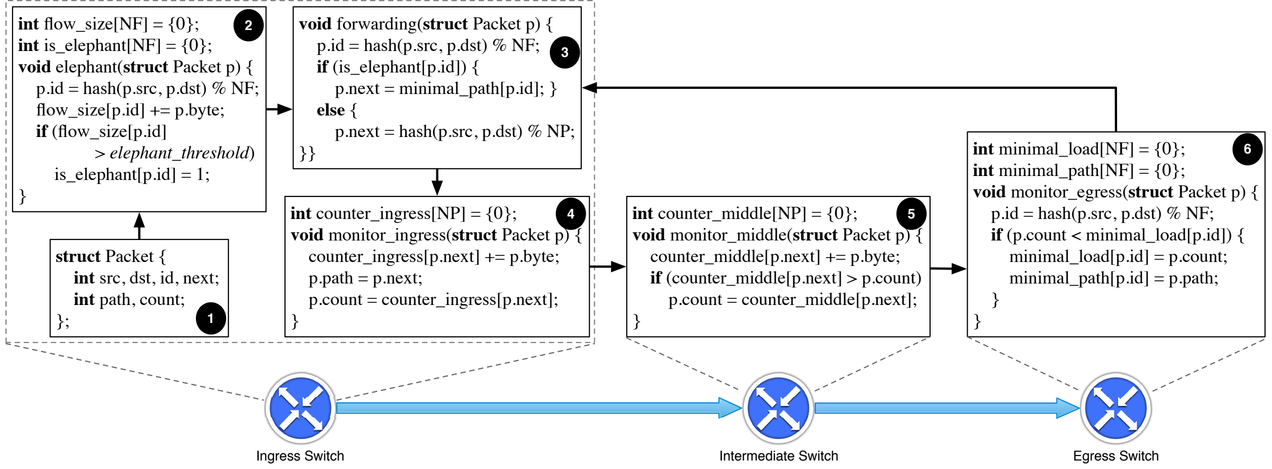

Fig. 1: Network application of "forwarding elephant flows to paths with minimal load" implemented using existing language Domino [28]. These programs would be compiled into low-level microcode that can run on programmable switches.

the bytes associated with each path to collect the necessary statistics. Implementing this functionality at the centralized controller would be easy, since the centralized controller takes responsibility for the collection of flow statistics available at the switches. However, this may slow down the packet processing since this functionality may be invoked on every packet, leading to serious overhead between the controller and the switches [1], [3]. Ideally, the programmer would like to install this functionality on every switches to collect the necessary statistics. However, performing this installation is non-trivial: the programmer needs to identify the *ingress* switch, the *intermediate* switch, and the *egress* switch; and to assign different functionalities to different switches according to the switch's location(❹, ❺, and ❻). Then, when a packet traversing the network, these switches will process the packet accordingly to get the total number of bytes (*i.e., p.counter*) on the outgoing path (*i.e., p.path*) for this packet.

The *elephant* function determines when and how to change the route to match the demand based on the current traffic flow and the total number of bytes on the outgoing path (snippet ❸). Function *forwarding* chooses a path with the minimal total number of bytes using the *minimal_path* for each elephant flow. Here, the programmer handles the complexity of the interactions between the ingress switch and the egress switch to transfer the value of *minimal_path* from ❻ to ❸.

In summary, programmers must manually map network functions to the underlying programming model using packet-processing primitives. This makes programs unnecessarily complicated. Hence, we are motivated to build a new abstraction that provides a simplified programming model so to relieve programmers from tasks such as maintaining global states, decomposing network function into distributed functionalities, and selecting path with a specific strategy. In the following sections, we will use the same example to show the benefits of our work.

III. EP2 Framework Overview

Our goal is to ease the work of programing network function, and to reduce the overhead at controller (so to improve the overall network performance) when running the programs. To achieve this goal, we have developed EP2 framework (see Figure 2), a framework that decouples the low-level implementation from the programming logic via a new programming model and a new programming language.

EP2 provides a domain-specific language for programming network functions. It advocates path computation over traffic semantics and path condition. The EP2 language achieves a higher level abstraction as a programming model, which allows the programmers to focus on the control logic of their network functions and ignore the low-level details. The intuition behind this programming model is to capture the actual structure of network function instead of the underlying switch packet-processing behaviors. Referring to the *elephant rerouting* example discussed in previous section, we see that in order to obtain the desired behaviors of a network function, the programmers have to manually map the network function into underlying packet-level programming model, in a switch-by-switch manner. This makes it hard to implement network function that is guaranteed to execute on the data plane correctly. The EP2 programming model, called *Flow-Path-Graph (FPG)* model, provides the programmers an opportunity to specify their network functions with an elevated abstraction.

In the EP2 language, the programmers describe a network function by defining its control logic, called *Flow-Path-Processing (FPP)* (see §V), in terms of *predicate* and *primitive*. Predicate enables the description of when to handle a flow by mixing flow semantics with path conditions. Primitive simplifies the expression of how to choose a path for a specific flow by packing path computation. The FPP can then be expressed by simply associating predicates with primitives.

In addition, EP2 provides a runtime system for executing the network functions in the stateful data planes. The low-level details involving the implementation of the functions defined by the predicates and primitives are handled by the runtime system. EP2 runtime system would map the high-level programs into the underlying stateful packet-processing applications. Thus, the EP2 runtime system removes programmers' burden of managing a collection of distributed switches.

