Toward an Abstract Model of Programmable Data Plane Devices

Debobroto Das Robin*, Dr. Javed I. Khan

Kent State University, Kent, Ohio, USA

ARTICLE INFO

Keywords:
Programmable data plane
Abstract model
Hardware abstraction layer
P4
Software Defined Network(SDN)

ABSTRACT

SDN divides the networking landscape into 2 parts: control and data plane. SDN expanded it's foot mark starting with Openflow based highly flexible control plane and rigid data plane. With the progress of SDN paradigm, data plane is becoming more programmable then ever before. This has opened the pathway for complex in-network computation. To understand and utilize data plane's programmability, a detailed structural analysis and an identifiable matrix to compare different devices are required. On the other hand, a common abstract model of data plane is required to develop complex application for heterogeneous data plane devices. This model is also required to develop: policy, control mechanism, testing and verification methods for those complex applications. It can also provide insight about performance optimization and bench-marking of programmable data plane devices. In this work, an improved and structured abstract model of the programmable data plane devices and features of its components are discussed. Several commercial programmable data plane devices are also compared based on those features. A high level packet processing work-flow based on this model is also presented using extended finite state machine.

1. Introduction

Data plane devices execute few common tasks- ***T1***: receive traffic from medium ***T2***: interpret traffic ***T3***: do computation on traffic ***T4***: forward traffic. Exact details of these tasks are ruled by the protocol stack a device is running. From the very beginning networking equipment were tightly bound with TCP/IP protocol stack. This rigidness hindered innovation in networking landscape for a long time. Effort for making this rigid environment more programmable can be traced back to different initiatives [23] starting around late 90's. They haven't gain momentum until OpenFlow [46] came to the theater. What OpenFlow offers can be identified as dynamic configurability over a broad set of protocol header fields rather than programmability in data plane. But soon everyone realized that, with it's ever growing protocol field set, OpenFlow can not achieve the goal of real programmable network [6]. For true network programmability, data plane hardware architecture should be decoupled from protocol and programmable in nature.

To achieve these goals, various technology, framework and architecture [28, 26, 35, 25, 27, 11, 45, 7, 13] with heterogeneous hardware internals have been developed both before and after OpenFlow. Based on them, several various programmable data plane devices are also available in market (with varying degrees of programmability) **add ref**. Exact details of **T1-T4** for these devices can be programmed *'on the fly'* during their life cycle. With advancement in hardware development and design, these programmable data plane devices are becoming more computationally capable and various complex application layer processing tasks are being pushed to data plane [65, 21, 42]. This is enabling emergence of a new paradigm *'in-network computation'* [59, 3]. Enabling such cross layer behavior and protocol independence in data plane makes OSI layer based definition of switch(L2, L3, L4 switch etc.) obsolete. This raises the necessity of structured discussion on: how to define programmable switch/data plane device and how to define it's programmability features? (*Through rest of the paper, programmable switch and programmable data plane device, these two terms are used interchangeably.*)

A Programmable switch needs a software stack for programming the data plane. Abstraction layer ***Device and resource Abstraction Layer (DAL)*** [33] is one of the key component of these stacks. It allows data plane application developers to write applications in more convenient way rather than dealing with actual hardware instructions. Prominent programmable data plane technologies have heterogeneous hardware internals and corresponding hardware abstraction layer (**HAL**) (Fig. 1)). These are different from one another(figure show)(except p4 version 16). Without a common abstraction layer both data and control plane application become tightly bound to target architecture. For example, a program developed for rmt based architectures can not be run directly as a dpdk based userpsace program. Lack of common abstraction layer and attempt to develop a programmable system directly over a specific architecture without an intermediate abstraction enormously increases complexity of development, testing, performance bench-marking and formal verification of any novel network functionality built over it. Lack of common abstraction layer also increase the cost and complexity of interoperability, portability, maintainability and extendibility. Besides this, network virtualization, network function virtualization and service chain composition are very important concept for dynamic and extendeble network**add ref to them**. They can bring rapid service innovation, improved resource usage, modular network function development and many other advantages. A well designed hardware abstraction layer (Logical Forwarding Plane [10]) is center piece for successful use of these concepts in networking domain [9].

*Corresponding author
drobin@kent.edu (D.D. Robin); javed@kent.edu (Dr.J.I. Khan)
ORCID(s):

Though a common abstraction layer over heterogeneous layer has various advantages, but restricting to only one hardware abstraction layer closes the door for future innovation in both hardware and abstraction layer design. Among current data plane programming software stacks, P4 is the most matured, flexible and widespread software stack for data plane programming. It's earlier version was tightly coupled with a single abstract hardware model(ref to p4 v14 arch). In latest version, P4 community has decoupled hardware architecture definition and data plane program definition. From version 16, P4 itself not provides any abstraction. It provides language constructs to express data plane hardware architectures and develop program based on those architectures. PSA [31] is one of the most matured among them.

Majority of the data plane programming stacks are tightly coupled with their own abstraction. Whereas, P4 (dominant data plane programming language) decoupled hardware architecture definition from packet processing behavior in it's latest version ($P4_{16}$ [18]). $P4_{16}$ provides separate language constructs to express abstract hardware architecture and develop program based on those architectures. Taking advantage of this feature, most of the data plane programming stacks have already developed interface with P4 [12, 69, 67, 56, 15, 61]. Besides this, several other architectures are also being proposed by P4 community [16]. PSA [31] is one of the most matured among them. PSA suggests a specification about how a programmable switch should work to conform with the P4 language specification. Hardware vendors can choose their own implemention. PSA can be termed as a common abstraction layer for data plane emerged from p4 community. Besides PSA, several other architectures are also proposed in literature **add ref**. Majority of them lacks programmability of two important components: buffer and scheduler. Moreover, these hardware architecture definitions and how they process a packet are described in an informal language. It creates several issues. Most important among them are- ***I1***) identifying clear definition of various components of a hardware architecture become hard ***I2***) boundary between 2 components and how they connect with each other become unclear ***I3***) exact details of how a packet is processed inside a component may differ from one hardware vendor to another. As consequence of ***I1***) and ***I2***), modular hardware/simulator/test-bed design become hard. And as consequence of ***I3***) application layer suffers. For example, if 2 hardware design not agrees about packet processing states inside the components, taking snapshot of a network(ref to motivation section) or testing/validating a data plane program for multiple architecture become extremely hard. To tackle these challenges, a generalized but flexible yet abstract model of the programmable data plane is necessary. It should be modular in nature and well defined interface among the components is necessary for independent development. Moreover, there should be a high level work flow of the components for ensuring uniform behavior across different hardware architectures.

