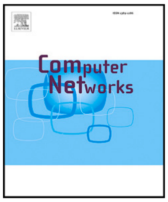
Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet



Survey paper

A survey on stateful data plane in software defined networks

Xiaoquan Zhang [a], Lin Cui [a,*], Kaimin Wei [a], Fung Po Tso [b], Yangyang Ji [a], Weijia Jia [c]

[a] *Department of Computer Science, Jinan University, Guangzhou, China*
[b] *Department of Computer Science, Loughborough University, LE11 3TU, UK*
[c] *BNU-UIC Institute of Artificial Intelligence and Future Networks, Beijing Normal University (BNU Zhuhai) and BNU-HKBU United International College, Zhuhai, China*

ARTICLE INFO

Keywords:
SDN
Stateful data plane
Data plane programmability
P4
OpenState

ABSTRACT

Software Defined Networking (SDN), which decouples control plane and data plane, normally stores states on controllers to provide flexible programmability and convenient management. However, recent studies have shown that such configuration may cause frequent and unnecessary interactions between data plane and controllers in some cases. For example, a DDoS detection installed on a controller needs to fetch information from data plane periodically, introducing additional network delay and controller overhead. Thus, stateful data plane is proposed to offload states and operation logics from controller to data plane. Stateful data plane allows switches to perform some operations independently, accelerating packets processing while reducing overhead on both controllers and networks. However, stateful data plane increases the complexity of network devices and imposes many new challenges to the management and schedule of SDN enabled networks. This paper conducts a comprehensive survey on the latest research works and provides insights into stateful data plane. Both stateful data plane platforms and compilers are extensively summarized and analyzed, as well as explicit design of applications based on them. Afterwards, we dwell on the fundamental technologies and research challenges, including implementation considerations of stateful data plane. Finally, we conclude this survey paper with some future works and discuss open research issues.

1. Introduction

Software Defined Networking (SDN) is an emerging network architecture that provides unprecedented network programmability by decoupling the control plane and data plane. OpenFlow [1], the first real implementation of the SDN paradigm, introduces a "match–action" paradigm to the SDN data plane wherein forwarding is simple but efficient. SDN controller, the control plane, can obtain global information of the whole network. This provides a convenient means for operators to adapt to the network dynamism and to extend new features by redefining control loops on the control plane. Clearly, in legacy SDN architecture, the controller manages network states while the data plane is only responsible for forwarding packets, which is also known as *stateless data plane* [2].

With an increasing adoption, it is soon realized that frequent interactions between data plane and control plane bring additional delay and overhead to the network (e.g., the overhead of periodic queries of counters produced by a DDoS detection [3]). On the other hand, experiments show that, even if preserving some network states and performing some simple operations, the data plane can still guarantee high-speed packet forwarding [4]. Thus, functions that require frequent interactions between controllers and data plane for state management can be offloaded to data plane to reduce overhead and increase efficiency. This novel architecture is called *stateful data plane*.

In contrast to the simple "*match–action*" paradigm of OpenFlow, stateful data plane proposes a new "*match-state–action*" paradigm that can both keep and manage states in the data plane [5]. It imposes higher requirements on network devices, such as requiring complex hardware design to support stateful operations (e.g., stateful elements are required in process pipelines [6]). Thus, network switches are no longer "dumb" but more "intelligent". Then, network applications can be deployed in stateful data plane and executed without explicit involvement of controllers [7]. For example, by eliminating delay to controllers, network monitoring implemented on stateful data plane can obtain better accuracy and efficiency [8]. However, managing states in data plane is hard because it is distributed [9]. Each switch keeps its own states, which may be quite distinct from others, then state inconsistency problems may arise. On the other hand, comparing with controller, it is difficult to realize too complex logic in data plane due

[*] Corresponding author.
E-mail address: tcuilin@jnu.edu.cn (L. Cui).

https://doi.org/10.1016/j.comnet.2020.107597
Received 25 May 2020; Received in revised form 8 October 2020; Accepted 8 October 2020
Available online 10 October 2020

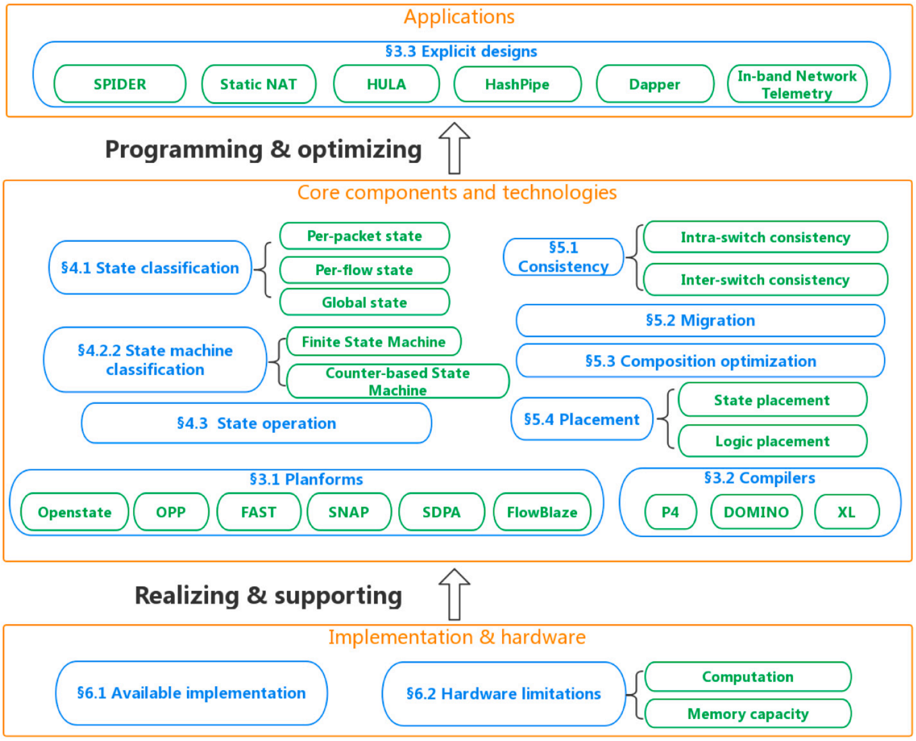


Fig. 1. Condensed overview of this survey on stateful data plane in SDN.

to limitations of network devices (e.g., difficult to implement multiplication/division while promising per-packet process speed [10]). Hence, stateful data plane also faces some challenges. For example:

1. Capabilities of network devices limit the implementation of complex stateful applications [11]. For example, applications only obtain limited resources of switches (e.g., memory).
2. A common interface for stateful data plane is still missing, making it a great challenge when implementing and deploying stateful application. For example, methods to locate state positions are varied in different hardware implementations [12].
3. Switch hardware structural design still have some issues when implementing stateful data plane, for example, race condition problem in pipeline process [6].

In fact, stateful data plane has received increasing attentions from research community. Shaghaghi et al. [13] analyzed three main types of vulnerabilities in stateful data plane, namely: unbounded flow state memory allocation, lack of authentication mechanisms and a central state management. They also gave some basic recommendations to cope with vulnerabilities of stateful data plane, including state consistency check and secure in-band signaling. Dargahi et al. [14] proposed schemes for stateful data plane and analyzed the vulnerabilities of existing stateful data plane proposals for security issues. They concluded that the possible vulnerabilities should be carefully taken into consideration in designs of current and future proposals for stateful data planes. Bifulco et al. [15] proposed a simple state classification, including packet state and global state, to define the data plane as stateful or stateless according to whether the global state is admitted to write. They also argued that finding an expressive yet simple model to handle state operations in the data plane is important. Kaljic et al. [16] pointed out that the future research direction of stateful data plane is the development of fully synchronized stateful data plane supporting state monitoring and management.

In this paper, we present a more comprehensive survey on stateful data plane, ranging from fundamental techniques of stateful data plane to existing platforms and applications. The comparison with other works is shown in Table 2. The taxonomy of stateful data plane discussed in this paper is shown in Fig. 1. The main contributions of this paper are as follows:

1. An extensive review of schemes on stateful data plane, and summary emerging stateful applications classified by distinct state machine.
2. An explicit definition of stateful data plane, and detailed exploration of basic components of stateful data plane to show flexibility and programmability.
3. A comprehensive analysis of schedule and optimization technologies and implementation considerations for stateful data plane, as well as future research directions of stateful data plane.

The remainder of this paper is organized as follows: Section 2 briefly provides an overview of stateless and stateful data plane. Section 3 lists existing platforms and applications and summarizes their features. Section 4 introduces basic components of stateful data plane. Section 5 summarizes schedule and optimization technologies and Section 6 discusses implementation considerations for stateful data plane. Finally, Section 7 summarizes several issues for future research directions on stateful data plane and Section 8 concludes this paper.

2. Overview of stateless and stateful data plane

In this section, we briefly outline the concepts of stateless data plane and stateful data plane, followed by a load balancer example to show their differences. Several commonly used acronyms in this paper are listed in Table 1.

2.1. Stateless data plane

The most prominent feature of traditional SDN is that the network control and states (i.e., the control plane) are centralized on one or more controllers for flexible network management. Switches send any required information to controller via *PACKET_IN* messages. Upon receiving those messages, controller updates network states. Such updates

Table 1
Main acronyms.

SDN	Software Defined Networking
DDoS	Distributed Denial of Service
TCAM	Ternary Content Addressable Memory
BRAM	Block Random Access Memory
SRAM	Static Random Access Memory
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
EFSM	Extended Finite State Machine
DAG	Directed Acyclic Graph
BMV2	Behavior Model v2
ALU	Arithmetic and Logic Unit
NAT	Network Address Translation
INT	In-band Network Telemetry
DSL	Domain Specific Language
ASIC	Application Specific Integrated Circuit
DPDK	Data Plane Development Kit
RDMA	Remote Direct Memory Access
NIC	Network Interface Card

may trigger actions such as modifying or installing new rules to the flow tables in switches. In comparison, data plane in switches only carries out simple "match–action" operations according to flow table without conducting any state management. This is referred as stateless data plane.

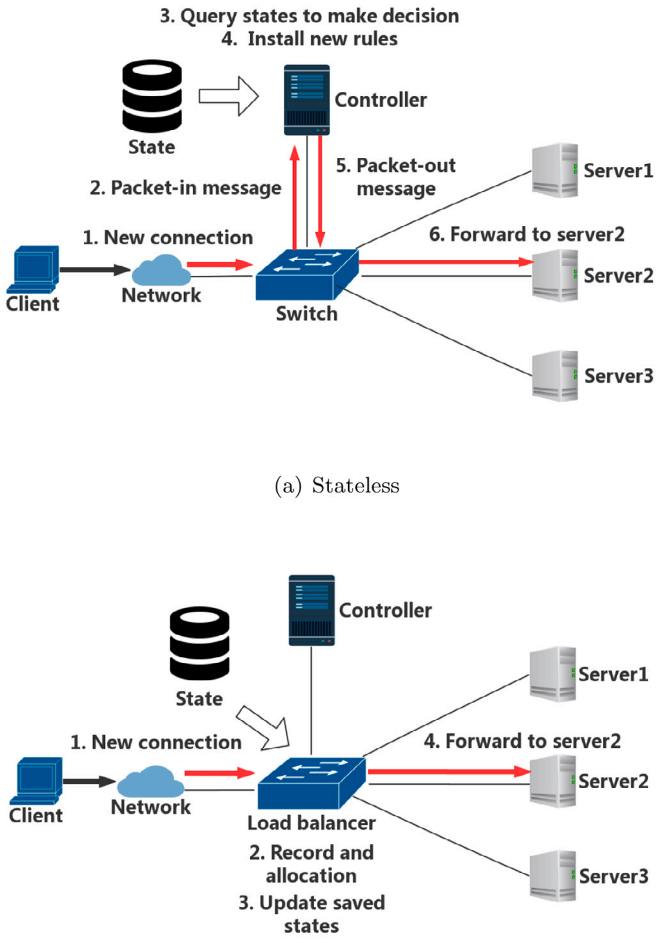
The development of traditional SDN [17–20] is becoming increasingly more mature as evidenced by many proposed and developed SDN frameworks such as NOX [21], POX [22], OpenDayLight [23], Floodlight [24] and Beacon [25]. The original OpenFlow versions adopt a simple and efficient "match–action" abstraction in data plane. This abstraction uses fixed match fields and corresponding actions to effectively forward packets. Network devices of stateless data plane are usually "dumb" devices.

However, stateless data plane, albeit simple and fast, can inevitably suffer from additional network delay and overhead, specially for applications that require frequent read/write on states, e.g., heavy hitter [27], where it needs to stop and wait for decisions from the controller. There are some attempts to fix this problem. For example, Han et al. [29] proposed a method that allows switches to indirectly participate in the management of states without storing states directly. To avoid the controller being a bottleneck, an open connection table is designed to manage states information interaction between controller and data plane. However, due to inevitable network delay, controller may still fail to update states in time, making decisions based on the out-of-date states.

2.2. Stateful data plane

Unlike stateless data plane, stateful data plane allows network switches to directly operate their own states with few or no intervention of controller. In this paradigm, a controller offloads two main components to data plane: state management and processing logic. With this, some network functions can be partially or even fully implemented in data plane. In addition, to correctly model these network functions, state machines (see details in Section 4) are introduced for analysis in stateful data plane. Hence, we define three key characteristics of stateful data plane:

1. **States can be stored in data plane**. This reduces data transmitted to controllers over the network. These states are meaningful for functions deployed on switches. For example, switches in HULA [30] save state information that indicates the next best hop for every entering flow.
2. **Data plane can update its own saved states**. For example, increasing a counter value when a given condition happens during packet processing. Thus, the decision-making policy of switches can be changed accordingly, which will affect the processing of subsequent received packets. This feature requires switches to offer certain capability of various state operations.
3. **Data plane is programmable**. Data plane has limited programmability and is able to change operation logics when needed.



(a) Stateless

(b) Stateful

Fig. 2. Load balancer in stateless and stateful data plane.

It is worth noting that there exist some works that only keep states on switches without providing state operations on the data plane. Such architectures are not classified as stateful data plane in this paper. For example, Nife et al. [31] provided the flow state-aware using tables to keep track of each specific flow. However, it only allows the controller to maintain states rather than the switch itself.

2.3. Stateless vs. stateful

Table 3 compares main features and properties of both stateless and stateful data plane. In addition to differences mentioned above, stateful data plane is distinct from stateless data plane in some aspects. The controller can provide global decide-making optimization to network. However, applications installing in stateless data plane requires the controller frequent involvement, which is undesirable since controller's design is not involved real-time packet processing but is supposed to generate rules. Moreover, overhead and latency of communication between switch and controller increases with the increase of the number of switches in stateless data plane, since more requests the controller needs to handle. Conversely, offloading simple logic in stateful data

Table 2
Comparison with other survey papers.