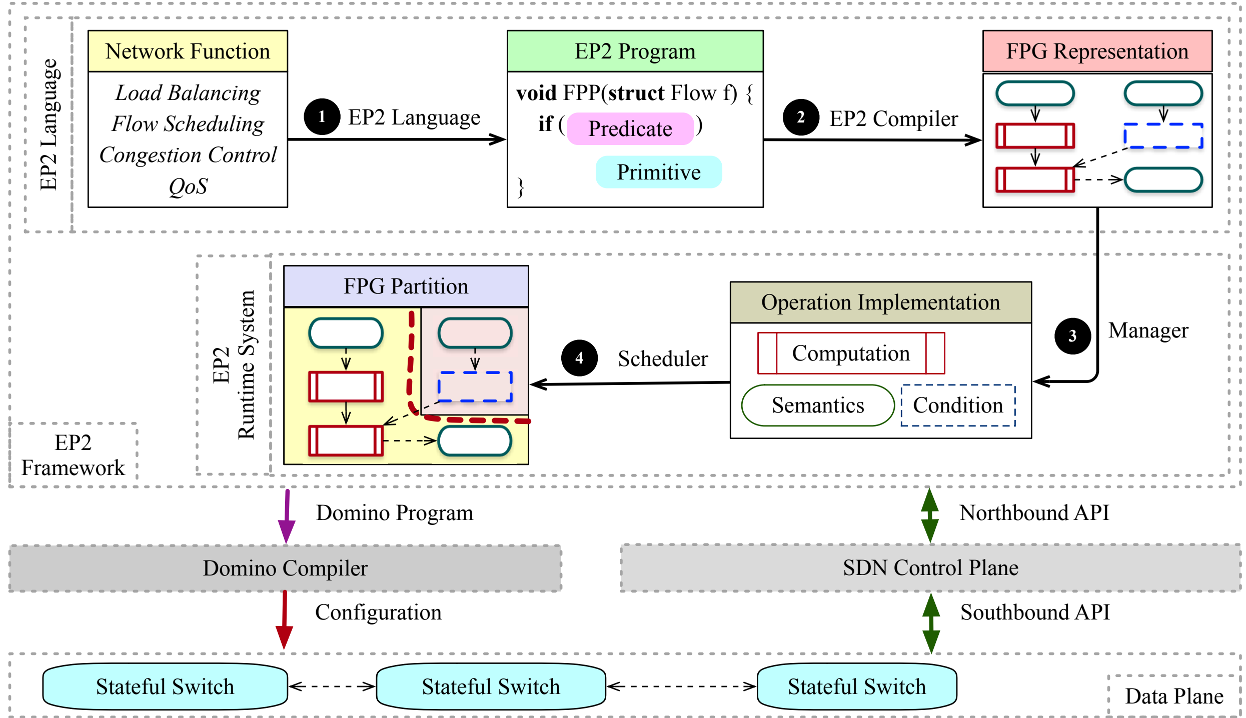

Fig. 2: The EP2 architecture.

Figure 2 gives a more detailed overview of how the programmers use the EP2 framework to implement their network functions. EP2 framework has multiple components. We develop efficient algorithms for these components. The first component *EP2 language* (❶) involves the expression of network function; which is then compiled into FPG representation by the *EP2 compiler* (❷). Furthermore, the *manager* (❸) component enables an efficient implementation of FPG operations on the programmable data planes. Finally, the *scheduler* (❹) makes decisions for the FPG operation placement and interconnection among multiple switches. The EP2 outputs are in the form of the data plane programing language Domino. The Domino compiler will configure the switches to run the needed packet processing logic and the interactions between the switches. The states are measured and exchanged between the switches according to the program. Note that when routing decisions can be made without the need for flow state information, they will be based on the rules imposed by the controller, as illustrated in the figure, which are the same as in the traditional SDN.

IV. EP2 Programming Model

Ideally, the programming model would allow programmers to focus on the logic of network functions while removing their burden of managing the flow details from the distributed switches. Hence, the programming model should capture the actual structure of network function instead of underlying switch packet-processing behaviors. Given such a programming model, the programmers would not need to translate a network function to the stateful per-packet processing programs. In other words, the question drives this work is: *Can we develop a programming model that spells out the actual requirements of network functions and allows intuitive expression of network functions?* This paper takes the first step towards addressing this question in the context of network functions that aim to reduce the average flow completion time (FCT) for short flows and to improve the throughput for long flows in the datacenter networks. Two widely studied functions are *load balancing* (being studied in ECMP [2], [33], Hedera [1] and CONGA [3]) and *flow scheduling* (being studied in PIAS [4], D3 [6] and PDQ [7]). We use them as examples to help introducing EP2. More general applicability of EP2 is discussed at the end of this section.

A. FPG Model

Rather than providing per-packet abstraction to the programmers, we introduce the *FPG* model as an intuitive high-level programming model. The Flow Path Graph model consists of operations that define the components of network functions and the relationships between components that represent the control and data dependency of network functions. Each operation can manipulate the values of network state including flow semantics and path conditions. Flow semantics refer to a set of packets to which a network functionality is applied to. Network states are then handled by the operations which compute a path for each flow entering the network. There are, in most cases, three specially designated operations including *semantics*, *condition*, and *computation*:

- *semantics:* predicating on flow states, through which programmers specify when to run path computation and assignment;
- *condition:* measuring path states, through which programmers determine which path to follow;

- *computation:* performing functionalities over flows at the data plane, including network path computation (through which programmers specify how to choose a path from a set of available paths) and flow path assignment (through which programmers specify the path a flow should go to).

Both *semantics* and *condition* operations are particularly important since many network functions are flow- and load-aware. The *computation* operation is particularly important for datacenter network since many network functions would provide better performance when handling traffic over small time scales. This is due to the important feature of datacenter network, *i.e.,* datacenter traffic is very bursty and unpredictable. Operating at small timescales makes network functions more adapt to traffic dynamics [3].

Given these operations, it is straightforward to capture the dependency graph that would specify the FPG model. The dependency graph links these operations based on the control logic of network function. Our dependency graph has two types of links: *data-flow links* and *control-flow*. Data-flow links capture the resource dependency that the parent operation must be available before the child operation can occur, and control-flow links capture the operation dependency that the parent operation must be done before the child.

B. Case Study

Network functions such as load balancing and flow scheduling are studied in recent papers. They exhibit high-level behaviors in terms of traffic semantics, path condition, and path computation. Here, we use these network functions to illustrate that FPG provides critical features to describe them.

Load Balancing. Load balancing aims to distribute traffic over network paths to reduce the level of congestion. Work form ECMP [33], Hedera [1], and the recently proposed CONGA [3] deal with the function differently. We show how each scheme would be expressed in FPG.

ECMP [33] evenly distributes the traffic over multiple equal cost paths. It can be easily decomposed into the following FPG components:

- *semantics:* relying on selected fields of packet headers to classify packets into flows;
- *condition:* requiring a set of available paths for each flow;
- *computation:* performing hash functionality on the data plane to forward flows.

Hedera [1] measures the bandwidth consumed by each flow and moves elephant flows to an alternate path with sufficient capacity for that flow. While Hedera is implemented with a centralized scheduler, we focus on the higher-order functionalities associated with Hedera: estimating demand for each flow, maintaining available bandwidth for each path, and selecting path for each flow. Thus, it can be decomposed into the following components:

- *semantics:* relying information on flow demand to detect elephant flows;
- *condition:* requiring maintenance of available bandwidth for each path;
- *computation:* performing elephant flow movements on the data plane.

CONGA [3] performs congestion-aware load balancing. It takes the form of a control loop that splits flows into flowlets, estimates real-time path congestion, and allocates flowlet path least loaded. It can be decomposed into the following components:

- *semantics:* relying on the idle interval information to detect flowlets;
- *condition:* requiring to measure the load for each path;
- *computation:* performing flowlet forwarding based on measured path load information on the data plane.

Flow Scheduling. Flow scheduling aims to optimize application performance in terms of reduced FCT by carefully scheduling flows across the network. For example, PIAS [4] minimizes FCT by simulating the shortest flow first scheduling strategy. With PIAS, flows start in the highest priority but are demoted to lower priorities as their sizes increase. PIAS's flow scheduling can be expressed via the following higher-order functionalities: estimating demand for each flow, maintaining available bandwidth for each path, and selecting path for each flow. Thus, PIAS can be decomposed into the following components:

- *semantics:* relying on flow sizes to assign priorities;
- *condition:* requiring to measure the load for each path;
- *computation:* performing path selection for high priority flows at the data plane.

C. Discussion

The FPG model provides the following features to simplify the task of sophisticated network functions programming:

1) FPG models the high-level behavior of a network function that captures the three common operations of existing network functions.
2) FPG enables a programmer to focus on the operations that are necessaries for network functions without worrying about the low-level details behind each operation. The programmer does not think about how to decompose this function into distributed functionalities, where to place those functionalities, and how to connect those functionalities to achieve the desired behavior.
3) FPG serves as an intermediate representation that enables an easy translation from an EP2 program to an efficient implementation on the programmable data planes (§VI).