What is an abstract model and what are its requirements? Any logical collection of data plane resources presenting

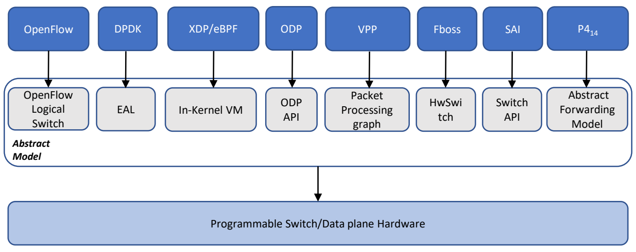


Figure 1: Abstraction layers of major SDN protocol (OpenFlow [28]) and programmable data plane technologies (DPDK [25], XDP/eBPF [35], ODP [26], VPP [27], Fboss [11], SAI [15], $P4_{14}$ [17])

view of a complete data plane device to the users can be termed as an ***abstract model*** of data plane device. Underlying resources can be components of single or multiple device or any form of hardware-software emulation of data plane functions. The abstract model should be generalized enough to hide low level hardware details from upper layer and be able to support a broad domain of use cases. It should be modular in nature and well defined interface among the components is necessary for independent development. Moreover, there should be a high level work flow of the components for ensuring uniform behavior across different hardware architectures.

Considering importance of an abstract model and current technology landscape, we think it is important to start discussion on a better model of abstraction layer for emerging programmable switches. In this work, we have attempted to shed light on this topic. In doing so, we make a number of important contributions

- Providing a unified definition of programmable switch which is not dependent on any protocol stack.
- The design of ***AVS***, an example model of ***Device and resource Abstraction Layer (DAL)*** for programmable data plane devices. It is modular in design and components are defined with a uniform and structured functional interface. This design brings hardware architecture independent model of network functions. Moreover it also enables independent design and optimization of the components. Moreover, this functional representation with hardware independent domain and range brings the scope of applying function composition to represent any network function(ref to section). This can lead to new programming language design for data plane.
- To cope with hardware architecture heterogeneity, work flow of the components are defined as extended finite state machine. Such kind of hardware independent representation helps to overcome disadvantages of vendor dependent implementation and increases interoperatibility & portability. Moreover, it opens the scope of static analysis. add log analysis in scenario.

- Analyzing programmability features of the components of abstract model. Based on these features, comparing leading programmable data plane devices to understand their programmability level. This gives a clear view of current state of the art and lays ground for bench marking of programmable data plane devices.

We do not claim the novelty of underlying components design. We are influenced by several existing work on this domain [7, 29, 48, 41, 43]. We believe our achievement is providing a broader picture with an improved and structured abstract model for the programmable switches.

The rest of the paper is organized as follows. In section 2, we start by looking at how abstraction layer is influencing current state of the art technologies in programmable data plane domain. It can help to understand the gap in current research and how our work tries to fill that gap without hampering scope of innovation. Next, our analysis and definition of programmable switch is presented in section 3 and proposed abstract model(AVS) is presented in section 4. A structural analysis and a novel functional representation of programmable components of AVS is presented in section 5. Section 6 explores the components in details. Section 7 presents comparison of few leading programmable data plane device based on the components of our proposed abstract model. In section 8 we analyzed the effectiveness of our proposed abstract model with few important use cases. We conclude the work in section 9.

2. Background and Related Work

OpenFlow [46] is the most successful name in SDN paradigm. Numerous OpenFlow supported switch exists in the market. But, the protocol itself, is described over an abstract model of switch named ***logical switch*** [28]. Despite having notion for logical abstraction of data plane, OpenFlow failed to come out with any programming model and language for data plane. This was more an attempt to separate control plane and data plane for enabling centralized network control. Soon academia and industry realized that, true potential of SDN can be leveraged only if data plane is fully programmable.

After the initial wave of OpenFlow protocol based SDN switches, work on programmable data plane device has gained significant momentum [5, 37]. The first consolidated effort for data-plane programmability is attributed to RMT [7], which proposed an architecture for programmable data plane devices. Based on RMT's architecture, an *'abstract forwarding model'* for data plane devices has been presented in [6]. In this work, authors have also presented a data plane programming language named P4 [6] to write program for RMT based programmable switches. Based on success of RMT and P4, several commercial programmable data plane devices has been emerged in the market. Major example of this category are: barefoot tofino [54], netronome smartNIC [51] etc. In initial version of P4(version 14) [17], programs are based on *'abstract forwarding model'* of [6]. At that time $P4_{14}$ was strictly coupled with the proposed abstract forwarding model. Although P4(version 14) came out with an abstract model and related programming construct to program them, it's high cohesion with proposed hardware with limited programmability made it unsuitable for design of new architectures. This clearly shows the requirement of dis-aggregation between hardware architecture and programming language for programmable data plane devices.

Currently RMT is the dominant programmable data plane architecture. But it has few limitations. each pipeline stage of RMT architecture can only access memory allocated for it. As a result cross component access of memory/data is also not possible [13]. Moreover, as components of RMT pipeline are connected linearly, a packet can not skip unnecessary stages in the pipeline. To overcome these limitations, authors have proposed dRMT architecture in [13]. It improves RMT architecture by disaggregating memory and compute resources inside switch. In [57] authors presented a conceptual model of data plane for supporting parallel processing. It is heavily based on RMT and dRMT architecture. But these works, lacks of concrete hardware abstraction layer.