Works	Published year	Main focus	Content of schemes and applications	Content of fundamental technologies analysis	Content of hardware limitations
Dargah et al. [14]	2017	Security of SDN stateful data plane	Introducing several typical schemes and applications	×	×
Shaghaghi et al. [13]	2018	Security of SDN data plane	Simply introduce few schemes. No applications	×	×
Bifulco et al. [15]	2018	Programmable data plane	Simply list schemes. No applications	State management	×
Kaljic et al. [16]	2019	Programmability in SDN data plane	Simply introduce few schemes. No applications	State management	×
Our work		Comprehensive survey of SDN stateful data plane	Elaborating schemes and applications	Comprehensive analysis	✓

Table 3
Stateless data plane and stateful data plane.

Description	Stateless data plane	Stateful data plane
State storage and management	Centrally stored and managed by the controller	Each switch stores and manages its own states
Switch-Controller communication	Frequent and essential	Stateful updates do not required to be triggered by controller
Network devices	"Dumb" [26], controller takes over logic	"Smart", switches can implement certain logic operation
Computing power	Controller provides global decision-making optimization and complex mathematical computation.	Controller can offer external computation for switches which only support simple mathematical operations and bitwise operations [27]
Effect of the number of switches	More switches means more overhead for controller.	The number of switches does not affect the overhead of the controller [14]
Application diversity	Support various applications in controller	Combining controller and switch makes more meaningful application [28]
Application placement	Only in controller	Mainly in switches (controller might be involved)

plane, the controller can preserve resources for more essential operations. Hence, due to logic handling in data plane, the controller would not be effected by the number of switches.

Besides completely installing in data plane, applications in stateful data plane also have an alternative option, which is installing in the controller and switches. Netcache [32] proposed a novel key–value store architecture for cloud services. In netcache, simple logic is installed in data plane to improve packet forwarding and processing performance, and the controller is responsible for updating cache in switches. The combination of the controller and data plane allows programmers to explore various and even more meaningful applications.

To show their differences clearly, Figs. 2(a) and 2(b) illustrate implementation of a load balancer with stateless and stateful data plane respectively. For the stateless implementation in Fig. 2(a), when a new connection arrives, the switch will send request to controller (step 2). The controller will make decision according to states it maintains and install new rules in the switch (step 3 and 4). Finally, the switch forwards packets to the assigned server (step 6). Unlike stateless implementation, the processing of stateful data plane is performed without involvement of the controller, as shown in Fig. 2(b). Upon receiving a new connection, the switch (load balancer) records information extracted from the packet header and assigns a server based on predefined logics, e.g., polling algorithm (step 2). Then, it updates its local states (step 3) and forwards packets to the assigned server (step 4).

Indeed, the legacy SDN architecture brings sufficient computing resources on the centralized controller for network management, and simple but effective abstraction on stateless data plane for rapid forwarding. However, the lack of independence causes that the data plane needs to communicate with controller frequently, which leads to overhead and potential bottleneck in the controller. By offloading state operations and logics to the data plane, stateful data plane greatly reduces response time and communication overhead between controller and data plane. For example, a port knocking application (Fig. 8) in the data plane can enormously avoid the network latency between controller and data plane [26].

Stateful data plane also has some limitations. For example,

1. **Some operations cannot be easily implemented in stateful data plane.** One typical example is applications that require modifying packet payload, e.g., intrusion detection application that needs to analyze information in payload [33].
2. **Complicated state recovery mechanism**. States stored on switches need to be restored when the switch fails. However, in order to ensure consistency and efficiency, existing state synchronization/migration methods have to make tradeoff between complexity of mechanisms and the time interval of recovery [12, 34].
3. **Complicated application design**. Since stateful data planes have more severe memory and computing constraints than stateless data planes, programmers sometimes need to consider these hardware limits and made some tradeoff on performance when designing applications on stateful data plane [10].

3. Existing platforms and applications for stateful data plane

In this section, we survey various existing platforms and compilers for stateful data plane, as well as several representative applications based on stateful data plane.

3.1. Stateful data plane platforms

Table 4 compares existing platforms of stateful data plane from six aspects: state machine implementation, implementing software switch, implementing hardware, hardware storage, global state in registers and controller involvement.

Table 4
Summary of stateful data plane platform.

Platform	State machine implementation	Implementing software switch	Implementing hardware	Hardware storage	Global state in registers	Controller involvement	Throughput
OpenState	EFSM table	OpenState softswitch [35]	FPGA-based [36,37]	TCAM RAM	×	Initiation & auxiliary state update	Unknown
OPP	EFSM table	CPqD OF1.3 switch [38]	NetFPGA SUME [39]	TCAM RAM Register	✓	Initiation & auxiliary state update	10~80 Mpps
FAST	EFSM table	Open vSwitch [40]	N/A	TCAM SRAM	×	Supplementary computation and storage	Unknown
SDPA	FP [5]	Open vSwitch [40]	ONetCard [41]	TCAM RAM	×	FP initialization [5]	0.5~10 Gbps
SNAP	xFDD [42]	NetASM's supported switch [43]	N/A	CAM Register	×	States placement	Unknown
FlowBlaze	EFSM table	mSwitch [44] eBPF/ XDP [45]	NetFPGA SUME [39]	BRAM TCAM Register	✓	Initiation & auxiliary state update	14.8 Mpps
Banzai	Match–action table	N/A	Specific hardware [11]	TCAM SRAM	×	×	Unknown

Benzai is compiled with DOMINO introduced in Section 3.2.2.

3.1.1. OpenState

OpenState [26] extends OpenFlow to configure stateful data plane, and has been implemented as an OpenFlow 1.3 experimenter extension. OpenState provides the ability to configure custom states inside switches and program how states should be evolved. OpenState relies on a simplified EFSM abstraction (Extended Finite State Machines), named Mealy Machine [46]. The EFSM is modeled as a 4-tuple (S, I, O, T), plus a default state S_0. S refers a finite set of states. I is a finite set of input symbols (events). O is a finite set of output symbols (actions). $T : S \times I \rightarrow S \times O$ is a transition function which maps $\langle state, event\rangle$ pairs into $\langle state, action\rangle$ pairs. OpenState simply provides programmers with different header fields to access to the state table: "lookup-scope" is used to access a state table for lookup operations, and "update-scope" is used to update the state table. The state table stores states of flows identified by a unique key composed of a subset of the information stored in the packet header (e.g., an IP address, a source/destination MAC pair, a 5-tuple flow identifier). EFSM table is used to implement state machines, which determines match keys and actions. Packets would trigger corresponding actions and state modifications after going through the EFSM table.

OpenState is one of early proposals for stateful data plane. It proposes primitives for switches to handle flow states and provides high programmability for users to express their network requirements [47]. However, since OpenState only provides a simple model that supports limited actions, it lacks expressiveness and can only abstract limited state machine [48].

3.1.2. OPP

OPP (Open Packet Processor) [4] proposes a programming abstraction which retains the platform independent features of the original "match–action" abstraction while bringing programmability of stateful packet processing tasks into network switches. The stateful process of OPP consists of four stages. In stage 1, a Flow Identification Key (FK) is extracted from packets and it identifies the entity to which a state may be assigned. The key is used to extract flow context including a state label s and an array of registers $R = \{r_0, r_1, \dots, r_{k-1}\}$. In stage 2, Condition Block is in charge of implementing the enabling functions specified by the EFSM abstraction according to compute conditions. Global registers are delivered to this block as an array $G = \{G_0, G_1, \dots, G_h\}$ of global variables and/or global switch states. The output of the condition block is a Boolean vector $C = \{c_0, c_1, \dots, c_{m1}\}$. In stage 3, EFSM Table describes the transition of a state machine. After matching in this table, output has three types: setting next state, executing actions, and updating registers. In state 4, Update Logic Block implements an array of Arithmetic and Logic Units (ALUs), which allows programmers to update the value of the registers (array R or array G).

OPP is an improvement of OpenState, providing more explicit executing actions and adding registers to expand programmability. In addition, OPP enriches EFSM formal notation, which permits programmers to implement more meaningful applications. However, some limitations of OPP architecture exist [4]. Firstly, OPP does not support some asynchronous events that trigger translation of states, e.g., timer's expiration. Secondly, OPP only deploys the ALU processing in the Update Logic Block for a cleaner abstraction and a simpler implementation, which means that it only supports simple calculations. Bianchi et al. [4] also discussed the hardware feasibility and devised a specific hardware for OPP architecture based on FPGA prototype.

3.1.3. FAST

FAST (Flow-level State Transitions) [27] is a new switch abstraction that allows operators to program their state machines for a variety of applications in data plane. FAST includes two key designs: control plane and data plane. The control plane compiles state machines for specified switch using high-level abstraction, and data plane is responsible for forwarding packets according to the state machines compiled by control plane. In the data plane, the implementation of state machines mainly bases on four tables: State table, State machine filter, State transition table and Action table. The State machine filter is used to filter different types of traffic. The State table stores the current state for each flow. It is worth pointing out that FAST decouples State transition table and Action table from the EFSM table, which offers more flexible programmability. The State transition table submits a received packet to Action table based on matched conditions. The Action table will execute specified actions, and the State table will update corresponding states based on the "next state" in state transition table (see Fig. 3). In FAST, controller is used to mitigate limitations of data plane. Due to insufficient computational complexity in data plane, FAST allows switches to upload unsupported computations (e.g., average) without guaranteeing the time interval of interaction with controller. In addition, controller can save states that are rarely used, which decreases memory consumption on switches.

3.1.4. SNAP

SNAP (Stateful Network-Wide Abstractions) [9] offers a high-level language that provides a simple "centralized" stateful programming model to achieve a stateful network-wide abstraction. In SNAP, a subset of switches is chosen for array placement, and other switches can still play a role in routing flows through state variables. By accessing and

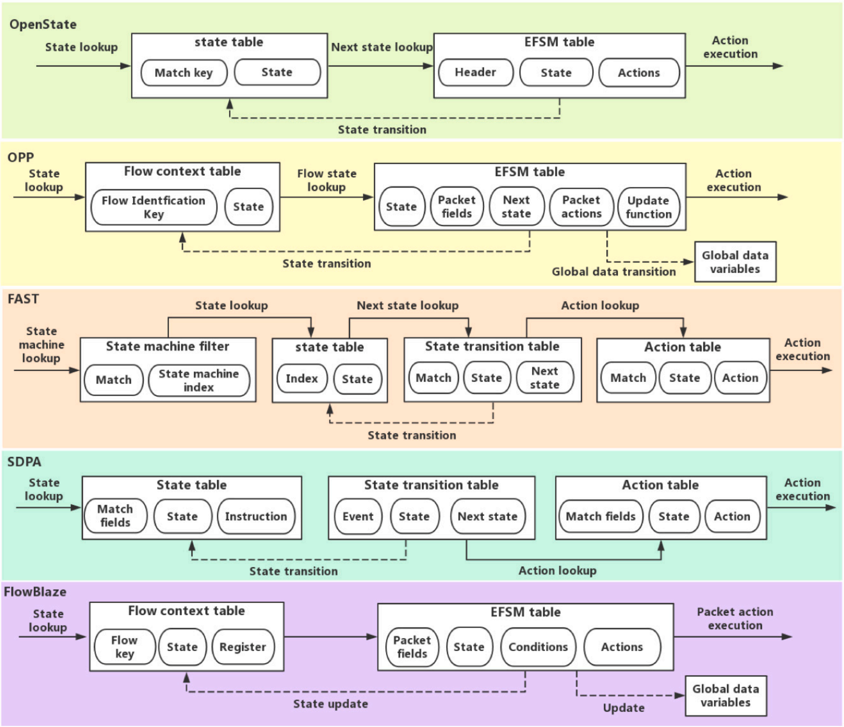


Fig. 3. Tables used in OpenState, OPP, FAST, SDPA and FlowBlaze architectures. Each table is represented as a rectangle containing the table name and corresponding table columns enclosed by smaller rounded rectangles. Dotted lines represent state transitions.

modifying the state stored in corresponding switches, a broad range of applications from firewall to fine-grained traffic monitoring can be implemented. Two key details in SNAP program are state placement and traffic routing. To support stateful packet processing, SNAP uses intermediate representation which is called extended forwarding decision diagram (xFDD) [42]. The SNAP program will be converted to xFDD, which determines state variables and processing operations for each packet before forwarding the packet to corresponding output port. In order to enforce state placement and traffic routing, the compiler uses a mixed-integer linear program (MILP) to decide state placement and routing.

In SNAP program, states storage is not distributed on each switch, but centralized on one switch. The main reason is that it is hard to simultaneously provide strong consistency when updating state variables. Although this centralized storage method saves a lot of storage spaces and reduces the complexity of network, it lacks reliability. Especially, when the switch storing states fails, these states are unrecoverable.

3.1.5. SDPA

SDPA (Stateful Data Plane Architecture) [5] is a platform that enables effective programming and stateful processing in data plane. SDPA proposes a "match-state–action" paradigm for the data plane. More specifically, a co-processing unit in switches named Forwarding Processor (FP) is designed to manage states. The FP includes three tables: state table (ST), state translation table (STT) and action table (AT). ST is used to store states. Applications deployed on controller can send messages to FP for dynamically initializing ST if stateful processing is needed. STT is designed to support stateful processing, which is configured by controller only once during initialization. STT contains three domains: state, event and next state. The state domain matches current states; the event domain makes a comparison with packet's flag or states to trigger state transition; and the next state domain can be a specified state or mathematical/logical operations (e.g., $state+1$). AT is used to record actions for incoming packets under different flows and it may transit the corresponding state in the ST.

SDPA proposes a more complicated but effective architecture for stateful processing. It supports processing of different applications by preserving and separating state information of different applications, which facilitates the requirement of isolation between different applications for programmers. States on an overloaded switch can be migrated with SDPA. Two types of migrations are considered in SDPA: (1) migration from one switch to another, (2) migration from controller to the switches, since SDPA allows applications to initiate in state table.

3.1.6. FlowBlaze

FlowBlaze [6] is an abstraction for designing stateful packet processing functions implemented on NetFPGA SmartNIC [39]. It achieves high performance and low latency while consumes very few power on newer FPGA models. In FlowBlaze, an Extended Finite State Machine (EFSM) [56] is introduced to build functions specified by users. The FlowBlaze machine model consists of stateless element and stateful element. Stateless element is the same as a "match–action" table. And stateful element is split into Flow Context Table, EFSM table and update function. The Flow Context Table is used to save states of flows. The EFSM table is used to implement functions abstracted by EFSM. And the update function is responsible for executing the state update. Besides, a small stash memory [6] is used to handle hash collision when too much entries inserting into hash table, which provides scalability for the Flow Context Table. FlowBlaze solves the race condition problem by a simple scheduler scheme to guarantee the consistency of flow states.