These features show that FPG is intuitive since it is derived from the common components of existing network functions. Although only load balance and flow scheduling are explicitly analyzed in this paper, FPG is general and applicable to other network functions. In §VII-A, an analysis of most reported network functions is given. The result shows that these functions can be expressed or approximated with matching performance using the proposed FPG operations of semantics, condition, and computation. For examples, congestion control and QoS can be approximated. While the related work usually apply to one function, FPG can support more network functions. We expect FPG model to describe new network functions proposed in the future.

$$\begin{aligned}
\textit{Predicate} \quad & d := c \,|\, d \; bop \; d \\
\textit{Primitive} \quad & p := f.next = s_l\,(f.path) \\
\textit{Selector} \quad & s := MIN \,|\, MAX \,|\, RAND \\
\textit{Path State} \quad & l := LOAD \,|\, ABW \,|\, MULTIPLEX \\
\textit{Flow Field} \quad & h := fi \,|\, fp \\
\textit{Flow Path} \quad & fp := next \,|\, path \\
\textit{Flow Info} \quad & fi := size \,|\, interval \,|\, priority \\
\textit{Condition} \quad & c := f.fi \; bop \; literals \,|\, f.fp.l \; bop \; literals \\
\textit{Statement} \quad & e := p \,|\, if\,(d)\{e\}\,else\{e\} \,|\, e; e \\
\textit{Function} \quad & k := void \;\; FPP(struct \;\; Flow \;\; f)\{e\}
\end{aligned}$$

Fig. 3: EP2 syntax.

V. EP2 Language Design

EP2 language provides language support to express the network functions in terms of the FPG programming model. The syntax of EP2 language is similar to C language, however, its semantics are tailored to programming with FPG programming model. EP2 language offers a number of features that allow programmers to use high-level primitives representing high-level behaviors when expressing processing logic. In this section, we describe the main features of EP2 language in detail.

A. EP2 Language Syntax

The EP2 language is mainly to express the processing logic of a network function (note the type annotations are elided for simplicity). The EP2 language is comprised of a collection of constructs, including *predicate*, *primitive*, and *FPP* functions. All of them allow programmers to specify the intended behaviors of their network functions at a high-level of abstraction. The predicates are used to query flow semantics and look up path conditions. The primitives are used to specify where to forward incoming flows. The FPP functions are used to express the control logic of a network function. Together, these abstractions hide the details of the entire network, while enabling programmers to retain the control over when and how to assign paths for interested flows. The core syntax of EP2 language is shown in Figure 3 with explanations below.

Flow *f*. Before writing an EP2 program, we use a concrete representation of traffic patterns and network states. Generally, a flow f is a record of fields $\{r_1, r_2, \cdots, r_n\}$, where field r_i represents properties such as size, current forwarding path, the set of available paths, etc. The values of fields can be accessed via the notation $f.r_i$, and updated via the notation $f.r_i = v$. Specifically, EP2 allows programmers to specify properties on either traffic patterns via $f.fi$ or forwarding paths via $f.fp$. Furthermore, programmers can access the network states associating with the forwarding paths $f.fp$ via $f.fp.l$, such as past load, available bandwidth, etc.

Predicate. EP2 supports a predicate language for classifying flows. Formally, the predicate denotes a filter, comprising a network state, a operator, and a guard, allowing programmers to select sets of flows that are of interest according to forwarding policies. There are two types of predicates: the predicates on flow information evaluating the guard against flow state, and the predicates on path information evaluating the guard against path condition. A flow information predicate is in the form of $f.fi$ bop $literals$, denoting the set of flows whose flow state $f.fi$ is *bop* to $literals$. EP2 provides flow information predicates for a number of flow properties including $size$, $interval$, and $priority$. For instance, the predicate (*f.size* > *threshold*) has two arguments: the demand (*f.size*) of the flow *f*, and a guard *threshold*. Here, the predicate tests whether the size field of the flow f being processed is greater than the guard value *threshold*. In other words, it tests whether the flow has sent the number of *threshold* packets out in the past. If it has, the predicate is satisfied; if not, it is unsatisfied. A path information predicate is in the form $f.fp.l$ bop $literals$, denoting the set of flows that is being forwarded through the network using path $f.fp$ whose state $f.fp.l$ is *bop* to $literals$. EP2 provides path information predicates for a number of path properties including $load$, abw, and $multiplex$. For instance, the predicate *f.next.abw* > *10* matches the available bandwidth of path *f.next* taken by flow *f*. Predicates can also be combined using logical operators including conjunction (&&), disjunction (||), and negation(!). More complex predicates are built up from simpler predicates using these logical operators.

Primitive. EP2 allows programmers to specify the desired forwarding path using primitives. Formally, a primitive in the form of $f.next = s_l\,(f.path)$ says that the flow f should be forwarded to the path $f.next$ whose state l satisfies the constraint specified in s in the context of $f.path$. The term $f.next$ identifies the fact that this is a primitive that assigns a new path to the flow f. The term $s_l\,(f.path)$ specifies how the flow f should be forwarded. It contains two components. The first component is a filter, written as l, extracting the path information from the set of available paths given by the argument $f.path$. The filter l feeds the extracted path information to the second component. The second component is a selector, written as s, which specifies how to select the path from the set of available paths given by the argument $f.path$. The path chosen by the selector s should match the path information specified by the filter l. Considering the example snippet code *MIN_LOAD(f.path)*, the composed primitive *MIN_LOAD* would first extract the load on each path, take the load as the input to the *MIN* selector to find the path with minimal load.

Function *FPP*. The algorithmic part of an EP2 program is expressed in the *FPP* function that takes a flow f as an input and is defined by the statements in e. Each statement specifies the handling of the incoming flow f. The struct Flow f is essential for modeling traffic patterns and network states, such as the past load on a particular path or packets sent. While the statement e is essential for modeling flow forwarding behaviors that depend upon the traffic patterns and the network states. A statement in the form of $if\,(d)\,\{p\}$ denotes that the primitive p is free to specify the forwarding path that should be applied to match flow f as long as the constraint expressed

Network Application	Operation	EP2 Program
Forward		**void** FPP(**struct** Flow f) {
elephant flows	Semantics	**if** (f.size > *elephant_threshold*)
to paths with		f.next = MIN_LOAD(f.path);
minimal	Computation	}
load	Condition	❶

Fig. 4: EP2 program for elephant rerouting application.

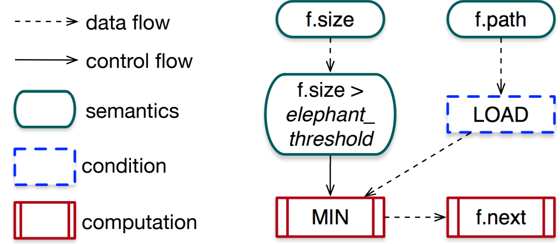

Fig. 5: FPG representation for elephant rerouting application.

by the predicate d is satisfied. Thus, the FPP function runs as this: taking a single flow f as the input and assigning the forwarding path through the network using the primitive p when the incoming flow f satisfies the predicate d. Note that the burden of managing all the details needed to ensure that each flow is forwarded out the correct path is left to the runtime system.