For server based networking environment, smartNICs have seen a steady growth. It enables datapath offloading both in CPU and/or smartNICs. CPU based datapath can be implemented both in kernel space(uses linux kernel's networking stack or ebpf in kernel virtual machine) and user space**ref to dpdk, vpp** . Both kernel space and user space **ref to dpdk page that says ebpf call is possible** technologies can be mixed even more flexibility and performance. But each of them relies on their own abstraction layer **Show a figure of list of abs layers**. Several frameworks (ODP, p4NFV **ref must**) over these abstractions are also implemented. ODP & SAI both defines a set of API to enable programmability over supported devices. For both of them DPDK based implementation are available. P4 programs also can be used over these cpu based technologies **ref to p4 dpdk, ebpf, vpp**. In fact P4 has become the most dominant language for data plane programmaing in recent times. Nearly all the major data plane prgrammablilty platform have a P4 implementation. As P4 is becoming more matured, several works [36, 22, 34, 71, 44, 55, 1] **add ref to p4nfv and mcasad** focusing on various high level aspects of data plane is rolling out. But, as all of these works depends on P4, they are limited by the abstract forwarding model of P4 version 14 or few of the present hardware model applicable **ref** supported by P4 version 16.

Initial version of P4(P4v14) was strictly coupled with the proposed *'abstract forwarding model'* and RMT architecture. But later, P4 community followed the proven path of virtual machine world. In current version of P4(version 16) [18] data plane device architecture and hardware specific features are decoupled from the core programming language. Currently in a P4 based data plane program, developer needs to define a model of hardware and the actual processing logic separately. Decoupling hardware architecture and data plane programming language definition gives 2 crucial advantage: a) new hardware architecture can be supported any time b) common architecture can be created by all relevant stake holders. As a result, P4 can keep the path open for innovation yet promoting inter operability through common hardware architectures. But their architecture and hardware features are proprietary[ref]. As a result, data plane application developers are forced to use development toolkit from those hardware vendors. Moreover because of heterogeneous architecture and programmability features application written for one hardware architecture can not be run in all programmable data plane devices. For example, a tofino chip based programmable switch has different capability then a tofino based smartnic(CPU offloading is available here). These are obstacle for independent and open source development of data plane programmaing language and other tool sets.

P4 community is actively working toward developing a common hardware architecture that can cover various programmable data plane devices ranging from core switches to smartNIC. PSA [31] is the most matured attempt from P4 community toward that goal. It aims to list several common packet processing paths inside programmable switch and smart NIC. These paths are composed from multiple programmable P4 blocks. It also lists several related stateful data structure and functions for use in data plane. PSA can be identified as the first structured effort toward a standard abstraction layer of data plane. But PSA specification describes the components and their work flow in a descriptive way. As a result there is always chance of confusion among various implementation of PSA and it suffers from problem I-a, b n c and b(**in intro bold and italic the problems**). Besides, PSA has no option of programmablity for buffer [43] and scheduler [48, 63] in the architecture. Moreover, in future more complex and hybrid architecture may emerge. In this work, we intends to overcome these limitations.

Our approach is aligned with P4 community, as the proposed abstract model(AVS) is inspired by PSA. Hence, compilers can easily translate AVS to PSA components and instructions. On the other hand, it enhances the ***PSA*** architecture, firstly by adding programmability to buffer and scheduler components. Secondly, by providing a common functional structure for each components of data plane processing pipeline, it allows to model any network functions in a target independent manner(section 8.6). Thirdly, by expressing work flow of each components as hardware architecture agnostic extended finite state machine, it increases interoperability. Lastly, our work provides a definition of programmable switch and characterize it's components programmability features in a structured manner. This layouts a framework for comparing different programmable switch implementations. Moreover, P4 doesn't imposes any restriction on use of hardware architecture and it has decoupled hardware dependent compiler back end for various architectures. Hence, use of ***AVS*** like abstract model doesn't restrict use of other hardware architectures and the scope of innovation in hardware design remains wide open.

3. Programmable Switch

A packet(PKT)'s life inside a switch starts with reaching through incoming(ingress) port as few bits of data and ends with exiting through outgoing(egress) port.

$$PKT = \{Bit_1, Bit_2, Bit_3, ..., Bit_{Packet\ Length}\}$$

. How a data plane device acts is defined by

- ***S1*- Interpreting Packet:** how to interpret incoming set of bits(PKT) as different meaningful fields.
- ***S2*- Packet Metadata:** a data plane device keeps few hardware dependent information about a packet(metadata) for use in different stages of the packet's life cycle. Example: arrival time of a packet, this is necessary for packet scheduling.
- ***S3*- Packet processing work-flow:** Set of operations executed based on different fields of packet and various data structures.
- ***S4*- Control plane configuration parameters:** data plane lacks of global knowledge about the network. Control plane needs to configure parameters so that ***S3*** can be adapted with dynamic network conditions.