FlowBlaze is a novel abstraction and mature stateful data plane platform implemented on hardware. It has been used to provide better performance to network functions in some projects, e.g., VPP functions [57]. It also discusses and solves several typical problems in stateful data plane, e.g., consistency and large amounts of flows. Although some issues are not resolved completely, FlowBlaze offers many inspirations for future designs of stateful data plane platform.

3.2. Stateful data plane compilers

In this section, we list some existing compilers for stateful data plane. Table 5 compares existing compilers, from six aspects: language

Table 5
Summary of stateful data plane compiler.

Compiler	Language type	State machine implementation	Implementing software switch	Implementing hardware	Stateful memories	Atomic operation
P4	Domain-specific language	DAG	BMV2 [49] PISCES [50]	RMT [51] dRMT [52] P4FPGA [53] P4-> FPGA [54]	Register [55]	✗
DOMINO	C-like imperative language	DAG	Banzai [11]	N/A	Atom [11]	✓
XL	Domain-specific language	EFSM	DPDK	FlowBlaze	Register	✗

DAG: Directed Acyclic Graph.

type, state machine implementation, implementing software switch, implementing hardware, stateful memories and atomic operation.

3.2.1. P4

P4 [58] is a high-level language for programming protocol-independent packet processors, which enables flexible reconfigurability in the field. In P4, programmers can not only devise header fields, but also define the packet parsing and processing in the fields. The processing of packets in P4 has four major phases: (1) Parsing of packet, a packet must be translated into a representation that can be processed in the next phase when it enters switch. In the meanwhile, the parser recognizes fields from header and extracts them for processing in the next stage but does not distinguish what protocol it is. (2) Apply the "match–action" table to ingress, the "match–action" table is divided into ingress and egress, both may modify packet headers. Ingress "match+action" may determine the egress port. Based on ingress processing, the packet may be forwarded, replicated, dropped, etc. (3) Apply the "match–action" table to the egress, which performs per-instance modifications to the packet headers, e.g., for multicast copies. (4) Deparsing, packets undergo decomposition as well as processing. After processing, the packet should be deparsed based on its current states before final forwarding.

Exploiting stateful applications in P4 is convenient. First, they are no longer limited by the fixed match fields (e.g., OpenFlow 1.3.4 supports 40 matching fields) as P4 enables flexible definition of headers. Second, P4 allows programmers to use metadata to transmit states in different stages, which provides great convenience for delivering state. Third, P4 version 1.1 [55] introduces a special stateful memory called registers. Registers can be defined as a global state accessed by multiple flows, and they can also be used to implement small dictionaries, or a hash table as sparse dictionaries.

3.2.2. DOMINO

DOMINO [11] is a data plane programming language which aims to achieve line-rate programmability for stateful algorithms. In order to simplify some sophisticated data plane algorithms, DOMINO introduces a new packet transition: a sequential packet-processing code block. A code block is atomic and isolated from other code blocks. It means that a packet only needs to consider its own processing without interference from the processing of other packets. Thus, DOMINO guarantees that packet process runs at line rate. DOMINO also introduces a machine model, named Banzai, for programmable line-rate switch. There are two constrains in the Banzai model: (1) different packet-processing units cannot share states; (2) any switch state modification is required to be visible to the next packet entering the switch. These two constrains ensure that code block is atomic. Thus, *atom* is introduced for storing and modifying states. It is a vector of processing unit used to handle stateful packet processing in Banzai, and each pipeline stage contains a vector of *atom*.

DOMINO guarantees state consistency and performance at the cost of limited flexibility. The atomic operation is unable to conduct tasks that cannot complete within the limited time budget of a single pipeline's stage [59]. DOMINO tries to explore a once-and-for-all way to cater to vendors' closed platform while provides an open programmable architecture, which compromises its flexibility [60].

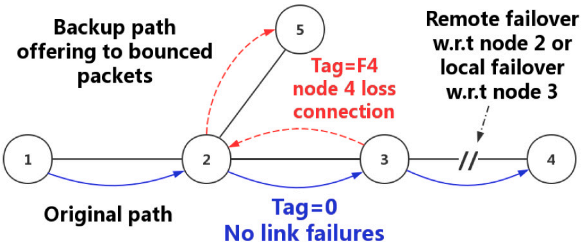

Backup path offering to bounced packets
5
Tag=F4
node 4 loss
connection
Remote failover w.r.t node 2 or local failover w.r.t node 3
1
2
3
4
Original path
Tag=0
No link failures

Fig. 4. A failover example [2]. Packets attaching to tag = 0 are forwarded along blue-solid lines to the original path, otherwise, packets attaching to tag = F4 are forwarded along red-dotted lines to the detour path.

3.2.3. XL

XL (XFSM Language) [61] is developed for the description of per-flow stateful network functions. It is suitable for FlowBlaze [6] to describe stateful and stateless network functions in the EFSM table. Since many platforms can execute network functions abstracted as EFSM, XL becomes a general compiler using EFSM abstraction, providing platform independent portable code. Furthermore, XL provides a compiler xlc to compile user's code into a JSON representation, which can be loaded into hardware or software (e.g., DPDK).

Compared with P4 and DOMINO using DAG as their abstractions, XL enriches the EFSM abstraction programming for hardware platforms that relies on the EFSM table, while it can also support programming in software. On the other hand, XL also provides a convenient way to compose multiple network functions for reducing the complexity of application design.

3.3. Applications based on stateful data plane

In this section, five explicit applications are introduced to show the programmability of stateful data plane. Due to page limitations, more other applications can be found in Table 6.

3.3.1. SPIDER

SPIDER [2] implements a link failure recovery that offers programmable detection and link reroute. It designs a packet processing pipeline implemented on OpenState. By sending probe packets to adjacent switches, SPIDER provides a recovery mechanism that has short delay of recovery and failure detection. However, it should consider the trade-off between overhead and probe packet frequency. SPIDER proposes two kinds of failover: local failover and remote failover. When a switch perceives that its neighbor is unreachable, this is a local failover. Remote failover refers that a switch receives a packet that indicates a failover happening at remote switches other than neighbors, and the packet is sent from the node with local failover (shown in Fig. 4). Moreover, SPIDER sends heartbeat packets to monitor whether adjacent nodes are alive.

SPIDER devises four different tables on OpenState. The *table 0* and *table 1* perform stateless forwarding (e.g., legacy OpenFlow), while *table 2* and *table 3* implement remote failure finite state machine (RF

Table 6
Summary of stateful data plane applications and state machines.

Application	Type	Stateful data plane solution	State transition	Ref.
Link failover	FSM	Switches save backup path and monitor link's status	Switches would notice that its adjacent switch is down and the deroute path is activated	[2,62,63]
Load balancing	FSM	Switches share network traffic with multiple links	By polling, or by the link information attached by received packets	[30,64,65]
NAT	FSM	Keep a state machine in the NAT switch for every flow	Allocate internal addresses in turn when receiving the first packet of a new flow	[7]
Stateful firewalls	FSM	Switch filters unsolicited inbound TCP connections without any outboard flow	Determine the legality of source and destination addresses, otherwise discard	[9,66]
Port knocking	FSM	Switches maintain a state machine for every flow determining whether it can obtain the permit	Determine if this connection can pass a series of specific ports in order, otherwise discard.	[4,47]
DNS detection	EFSM	Assign a counter to keep track of all the resolved IP addresses for clients	The client is judged to be a malicious user if its accesses exceed the threshold	[5]
DDoS detection	EFSM	Switches count the features of the background traffic to detect potential attacks	If the counter exceeds the threshold, it would be signed as a attacker	[27,67,68] [69–71]
Super-spreader detection	EFSM	Detect flows from one source with TCP connection exceeding threshold [72]	Calculate counters of connection number of a source and send to controller if it exceeds threshold	[5,9,27]
Flow size counter	EFSM	Report to the controller after the completion of collecting the size of flow in data plane	Keep a counter of flow size and send to the controller upon received FIN signal	[27,73,74]
SYN-flood detection	EFSM	Switches maintain a counter for every flow to detect SYN-Flood	Compare the counter of SYN packets saved in data plane with the threshold	[5]
Heavy-hitter detection	EFSM	Save a counter for every flow	The switch uploads every counter to controller once the counter exceeds the threshold	[75–77]

FSM) and local failure finite state machine (LF FSM) respectively. More specifically, when remote failure happens in the network, the *table 2* is responsible for the transition of state (from normal to F_i), and reroutes packets to backup path. The state transition can also be triggered upon receiving a bounce packet, which is sent back across its original path until it arrive the switch 2, shown by the red dotted line in Fig. 4. The *table 3* mainly processes heartbeat packets to detect if adjacent nodes enable communication and implements the FSM with two macro states: *UP* and *DOWN*. Initially, all neighbors are in state *UP* and need heartbeat. When the first packet is matched in this table, the state will be updated to *UP: heartbeat request*, which indicates that packets will be normally forwarded to primary path and the switch is waiting for heartbeat packet. Otherwise, if a heartbeat reply is time-out, the state will be updated to *DOWN: need probe*. Under such state, packets would be forwarded to detour and the switch will persistently send the probe to monitor status of links.

Solving failover problems in stateless data plane mainly rely on the reaction of the controller. Links may not be recovered in time and packet loss happens when the latency between switches and the controller rises. SPIDER utilizes the collaboration between different switches to rapidly recover network while considerably mitigates packet loss.

3.3.2. Static NAT

Bonola et al. [7] implemented a static NAT based on OPP platform. The static NAT keeps track of states for TCP connections in round-robin fashion for assigning TCP connections to a set of web servers in a private LAN. The core of this design is to track two states: global state and per-flow state. More specifically, static NAT requires two stateful tables (table 0 and table 1) and one stateless table (table 2). The table 0 mainly enforces initialization of a new flow upon receiving its first packet. When submitting to the next table, the value of the global variable G_0 is used as the metadata labeled in the packet. Once the flow is bound to its assigned server through its first packet, subsequent packets will be forwarded to the same server. The table 1 is used to translate destination address from external address to one of internal server's addresses. The table 2 processes stateless forwarding on reverse direction. A concise description of creating a new flow is that the first packet of the new flow entering table 0, which will be labeled as a metadata with the value of G_0 for enforcing server assignment decision. Then it will be submitted to the next table for assignment based on the last bit of packet's metadata.

Some network functions repeat simple logic to every flow (e.g., port knocking), which can be completely implemented in data plane to avoid the communication overhead between the controller and switches.

3.3.3. HULA

HULA [30] is a scalable load balance scheme written in P4. HULA adopts ECMP (Equal Cost Multipath) strategy, and performs the distance vector algorithm in switches. HULA uses the distribution of network link utilization information to obtain the best next hop and uses probe packets to advertise its own link status. Moreover, the information stored in switch gives the best next hop towards any destination instead of calculating the whole path for every flow.

Two key phases in HULA are processing probe packets and flowlet forwarding. (1) The HULA probe packet, which carries the value of maximum link utilization, is forwarded to all paths for updating information on the switch. (2) Flowlet forwarding, which prevents packet reorder, uses a hash table to save information: the last time a packet was seen for the flowlet, and the best hop assigned to that flowlet. To achieve these phases, HULA devises a new header, a metadata for probe packets, and several register arrays. The $hula\_header$ consists of two fields: dst_tor and $path\_util$. The dst_tor represents the destination of the packet, and the $path\_util$ represents the path utilization of previous switches. The nxt_hop in metadata devised for a normal packet represents the best next hop of this packet, which would be modified after the process of pipeline. Besides, in order to support stateful operations in data plane, five state variables (registers) are defined: min_path_util, $best\_hop$, $update\_time$, $flowlet\_hop$, and $flowlet\_time$. (1) Both $flowlet\_time$ and $update\_time$ record the last time at which states are changed. For example, when a packet arrives, it needs to update

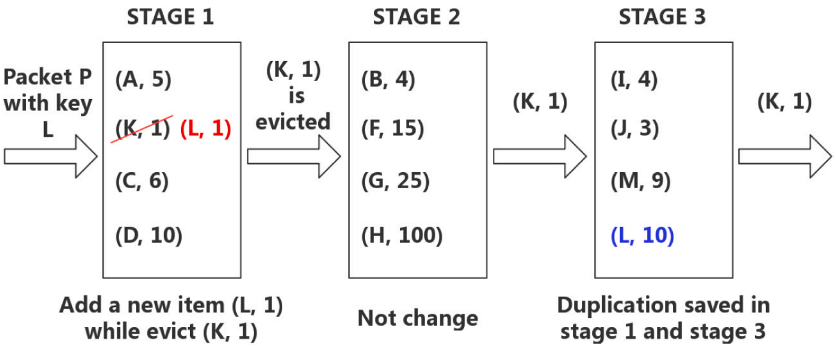


Fig. 5. HashPipe [75] consists of several stages. In stage 1, a new flow (e.g., flow with key L) will initiate a new item and replace an existing item with the minimum value (e.g., item with key K). Afterwards, the replaced item will be evicted to the next stage and same operations will conducted. Lastly, an item would be deleted from the three stages. HashPipe may generate duplications since different stages may save same flows (e.g., item with key L simultaneously exists in $stage1$ and $stage3$).

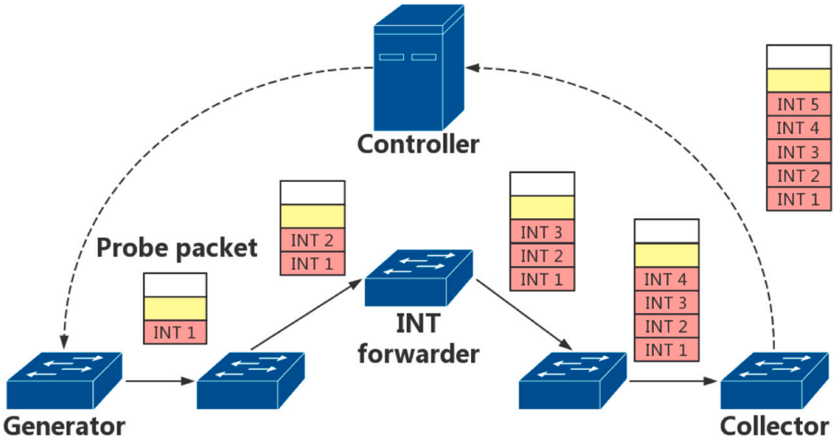


Fig. 6. An example of INT execution [78].

the time of the flowlet in which the packet is located. (2) $flowlet\_hop$ variable records the next hop of packet to avoid packet reordering. (3) *best_hop* tracks the best next hop, and can be changed if the value of the *min_path_util* is updated. (4) *min_path_util* stores the information of utilization on the best path, which is updated by the maximum value comparing with the specified state stored in header fields of the incoming probe packet.

Implemented with P4 in stateful data plane, HULA beats previous load balanced algorithms implemented in custom silicon on a switching chip (e.g., CONGA [79]), which have a long hardware design periods and cannot be modified once implemented.