An EP2 Example. We illustrate how program in EP2 language using the same elephant rerouting example, the one presented in Figure 1 as our motivation. The corresponding EP2 program is shown in Figure 4 and elaborated below.

The data type used in *elephant* function (snippet ❷ in Figure 1), *i.e.,* the *Packet*, is defined at a very low level of abstraction. In order to maintain the demand of a specified flow, the program first *hashes* the packets into flows, and then accesses and modifies the corresponding data array *flow_size*. The program shows the burden of programming in maintaining global arrays and assigning offsets. In contrast to this packet-processing abstraction, with EP2, a conventional data object *Flow* is used for flow-processing abstraction. Thus, the programmer can query the information of a specified flow by referring to the field of struct *Flow*, one information per field. The *runtime system* is responsible for the implementation of corresponding routines. In this case, we would be able to pack the function body of maintaining *flow_size* into the express *f.size* that yields the demand of flow *f*.

Rather than implementing the *monitor* function (snippet ❹, ❺ and ❻ in Figure 1) at the centralized controller, which forces the controller to process far more packets than necessary, we install the monitor function at the distributed data planes. Using the previous programming model, the programmer manually and carefully specifies the functionality executed on each switch. In addition, the programmer must also specify the communication patterns between the different functionalities, *i.e.*, the way they interact with each other, to obtain the desired behavior. However, using EP2, programmers are provided a set of declarative query operators to obtain the desired behavior. In this case, we would be able to query the total number of bytes of outgoing path for flow f with a simple query expression: *f.next.load*.

For the *forwarding* function (snippet ❸) shown in Figure 1, the programmer needs to manually implement the path selecting functionality *minimal_path* depending on network state, which makes the program unnecessarily complicated. In EP2, there is no need to implement the *minimal_path* functionality. Rather, the programmer can choose from a set of declarative query operators to obtain the desired path. In this case, we would be able to choose a path for flow f compliant with the *min_load* function using: *MIN_LOAD(f.path)*, where *f.path* represents the set of available paths for *f*.

Furthermore, to specify the *forwarding* function using EP2 language, a programmer defines a function *FPP* to be invoked on every flow f (Figure 4). This flow f has demand *f.size* and should be forwarded to path *f.next* which is selected from available paths *f.path*. The program shown above uses the predicate (*f.size* $>$ *elephant_threshold*) to detect elephant flows: as soon as the flow size exceeding the threshold we characterize it as an elephant. The program then selects the least loaded path with primitive: *MIN_LOAD*, which takes *f.path* as input and returns the least loaded path to *f.next*. In this code, the special keyword *LOAD* matches *f.path.load*, while the keyword *MIN* ranks the set of paths with key *LOAD*. Thus, the statement *MIN_LOAD* says that the flow f should follow the path with minimal total number of bytes. As such, by associating predicates *f.size* $>$ *elephant_threshold* with primitive *MIN_LOAD*, the programmer defines the network application shown in Figure 4.

B. EP2 Compiler

Given an EP2 program, the compiler translates it into FPG. The compiler first performs *code analysis* to determine the operations required to implement a network application. Any *predicates* of the program should be translated into the *semantics* operations. The compiler then partitions any *primitives* of the program into three operations: the arguments of the *primitive* are translated into the *semantics* operations, the selector part of the *primitive* is translated into the *computation* operation, and the path state part of the *primitive* is translated into the *condition* operation. Finally, the compiler interconnects these operations to build the intermediate representation of the program, *i.e.,* FPG.

To illustrate the workflow of the compiler, consider the *elephant rerouting* example shown in Figure 4. To begin, the compiler would identify six operations: (*f.size* $>$ *elephant_threshold*) predicating flow semantics *f.size*, *MIN* computing path over *f.path* using path condition *load*, and *f.next* updating the path for flow *f*. The FPG representation can then be obtained by drawing directed edges between those operations and related dependencies, as shown in Figure 5, representing possible logic for flow handoff.

VI. EP2 Runtime System

EP2 provides high-level programming model that captures the logic of network functions instead of behaviors of the

physical switches. However, the requirements of handling low-level details do not just disappear. There is still the need of mapping the high-level FPG representation to the low-level packet-processing behavior. Rather than requiring the programmers to manually interact with switch-level primitives, the *runtime system* takes the responsibility of implementing the low-level details in order to provide an efficient deployment for high-level FPG model.

EP2 *runtime system* manages all of the functionalities related to the network state measurement and the path selection. It also generates the necessary communication patterns between these functionalities. Figure 2 shows an overview of the EP2 runtime system. The runtime system is designed around the FPG. It is composed of tasks that decompose and place the FPG to the stateful SDN for execution. It handles operations, the communication between operations (*i.e., manager*) and the assignment of operations to programmable switches and SDN controllers (*i.e., scheduler*).

The *manager* (❸ in Figure 2) measures the flow semantics and path conditions to specify when to run path computation and assignment. For each flow, the manager provides predicates on flow semantics and path conditions to specify when to run path computation and determines the path from a set of available paths the flow should go to. The path assigned by the manager is compliant with the network function that the programmer defined. Specifically, the manager aims to achieve the desired behavior specified by the programmer based on the FPG.

The *scheduler* (❹ in Figure 2) determines the appropriate assignment of the FPG to the controller and the underlying switches to support high-level network function in a scalable way. Since we focus on the network function that advocates stateful packet processing in the network data planes, we do not involve the controller in handling data packets. Specifically, the scheduler runs on top of the controller to divide the FPG across the switches with a goal of minimizing the fragmentation of the FPG. This is achieved by exploiting locality, *i.e.*, the operations handled locally within a switch. In this way, a substantial fraction of the operations will be placed within individual switch which reduces the inter-operation communications. Hence, the scheduler significantly lowers the communication overhead among operations.

A. Manager

The *manager* first measures the network states including those in *semantics* and *condition* belong to the FPG specified by the programmer. Then, the *manager* performs network state aware computation. We briefly describe how the manager works in terms of *semantics, condition,* and *computation*. Throughout this section, we will use the elephant rerouting application (Figure 4) as the illustration example.

Measuring Semantics. For the semantics, *manager* needs to mark all its traffic and to identify each semantic information separately. The semantic information is carried with the flow and is used to determine the operations to apply. In addition, *manager* keeps tracking of the historical semantics for necessary, and associates the historical semantics with flows by using a unique identifier. For example, *f.interval* measures the idle interval between two bursts of packets from the same flow *f*. The *manager* would keep tracking the arrival time of the last packet from flow *f*, and update the interval time *f.interval* when the new packet arrives.

Consider the elephant flow detection part of the elephant rerouting application. In this case, we assume that the threshold, beyond which a flow is characterized as elephant, can be determined by analyzing the flow size distribution. For each flow, *manager* updates the size of the flow and then test the flow size on the threshold by using predicate (*f.size* $>$ *elephant_threshold*). This predicate triggers a reroute operation if a flow matches the threshold.

Measuring Condition. The *manager* uses switch-to-switch feedback mechanism to measure the path conditions. The knowledge about path conditions are essentially needed in order to handle network dynamics. For example, with asymmetric network topology, a switch is hard to balance traffic without the knowledge about downstream congestion. The *manager* measures path conditions using a CONGA-like condition estimation algorithm, where the egress switch feeds the path conditions to the ingress switch.