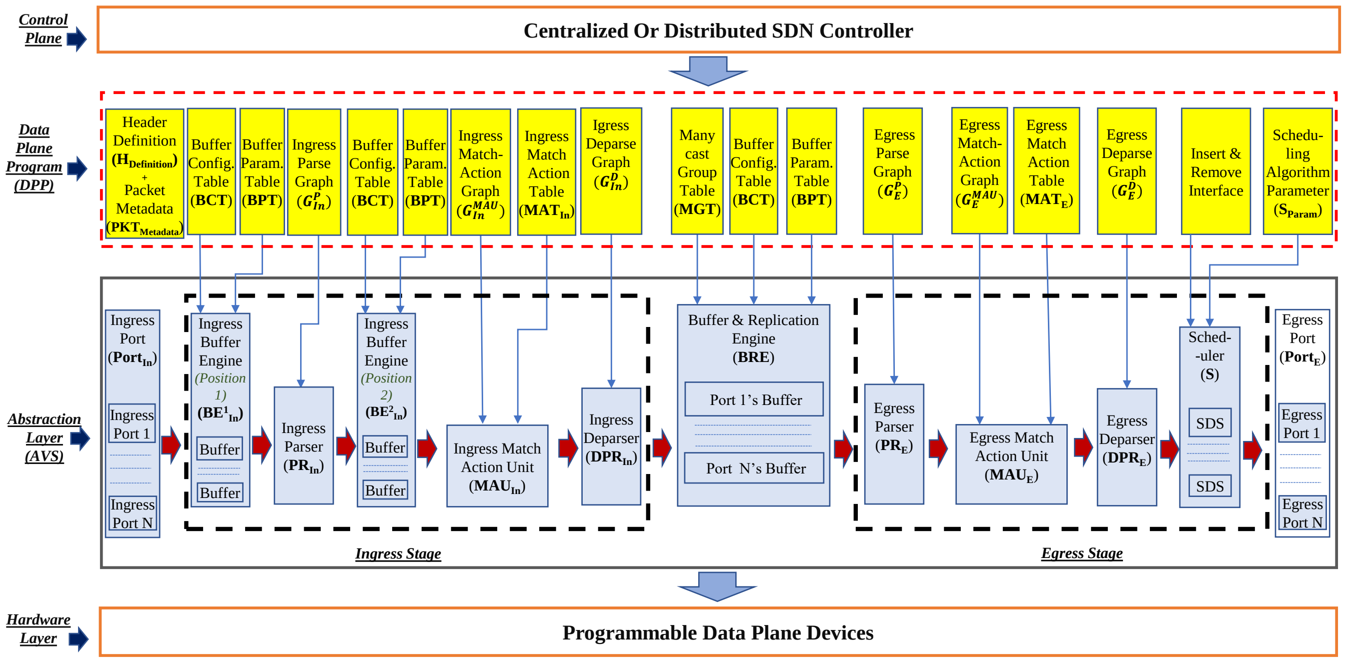
In legacy fixed function ASIC switches, ***S1-S3*** are static. Though ***S4*** is available, they are rather configuring parameters for fixed ***S3***. Hence, in legacy switches, once ***S1-S4*** are loaded, they can't be changed. These switches are optimized for specific protocol. But in programmable switch, ***S1-S4*** are not static and they can be modified at switch lifetime. Formally, any combination of hardware and/or software is a ***programmable switch or programmable data plane device***, if it fulfills following properties

- ***P1:*** Not bound to any specific protocol
- ***P2:*** Can execute any program, which
  - ***P2-1:*** contains logic and interface for ***S1-S4***
  - ***P2-2:*** is not coupled to hardware architecture
  - ***P2-3:*** can be loaded and unloaded at runtime

Any logical abstraction that can provide an uniform view and functionality of a *'programmable switch'* can be termed as an ***abstract model*** of programmable switch.

4. The Abstract Model(*AVS*)

Processing path of a packet inside a programmable switch can be modeled as an abstract pipeline of serially connected programmable components. Our proposed pipeline based abstract model of programmable switch(***'Abstract Virtual Switch(AVS)'***) is represented along with other components of SDN stack in Figure 2. Components(C) of ***AVS*** are a) ingress port($Port_{In}$) b) ingress parser(PR_{In}) c) ingress buffer engine(BE_{In}) d) ingress match action unit(MAU_{In}) e) ingress deparser(DPR_{In}) f) buffer and replication engine(BRE) g)



Control Plane
Centralized Or Distributed SDN Controller
Data Plane Program (DPP)
Header Definition ($H_{Definition}$) + Packet Metadata ($PKT_{Metadata}$)
Buffer Config. Table (BCT)
Buffer Param. Table (BPT)
Ingress Parse Graph (G^P_{In})
Buffer Config. Table (BCT)
Buffer Param. Table (BPT)
Ingress Match-Action Graph (G^{MAU}_{In})
Ingress Match Action Table (MAT_{In})
Igress Deparse Graph (G^D_{In})
Many cast Group Table (MGT)
Buffer Config. Table (BCT)
Buffer Param. Table (BPT)
Egress Parse Graph (G^P_E)
Egress Match-Action Graph (G^{MAU}_E)
Egress Match Action Table (MAT_E)
Egress Deparse Graph (G^D_E)
Insert & Remove Interface
Scheduling Algorithm Parameter (S_{Param})
Abstraction Layer (AVS)
Ingress Port ($Port_{In}$)
Ingress Port 1
Ingress Port N
Ingress Buffer Engine (Position 1) (BE^1_{In})
Buffer
Buffer
Ingress Parser (PR_{In})
Ingress Buffer Engine (Position 2) (BE^2_{In})
Buffer
Buffer
Ingress Match Action Unit (MAU_{In})
Ingress Deparser (DPR_{In})
Buffer & Replication Engine (BRE)
Port 1's Buffer
Port N's Buffer
Egress Parser (PR_E)
Egress Match Action Unit (MAU_E)
Egress Deparser (DPR_E)
Sched--uler (S)
SDS
SDS
Egress Port ($Port_E$)
Egress Port 1
Egress Port N
Ingress Stage
Egress Stage
Hardware Layer
Programmable Data Plane Devices

Figure 2: An Abstract Model for Programmable Switch

Symbol	Meaning
Simple State	**Simple State:** A state representing a stable situation of the workflow.
Complex State	**Complex State:** A state which refers to an entire State Machine. This sub state machine is shown as separate state machine in next parts.
Final State	**Final State:** A special kind of State (not Pseudo State) represents "a workflow has been completed".
State 1 transition = [trigger] [guard] [behavior-expression] [output] State 2	**Trigger:** An event causing transition from one state to another. **Guard:** A Boolean expression, which enables the trigger when evaluated TRUE. **Behavior-expression:** An expression specifying what happens when transition occurs. **Output**: is any kind of data structure or value that is passed as result of trigger action to next state If any of these 4 is not present for a transition, it is marked as "-".

Figure 3: EFSM Notations

egress parser(PR_E) h) egress match action unit(MAU_E) i) egress deparser(DPR_E) j) scheduler(S) k) egress port($Port_E$).