3.3.4. HashPipe

Sivaraman et al. [75] proposed a heavy-hitter detection algorithm, HashPipe, implemented based on P4. The HashPipe tracks the k-heaviest flows with high accuracy by maintaining both flow keys and counts of heavy flows in data plane (shown in Fig. 5). HashPipe manages multiple stages to store heavy flows' information by dividing the switch process per packet into the first stage and next stages. When a packet arriving in the first stage, its counter is updated if its key matches a hit. If it fails to match, it would be initiated when there are available slots. Otherwise, it will replace the originated key. At the same time, the metadata in the packet would carry the originated key's information to the next stage. In all downstream stages, if the packet matches successfully and the corresponding counter in the matched table item is bigger, it would replace the item as in the first stage, otherwise it is directly forwarded to the next stage. Through this “smoke out” method to transfer heavy keys, HashPipe implements simple heavy-hitter detection under the condition of limited available memory. However, HashPipe may produce duplication in different stages (shown in Fig. 5) due to its “smoke out” method (as shown in Fig. 5).

Previous works need to consider the tradeoff of reasonable accuracy and acceptable overhead when monitoring heavy flow (e.g, netflow [80]). The programmability of P4 enables HashPipe to implement the heavy hitter in each switch with few memory consumption and high accuracy.

3.3.5. Dapper

Dapper [8] is a TCP diagnosis that monitors TCP performance accurately in real time at network edge. Once Dapper identifies network bottleneck, its specialized tools can find out what causes the bottleneck and offer a solution. Moreover, Dapper monitors traffic close to the server, which provides accuracy for measurement without monitoring client directly. There are two types of core metrics required to infer network problems, including easy to infer (e.g., counting number of bytes) and hard to infer (e.g., congestion). Dapper devises a packet process to obtain these metrics, which is first hashed to either initialize a new flow or check the statistics storing current information. To reduce the data-plane state requirements, Dapper adopts a two-phase monitoring technique. The first phase monitors all connections continuously but only collects low-overhead metrics (e.g., average rate of the flow). When a flow needs to be diagnosed due to poor performance, the second phase will be initialized to investigate it with more states (e.g., flow statistics).

Dapper utilizes flexible packet processing to obtain a fine-grained metrics for diagnosis in data plane. The challenge is the diagnosis needed to be lightweight in order to run across a wide range of devices with limited capabilities.

3.3.6. INT

In-band Network Telemetry (INT) is a new method of transmitting network measurement that enables packets to query switch-internal states (e.g., link utilization, queuing latency) to achieve fine-grained real-time monitoring [28]. The basic idea is that data plane allows probe packets attached its internal states to traverse whole network. In the last position of INT path, operators or controller can obtain all information of data planes by extracting the probe packets. INT requires the support of data plane for internal state exposure.

Fig. 6 shows an example of INT execution. The probe packet is sent from the INT generator to the INT collector after goes through several INT forwarders. The probe packet attaches INT metadata information in each arriving node and finally is extracted by the controller for network analysis. Researches have proved that it does work well in addressing some network problems [81]. However, due to the diversity of network deployment and the overhead brought by probe packet headers, the challenge of INT is how to optimize the problem of probing paths [78].

INT is a novel application that benefits from stateful data plane. Network operators can easily define what switch-internal states they need and design how to process probe packet for reduce of latency and overhead via flexible packet processing. This technology has drawn academic attention in recent years [82].

3.4. Remarks

Many platforms have implemented dozens of stateful applications in their architecture [6,9]. From the typical application in stateful data platform, e.g., port knocking, to today's variety of applications, they not only just leverage the property of stateful data plane to avoid unnecessary involvement of the controller, but also utilize the programmability and flexibility of stateful data plane to exploit fine-grained and meaningful designs.

4. Basic components of stateful data plane

To exploit the flexibility and programmability of stateful data plane, this section describes necessary definitions and designs, focusing on three aspects: state classification, state machine and state operation. Understanding the relation and distinct functions of different states can help researchers exploit their applications and improve their stateful processing architecture [83].

4.1. State classification

As the carrier of network information, state is the key component of stateful data plane. According to its effective scope, in this paper, states are divided into three types: per-packet state, per-flow state and global state.

4.1.1. Per-packet state

Per-packet state, which is maintained by each packet, is essential for delivering state information within each switch or across the network. States for different packets may be different.

On the one hand, when states are transferred as metadata within a switch, they can greatly facilitate the exchange of state information in different stages, e.g., the key–value tuple evicted from the previous stage can be carried in metadata of a packet traversing to the next stage [75].

On the other hand, when states are stored in packet's header, they can be transmitted outside a switch, allowing different switches to exchange information. First, a switch can change its optimization strategy by extracting packet's information. For example, in HULA [30], switches obtain the information of path utilization from received packet sent by the previous switch and such information is used to determine the next optimal router. Second, the switch can announce network information through these states proactively. For example, in SPIDER [2], switch announces the location of a link failure to others by sending a special packet attaching pre-defined states.

4.1.2. Per-flow state

A flow is identified by a unique key (e.g., IP addresses, source/destination MAC pair or 5-tuple flow identifier) in many platforms [26]. Per-flow state can be used to track status of each flow in a switch. Such state can be used and updated when processing packets of corresponding flow. For example, in the application of port knocking, shown in Fig. 8, a flow would be marked as the “open” state by a switch after verifying a series of packets in the correct sequence in the switch. Besides, per-flow states reside inside switches, usually implemented in memory or registers [6].

On the other hand, the states can also be used to update other flows' states. For example, in MAC learning, traffic from a bidirectional flow has two key based on our flow definition. Therefore, the application requires cross-state update [26].

4.1.3. Global state

Global state is a special type of state with various definitions in different platforms or hardware architectures implementation [4,43]. Here, we define that global state can be accessed and updated by multiple flows. For example, a global state G_0 is used to count the number of received flows [7]. Global state is essential in many network applications. For example, a global state can be used to save the port utilization for load balance [7].

On the other hand, many flows show the same behavior in network [84]. A state machine can be used in different flows. Assuming that programmers intend to record the overall information for all flows described by a state machine, it would be difficult to obtain states for only portion of these flows through extending the state machine. For example, programmers require to obtain the total number of flows transferred in the state 2 (shown in Fig. 8). To address this issue, FlowBlaze [6] proposes a new notion of global state, which is implemented in registers and can be read/modified by all the state machine instances (each flow associates with a state machine instance) originated from the same state machine definition. The previous problems can be easily addressed by using global state: programmers can only specify a global state to update it when a flow enters the state 2.

4.1.4. Discussions

Table 7 shows the differences among all three types of states above. The main difference between per-packet state and two other states is that per-packet state locates on packets. And we distinguish the per-flow state from the global state by whether the states can be accessed and modified by multiple flows.

In addition to classifications of state above, other classification methods also exist for different objectives and applications. For example, considering whether states need to be migrated, they can be divided into soft state and hard state. Soft state can be recovered through each packet easily, while careful migration needs to be considered for hard state since it requires to be copied from other switches [12]. Bifulco et al. [15] classified state information to two categories: packet state and global state. The packet state is associated with a single packet and the global state is associated with the device while it persists across packets. Pontarelli et al. [6] proposed a distinct notion of global state, which can be read and modified by all flows generated from the same EFSM definition.

4.2. State machines

Stateful applications can be abstracted as state machines. Hence, how to correctly design a proper state machine is essential in stateful data plane.

Table 7
Three types of state.

Type	Location	Usage	Examples
Per-packet state	Metadata header	(1) Facilitate the exchange of state information in different stages (2) Exchange information between switches	(1) Stages in pipeline transfer information by saving key–value pairs in metadata of packets [75] (2) Notice the path utilization for the next switch by packets [30] (3) Announce network the locations of failover by spreading special packets [2]
Per-flow state	Register RAM/SRAM/BRAM	(1) Track status of the state machine of a flow (2) Flow counter (3) Track some elaborate flow information (4) Temporary variables per flow	(1) Represent the certain status of state machine in port knocking [26] (2) A counter can be used in various applications, e.g., heavy hitter [75], DDoS detection [67], SYN-flood detection [5] (3) Save the timestamp of the last received packet (HULA [30]) (4) The key of hash algorithm to identify each flow [75] (5) Save features of flows for machine learning prediction [85]. (6) Temporary variables for Saving flow computing results [85]
Global state	Register	(1) Port status (2) Global counter (3) Global temporary variables	(1) Save the utilization of port [7] (2) Count the number of all flows (Static NAT [7]) (3) save machine learning models predicting for flows [85]

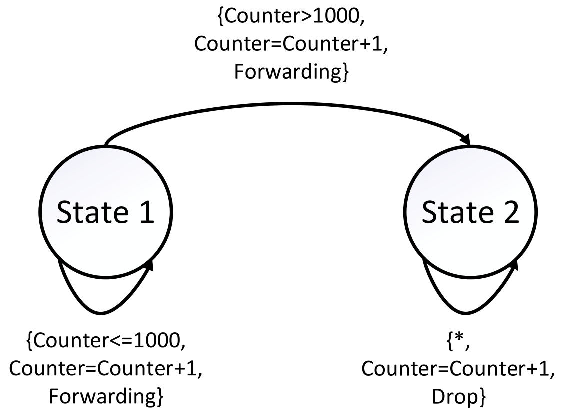


Fig. 7. State machine of a flow counter [6]. Packets of a flow will be dropped after receiving more than 1000 packets from the same flow.

4.2.1. *State machine abstraction*

Network functions, i.e., applications, deployed on stateful data plane have different logics. To correctly abstract logics of stateful network functions, Finite State Machine abstraction is usually used [86, 87]. OpenState adopts Mealy Machine [46] as its state machine abstraction. In short, a transition in state machine refers to changing from one state to another certain state, which can be described as $T : S \times I \to S \times O$ (introduced in Section 3). For example, in a port knocking shown in Fig. 8 (see details in Section 4.1.2), nodes are states and edges represent transitions. Transitions are marked with the tuple consisting of the condition and an action. In this port knocking, when a switch receives the packet of a flow that satisfies the corresponding condition under its current state, it enters the next state and the switch drops these probe packets. Until the state is "Open", the switch would forward those packets with port = 22.

Although Mealy Machine is suitable for transforming the stateless "match–action" table into stateful process [88], some network functions are still not supported since per-packet processing interval in them may be too long to affect processing latency. Another problem is that Mealy Machine needs to explicitly define all the possible states, which may lead to state explosion [6]. Hence, researchers consider better abstraction to simplify the design of network functions' state machine. Flowblaze [6] resorted to Extended Finite State Machines (EFSMs) [56], and OPP [4] adopted eXtended Finite State Machines (XFSM) [46].[1] They extended the Mealy Machine model by introducing: (1) variables D to describe the state; (2) a set of enabling functions $F(f_i : D \to 0, 1,\ f_i \in F)$ to trigger transitions; (3) a set of update functions $U(u_i : D \to D,\ u_i \in U)$. Thus, the transition is expressed as $T : S \times F \times I \to S \times U \times O$.

The improved extended finite state machine improves enabling functions for triggering transitions, which enables programmers to exploit more meaningful stateful applications. Afterwards, the state transition is no longer just "match" next state, but can also determine the next state based on "conditions". For example, a flow counter is shown in Fig. 7. The condition of triggering transition from state 1 to state 2 is that the counter of a flow is more than 1000. Besides, the improvement will also augment the complexity in hardware implementation (e.g., condition blocks) [4].

4.2.2. *State machine classification*

Most of stateful network applications can be abstracted into state machines, which can be classified into two types: **finite state machine** and **extended finite state machine**.

A state machine may perform the whole process of network functions from initialization to extinction, in which every phase is explicitly defined. Such state machines are referred as finite state machines. Finite state machines may need to work with some specific protocol states (e.g., TCP three-way handshake), or can be abstracted into certain network functions (e.g., link failover and port knocking). For example, in the port knocking shown in Fig. 8 (see details in Section 4.1.2), the initialization of a new flow would be marked "default". After the flow going through three state transitions, it would become the final state "open". This indicates that the port is open and the whole process of a finite state machine is completed.

On the contrary, some applications are defined as an extended finite state machine whose core logic requires data variables to model counters. Specifically, these applications monitor the value of counters and

[1] EFSM and XFSM are the same abstraction and simply use different acronyms, and we use EFSM in this paper.

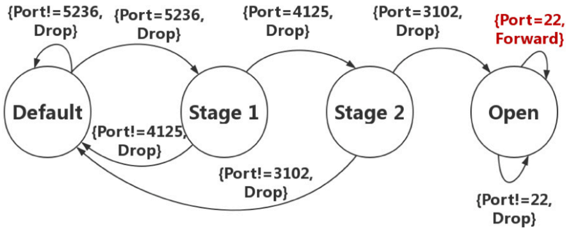


Fig. 8. State machine of port knocking [26].

execute pre-defined operations when these counters exceed pre-defined thresholds.

An extended finite state machine is used to abstract a network monitoring that needs to consistently supervise network traffic (e.g., heavy hitter detection, DDoS detection and flow size monitoring). Compared to traditional stateless data plane architecture that controller needs to periodically query statistics from switches, stateful data plane with this state machine can significantly reduce bandwidth consumption and controller overhead [5].

Table 6 summarizes details of network applications that are implemented based on these two types of state machines.

4.2.3. Remarks

Basically, the relations between flows and state machines can be one-to-one, many-to-one and one-to-many. Considering limited memory space, e.g., TCAM (Ternary Content Addressable Memory), a state machine usually can be shared among many flows [6]. A flow can also connect to two or more state machines simultaneously, which is very common in today's network requirements (e.g., a MAC learning and firewall in a switch). And the problem for network operator is how to let the switch processing packets correctly and rapidly. The competition of state machines is effective way (Section 5.3).

4.3. State operations

How to deploy state machines in data plane is critical for success of stateful data plane. In the OpenFlow version 1.5 [89], a group table is used to support stateful operations, which can solve problems such as fast failover. However, its flexibility is limited because it cannot provide enough programmability to satisfy diversified requirements of stateful applications. Therefore, several approaches are proposed to enforce deployment of stateful applications: extensible match/action table and control flow.

4.3.1. Extensible match–action table

OpenFlow's "match–action" abstraction is innovative to permit a certain level of programmability. However, it is still not flexible enough to satisfy demand of network applications. Thus, appropriate forwarding abstractions in data plane are required. A number of platforms apply the stateful table to implement state machines, which is originated from OpenFlow's "match–action" abstractions [4,5,26]. Bianchi et al. [26] found that OpenFlow "match–action" primitive can be reused with a broadened semantic. OPP adopts the EFSM table to finish its stateful operations, which is also an extended version of the "match–action" table. SDPA proposes a new "match-state–action" paradigm in which state information is maintained and modified within the data plane. The major difference of "match–action" abstraction compared to "match-transition-action" abstraction is the addition of a stateful modification operation. State modification in the extended "match–action" table usually occurs after matching packets. In the meantime, the extended "match–action" table splits "actions" into two types: one is the normal action executing on the packet, and another is the state action modifying the original state saved in the switch (shown in Fig. 3). For example, the port knocking (Fig. 8) may contain three tables. A packet would match a flow by the key in the *State lookup table*, query the current state in the *State machine table* and finally update its state by matching actions in the last table.