To illustrate the procedure of path condition measurement, we use the *load* functionality (see Figure 4) from the elephant rerouting example. Measuring the level of load requires coordination among switches to identify the bottleneck for each path. Thus, per-switch monitor does not satisfy this requirement. Specifically, in order to measure the load level, the source switch should attach a load field to the packet header that contains the current state of load level (*e.g., f.next.load*). Each switch then maintains this state for the path *f.next*. Since this state is needed for the ingress switch, the egress switch feeds this state back to the ingress switch. The following steps can be used to measure the level of load:

1) For each intermediate switch, keeps tracking of the total number of bytes for each upstream link;
2) For each arrival flow *f*, updates *f.next.load* if upstream link has higher load level than the value in *f.next.load*;
3) For each egress switch, feedbacks the value in *f.next.load* to the source switch with opportunity;
4) For each ingress switch, update the *load* table for each path upon receiving feedback from the egress switch.

The *manager* must maintain states over time because flow semantics or network conditions change. The main concern is the overhead incurred by such fine-grained recording. To reduce the overhead, the *manager* requires that every flow must begin with a SYN and terminate with a FIN so that *manager* can perform in-network book-keeping.

Performing Computation. The computational functionality is conceptually a collection of network state operations. The state of the network is indexed by referring to the programmer specified fields. Intuitively, a computation functionality takes a set of paths plus the current state of the network as input, and determines the path characterized by the properties specified by the programmer.

To illustrate how the *manager* handles the computational operations, consider the example of *elephant rerouting* again.

To begin, the manager starts by learning the size of a flow. If the flow's size is already larger than a predefined threshold, the manager reads the path conditions to find a path that this flow should go to. In this case, the code snippet (*f.next* = *MIN_LOAD* (*f.path*)) in the *forwarding* function would check two kind of network states: the set of possible paths *f.path* and the load level for each path in *f.path* (e.g, *f.path.load*). The operation *MIN_LOAD* sorts these paths (*i.e., f.path*) based on their load extent and returns the path with minimal load, which is then assigned to flow *f* (*i.e., f.next*). If a flow has been moved to a new path, the manager updates the path that carries this flow for total number of bytes automatically. Consequently, flows can continue to forward along this path without causing congestion even when other flows arrive.

B. Scheduler

The objective of the *scheduler* is to maximize the number of the operations handled locally within switches. It determines for each FPG the locations of its operations. The placement is synergistic with the *manager*, as it satisfies the placement of the operations assigned by the manager. Specifically, the scheduler needs to ensure that it places operations in the network (*i.e.,* ingress/egress switches) specified by the manager (called *local* operations). For the remaining operations not assigned by the manager (called *residual* operations), the scheduler performs a greedy algorithm that exploits the locality to minimize the overhead of data transfer as follows:

1) Let G_l be the set of *local* operations with location l assigned by the manager.
2) Residual operations become assignable after all its parent is scheduled.
3) The assignable operation would be added to G_l if it has the largest connectivity with the scheduled operations in G_l.

To illustrate the placement algorithm, consider the example of *elephant rerouting* application. First, the scheduler assigns the set of operations {*f.size*, *f.next*} and {*load*, *f.path*} to the ingress switch and egress switch respectively, based on the feedback from the manager. Then, the scheduler iterates over remaining operations and assigns them if their parent dependency are satisfied. We see that (*f.size* $>$ *threshold*) does not depend on other operations, so it can be scheduled immediately. In this case, the scheduler would add the operation of (*f.size* $>$ *threshold*) to the ingress switch, as it depends on the operation of *f.size*. For the remaining operation *MIN*, the scheduler checks its dependency on (*f.size* $>$ *threshold*) and *load*, and finds that it has a child node *f.next*. So, this operation would be assigned to the ingress switch. Finally, the scheduler generates the communication pattern between the ingress and egress switches, as the operation of *MIN* depends on the operation of *load*.

VII. Evaluation

The central piece of EP2 implementation is the compiler, which translates the EP2 program into the FPG representation. We build a prototype implementation for the compiler described in section V-B by using Antlr [32]. This tool accepts an EP2 program and produces the corresponding FPG. The other piece of EP2 implementation is the runtime system, which sits between the FPG representation and the stateful SDN. We implement the runtime system described in section VI on top of Domino. The Domino language provides features such as running at line rate, expressing algorithms with an imperative language, and manipulating states at the computation.

TABLE I: Example network functions. Where ✔ indicates supported, and ✶ indicates approximated.

Function	Scheme	EP2		
		Semantics	Condition	Computation
Load balancing	WCMP [2]	✔	✔	✔
	Hedera [1]	✔	✔	✔
	CONGA [3]	✔	✔	✔
Flow scheduling	PIAS [4]	✔	✶	✶
	pFabric [5]	✔	✶	✶
Congestion control	D3 [6]	✔	✶	✶
	PDQ [7]	✔	✶	✶
QoS	QJUMP [8]	✔	✶	✶

In this section, we evaluate the expressiveness of EP2 language and the performance of network functions achieved by their implementations on top of EP2 runtime system. We answer three key questions:

1) Does EP2 allow programmers to easily write real-world network functions?
2) What is the overhead of EP2 runtime system on the programmable switches?
3) How efficient are the implementations of network functions generated by EP2?

We address #1 through quantitative analysis and #2-3 through case studies on real-world network fucntion examples.

A. Expressiveness

We have implemented several network functions (Table I) that are typically related to routing algorithms. Table I gives the expressiveness of EP2 language in terms of whether EP2 can support operations of those network functions. The listed network functions are those studied in related work, which are cited. EP2 is able to express most of them, while each related work only covers one network function. For those that EP2 can not express fully, which are the functions performing packet scheduling and queue management, EP2 can approximate them. The approximation can simplify their implementation.

B. Case Study

The case study is to evaluate the runtime system and the resulting network functions with comparisons to the related work. Our rational for choosing the candidate existing work from Table I for comparison is justified in term of the network function and the related scheme implementing the selected function. Given the four network functions, we observed that a large fraction of load balancing schemes are already available as routing algorithms; translating them to FPG model is straightforward. In contrast, expressing flow scheduling function demonstrates high complexity. It requires a support for flow prioritization at the switches, which may be unavailable at the path selection component. Specifically,

```
1 void pFabric(struct Flow f){
2   if (f.interval > flowlet){
3     if (f.priority <= threshold){
4       f.next = MIN_LOAD(f.path);
5 }}}
```

Fig. 6: pFabric implementation in EP2.

flow prioritization requires several properties, such as, strict priority to prioritize one flow over another, and preemption to allow higher priority flows to preempt lower priority ones if needed. The main reason accounting for this difficulty is that flow-based policies schedule flows one at a time. This can help finish flows faster by reducing the amount of contention in the network. In addition, these two properties are crucial in supporting flow prioritization mechanisms in the other network functions of congestion control and QoS (in PDQ [7] and QJUMP [8] respectively). Hence, it is sufficient when using flow scheduling function in our evaluation. Further, the flow scheduling function in pFabric implementation has been used in performance comparisons with other network functions by multiple researchers. It has shown promising performance in these comparisons. Thus, we also choose to compare EP2 implementation with pFabric implementation. With a comparison to pFabric, one is able to estimate the performance of EP2 with those related work.