Based on egress port selection for a packet, the ***AVS*** pipeline is logically divided into 2 stages: **ingress stage**- before selecting the egress port and **egress stage**- after selecting egress port. After entering egress stage a packet's egress port can not be changed. This is particularly important when packets are replicated (clone, broadcast, multicast etc.) in egress stage. On such cases, packets may require further match action processing in egress stage. For example, traffic rate controlling at each outgoing port requires action at egress stage. If egress processing is not required in a hardware implementation, vendors may skip that part.

$$AVS = f_{Egress\ Stage}(f_{BRE}(f_{Ingress\ Stage}(PKT)))$$

Where,

$$f_{Ingress\ Stage}(PKT) = f_{Port_{in}}(fPR_{in}(f_{B_i}()))\cup MAU_{in}\cup DPR_{in}$$

$$f_{Ingress\ Stage}(PKT) = PKT_Ingress = f_{DPR_{in}}(f_{MAU_{in}}(f_{B_i}(fPR_{in}(f_{Port_{in}}(PKT)))))$$

and

$$f_{Egress\ Stage}(PKT) = f_{DPR_e}(f_{MAU_e}(fPR_e(f_{Port_{in}}(PKT_ingress))))$$

AVS may represent a single non virtualized programmable switch or a slice in a virtualized programmable switch or just a software switch. It is compiler's duty to map components of ***AVS*** to actual hardware resources. ***AVS*** provides a uniform view of data plane over heterogeneous programmable switches. How the data plane will behave is defined by **data plane program(DPP)** (Figure 2). Instead of heterogeneous hardware architecture, DPP contains logic to program AVS components. Management plane handles (un)loading of DPP. On the other hand control plane controls runtime behavior of ***AVS*** by configuring parameters.

5. AVS Components

Each component(C) of ***AVS*** represents a programmable unit and they are needed to be programmed from outside before a packet processing starts. These are supplied as DPP. On the other hand CP controls each component's behavior by configuring parameters through southbound interface. Degree of programmability of ***AVS*** and it's components depends on these 2 kind of features. For formal discussion we classified these features as following:

- ***Compile Time Programmability(CTP) Features***: Set of instructions a programmable component can execute(comparable to cpu instruction set). How C will behave at run time is defined through these features.
- ***Run Time Configurability(RTC) Features***: Capability of adjusting run-time behavior of **CTP** features through configuring parameters. Control plane uses these to manage the behavior of a component C.

Irrespective of hardware implementation, CTP and RTC features can be exposed to upper layer as an uniform API. Compiler translates these API call to actual hardware instruction. Therefore, DPP is a program expressed through CTP features and contains runtime processing logic of a component C. And control plane application controls behavior of those processing logic at run time through RTC features. DPP also contains data structures for facilitating control plane communication with C through RTC features.

Formally, a component(C) can be represented as a component function f_c,

$$\begin{aligned} f_c &: \mathcal{I} \rightarrow \mathcal{O} \\ f_c(X, ProcLogic, Conf_{param}) &= Y \end{aligned} \tag{1}$$

Here,

- $\mathcal{I}$ is the domain of f_c, it represents the set of all possible values that C accepts.
- $\mathcal{O}$ is the co-domain(range) of f_c, it represents the set of all possible values that C can emit as output.
- $ProcLogic$ is the processing logic to be executed by the component. It is represented using CTP, RTC and control flow
- X is the input to the component, $X \in I$
- Y is the output of the component, $Y \in O$
- $Conf_{param}$ is the set of parameters ***CP*** can configure to control the behavior of the component at run time. This is a subset of CTP features

A component can modify it's input(X) and return a modified version of the input(X) as output(Y). On such case, we used $X\ *$ in place of Y to represent the return value of a function(* signifies an object is modified inside the component). Parser, match action unit and deparser exists both in *'ingress'* and *'egress'* stage. Their internal structure and parameters structures are same for both the stage but parameter names are different. In section 6, those components are discussed in details only for ingress stage. To differentiate between ingress and egress stage components and parameters ***In*** and ***E*** subscripts are used respectively.

In next few subsections, each component of ***AVS*** are discussed in details. How they work and their programmability features are also discussed. For expressing work-flow of the components extended finite state machine(EFSM) approach is used. In Figure 4, an EFSM is presented to define how a packet(***PKT***) goes through different components of the **AVS**. Separate sub state machines are used when necessary. Table 1 contains complex states of this state machine and reference to corresponding sub-state-machine with section number where the components are discussed. Notations used in the EFSMs are described in Figure 3.

Table 1
Complex state of Figure 4 and corresponding sub-state-machines

State	Sub statemachine	Section
S2,S5, S10,S11	Figure 8	Section 6.3
S3,S6,S12	Figure 9	Section 6.3
S4,S13	Figure 7	Section 6.2, 6.7
S7,S14	Figure 10	Section 6.4, 6.8
S8 ,S15	Figure 11	Section 6.5, 6.9

5.1. Related Terminology

Before driving into details of each component, few relevant terminologies and notations are discussed in this section.

5.1.1. Bit Space($\mathcal{BS}$)

Upon receipt a packet(PKT) is a sequence of one and zero.

$$PKT = \{Bit_1, Bit_2, Bit_3, ..., Bit_{Packet\ Length}\}$$

Formally, a packet(PKT) is a point in the space $BS = \{0,1\}^{Maximum\ Packet\ Length}$.

5.1.2. Header Definition($H_{Definition}$)

To do meaningful operation on a PKT, it is needed to be interpreted as fields of different protocols. Header definition provides structure of these fields. Definition of i'th header field

$$H_f^i = (unique\ ID/name,\ starting\ position\ in\ packet,\ length)$$

Here,

$$(1 \le i \le p)\ and$$

$$p\ =\ total\ number\ of\ fields\ in\ H_{Definition}$$

Starting position of H_f^i provides relative order of the field in packet. All H_f^i together from packet header,