4.3.2. Control flow

In addition to methods above, the programmable data planes provide more easy ways to achieve stateful operations [11,58]. P4 provides a DSL (Domain-specific language) that enables network administrators to design their own packet process for switches. Specifically, a P4 program should define a Direct Acyclic Graph (DAG) of "match–action" stages, named control flow, which defines how packets are processed [90]. The control flow may contain an arbitrary number of stages. Therefore, how to design multiple suitable stages to control the correct logic is the key to achieve a complete stateful network application. For example, HULA [30] designs several stages for different matching, where the get_dst_tor is used to extract the destination of packets, the $hula\_logic$ is used as load balancer and the $hula\_to\_host$ is used to forward packets. Besides, P4 compiler also offers a special stateful memory, called registers. Their values can be read and written in actions. The transition of states in P4 represents how to operate the modification of registers, e.g., HULA defined five register arrays for storing link and packets information.

5. Schedule and optimization technologies

Considering the distributed property of stateful data plane, e.g., distributed state management and logic operations, it is very important and full of great challenges to achieve correctness, effectiveness and efficiency. This section will discuss several fundamental technologies on these issues.

5.1. Consistency

Inconsistency may lead to error states in state data plane, resulting in executing improper logics. Consistent problem is important in stateful data plane. Two types of consistency are considered in this paper: **intra-switch consistency** and **inter-switch consistency**. On one hand, due to state modify abstraction on the data plane, inconsistency may occur when packets of the same flow are read and written in an improper order within a switch. In the meantime, the inherent consistency problem among switches in stateful data plane is inevitable due to distributed state management among switches. These two consistency problems will shake the stability and availability of the whole network in varying degrees, and even cause network faults (e.g., incorrectly skipping firewall policies).

5.1.1. Intra-switch consistency

Inside switches, the state stored in a given memory is accessed and modified at a rate that belongs to the fraction of packet processing rate, while read/write operations may occupy multiple clock cycles [37]. The interval time from reading a memory location to completing the modification may depend on traffic pattern, which may potentially lead to concurrent read/write of the same location. It will cause inconsistency problem if a read operation precedes the completion of a write operation in the same memory location. Such situation usually occurs when packets of the same flow are processed in parallel. For example, assuming that two packets (say p_1 and p_2) of the same flow consecutively enter the pipeline at the same time. p_1 needs N clock cycles to complete the memory modification, while p_2 only needs M (assume $M < N$) clocks to access the same memory location. At this time, p_1's modification is not completed yet, resulting in inconsistency.

Several solutions have been proposed to solve this issue (see Table 8). For example, Pontarelli et al. [37] proposed a scheme of using the mixer's round robin policy to separate two packets coming from the same flow by N clock cycles (the time interval from the first table lookup to the last state update).

Table 8
Schedule and optimization technologies.

	Ref.	Solution		Affect line rate
Solutions for intra-switch consistency	OPP [4] FlowBlaze [6] OpenState [37]	Schedule packets' order		✓
	DOMINO [11]	Atom operation		✕
	Lock scheme [59]	Lock scheme		✓
	Ref.	Solution	Advantage	Disadvantage
Solutions for inter-switch consistency	SNAP [9]	Centralized storage	Easy to manage	Easy to be the bottleneck
	Eventual consistency [91]	State duplication scheme	Easy to realize	May be in transient inconsistency
	LODGE [92]	Synchronization messages	Easy to realize	Consume bandwidth
	Ref.	Solution		Degree of difficulty
Composition optimization	Hyper4 [93]	Using the port to isolate		Easy
	HyperV [90]	Simple DAG composition		Simple
	P4Visor [94]	Weight DAG composition		NP-complete problem

DOMINO [11] offers a packet-level solution. Each state is installed in an atom, which is a special stored cell. Each atom can only modify its own state. Thus, states are not allowed to be accessed or modified by multiple stages. This scheme can eliminate inconsistency problem when updating states, but reduces the flexibility and programmability.

Cascone et al. [59] came up with a locking scheme. If two packets that require access to the same portion of the memory arrive back-to-back, processing is paused for the second packet until the first one has updated the memory. This seems to be a compromising plan. However, it would affect the line rate at which the switch sustains one packet per clock cycle.

Pontarelli et al. [6] pointed out that it is possible to modify their respective flow states in parallel when processing packets belonging to different flows. Hence, they used a scheduler to guarantee that there are no two packets from the same flow are processed in parallel in the pipeline. However, when burst flow happens, i.e., a large number of packets of the same flow arrive, the latency will be increased.

In conclusion, above solutions have both advantages and disadvantages. Scheduling packets' order is easy to implement but may cause latency when packets originated from the same flow burst. Lock scheme violates the principle of processing packet at line rate. Atom operation overcomes the concern that a packet stays too long in switch, but it may lack flexibility in programming. Intra-switch consistency problem severely affects the feasibility of network application [95]. Hence, it is necessary to investigate an effective way to solve this problem.

5.1.2. *Inter-switch consistency*

Distributed state management imposes potential inconsistency risks, which may cause incorrect strategies to switches and even lead to network instability. However, it is impossible that states saved in the data plane completely achieve strong consistency without enormous cost. Specifying a location for storing all states seems to be an effective way. SNAP [9] puts forward an idea of one big switch, which adopts a method that all states are saved on a specified switch, and drains all the packets that need to be processed into the switch. This strategy can reduce memory consumption of most switches, but the switch that is designated to keep all states would become a bottleneck. Moreover, network problems would be arisen if the switch is down (e.g., states are unrecoverable). How to collect exact states from data plane is also a challenging issue.

Muqaddas et al. [91] proposed a state duplication scheme based on eventual consistency in SNAP platform. They pointed out that switches with limited amount of hardware resources are difficult to execute complex algorithms for employing strong consistency. Hence, they considered the replication scheme based on eventual consistency, and demonstrated that the method can bring low complexity while maintaining small replication error among replicas.

Sviridov et al. [92] proposed a relatively simple way to ensure consistency in network-wide by sending update messages and state synchronization triggered by certain conditions. They designed a state replication scheme based on the eventual consistency to provide state synchronization, and foresaw three different scenarios in order to achieve state synchronization.

Achieving strong consistency requires some consumption of switch resources. Accomplishing eventual consistency may lead transient error. The involvement of controller can apply in delay insensitive tasks. How to synchronize stateful data plane with support for state monitoring and management is still a challenge [96–98].

5.2. *Migration*

Migration plays an important role in ensuring high availability of network. When switch fails or network server requirements are changed (e.g., network update), migration is needed. For example, states in switches need to be migrated from one switch to another, which should guarantee correct network behaviors. In stateless data plane, the execution of migration is mainly determined by the controller [99] or middlebox [100]. The controller monitors the changes of the environment in real time and produces corresponding countermeasures. However, since each switch maintains its own states in stateful data plane, migration becomes much more complex. The major challenge is that how to consistently migrate states from the original switch to the new switch without affecting the network.

Fortunately, stateful data plane has its own unique mode of state transmission, some states maintained by switch can be learned by observing incoming traffic [2]. In other word, switch can correctly update some own states based on current received packets without acknowledging any phase of installed state machines. For example, the variable $flowlet\_time$ in HULA [30] can be overwritten by incoming packets.

Nevertheless, not all states in switches can be synchronized by the overridway of packet-carrying propagation. Luo et al. [12] proposed a consistent migration for data plane applications that requires strong-consistency network-wide. At first, they categorized P4 states into two types: soft state and hard state. A state is soft if its value is computed from random variables, which is usually used for optimization purposes in congestion control algorithms or scheduling. For example, the states stored in the variable $flowlet\_time$ [30] depend on packet arrival times. Soft states need not to be migrated since the functions can tolerate inconsistency by design. In contrast, hard states are maintained deterministically and explicitly. They cannot be correctly updated from

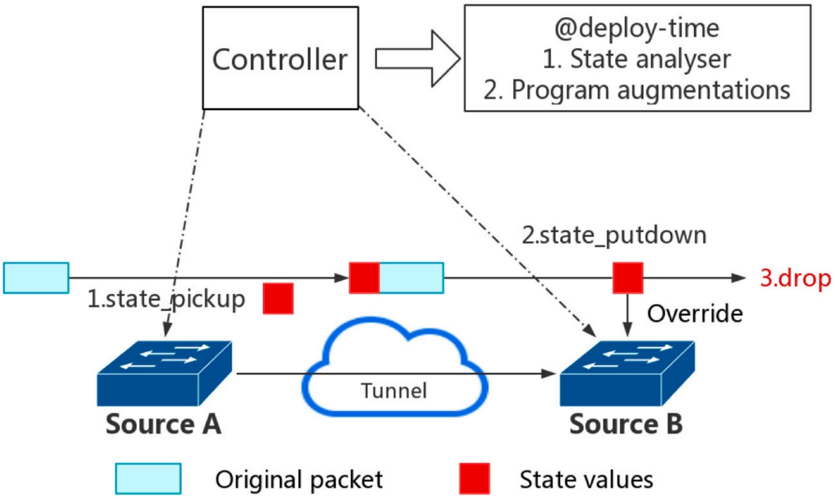


Fig. 9. Process of migration in Swing state [12].

packets or flow traffic. For example, a switch cannot recover which phase of safety certifications justified by current traffics in port knocking (shown in Fig. 8). And then, they consider two forwarding modes: *state_pickup* and *state_putdown*. A migration's source device in the mode of *state_pickup* will record state values, and tunnel a clone packet encapsulating the state values to the destination device. In the mode of *state_putdown*, the destination switch of migration will decapsulate the packet to obtain the state values while overwriting its variables. These two modes define state migration from source device to destination device. Finally, they design an augmentation procedure to support these forwarding modes for automatic migration at runtime. Before migration, controller classifies state types and augments P4 programs at deploy time. The process of migration is shown in Fig. 9. Inconsistent migration is rare, mainly caused by two issues: (1) packet re-ordering and loss, and (2) inconsistent hash collisions between different hash implementations.

He et al. [34] discussed two alternatives to migrate states in P4. One is transferring states directly in data plane. Another is collecting states from the data plane and installing on target switches by controller. They recommended the second option since the first method should require network function nodes to generate data plane packets with payloads of state information. The controller periodically fetches states and stores in database. Controller only redistributes states that are involved in migration. Experiments showed that it has no migrated packet loss while introducing additional forwarding latency. However, since the total migration time increases with the number of states, the latency will be unacceptable when there are too many states needed to be migrated.

Recently, He et al. [101] proposed a state management framework, which provides an automated and consistent state management in P4. They analyzed a P4 program as a control flow graph and excluded all stateless nodes to provide all stateful operations, which quickly identifies the network states that are critical for data plane reconfigurations (e.g., migration). This is an important step to automatically analyze states from source codes for migration.

5.3. Composition optimization

The trend of network application development is diversified and complicated. A single network device may carry multiple network programs with distinct functions. Composition optimization becomes critical to their performance. It can not only reduce extra overhead generated by multiple programs in the data plane and the assumption of switch resources, but also speed up processing and optimize overlapped logics among network functions. In addition, careful inspection for composition can also avoid conflicts that may occur when multiple state machines co-exist in a single switch [27].

The idea of composition in SDN can provide inspiration for composition in stateful data plane [102,103]. To let multiple programs working mutually without conflict, some platforms have discussed the composition of network functions in compilers [9,27]. Unfortunately, they did not provide solutions on composition in detail. Table 8 lists three solutions for composition optimization.

Hancock et al. [93] proposed a portable virtualization solution in data plane to offer simple composition. By virtualizing a device into multiple devices through different ports, multiple programs can be run on one device. This method is simple, effective and easy to implement. However, it is unscalable since it use software to emulate hardware to provide full virtualization [94].

Zhang et al. [90] drove the composition happening on pipelines. Considering that a control flow in P4 programs is a DAG (Directed Acyclic Graph), two steps are required for composition. Firstly, each control flow is analyzed and a "match–action stage" is divided into three fixed functional pipelines. Next, a DAG is converted into a uniform linear sequence using topological sorting algorithm. The authors proved that it prevails Hyper4 [93] in performance. However, this approach would waste a lot of resources in virtualization.

Zheng et al. [94] explored an optimization problem: merging two weighted DAGs into one while maximizing overlap. This problem is proved to be NP-Complete. Then, a heuristic algorithm is proposed to solve this problem, and the feasibility of the algorithm is demonstrated through experiments.

Above solutions are all designed for P4 programs and enlighten researchers on the problem of composition, especially the composition of DAG. But they are not suitable for other platforms, e.g., network functions exploited by the extensible match–action table. Hence, due to the limitation of different platforms with their own distinct compilers, it is still significant to explore a more general way of composition optimization.

5.4. Placement

The most obvious feature of stateful data plane is that state management and stateful processing can be offloaded to data plane, which brings many advantages. However, due to limitations of switches, e.g., limited memory and computation sources, controller is still indispensable for sophisticated functions. In this section, we discuss about two kinds of placement issues related to stateful data plane: state placement and logic placement.

5.4.1. State placement

Network devices keep their own states independently in stateful data plane, and the value of states on each device may be different. Such distributed scheme is widely used. However, the challenge is that the states are difficult to manage or even synchronize if it needs. In order to facilitate state management, some proposals offer a centralized management. For example, SNAP [9] uses a centralized scheme that uniformly places states in a specific switch in the data plane for maintenance. It is easy to manage and synchronize different types of network states in data plane directly. However, the switch may become bottlenecks. An improved solution in [91] proposed a backup placement and discussed the state migration based on eventual consistency to improve robustness of the centralized management.

On the other hand, not all states need to be stored on switches. Two types of states can be considered to inhabit on controllers. First, states that may have low usage can be placed on controllers to save memory resources on switches. For example, a phantom state on controller is used to solve this problem in FAST [27]. Second, states, which require complex computation that exceeds abilities of switches, can also be stored on controllers. For example, if computing the average flow rate using a counter is hard in switches, the controller could periodically fetch the counter to compute the result. It is worth noting that these states should allow inconsistency for a certain period of time, otherwise it will arise risk of network errors.

Storing and managing states in data plane can accelerate execution of operation logics, but not all operations are supported due to hardware limitation of switches. Maintaining state on the controller can take full advantage of the computing power on controllers. However, it only allows delay insensitive states. In practice, the placement of state could consider a combination of both switches and controllers according to requirements of applications.

5.4.2. Logic placement

By elaborating design, network functions can be deployed in data plane directly. However, in order to ensure the performance and efficiency of packet forwarding, existing packet processing components of switches are usually designed as simple as possible [16]. In other words, it is expensive, if not impossible, to implement sophisticated functions on a switch while guaranteeing forwarding speed. Hence, complex logics can consider to be implemented on controllers, such as complex mathematical calculation and dynamic network scheduling.