Identical implantation of the flow scheduling function in EP2 as it in pFabric is impossible due to the different levels of abstraction of the two languages. Guided by pFabric, which prioritizes small flows over large flows, we consider forwarding small flows over less loaded paths. By examining pFabric, we learned that it schedules packets using the Shortest Remaining Processing Time First policy. Hence, as our case study, we describe how EP2 can approximate pFabric in expressing the network function that schedules flows so as to reduce the FCT.

Figure 6 shows the pFabric function written in EP2 language. Note that we omit the maintenance of the *threshold* variable used as guard in line 3. This function makes two decisions to approximate pFabric: which flow to move away and which path to take. For the former, we assume that the data center has the knowledge about the distribution of the flow size. Based on this information, we identify a set of thresholds that define the priority for a flow given its current size. For each flow, the program then searches the set of thresholds to find the priority corresponding to the flow size. Then the function performs a priority-based flow scheduling in which only the highest priority flow should be forwarded along the least loaded path. This design guarantees that the highest priority (smallest) flows encounter small buffers and consequently entail small latencies. This, in turn, helps to reduce the FCT of large flows, since the contention on the path taken by large flows would be reduced as we move the small flows away.

Runtime system overhead and programmability. The EP2 runtime system program of the *pFabric* function is given in Figure 8. As shown, it is implemented on top of the Domino language. The program implemented in Domino has

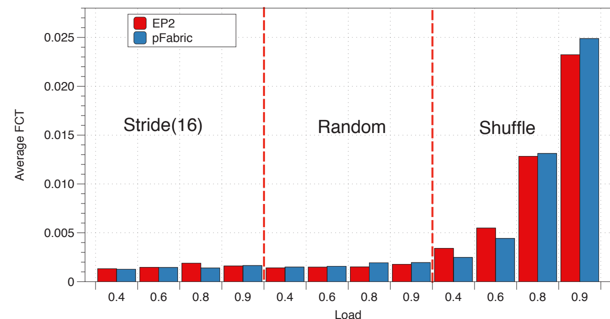

Fig. 7: Implementation efficiency.

TABLE II: Overhead of EP2 runtime system.

Component	Most expressive atom [28]	Delay
Semantics	ReadAddWrite (RAW)	316ps
Condition	Paired updates (Pairs)	606ps
Computation	ReadAddWrite (RAW)	316ps

the details for *semantic, condition* and *computation*. For the programmability, we count the number of lines in the original EP2 program implementation of *pFabric* function in Figure 6, and compare the number with the number of lines in Figure 8. The result shows that EP2 is more concise, using five time less number of lines in the program.

To evaluate the overhead of the runtime system, we check whether the Domino implementation allows the programmable switch to run at line rate following the method used in previous works [28]. We report the overhead of running the *semantic*, *condition*, and *computation* functions in terms of most expressive atom, as shown in Table II. In the table, *RAW* indicates the ability to read a packet field, add the value of the packet field to the state variable, and write back to the state variable. *Pairs* extends the ability of *RAW* to allow conditional operations based on the value of the state variable. Essentially, an atom reflects the delay to run the function over programmable switch and the area overhead incurred in silicon. Specifically, *RAW* has a delay of 316ps and an area of $431\mu m^2$; *Pairs* has a delay of 606ps and an area of $5997\mu m^2$. All atoms meet timing at 1GHz. The result shows that both of these functions can run at a 1GHz clock frequency.

Implementation performance. To assess the performance of our algorithms used in implementing pFabric, we compare the EP2 implementation with pFabric implementation in NS2 [34].

We use the leaf-spine topology that has 9 leaf switches connected to 4 spine switches. Each leaf switch has sixteen 10Gbps downlinks to the hosts (144 hosts) and four 40Gbps uplinks to the spines. Our simulation setup is based on the NS2 implementation from [35]. The workload used in the experiments is based on the observation that most of the traffic are generated from a small fraction of the flows. Specifically, both the flow size and sending rate are generated according to the distribution modeled after realistic traffic pattern in the deployed datacenters, as reported in [10]. Similar to the previous works [3], [7], [36], we study the impact of the following sending patterns:

- *Stride(i)*: servers in the network are indexed from left to

TABLE III: Comparison of EP2 with pFabric in terms of FCT for small and intermediate flows with *shuffle* sending pattern.

Load	Metric	Flow type	EP2/pFabric
0.8	Average FCT	Short	108%
		Intermediate	100%
	Tail FCT	Short	102%
		Intermediate	107%

right; a server with index x sends to the server with index (x+i) mod N, where N is the total number of servers.

- *Random*: a server sends to another randomly-selected server not under the same leaf switch.
- *Shuffle*: each server in the network sends data to every other server in the network, with a constraint that the source and the destination are under different leaf switches.

Figure 7 shows the overall average flow completion time (in seconds) for each sending pattern at different traffic loads. From the figure, we find that the overall average flow completion time is similar for both schemes in our experiments. First, under low network load, pFabric has some advantages over EP2. Second, EP2 is comparable to pFabric with increasing load. This is because the throughput is mainly bottlenecked at the leaf-to-host links at the low load and at the leaf-to-spines links at the high load. The latter enables EP2 to optimize average FCT with increased load since EP2 focuses on scheduling leaf-to-spine links. Further, we breakdown the FCT statistics for two classes of flows: the small flows (<10KB) and the intermediate flows (10KB — 1MB). Table III compares the average and the 99th-percentile of FCT for the two classes when executing EP2 and pFabric programs with the shuffle sending pattern. As expected, by enabling prioritization, pFabric reduces the average flow completion time. EP2, by fairly sharing bandwidth among flows, provides comparable performance with less variability. In all, it is safe to say that EP2 can approximate pFabric by prioritizing small flows over large flows. This is enabled by always selecting the least loaded paths for small flows.

VIII. Related Work

EP2 relates to existing work on network programming languages in different ways. Table IV briefly summarizes and compares these related work against the design goals of the network programming languages in terms of programmability and performance.

OpenFlow-Based SDN: Several programming languages have been proposed to offer a level of abstraction for programming the network. They include Frenetic [16], Pyretic [17], and Maple [22]. These languages are limited to OpenFlow networks: any stateful processing intelligence of network services is delegated to the centralized controller, leaving the OpenFlow switches dumb. The centralized controller is responsible for the correctness of the switch behavior. EP2 is similar to these work in the sense that the FPG programming model allows programmers to focus on the logic of network functions while removing their burden of managing the collection of distributed switches. But, EP2 enables the implementation of network functions in the data plane by leveraging stateful platforms, which is not the case for these languages. Although Frenetic also aims to handle as much packets as possible at the data plane to reduce the amount of packets handled by the controller, it does not allow programmers to specify switch functionalities as EP2 does. Readers interested in a comprehensive survey about programming languages in OpenFlow-based SDN may consult the paper by He *et al.* [37].