$$PKT_{header} = \bigcup_{i=1}^{p} H_f^i$$

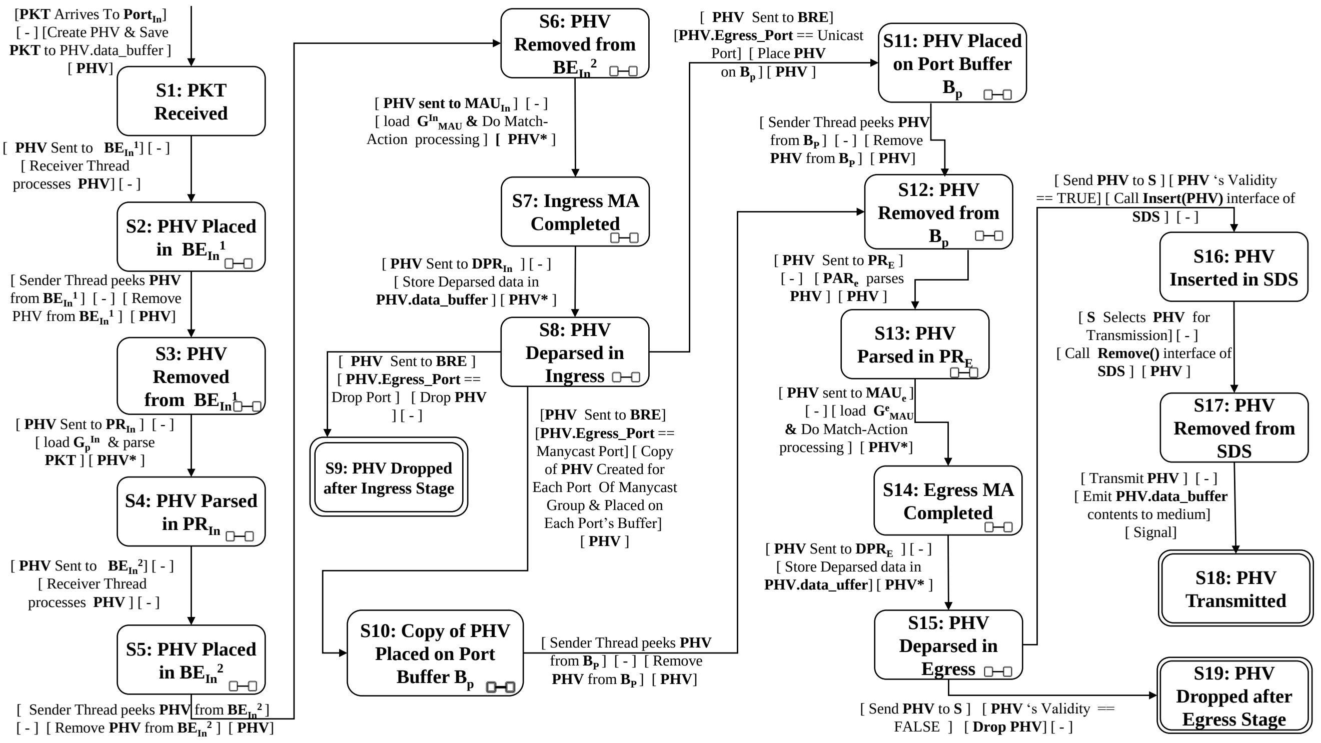

[PKT Arrives To Port_In] [-] [Create PHV & Save PKT to PHV.data_buffer] [PHV]
S1: PKT Received
[PHV Sent to BE_In^1] [-] [Receiver Thread processes PHV] [-]
S2: PHV Placed in BE_In^1
[Sender Thread peeks PHV from BE_In^1] [-] [Remove PHV from BE_In^1] [PHV]
S3: PHV Removed from BE_In^1
[PHV Sent to PR_In] [-] [load G_p^In & parse PKT] [PHV*]
S4: PHV Parsed in PR_In
[PHV Sent to BE_In^2] [-] [Receiver Thread processes PHV] [-]
S5: PHV Placed in BE_In^2
[Sender Thread peeks PHV from BE_In^2] [-] [Remove PHV from BE_In^2] [PHV]
S6: PHV Removed from BE_In^2
[PHV sent to MAU_In] [-] [load G^In_MAU & Do Match-Action processing] [PHV*]
S7: Ingress MA Completed
[PHV Sent to DPR_In] [-] [Store Deparsed data in PHV.data_buffer] [PHV*]
S8: PHV Deparsed in Ingress
[PHV Sent to BRE] [PHV.Egress_Port == Drop Port] [Drop PHV] [-]
S9: PHV Dropped after Ingress Stage
[PHV Sent to BRE] [PHV.Egress_Port == Manycast Port] [Copy of PHV Created for Each Port Of Manycast Group & Placed on Each Port's Buffer] [PHV]
S10: Copy of PHV Placed on Port Buffer B_p
[Sender Thread peeks PHV from B_p] [-] [Remove PHV from B_p] [PHV]
[PHV Sent to BRE] [PHV.Egress_Port == Unicast Port] [Place PHV on B_p] [PHV]
S11: PHV Placed on Port Buffer B_p
[Sender Thread peeks PHV from B_p] [-] [Remove PHV from B_p] [PHV]
S12: PHV Removed from B_p
[PHV Sent to PR_E] [-] [PAR_e parses PHV] [PHV]
S13: PHV Parsed in PR_E
[PHV sent to MAU_e] [-] [load G^e_MAU & Do Match-Action processing] [PHV*]
S14: Egress MA Completed
[PHV Sent to DPR_E] [-] [Store Deparsed data in PHV.data_uffer] [PHV*]
S15: PHV Deparsed in Egress
[Send PHV to S] [PHV 's Validity == TRUE] [Call Insert(PHV) interface of SDS] [-]
S16: PHV Inserted in SDS
[S Selects PHV for Transmission] [-] [Call Remove() interface of SDS] [PHV]
S17: PHV Removed from SDS
[Transmit PHV] [-] [Emit PHV.data_buffer contents to medium] [Signal]
S18: PHV Transmitted
[Send PHV to S] [PHV 's Validity == FALSE] [Drop PHV] [-]
S19: PHV Dropped after Egress Stage

Figure 4: Packet's Life Cycle EFSM

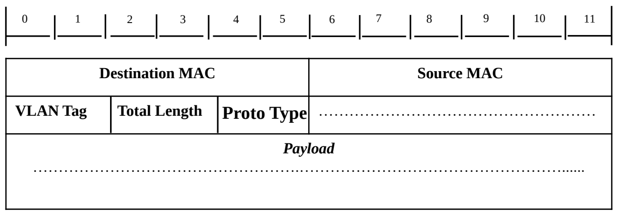


Figure 5: A sample packet format

Rest of the packet is considered as payload. Thus, $H_{Definition}$ divides a packet in 2 parts: header and payload.

$$Header_{length} = \sum_{i=1}^{p} H_f^i.length$$

5.1.3. Packet Metadata($PKT_{Metadata}$)

Depending on actual hardware implementation, switches maintain some metadata ($PKT_{Metadata}$) about a packet (ingress or egress port, time of arrival, packet unicast or multicast type etc.). I'th metadata field

$$M_f^i = (uniqueID/name, data\ type/length)$$

Here,

$$(1 \le i \le q)\ and$$

$$q\ =\ total\ number\ of\ fields\ in\ PKT_{Metadata}$$

All M_f^i together from packet metadata,

$$PKT_{Metadata} = \bigcup_{i=1}^{q} M_f^i$$

A sample packet format is shown in 5, corresponding header definition and a sample packet metadata are shown in Figure 6.(a).