Network monitoring is intrinsically suitable for deploying on stateful data plane. However, some key components of network monitoring are too complex to implement on switches, e.g., multiplications and divisions are required in average and EWMA [10]. The usual way is to leave them to controller. Some stateful data plane platforms and applications involve controller in their design and implementation [27, 76].

When network strategies need to be changed, controller can provide real-time dynamic scheduling. Firstly, it can dynamically modify internal processing logic of switches (e.g., state machines). Some platforms allow the operator to replace old logics to satisfy changing environment [5,27]. In SDPA, the controller proactively initiates a new record of states when an application needs stateful processing. FAST offers dynamically management to local state machines at individual switches. Secondly, when network is instable, controller can quickly lead the process of network recovery. For example, controller is involved when migration is needed [12].

5.5. Remarks

In this section, several fundamental technologies are discussed to enhance the performance of stateful data plane. The consistency problems is critical for the correctness and stability of network. Migration improves fault tolerance of states. The composition optimization can enhance the speed of stateful processing and prevent errors during processing multiple applications. The placement separately discusses different positions of states and logics. In the most of these fundamental problems and technologies, controller still plays an important role in implementations of stateful data plane. Most platforms let controller initiate network functions in data planes. Dynamic changing of functions is also supported widely [104], which can be achieved through network update (e.g., migration). Moreover, controller can provide complex computations and optimization schemes for correctness, effectiveness and stability of networks. Hence, currently, controller is still important and indispensible for stateful data plane.

6. Implementation considerations for stateful data plane

In recent years, programmable switches have gradually aroused interest among academia and industry. Stateful data plane platforms and applications can be implemented based on these programmable switches. In this section, we will discuss current network devices for stateful data plane, as well as their hardware limitations including computation and memory.

6.1. Available implementations

Benefiting from the flexibility and programmability offered by current novel programmable switches, it is convenient for researchers to implement switch architecture with customized functions and structures running on stateful data plane [105]. Next, we will discuss both software and hardware programmable switches [15,106] for stateful data plane.

A software switch executes entire processing logic on a commodity CPU on top of a fast packet-classification algorithm/data structure [40]. Currently, Open vSwitch (OVS) and CPqD switches [38] are the most popular OpenFlow software switches. Stateful data plane platforms use the programmability of them to realize specific functionality of switches [5,27,35]. OpenState softswitch is an earlier software switch developed based on CPdQ to provide stateful processing. SDPA extends Open vSwitch to support "match-state–action" paradigm for stateful forwarding. The secure channel for communication with controller is modified to allow controller to be able to directly initialize and configure stateful applications on switches. Besides, the proprietary switch construct can be easily designed via using the programmability of Open vSwitch. PISCES [50] is a P4 customized software switch derived from Open vSwitch, which is the first software switch that allows custom protocol specification in a high-level DSL without requiring direct modifications for switch source code. Evaluation results show that PISCES programs are about 40 times shorter than equivalent changes to Open vSwitch source code.

For hardware programmable switches, RMT (Reconfigurable Match Tables) [51] offers flexible match table configuration, definition of arbitrary headers and header sequences, and state update per packet. Alternatively, dRMT [52] has significant flexibility due to its memory and compute disaggregation. However, both RMT and dRMT cannot allow multiple stages to access the same state block mutually without consuming too much memory space. On the hardware implementation, FPGA [39,41] can be used to implement data plane functionality, while using a dedicated packet classification engine achieved in TCAM chips. Pontarelli et al. [37] showed the feasibility of hardware implementation based on FPGA and also discussed on the performance achievable by using an ASIC to implement OpenState switch. FlowBlaze [6] discusses the implementation on NetFPGA SmartNIC to support a wide range of complex network functions, which achieves low latency and consumes relatively few energy. In addition to FPGA, Li et al. [107] proposed a heterogeneous programmable hardware architecture consisting of a CPU and a GPU. Besides, there are also a number of vendors dedicated hardware switches that can provide high-performance programmability [108].

Table 9
Hardware limitation.

Type	Problems	Negative affection	Solution
Computation	Limited stages in pipeline	Restrict flexible functionalities	(1) Composition (2) Concatenating pipeline
	Limited operations	Cannot realize some applications	(1) look up table [10,111]
Memory capacity	Scarce available memory	Deteriorate application performance	(1) Controller involvement [27] (2) External DRAM access [112–114]

6.2. Hardware limitations

Current innovations in switching hardware allow flexible per-packet processing and the ability to maintain limited mutable state at switches without sacrificing performance (e.g. RMT [51], FlexPipe [109], Barefoot's Tofino2 switch [108], Cavium XPliant switches [110]). But there still exist some limitations in these designs, which directly affects the feasibility and complexity of implementing meaningful programs in stateful data plane. Here, we mainly focus on limitations of computation and memory (see Table 9).

6.2.1. Compute capacity

In order to process packets at line rate, today's programmable switching hardware has computing limitations. For example Barefoot Tofino [108] supports 12 stages per pipeline and multiple pipelines (e.g., 2 to 4) per device. This limitation restricts functionalities implementing in data plane. For example, a server function chain includes multiple network functions implementing in a single switch [34]. The problem can be mitigated by application composition (Section 5.3). But if there is no or few overlap between different applications, the method is not effective. Another way is to concatenate several pipelines to prolong processing stages. But it incurs latency per packet while reduces throughout.

Another issue is that programmable switches only support basic operations, e.g., addition and subtraction, bitwise operation. It is difficult to provide complex operations since they are expensive executing in hardware chip, e.g., multiplication, division and loop. These complex computations are particularly essential in some applications [10,111]. For example, network traffic entropy is a well indication on traffic distribution across the network (e.g., DDoS detection). The entropy computation consists of two complex operations, i.e., logarithm and division, which can be approximate to exponential functions. Ding et al. [111] proposed two novel algorithms, P4Log and P4Exp, to estimate logarithms and exponential functions with a given precision by only using P4-supported arithmetic operations. They successfully implemented entropy-based applications in programmable switches. Sharma et al. [10] concluded several requirements of complex computation in switches. For example, RCP (Rate Control Protocol) needs to compute the fair rate by multiplication and division operations for optimizing the link utilization [115]. They transferred these complex computation to table lookup which pre-defines logarithms mapping results.

6.2.2. Memory capacity

A very fast memory is essential for packet processing at high speed, e.g., TCAM or SRAM, which is expensive and only available in small capacities [116]. For example, Barefoot Tofino [108] provides few tens of MBs of available memory. This limitation is manifested in restricting to the amount of storing states. Applications that are sensitive to memory size would be affected or even infeasible. For example, load balancing [30] lapses into slower, accuracy of sketching or monitoring applications declines, and even network diagnosis [8] that relies on per-flow or per-packet monitoring would be infeasible if the number of connections is large [117]. Hence, most of the applications need to make tradeoffs between performance and memory usage.

Simply increasing the memory size on switches brings challenges to the design of switches, e.g., consuming additional chip area [11], not matching packet processing speed. The controller can be used as an auxiliary memory to store states that are rarely read [27]. For example, a NAT maintains a statistics state of the number of total packets [118]. However, this method does not suitable for states that requires many writing/reading, since latency between switches and the controller CPU is high and unpredictable.

Other works try to enable network switches to access external memory [112–114]. DRAM can be used as an external memory since it is more affordable than on-chip buffer memory. Kim et al. [112] aimed to the feasibility of accessing remote memory from programmable switches. They assume that RDMA-capable NICs in remote memory servers directly connect switches. So switches can access the remote memory via the channel between the RDMA-capable NICs and switches while processing packets by DRAM primitives without any involvement of CPU. Beckmann et al. [113] envisioned a combination of a P4-capable ASIC with a DRAM scale match–action table. A packet firstly is preconstructed the match key field in ASIC, and sent to a FPGA which stores network states in DRAM. Secondly, the FPGA sends back the original packet with matching results. Finally, corresponding actions would be executed when the ASIC received the packet. In this work, ASIC needs to consume 100 Gbps Ethernet ports for high bandwidth connection to FPGAs. Kim et al. [114] explored a new approach that switch ASICs can access external DRAM purely in the data plane without involving CPUs on servers. If the data plane does not need to access DRAM, packets will be forwarded normally. Otherwise, it crafts a packet with DRAM header in pipeline and sends it to the DRAM server. Then the server replies the packet with matching results that needs to be processed in pipeline again. Therefore, this method will incur extra latency.

The basic idea of the three methods is setting an external DRAM memory. Although they would incur extra latency, they all extend memory in switches while keep processing at line rate. However, they should consider the re-order of packets because some packets need to enter in the pipeline twice.

6.3. Remarks

The performance of switches greatly affects deployment of stateful data plane applications. This section lists a number of programmable switches and analyzes hardware limitations of current programmable switches. Both computation and memory limitations can affect the feasibility and flexibility for the design of stateful applications.

7. Future research discussions

There are many aspects need to be further improved in stateful data plane, as we conclude in Fig. 10. How to optimize stateful data plane on appropriate switches, and develop high performance applications on stateful data plane are the mainstreams of stateful data plane research. Several potential future research issues on stateful data plane are summarized as below:

1. **A unified standard for stateful data plane.** There is no overall winner for stateful data plane today. P4 seems to be a main trend compiler in recent years. Although the appearance of P4 is to tackle the shortage of openflow switch and provide a flexible processing pipeline, it has developed to a popular switching architecture following by many famous vendors and group (e.g., VMware, Google). A number of mature network applications has exploited by P4 (e.g., Hula [30]). Researchers also have leveraged it to address many network problems [81,

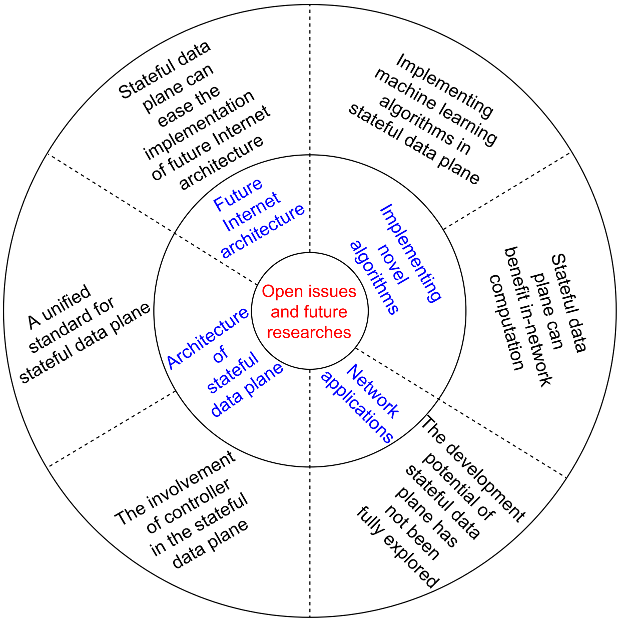

Open issues and future researches
Future Internet architecture
Implementing novel algorithms
Architecture of stateful data plane
Network applications
Stateful data plane can ease the implementation of future Internet architecture
Implementing machine learning algorithms in stateful data plane
Stateful data plane can benefit in-network computation
A unified standard for stateful data plane
The involvement of controller in the stateful data plane
The development potential of stateful data plane has not been fully explored

Fig. 10. Summary of potential future research issues on stateful data plane.

117]. We believe its potential has not been explored completely. However, some novel stateful packet processing architectures also show remarkable performance [6,11]. More generic and universal programming languages and switch models for stateful data plane are expected in future. For example, sluice [119] is a network-wide specification of the data plane whose aim is to offer more generic network tasks.

2. **The involvement of controller in the stateful data plane.** Although network functions can be implemented directly in stateful data plane without the intervention of controller, controller is still important for stateful data plane architecture today. Leveraging respective characteristics of the controller and data plane, researchers have deployed a routing strategy by mutual cooperation of the controller and data plane [120,121]. The data plane executes simple machine learning models with low accuracy to decrease the number of monitoring flows that need to upload to the controller. On the other hand, the controller implements complicated models with high accuracy for flow prediction. The cooperation makes flow prediction via machine learning in data center network possible, since it decreases the communication overhead between the controller and data plane. Whether and to what extent controller should be involved in implementation of stateful data plane applications still remains to be an open question.
3. **The development potential of stateful data plane has not been fully explored.** Ranging from port knocking, which is definitely suitable for offloading to stateful data plane, to today's various complicated applications executed on stateful data plane, a number of talent ideas have come true. There will be more explicit network application requirements emerging in the future (e.g., In-band Network Telemetry [78,122]), or traditional available protocol (e.g., FRR in P4 [123]). How to fully exploit the programmable potential of stateful data plane to provide more powerful and diversified functions will be one of the concerns in future.
4. **Implementing machine learning algorithms in stateful data plane is a challenge.** In recent years, machine learning algorithms do have offered distinct solutions to improve SDN network performance [124]. Researchers consider directly offloading machine learning algorithm to switches to optimize network [85,120,125]. pForest [85] tries to implement random forest in data plane and experiments prove it has high accuracy and flexibility. However, limited computations (e.g., no floating) and memory introduce great challenges to implementation of machine learning algorithms in stateful data plane. Setting a look up table to store results of complex mathematical operations is an effective way to satisfy the need of different machine learning algorithms [126]. The tradeoff of accuracy and memory usage still needs to be considered. Hence, designing suitable machine learning models and algorithms that fit stateful data plane, or enhancing stateful data plane's capability to support these models and algorithms, will have an important impact on the future ecology of stateful data plane applications.
5. **Stateful data plane can benefit in-network computation.** Offloading a set of compute operations from end hosts into stateful data plane is feasible and can provide considerable performance benefits [116]. Currently, the bottleneck in distributed machine learning training shifts from computation to communication. Experiments show that implementing stateful data plane as accelerators can speed up machine learning training [127,128]. On the other hand, stateful data plane can also help to take up the performance of some essential applications in cloud service (e.g., netcache [32], mapreduce [129]). Except for tackling traditional network problems, how stateful data plane can improve communication problems in emerging technologies is also an open issue.
6. **Stateful data plane can ease the implementation of future Internet architecture.** ICN (Information Centric Networking) is a networking paradigm that breaks the host centered connection mode of TCP/IP and becomes the information (or content) centered mode. In NDN (Named Data Networking), which is a representative of ICN instantiations, the problem is that current network equipment cannot be seamlessly extended to offer NDN data plane functions. To solve this problem, researchers have implemented NDN router via stateful data plane that offers programmabilities to satisfy frequent and drastic change in devices' behavior while keeps high processing speed [130]. On the other hand, the emerging architecture SD-ICN [131] integrates the thought of SDN's central management into ICN, which realizes some important network applications that have not been well considered in ICN (e.g., QoS). SD-ICN also faces challenges in the data plane, e.g., the OpenFlow-based data plane fails to consider the evolution of both ICN protocols and the OpenFlow protocol [132]. Inspired by the successful implementation above, stateful data plane can be considered to improve such future Internet architecture.