Stateful SDN: There are some new languages that offload the programs requiring stateful traffic processing to the switches. They include OpenState [38], FAST [39], P4 [26], and Domino [28]. The P4 language represents a switch as an abstract forwarding model which allows the expression of how packets should be processed by the network switches. Programmers can program a set of header fields to be matched and a set of actions to be applied. Programs written in the P4 language can be mapped to several kinds of devices (e.g., NPU, FPGA, and Open vSwitch [40]). The Domino language proposes the packet transaction abstraction to represent sequential packet processing. It allows the programmers to write stateful data-plane packet processing programs without worrying about other concurrent packets. Therefore, the programmers only concern about the operations on one single packet. Unfortunately, while these mechanisms make it possible to implement traffic processing tasks on the data plane, both P4 and Domino do not make it easy. They both target at a single device, while lacking the abstraction to help programmers managing the complexity of programming a network of switches. The main difference between EP2 and these languages is that EP2 seeks to address the issue of how to program a collection of interconnected switches. On the other hand, EP2 benefits from these languages in that they can be used as an intermediate language for EP2 programs.

One-Big-Switch: The closest related work to EP2 is SNAP [27], which also offers a new language and an abstraction model to reduce the complexity of network programming. It does so by representing the whole network as one big switch. With SNAP, programmers can easily write programs involving stateful operations at the data plane, without knowing how or where to store the state information. Similar to SNAP, EP2 provides the programmer a centralized view of network states and allows the programmer to manage network states globally. While the high-level architecture is similar, EP2 emphasizes on the details that are significant to the network functions handling the datacenter traffic, e.g., the load balancing, which is critical to avoid network congestion at short timescales. Moreover, EP2 also emphasizes on the functional properties of when and how to choose a path for a specified flow while SNAP does not.

IX. Conclusion

This paper has presented EP2 framework with the goal to ease the program developing process for the network functions. Specifically, EP2 consists of a language, a compiler and a runtime system for implementing network-wide applications using a distributed set of programmable devices. The details relating to programming with EP2 high-level language and

Proposal	Programming Model	Programmability	Performance
Frenetic [16]	OpenFlow-based SDN	Simple: - Domain specific language - Global view of network state	Low: Indirect control (stateful packet processing in the controller)
NetKat [20]			
Maple [22]			
P4 [26]	Stateful SDN	Complex: - Need for cooperation - Per-device programming	High: Direct control (stateful packet processing in the switches)
Domino [28]			
OpenState [38]			
FAST [39]			
SNAP [27]	One-Big-Switch	Simple: Domain specific language	
EP2	Flow-Path-Graph	Simple: - Network function driven	High: Direct control & Global network state aware packet processing

TABLE IV: Comparison of related work in terms of programming model, programmability, and performance.

supporting from its own runtime system are given using examples with comparisons to related work. Moreover, the paper has shown that EP2 can express or approximate the various network functions studied in the literature with matching performance. This shows that EP2 framework simplifies the example programing cases, and is general and applicable to other network functions. In our future work, more network functions will be instantiated.

Acknowledgment

This work is supported by the National Key Research and Development Program of China (2016YFB0800102, 2016YFB0800201), the Key Research and Development Program of Zhejiang Province (2017C01064, 2017C01055), the National Natural Science Foundation of China (61379118) and the Fundamental Research Funds for the Central Universities. Xiaoyan Hong's work is supported partly by National Science Foundation award #1541462.

```
int first_time [NUM_FLOWS] = {0};
void semantic(struct Packet pkt) {
  pkt.id = hash2(pkt.sport, pkt.dport) % NUM_FLOWS;
  pkt.interval = pkt.arrival - first_time[pkt.id];
  pkt.priority = 0;
  if (pkt.interval > THRESHOLD_1) pkt.priority = 1;
  if (pkt.interval > THRESHOLD_2) pkt.priority = 2;
  if (pkt.interval > THRESHOLD_3) pkt.priority = 3;
  if (pkt.interval > THRESHOLD_4) pkt.priority = 4;
  if (pkt.interval > THRESHOLD_5) pkt.priority = 5;
  if (pkt.interval > THRESHOLD_6) pkt.priority = 6;
  if (pkt.interval > THRESHOLD_7) pkt.priority = 7;
}
int old_path_util[NUM_PATHS] = {100};
int best_path_util[NUM_SRCS] = {100};
int best_path[NUM_SRCS] = {0};
void condition(struct Packet pkt) {
  pkt.old_path_id = pkt.old_path > 0 ? pkt.old_path
      % NUM_PATHS : 0;
  pkt.src_id = pkt.old_path > 0 ? (pkt.old_path /
      NUM_PATH_PER_SRC) % NUM_SRCS : 0;
  pkt.old_path_util = old_path_util[pkt.old_path_id]
       * ALPHA / 16 + pkt.mark * BETA / 16;
  old_path_util[pkt.old_path_id]=pkt.old_path_util;
  if(pkt.old_path_util<best_path_util[pkt.src_id]){
    best_path_util[pkt.src_id] = pkt.old_path_util;
    best_path[pkt.src_id] = pkt.old_path_id;
  }else if(pkt.old_path_id==best_path[pkt.src_id]){
    best_path_util[pkt.src_id] = pkt.old_path_util;}
}
void computation(struct Packet pkt) {
  if (pkt.priority < THRESHOLD) {
    pkt.new_path = best_path[pkt.src_id];
  } else { pkt.new_path = 1;}
}
```

Fig. 8: pFabric implementation in Domino. Packet and constant definitions are elided for simplicity. The measurement of path utilization is also omitted, which is similar to the monitor function defined in Figure 1.

References

[1] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic Flow Scheduling for Data Center Networks," in *7th USENIX Symposium on Networked Systems Design and Implementation (NSDI 10)*, (Berkeley, CA, USA), pp. 19–19, USENIX Association, 2010.

[2] J. Zhou, M. Tewari, M. Zhu, A. Kabbani, L. Poutievski, A. Singh, and A. Vahdat, "WCMP: Weighted Cost Multipathing for Improved Fairness in Data Centers," in *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, (New York, NY, USA), pp. 5:1–5:14, ACM, 2014.

[3] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese, "CONGA: Distributed Congestion-aware Load Balancing for Datacenters," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, (New York, NY, USA), pp. 503–514, ACM, 2014.

[4] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang, "Information-Agnostic Flow Scheduling for Commodity Data Centers," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, (Oakland, CA), pp. 455–468, USENIX Association, 2015.

[5] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pFabric: Minimal Near-optimal Datacenter Transport," in *Proceedings of the 2013 ACM Conference on SIGCOMM*, SIGCOMM '13, (New York, NY, USA), pp. 435–446, ACM, 2013.

[6] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, "Better Never Than Late: Meeting Deadlines in Datacenter Networks," in *Proceedings of the 2011 ACM Conference on SIGCOMM*, SIGCOMM '11, (New York, NY, USA), pp. 50–61, ACM, 2011.

[7] C.-Y. Hong, M. Caesar, and P. B. Godfrey, "Finishing Flows Quickly with Preemptive Scheduling," *SIGCOMM Comput. Commun. Rev.*, vol. 42, pp. 127–138, Aug. 2012.

[8] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. M. Watson, A. W. Moore, S. Hand, and J. Crowcroft, "Queues Don't Matter When You Can JUMP Them!," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, (Oakland, CA), pp. 1–14, USENIX Association, 2015.