5.1.4. Packet Header Vector(PHV) and Space(PHVS)

Header definition and metadata defines a $p + q$ dimensional space named Packet Header Vector Space(PHVS). Each point in this space is termed as packet header vector(PHV). PHV can be considered as a container for all the attributes of a packet in key-value format. Where, key represents a header field(H_f) or metadata field (M_f) or any other field derived in the pipeline and value represents corresponding data.

$$PHV = PKT_{header} \cup PKT_{Metadata}$$

Each PHV represents a point and flow represents a region in PHVS. In a PHV, all the fields in definition may not be present at some point in AVS. They may be filled up by different components in pipeline at different stage of the packet's life-cycle.

5.1.5. Ordered PHV Set(PHV_{set}, <)

Let PHV_{set} is a set of PHV. An *'Ordered PHV Set'* is an ordered pair ($PHV_{set}, \le$) of set PHV_{set} and the binary relation $\le$ contained in $PHV_{set} \times PHV_{set}$, such that

- **Reflexive:** $\forall PHV \in PHV_{set}\ :\ PHV \le PHV$

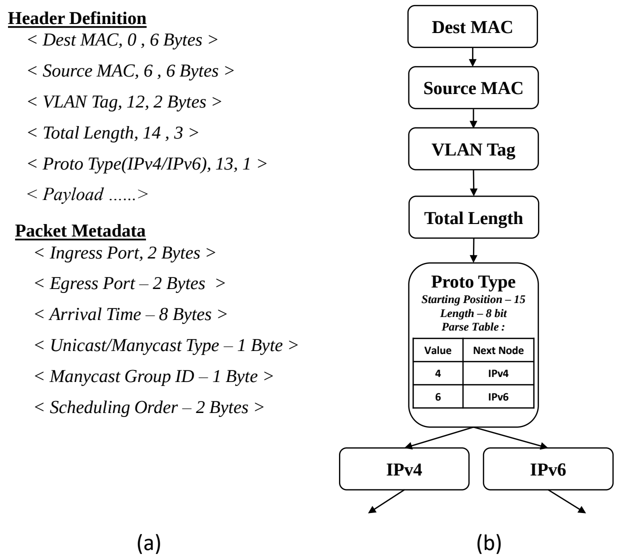

Header Definition
< Dest MAC, 0 , 6 Bytes >
< Source MAC, 6 , 6 Bytes >
< VLAN Tag, 12, 2 Bytes >
< Total Length, 14 , 3 >
< Proto Type(IPv4/IPv6), 13, 1 >
< Payload ......>
Packet Metadata
< Ingress Port, 2 Bytes >
< Egress Port – 2 Bytes >
< Arrival Time – 8 Bytes >
< Unicast/Manycast Type – 1 Byte >
< Manycast Group ID – 1 Byte >
< Scheduling Order – 2 Bytes >
Dest MAC
Source MAC
VLAN Tag
Total Length
Proto Type
Starting Position – 15
Length – 8 bit
Parse Table :
Value
Next Node
4
IPv4
6
IPv6
IPv4
IPv6
(a)
(b)

Figure 6: (a) Header definition and metadata for the sample packet of Figure 5 (b) Parse graph for sample packet of Figure 5(only "Proto Type" node is shown in expanded format)

- **Transitive:** $\forall PHV_i, PHV_j, PHV_k \in PHV_{set}$: $[(PHV_i \leq PHV_j) \&\& (PHV_j \leq PHV_k)]$ $\Rightarrow ((PHV_i \leq PHV_k))$
- **Anti-symmetry:** $\forall PHV_i, PHV_j \in PHV_{set}$: $[(PHV_i \leq PHV_j) \&\& (PHV_j \leq PHV_i)] \Rightarrow$ $((PHV_i = PHV_k))$

Here the relation is defined based on one or more fields of PHV such that, for a field $x \in PHV$ and 2 elements of the set $PHV_m, PHV_n \in PHV_{set}$, $(PHV_m.x.value \leq PHV_n.x.value) \Rightarrow (PHV_m \leq PHV_n)$ and if $(PHV_m.x.value = PHV_n.x.value)$ then another filed $y \in PHV$ can be used to break tie. As packets received from each port have distinct arrival time, and each port has different number in a switch, we can always get a total order on a set of PHV

Header Definition not represents any component of AVS pipeline. But along with packet metadata it forms PHVS. Metdata is fixed for an architecture and therefore PHVS is mainly dependent on Header definition. All most all the component's domain and range is PHVS, hence header definition is a crucial part of DPP. Because, It lays the foundation of a programmable switch's work flow by defining how to interpret a raw packet.

6. Analysis of Components

6.1. Ingress Port($Port_{In}$)

In ***AVS***, sole task of an ***ingress port($Port_{In}$)*** is to receive a set of bits(PKT) and store as a PHV. How a hardware level frame is received and format of the frame is out of the scope of our discussion. After receiving, a new PHV is initiated and PKT is stored in ***data_buffer*** variable of PHV. ***data_buffer*** is a storage for an array of bits. In this phase,