8. Conclusions

SDN provides a convenient state management to improve the network utilization efficiency. However, unnecessary interactions between controller and data plane brings additional overhead and delay to network. Stateful data plane architecture allows applications to be deployed directly in data plane without explicit involvement of controller. Thus network delay and controller overhead can be reduced. In this paper, a comprehensive survey on recent research works of stateful data plane is conducted. Several existing aspects for stateful data plane such as basic components, schedule and optimization technologies and implementation consideration are introduced and summarized. Also, the strengths and weaknesses of existing relevant research results are analyzed.

CRediT authorship contribution statement

Xiaoquan Zhang: Conceptualization, Writing - original draft. **Lin Cui:** Supervision, Funding acquisition. **Kaimin Wei:** Writing - review & editing. **Fung Po Tso:** Resources, Writing - review & editing. **Yangyang Ji:** Writing - review & editing. **Weijia Jia:** Writing - review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This work has been partially supported by Chinese National Research Fund (NSFC) No. 61772235, 61532013 and 61872239; Natural Science Foundation of Guangdong Province No. 2020A1515010771; Science and Technology Program of Guangzhou No. 202002030372; the UK Engineering and Physical Sciences Research Council (EPSRC) grants EP/P004407/2 and EP/P004024/1; the Innovate UK project 47198.

References

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner, Openflow: enabling innovation in campus networks, ACM SIGCOMM Comput. Commun. Rev. 38 (2) (2008) 69–74.
[2] C. Cascone, D. Sanvito, L. Pollini, A. Capone, B. Sansò, Fast failure detection and recovery in SDN with stateful data plane, Int. J. Netw. Manage. 27 (2) (2017) e1957.
[3] S.M. Mousavi, M. St-Hilaire, Early detection of ddos attacks against SDN controllers, in: 2015 International Conference on Computing, Networking and Communications (ICNC), IEEE, 2015, pp. 77–81.
[4] G. Bianchi, M. Bonola, S. Pontarelli, D. Sanvito, A. Capone, C. Cascone, Open packet processor: a programmable architecture for wire speed platform-independent stateful in-network processing, 2016, in: arXiv Preprint arXiv:1605.01977.
[5] C. Sun, J. Bi, H. Chen, H. Hu, Z. Zheng, S. Zhu, C. Wu, SDPA: Toward a stateful data plane in software-defined networking, in: IEEE/ACM Transactions on Networking, 2017.
[6] S. Pontarelli, R. Bifulco, M. Bonola, C. Cascone, M. Spaziani, V. Bruschi, D. Sanvito, G. Siracusano, A. Capone, M. Honda, et al., Flowblaze: Stateful packet processing in hardware, in: Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation, USENIX Association, 2019.
[7] M. Bonola, R. Bifulco, L. Petrucci, S. Pontarelli, A. Tulumello, G. Bianchi, Implementing advanced network functions for datacenters with stateful programmable data planes, in: Local and Metropolitan Area Networks (LANMAN), 2017 IEEE International Symposium on, IEEE, 2017, pp. 1–6.
[8] M. Ghasemi, T. Benson, J. Rexford, Dapper: Data plane performance diagnosis of tcp, in: Proceedings of the Symposium on SDN Research, ACM, 2017, pp. 61–74.
[9] M.T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, D. Walker, SNAP: Stateful network-wide abstractions for packet processing, in: Proceedings of the 2016 ACM SIGCOMM Conference, ACM, 2016, pp. 29–43.
[10] N.K. Sharma, A. Kaufmann, T. Anderson, C. Kim, A. Krishnamurthy, J. Nelson, S. Peter, Evaluating the power of flexible packet processing for network resource allocation, in: Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation, USENIX Association, 2017, pp. 67–82.
[11] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, S. Licking, Packet transactions: High-level programming for line-rate switches, in: Proceedings of the 2016 ACM SIGCOMM Conference, ACM, 2016, pp. 15–28.
[12] S. Luo, H. Yu, L. Vanbever, Swing state: Consistent updates for stateful and programmable data planes, in: Proceedings of the Symposium on SDN Research, ACM, 2017, pp. 115–121.
[13] A. Shaghaghi, M.A. Kaafar, R. Buyya, S. Jha, Software-defined network (SDN) data plane security: Issues, solutions and future directions, 2018, in: arXiv Preprint arXiv:1804.00262.
[14] T. Dargahi, A. Caponi, M. Ambrosin, G. Bianchi, M. Conti, A survey on the security of stateful SDN data planes, IEEE Commun. Surv. Tutor. 19 (3) (2017) 1701–1725.
[15] R. Bifulco, G. Rétvári, A survey on the programmable data plane: Abstractions architectures and open problems, in: Proc. IEEE HPSR, 2018.
[16] E. Kaljic, A. Maric, P. Njemcevic, M. Hadzialic, A survey on data plane flexibility and programmability in software-defined networking, IEEE Access 7 (2019) 47804–47840.
[17] F. Bannour, S. Souihi, A. Mellouk, Distributed SDN control: Survey, taxonomy, and challenges, IEEE Commun. Surv. Tutor. 20 (1) (2017) 333–354.
[18] W. Xia, Y. Wen, C.H. Foh, D. Niyato, H. Xie, A survey on software-defined networking, IEEE Commun. Surv. Tutor. 17 (1) (2015) 27–51.
[19] D. Kreutz, F.M. Ramos, P. Verissimo, C.E. Rothenberg, S. Azodolmolky, S. Uhlig, Software-defined networking: A comprehensive survey, Proc. IEEE 103 (1) (2015) 14–76.
[20] J. Xie, D. Guo, Z. Hu, T. Qu, P. Lv, Control plane of software defined networks: A survey, Comput. Commun. 67 (2015) 1–10.
[21] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, S. Shenker, NOX: towards an operating system for networks, ACM SIGCOMM Comput. Commun. Rev. 38 (3) (2008) 105–110.
[22] POX, 2019, https://github.com/noxrepo/pox, Access on: 2020.
[23] J. Medved, R. Varga, A. Tkacik, K. Gray, Opendaylight: Towards a model-driven SDN controller architecture, in: Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014, IEEE, 2014, pp. 1–6.
[24] Project floodlight, 2019, https://floodlight.atlassian.net/wiki/home, Access on: 2020.
[25] D. Erickson, The beacon openflow controller, in: Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, ACM, 2013, pp. 13–18.
[26] G. Bianchi, M. Bonola, A. Capone, C. Cascone, Openstate: programming platform-independent stateful openflow applications inside the switch, ACM SIGCOMM Comput. Commun. Rev. 44 (2) (2014) 44–51.
[27] M. Moshref, A. Bhargava, A. Gupta, M. Yu, 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, ACM, 2014, pp. 61–66.
[28] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, L.J. Wobker, In-band network telemetry via programmable dataplanes, in: ACM SIGCOMM, 2015.
[29] W. Han, H. Hu, Z. Zhao, A. Doupé, G.-J. Ahn, K.-C. Wang, J. Deng, State-aware network access management for software-defined networks, in: Proceedings of the 21st ACM on Symposium on Access Control Models and Technologies, ACM, 2016, pp. 1–11.
[30] N. Katta, M. Hira, C. Kim, A. Sivaraman, J. Rexford, Hula: Scalable load balancing using programmable data planes, in: Proceedings of the Symposium on SDN Research, ACM, 2016, p. 10.
[31] F. Nife, Z. Kotulski, Multi-level stateful firewall mechanism for software defined networks, in: International Conference on Computer Networks, Springer, 2017, pp. 271–286.
[32] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, I. Stoica, Netcache: Balancing key-value stores with fast in-network caching, in: Proceedings of the 26th Symposium on Operating Systems Principles, 2017, pp. 121–136.
[33] S. Goswami, N. Kodirov, C. Mustard, I. Beschastnikh, M. Seltzer, Parking packet payload with p4, 2020, in: arXiv preprint arXiv:2006.05182.
[34] M. He, A. Basta, A. Blenk, N. Deric, W. Kellerer, P4NFV: An NFV architecture with flexible data plane reconfiguration, in: 2018 14th International Conference on Network and Service Management (CNSM), IEEE, 2018, pp. 90–98.
[35] Openstate softswitch, 2019, https://github.com/OpenState-SDN/ofsoftswitch13, Access on: 2020.
[36] G. Bianchi, M. Bonola, A. Capone, C. Cascone, S. Pontarelli, Towards wire-speed platform-agnostic control of openflow switches, 2014, in: arXiv Preprint arXiv:1409.0242.
[37] S. Pontarelli, M. Bonola, G. Bianchi, A. Capone, C. Cascone, Stateful openflow: Hardware proof of concept, in: 2015 IEEE 16th International Conference on High Performance Switching and Routing (HPSR), IEEE, 2015, pp. 1–8.
[38] Openflow 1.3 software switch, 2019, https://github.com/CPqD/ofsoftswitch13, Access on: 2020.
[39] N. Zilberman, Y. Audzevich, G.A. Covington, A.W. Moore, NetFPGA SUME: Toward 100 gbps as research commodity, IEEE Micro 34 (5) (2014) 32–41.
[40] Open vswitch, 2016, https://github.com/openvswitch/ovs, Access on: 2020.
[41] Onetcard, 2019, https://www.xilinx.com/products/boards-and-kits.html, Access on: 2020.
[42] S. Smolka, S. Eliopoulos, N. Foster, A. Guha, A fast compiler for netkat, ACM SIGPLAN Not. 50 (9) (2015) 328–341.
[43] M. Shahbaz, N. Feamster, The case for an intermediate representation for programmable data planes, in: Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research, ACM, 2015, p. 3.
[44] M. Honda, F. Huici, G. Lettieri, L. Rizzo, Mswitch: a highly-scalable, modular software switch, in: Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research, ACM, 2015, p. 1.
[45] Linux socket filtering aka berkeley packet filter (BPF), 2019, https://www.kernel.org/doc/Documentation/networking/filter.txt, Access on: 2020.
[46] K.-T. Cheng, A.S. Krishnakumar, Automatic functional test generation using the extended finite state machine model, in: 30th ACM/IEEE Design Automation Conference, IEEE, 1993, pp. 86–91.
[47] Openstate demo, 2015, http://www.beba-project.eu/presentations/2015-openstate-live-demo.pdf, Access on: 2020.
[48] From dumb to smarter switches in software defined networks: towards a stateful data plane, 2015, http://www.beba-project.eu/, Access on: 2020.
[49] P4lang/behavioral-model, 2019, https://github.com/p4lang/behavioral-model, Access on: 2020.
[50] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, J. Rexford, PISCES: A programmable, protocol-independent software switch, in: Proceedings of the 2016 ACM SIGCOMM Conference, ACM, 2016, pp. 525–538.

[51] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, M. Horowitz, Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN, ACM SIGCOMM Comput. Commun. Rev. 43 (4) (2013) 99–110.

[52] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, et al., DRMT: Disaggregated programmable switching, in: Proceedings of the Conference of the ACM Special Interest Group on Data Communication, ACM, 2017, pp. 1–14.

[53] H. Wang, R. Soulé, H.T. Dang, K.S. Lee, V. Shrivastav, N. Foster, H. Weatherspoon, P4FPGA: A rapid prototyping framework for P4, in: Symposium on SDN Research, 2017.

[54] S. Ibanez, G. Brebner, N. McKeown, N. Zilberman, The P4->netFPGA workflow for line-rate packet processing, in: Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ACM, 2019, pp. 1–9.

[55] The p4 language specification version 1.1.0, 2016, https://p4.org/p4-spec/p4-14/v1.1.0/tex/p4.pdf, Access on: 2020.

[56] V. Alagar, K. Periyasamy, Extended finite state machine, in: Specification of Software Systems, Springer, 2011, pp. 105–128.

[57] L.foundation. fd.io, 2019, https://fd.io/, Access on: 2020.

[58] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, et al., P4: Programming protocol-independent packet processors, ACM SIGCOMM Comput. Commun. Rev. 44 (3) (2014) 87–95.

[59] C. Cascone, R. Bifulco, S. Pontarelli, A. Capone, Relaxing state-access constraints in stateful programmable data planes, ACM SIGCOMM Comput. Commun. Rev. 48 (1) (2018) 3–9.

[60] G. Bianchi, M. Welzl, A. Tulumello, F. Gringoli, G. Belocchi, M. Faltelli, S. Pontarelli, Xtra: Towards portable transport layer functions, IEEE Trans. Netw. Serv. Manag. 16 (4) (2019) 1507–1521.

[61] A. Tulumello, G. Belocchi, M. Bonola, S. Pontarelli, G. Bianchi, Pushing services to the edge using a stateful programmable dataplane, in: 2019 European Conference on Networks and Communications (EuCNC), IEEE, 2019, pp. 389–393.

[62] C. Cascone, L. Pollini, D. Sanvito, A. Capone, Traffic management applications for stateful SDN data plane, in: Software Defined Networks (EWSDN), 2015 Fourth European Workshop on, IEEE, 2015, pp. 85–90.

[63] A. Capone, C. Cascone, A.Q. Nguyen, B. Sanso, Detour planning for fast and reliable failure recovery in SDN with openstate, in: Design of Reliable Communication Networks (DRCN), 2015 11th International Conference on the, IEEE, 2015, pp. 25–32.

[64] C.H. Benet, A.J. Kassler, T. Benson, G. Pongracz, MP-HULA: Multipath transport aware load balancing using programmable data planes, in: Proceedings of the 2018 Morning Workshop on in-Network Computing, ACM, 2018, pp. 7–13.

[65] V. Olteanu, A. Agache, A. Voinescu, C. Raiciu, Stateless datacenter load-balancing with beamer, in: Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation, USENIX Association, 2018, pp. 125–139.

[66] M. Caprolu, S. Raponi, R. Di Pietro, Fortress: An efficient and distributed firewall for stateful data plane SDN, Secur. Commun. Netw. 2019 (2019).

[67] F. Rebecchi, J. Boite, P.-A. Nardin, M. Bouet, V. Conan, Traffic monitoring and ddos detection using stateful SDN, in: Network Softwarization (NetSoft), 2017 IEEE Conference on, IEEE, 2017, pp. 1–2.

[68] F. Rebecchi, J. Boite, P.-A. Nardin, M. Bouet, Conan, Ddos protection with stateful software-defined networking, Int. J. Netw. Manage. 29 (1) (2019) e2042.

[69] M. Zhang, G. Li, S. Wang, C. Liu, A. Chen, H. Hu, G. Gu, Q. Li, M. Xu, J. Wu, Poseidon: Mitigating volumetric ddos attacks with programmable switches, 2020.

[70] M. Kuka, K. Vojanec, J. Kučera, P. Benáček, Accelerated DDoS attacks mitigation using programmable data plane, in: 2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), IEEE, 2019, pp. 1–3.