[9] L. Popa, P. Yalagandula, S. Banerjee, J. C. Mogul, Y. Turner, and J. R. Santos, "Elasticswitch: Practical Work-Conserving Bandwidth Guarantees for Cloud Computing," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 351–362, 2013.

[10] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data Center TCP (DCTCP)," in *Proceedings of the 2010 ACM Conference on SIGCOMM*, SIGCOMM '10, (New York, NY, USA), pp. 63–74, ACM, 2010.

[11] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The Nature of Data Center Traffic: Measurements and Analysis," in *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, pp. 202–208, ACM, 2009.

[12] T. Benson, A. Anand, A. Akella, and M. Zhang, "Understanding Data Center Traffic Characteristics," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 1, pp. 92–99, 2010.

[13] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri,

D. A. Maltz, P. Patel, and S. Sengupta, “Vl2: A Scalable and Flexible Data Center Network,” in *ACM SIGCOMM computer communication review*, vol. 39, pp. 51–62, ACM, 2009.
[14] D. Kreutz, F. M. V. Ramos, P. E. Verssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, “Software-Defined Networking: A Comprehensive Survey,” *Proceedings of the IEEE*, vol. 103, pp. 14–76, Jan 2015.
[15] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, “Openflow: Enabling Innovation In Campus Networks,” *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.
[16] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, “Frenetic: A Network Programming Language,” in *ACM Sigplan Notices*, vol. 46, pp. 279–291, ACM, 2011.
[17] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker, “Modular SDN Programming With Pyretic,” *Technical Reprot of USENIX*, 2013.
[18] R. Soulé, S. Basu, R. Kleinberg, E. G. Sirer, and N. Foster, “Managing the Network with Merlin,” in *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, (New York, NY, USA), pp. 24:1–24:7, ACM, 2013.
[19] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster, “Merlin: A Language for Provisioning Network Resources,” in *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT ’14, (New York, NY, USA), pp. 213–226, ACM, 2014.
[20] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, “NetKAT: Semantic Foundations for Networks,” in *ACM SIGPLAN Notices*, vol. 49, pp. 113–126, ACM, 2014.
[21] M. Reitblatt, M. Canini, A. Guha, and N. Foster, “Fattire: Declarative Fault Tolerance for Software-Defined Networks,” in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pp. 109–114, ACM, 2013.
[22] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak, “Maple: Simplifying SDN Programming Using Algorithmic Policies,” in *Proceedings of the 2013 ACM Conference on SIGCOMM*, SIGCOMM ’13, (New York, NY, USA), pp. 87–98, ACM, 2013.
[23] F. Chen, C. Wu, X. Hong, Z. Lu, Z. Wang, and C. Lin, “Engineering Traffic Uncertainty in the OpenFlow Data Plane,” in *Computer Communications, IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on*, pp. 1–9, IEEE, 2016.
[24] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, “Scalable Flow-Based Networking With DIFANE,” *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4, pp. 351–362, 2010.
[25] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, “DevoFlow: Scaling Flow Management for High-performance Networks,” in *SIGCOMM*, pp. 254–265, ACM, 2011.
[26] 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,” in *Proceedings of the 2013 ACM Conference on SIGCOMM*, SIGCOMM ’13, (New York, NY, USA), pp. 99–110, ACM, 2013.
[27] 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 Conference on SIGCOMM*, SIGCOMM ’16, (New York, NY, USA), pp. 29–43, ACM, 2016.
[28] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking, “Packet Transactions: High-Level Programming for Line-Rate Switches,” in *Proceedings of the 2016 ACM Conference on SIGCOMM*, SIGCOMM ’16, (New York, NY, USA), pp. 15–28, ACM, 2016.
[29] J. McClurg, H. Hojjat, N. Foster, and P. Černỳ, “Event-driven Network Programming,” in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 369–385, ACM, 2016.
[30] H. Ballani, P. Costa, C. Gkantsidis, M. P. Grosvenor, T. Karagiannis, L. Koromilas, and G. O’Shea, “Enabling End-Host Network Functions,” in *Proceedings of the 2015 ACM Conference on SIGCOMM*, SIGCOMM ’15, (New York, NY, USA), pp. 493–507, ACM, 2015.
[31] N. K. Sharma, A. Kaufmann, T. Anderson, A. Krishnamurthy, J. Nelson, and S. Peter, “Evaluating the Power of Flexible Packet Processing for Network Resource Allocation,” in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, (Boston, MA), pp. 67–82, USENIX Association, 2017.
[32] Antlr, “http://www.antlr.org/.”
[33] C. E. Hopps, “Analysis of An Equal-Cost Multi-Path Algorithm,” 2000.
[34] NS2, “http://www.isi.edu/nsnam/ns/.”
[35] QJUMP-NS2, “https://github.com/camsas/qjump-ns2.”
[36] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella, “Presto: Edge-based load balancing for fast datacenter networks,” in *Proceedings of the 2015 ACM Conference on SIGCOMM*, SIGCOMM ’15, (New York, NY, USA), pp. 465–478, ACM, 2015.
[37] C. Trois, M. D. D. Fabro, L. C. E. de Bona, and M. Martinello, “A Survey on SDN Programming Languages: Toward a Taxonomy,” *IEEE Communications Surveys Tutorials*, vol. 18, no. 4, pp. 2687–2712, 2016.
[38] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, “OpenState: Programming Platform-Independent Stateful OpenFlow Applications Inside the Switch,” *acm special interest group on data communication*, vol. 44, no. 2, pp. 44–51, 2014.
[39] M. Moshref, A. Bhargava, A. Gupta, M. Yu, and R. Govindan, “Flow-Level State Transition as a New Switch Primitive for SDN,” in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN ’14, pp. 61–66, ACM, 2014.
[40] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, “The Design and Implementation of Open vSwitch,” in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, (Oakland, CA), pp. 117–130, USENIX Association, 2015.

Fei Chen received his B.S. degree from North-Eastern University, China, in 2012, and the Ph.D. degree from Zhejiang University, China, in 2017, all in computer science. His research interests span networking and distributed computing, with a recent focus on software-defined networking, data-plane programming, and datacenter networks.

Chunming Wu received the Ph.D. degree in computer science from Zhejiang University in 1995. He is now a Professor in the College of Computer Science and Technology, Zhejiang University. His research fields include software-defined networks and network virtualization. He has published over 80 peer-review papers at international conferences and journals, which include INFOCOM, IWQoS, GLOBECOM, IEEE/ACM TRANSACTIONS ON NETWORKING, IEEE Communications Magazine, Elsevier Computer Networks, etc.

Xiaoyan Hong is an Associate Professor in the Department of Computer Science at the University of Alabama and directs the Wireless, Mobile and Networking Research Lab (WiMaN). She received her Ph.D. degree in Computer Science from the University of California at Los Angeles in 2003. Her research interests include mobile and wireless networks, connected vehicular and transportation systems, underwater acoustic communication networks and software defined networks.

Bin Wang is now the director of network and information security laboratory of Hangzhou Hikvision Digital Technology Co.,Ltd. His current research is in next generation network technology, information security, IoT security and network routing.