[71] S. Mahrach, A. Haqiq, Ddos flooding attack mitigation in software defined networks, Int. J. Adv. Comput. Sci. Appl. 11 (1) (2020) http://dx.doi.org/10.14569/IJACSA.2020.0110185.

[72] S.A. Mehdi, J. Khalid, S.A. Khayam, Revisiting traffic anomaly detection using software defined networking, in: International Workshop on Recent Advances in Intrusion Detection, Springer, 2011, pp. 161–180.

[73] A. Bianco, P. Giaccone, S. Kelki, N.M. Campos, S. Traverso, T. Zhang, On-the-fly traffic classification and control with a stateful SDN approach, in: Communications (ICC), 2017 IEEE International Conference on, IEEE, 2017, pp. 1–6.

[74] D. Sanvito, D. Moro, A. Capone, Towards traffic classification offloading to stateful SDN data planes, in: 2017 IEEE Conference on Network Softwarization (NetSoft), IEEE, 2017, pp. 1–4.

[75] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, J. Rexford, Heavy-hitter detection entirely in the data plane, in: Proceedings of the Symposium on SDN Research, ACM, 2017, pp. 164–176.

[76] R. Harrison, Q. Cai, A. Gupta, J. Rexford, Network-wide heavy hitter detection with commodity switches, in: Proceedings of the Symposium on SDN Research, ACM, 2018, p. 8.

[77] B. Turkovic, J. Oostenbrink, F. Kuipers, Detecting heavy hitters in the data-plane, 2019, in: arXiv Preprint arXiv:1902.06993.

[78] T. Pan, E. Song, Z. Bian, X. Lin, X. Peng, J. Zhang, T. Huang, B. Liu, Y. Liu, Int-path: Towards optimal path planning for in-band network-wide telemetry, in: IEEE INFOCOM 2019-IEEE Conference on Computer Communications, IEEE, 2019, pp. 487–495.

[79] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V.T. Lam, F. Matus, R. Pan, N. Yadav, et al., Conga: Distributed congestion-aware load balancing for datacenters, in: Proceedings of the 2014 ACM Conference on SIGCOMM, 2014, pp. 503–514.

[80] Cisco IOS netflow, 2020, https://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow/index.html, Access on: 2020.

[81] K.A. Vardhan, M. Jakaraddi, G. Shobha, J. Shetty, A. Chala, D. Camper, Design and development of iot plugin for hpcc systems, in: 2019 IEEE 4th International Conference on Big Data Analytics (ICBDA), IEEE, 2019, pp. 158–162.

[82] R. Ben Basat, S. Ramanathan, Y. Li, G. Antichi, M. Yu, M. Mitzenmacher, Pint: Probabilistic in-band network telemetry, in: Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication, 2020, pp. 662–680.

[83] B. Stephens, A. Akella, M.M. Swift, Loom: flexible and efficient nic packet scheduling, in: Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation, USENIX Association, 2019, pp. 33–46.

[84] P. Kazemian, G. Varghese, N. McKeown, Header space analysis: static checking for networks, in: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, USENIX Association, 2012, p. 9.

[85] C. Busse-Grawitz, R. Meier, A. Dietmüller, T. Bühler, L. Vanbever, Pforest: In-network inference with random forests, 2019, in: arXiv Preprint arXiv:1909.05680.

[86] S.J. Moon, J. Helt, Y. Yuan, Y. Bieri, S. Banerjee, V. Sekar, W. Wu, M. Yannakakis, Y. Zhang, Alembic: automated model inference for stateful network functions, in: Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation, USENIX Association, 2019, pp. 699–718.

[87] S.K. Fayaz, T. Yu, Y. Tobioka, S. Chaki, V. Sekar, BUZZ: Testing context-dependent policies in stateful networks, in: Proceedings of the 13th USENIX Conference on Networked Systems Design and Implementation, USENIX Association, 2016.

[88] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, R. Clark, Kinetic: Verifiable dynamic network control, in: Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation, 2015, pp. 59–72.

[89] Openflow switch specification version 1.5.0 (protocol version 0x06), 2019, http://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.0.noipr.pdf, Access on: 2020.

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

[91] A.S. Muqaddas, Control plane in software defined networks and stateful data planes, 2019.

[92] G. Sviridov, M. Bonola, A. Tulumello, P. Giaccone, A. Bianco, G. Bianchi, LODGE: LOcal decisions on global states in progrananaable data planes, in: 2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft), IEEE, 2018, pp. 257–261.

[93] D. Hancock, J. Van der Merwe, Hyper4: Using P4 to virtualize the programmable data plane, in: Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies, ACM, 2016, pp. 35–49.

[94] P. Zheng, T. Benson, C. Hu, P4visor: lightweight virtualization and composition primitives for building and testing modular programs, in: Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies, ACM, 2018, pp. 98–111.

[95] B. Turkovic, F. Kuipers, N. van Adrichem, K. Langendoen, Fast network congestion detection and avoidance using p4, in: Proceedings of the 2018 Workshop on Networking for Emerging Applications and Technologies, ACM, 2018, pp. 45–51.

[96] A. Shukla, S.J. Saidi, S. Schmid, M. Canini, T. Zinner, A. Feldmann, Towards consistent SDNs a case for network state fuzzing, in: IEEE Transactions on Network and Service Management, IEEE, 2019.

[97] G. Sviridov, M. Bonola, A. Tulumello, P. Giaccone, A. Bianco, G. Bianchi, Local decisions on replicated states (LOADER) in programmable data planes: programming abstraction and experimental evaluation, 2020, in: arXiv Preprint arXiv:2001.07670.

[98] A. Shukla, S. Fathalli, T. Zinner, A. Hecker, S. Schmid, P4CONSIST: Towards consistent P4 SDNs, in: IEEE Journal on Special Areas in Communication (JSAC)-NetSoft, IEEE, 2020.

[99] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, A. Akella, OpenNF: Enabling innovation in network function control, in: ACM SIGCOMM Computer Communication Review, Vol. 44, ACM, 2014, pp. 163–174.

[100] S. Rajagopalan, D. Williams, H. Jamjoom, A. Warfield, Split/merge: system support for elastic execution in virtual middleboxes, in: Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation, USENIX Association, 2013, pp. 227–240.

[101] M. He, A. Blenk, W. Kellerer, S. Schmid, Toward consistent state management of adaptive programmable networks based on p4, in: Proceedings of the ACM SIGCOMM 2019 Workshop on Networking for Emerging Applications and Technologies, ACM, 2019, pp. 29–35.

[102] C. Monsanto, J. Reich, N. Foster, J. Rexford, D. Walker, Composing software-defined networks, in: Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation, USENIX Association, 2013, pp. 1–14.

[103] C. Monsanto, N. Foster, R. Harrison, D. Walker, A compiler and run-time system for network programming languages, ACM SIGPLAN Not. 47 (1) (2012) 217–230.

[104] P4 runtime, 2019, https://p4.org/p4-runtime/, Access on: 2020.

[105] J. Sonchack, A.J. Aviv, E. Keller, J.M. Smith, Turboflow: Information rich flow record generation on commodity switches, in: Proceedings of the Thirteenth EuroSys Conference, ACM, 2018, p. 11.

[106] H. Farhad, H. Lee, A. Nakao, Data plane programmability in SDN, in: IEEE International Conference on Network Protocols, 2014.

[107] P. Li, Y. Luo, P4GPU: Accelerate packet processing of a p4 program with a CPU-gpu heterogeneous architecture, in: Proceedings of the 2016 Symposium on Architectures for Networking and Communications Systems, ACM, 2016, pp. 125–126.

[108] Barefoot tofino, 2020, https://barefootnetworks.com/products/brief-tofino//, Access on: 2020.

[109] Intel flexpipe, 2019, https://www.intel.com/content/www/us/en/ethernet-products/switch-silicon/ethernet-switch-fm6000-series-brief.html?wapkw=flexpipe, Access on: 2020.

[110] Xpliant ethernet switches, 2019, https://www.marvell.com/switching/, Access on: 2020.

[111] D. Ding, M. Savi, D. Siracusa, Estimating logarithmic and exponential functions to track network traffic entropy in p4, in: IEEE/IFIP Network Operations and Management Symposium (NOMS), 2019.

[112] D. Kim, Y. Zhu, C. Kim, J. Lee, S. Seshan, Generic external memory for switch data planes, in: Proceedings of the 17th ACM Workshop on Hot Topics in Networks, 2018, pp. 1–7.

[113] C. Beckmann, R. Krishnamoorthy, H. Wang, A. Lam, C. Kim, Hurdles for a dram-based match-action table, in: 2020 23rd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN), IEEE, 2020, pp. 13–16.

[114] D. Kim, Z. Liu, Y. Zhu, C. Kim, J. Lee, V. Sekar, S. Seshan, Tea: Enabling state-intensive network functions on programmable switches, in: Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication, 2020, 90–106.

[115] N. Dukkipati, Rate Control Protocol (RCP): Congestion Control to Make Flows Complete Quickly, Citeseer, 2008.

[116] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, P. Kalnis, In-network computation is a dumb idea whose time has come, in: Proceedings of the 16th ACM Workshop on Hot Topics in Networks, 2017, pp. 150–156.

[117] Z. Liu, S. Zhou, O. Rottenstreich, V. Braverman, J. Rexford, Memory-efficient performance monitoring on programmable switches with lean algorithms, 2019, in: arXiv Preprint arXiv:1911.06951.

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

[119] V. Natesh, P.G. Kannan, A. Sivaraman, R. Netravali, Sluice: Network-wide data plane programming, in: Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos, ACM, 2019, pp. 156–158.

[120] S.-C. Chao, K.C.-J. Lin, M.-S. Chen, Flow classification for software-defined data centers using stream mining, IEEE Trans. Serv. Comput. 12 (1) (2016) 105–116.

[121] Y.-H. Huang, W.-Y. Shih, J.-L. Huang, A classification-based elephant flow detection method using application round on SDN environments, in: 2017 19th Asia-Pacific Network Operations and Management Symposium (APNOMS), IEEE, 2017, pp. 231–234.

[122] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, W. Willinger, Sonata: Query-driven streaming network telemetry, in: Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, ACM, 2018, pp. 357–371.

[123] R. Sedar, M. Borokhovich, M. Chiesa, G. Antichi, S. Schmid, Supporting emerging applications with low-latency failover in p4, in: Proceedings of the 2018 Workshop on Networking for Emerging Applications and Technologies, ACM, 2018, pp. 52–57.

[124] J. Xie, F.R. Yu, T. Huang, R. Xie, J. Liu, C. Wang, Y. Liu, A survey of machine learning techniques applied to software defined networking (SDN): Research issues and challenges, IEEE Commun. Surv. Tutor. 21 (1) (2018) 393–430.

[125] Y.-S. Lu, K.C.-J. Lin, Enabling inference inside software switches, in: 2019 20th Asia-Pacific Network Operations and Management Symposium (APNOMS), IEEE, 2019, pp. 1–4.

[126] Z. Xiong, N. Zilberman, Do switches dream of machine learning? Toward in-network classification, in: Proceedings of the 18th ACM Workshop on Hot Topics in Networks, 2019, pp. 25–33.

[127] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D.R. Ports, P. Richtárik, Scaling distributed machine learning with in-network aggregation, 2019, arXiv preprint arXiv:1903.06701.

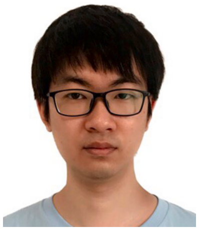
[128] Y. Li, I.-J. Liu, Y. Yuan, D. Chen, A. Schwing, J. Huang, Accelerating distributed reinforcement learning with in-switch computing, in: 2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA), IEEE, 2019, pp. 279–291.

[129] L. Chen, G. Chen, J. Lingys, K. Chen, Programmable switch as a parallel computing device, 2018, arXiv preprint arXiv:1803.01491.

[130] S. Signorello, R. State, J. François, O. Festor, Ndn. p4: Programming information-centric data-planes, in: 2016 IEEE NetSoft Conference and Workshops (NetSoft), IEEE, 2016, pp. 384–389.

[131] M. Arumaithurai, J. Chen, E. Monticelli, X. Fu, K.K. Ramakrishnan, Exploiting ICN for flexible management of software-defined networks, in: Proceedings of the 1st ACM Conference on Information-Centric Networking, 2014, pp. 107–116.

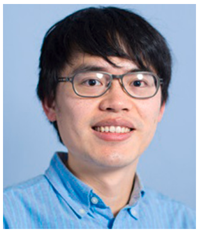
[132] Q.-Y. Zhang, X.-W. Wang, M. Huang, K.-Q. Li, S.K. Das, Software defined networking meets information centric networking: A survey, IEEE Access 6 (2018) 39547–39563.



Xiaoquan Zhang is currently a postgraduate student in Jinan University. He received the B.E. degree in computer science and technology from Chengdu University of Technology, China, in 2016. His current research interests includes stateful data plane, software-defined networking, and machine learning in computer network.

Lin Cui is currently with the Department of Computer Science at Jinan University, Guangzhou, China. He received the Ph.D. degree from City University of Hong Kong in 2013. He has broad interests in networking systems, with focuses on the following topics: cloud data center resource management, data center networking, software defined networking (SDN), virtualization, distributed systems as well as wireless networking.

Kaimin Wei joined the State Key Lab. of Software Development Environment (SKLSDE) in 2010, and is currently a Ph.D. Student at the Beihang University in Beijing, majoring in computer science. His current research interests are in the areas of wireless mobile networks, delay/disruption tolerant networks, mobile social network and algorithm design.



Fung Po Tso received his B.Eng., M.Phil. and Ph.D. degrees from City University of Hong Kong in 2006, 2007 and 2011 respectively. He is currently a Senior Lecturer (Associate Professor) in the Department of Computer Science at the Loughborough University. Prior to joining Loughborough, he worked as SICSA Next Generation Internet Fellow at the School of Computing Science, University of Glasgow during 2011 and 2014 and lecturer in Liverpool John Moores University during 2014 and 2017. He has published more than 30 research articles in top venues and outlets. His research interests include: network policy management, network measurement and optimization, cloud data center resource management, data center networking, software defined networking (SDN), distributed systems as well as mobile computing and system.

Yangyang Ji is currently a postgraduate student in Jinan University. She received the B.E. degree in network engineering from Nantong University, China, in 2018. Her current research interests includes mobile edge computing and multicast routing.

Weijia Jia is currently a Chair Professor and Director of BNU-UIC Institute of Artificial Intelligence and future Networks, Beijing Normal University at Zhuhai; VP for Research of BNU-HKBU United International College. His research interests include smart city, IoT, knowledge graph constructions, multicast and anycast QoS routing protocols, wireless sensor networks, and distributed systems. He has over 500 publications in prestigious international journals/conferences and research books and book chapters.