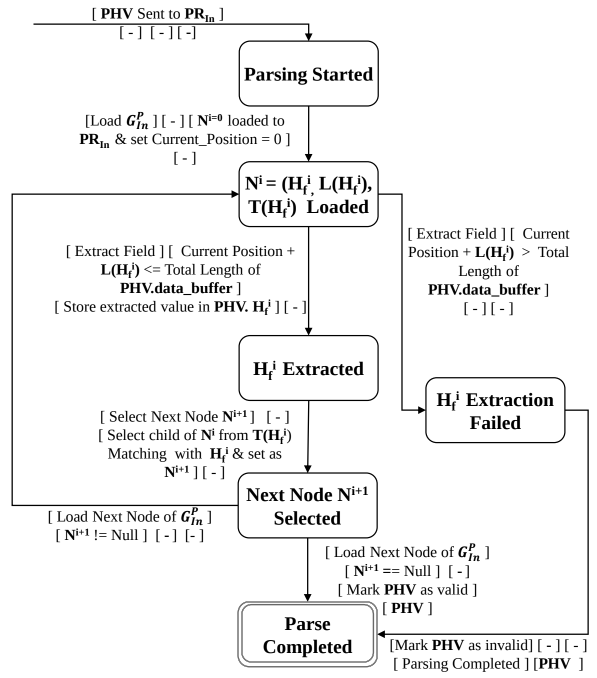

[PHV Sent to PR_In]
[-] [-] [-]
Parsing Started
[Load G^P_In] [-] [N^i=0 loaded to PR_In & set Current_Position = 0]
[-]
N^i = (H_f^i, L(H_f^i), T(H_f^i) Loaded
[Extract Field] [Current Position + L(H_f^i) <= Total Length of PHV.data_buffer]
[Store extracted value in PHV. H_f^i] [-]
[Extract Field] [Current Position + L(H_f^i) > Total Length of PHV.data_buffer]
[-] [-]
H_f^i Extracted
H_f^i Extraction Failed
[Select Next Node N^i+1] [-]
[Select child of N^i from T(H_f^i) Matching with H_f^i & set as N^i+1] [-]
Next Node N^i+1 Selected
[Load Next Node of G^P_In]
[N^i+1 != Null] [-] [-]
[Load Next Node of G^P_In]
[N^i+1 == Null] [-]
[Mark PHV as valid]
[PHV]
Parse Completed
[Mark PHV as invalid] [-] [-]
[Parsing Completed] [PHV]

Figure 7: Programmable Parser EFSM

necessary metadata(ingress port, arrival time etc.) are also stored in the PHV. Then the PHV is passed to next component in pipeline. In AVS, ingress ports do not have any kind of programmability in terms of CTP and RTC features.

$$f_{Port_{In}} : BS \rightarrow PHVS$$
$$f_{Port_{In}}(PKT, Null, Null) = PHV,$$

6.2. Ingress Parser(PR_{In})

PR_{In} parses array of bits stored in $PHV.data\_buffer$ to different header fields. Parsing logic is provided through parse graph(G^p_{In}). Each node(N^i) $\in G^p_{In}$ contains: a) Header field information: $H^i_f \in PHV$ to be parsed, starting position in PKT and length of $(L(H^i_f))$ and b) Parse table($(T(H^i_f))$: Lists possible values of H^i_f and corresponding next node [29]. Work-flow of a programmable parser is shown in fig. 7 as EFSM. After parsing, PHV is passed to next component in pipeline.

$$f_{PR_{In}} : BS \rightarrow PHVS$$
$$f_{PR_{In}}(PHV.data\_buffer, G^p_{In}, Null) = PHV^*$$

CTP features: a) maximum length limit: maximum length of PKT that can be parsed by PR_{In}. This limit is important for bounded runtime. b) supported data types: different data types(including all the standard primitive data types: int, float, char etc) that can be parsed by the parser c) granularity of field parsing: can the parser circuit parse individual bit or variable number number of bits to a field. d) loop support: Can the parser support loop. This is important because some parser has circuit for parsing whole packet in a single cycle. for example assume a parser can only support 4000 bytes and it can parse whole 4000 bytes in a fixed number of cycle. This will give a bounded run time. but if a parser support loop then 2 cases may happen. A parser can parse 4000 bytes in n iteration of loop. Then in each iteration 4000/n bytes will be parsed which will take less time if the packet is small. in another case, if it supports loop then running the parser 'x' times will parse $x * (4000/n)$ bytes which will take more time and most importantly this is not fixed for all parsing, hence the runtime is not bounded. This may effect the line rate.

RTC features: a) modifiability of G^{p}_{In} : is the parse graph($G_p{}^{In}$) and it's nodes contents are modifiable at runtime by CP. Assume a parser is configured to parse a filed based on some condition. If the branches of prase graph can be modified at run time, then frameworks can inject code in the switches at run time.

6.3. Ingress Buffer Engine(BE_{In})

General role of a buffer is to temporarily hold packets. Though a programmable buffer can be expressed under match-action semantics, but its significance in various important applications **ref to prog buffer and any buffer sizing paper** warrants a separate component for buffer. Ingress buffer engine(BE_{In}) consists of a set of buffer(B_{set}). Assuming n individual buffers(B_i) in the engine

$$B_{set} = \bigcup_{i=1}^{n} B_i$$

Their size may be fixed or programmable. CP can control these sizes through configuring ***'Buffer Parameter Table (BPT)'***(Table 3). There are 2 possible positions of BE_{In} in AVS. Any one or both can be used.

First one(BE^1_{In}) is just after the $Port_{In}$. BE^1_{In} holds PHV received from ports. CP controls PHV from which ingress port should be stored to which buffer through ***'Buffer Configuration Table(BCT)'***.

Second one(BE^2_{In}) is after ingress parser. This is a more generalized and useful implementation which can store PHV to a buffer based on PHV fields. Here, instead of only ingress port, CP configures based on which header/metadata field, a PHV should be sent to which buffer through ***'Buffer Configuration Table(BCT)'***(Table 2). Priority for matching PHV's also can be assigned through BCT.

Buffer engine's work-flow can be expressed as 2 threads: a) ***Receiver Thread***(Figure 8: for inserting PHV) in buffers and b) ***Sender Thread*** (Figure 9): for moving out PHV from buffers and sending to next component in pipeline. These 2 threads behavior are controlled by 2 separate state variable for each of the buffer($B_i \in B_{Set}$). These are configurable from control plane via ***'Buffer Parameter Table(BPT)'***. These 2 are: a) RX mode: when RX of a B_i is true, receiver thread either receive PHV and store them (from port or ingress parser) or drop them if false b) TX mode: hold flow from buffer(buffer in pause state) when TX is false or release(buffer in resume state) packet/PHV to next component in pipeline when TX of the buffer(B_i) is true.

Table 2
Buffer Configuration Table(BCT) for sample packet of Figure 5

PHV field Name	PHV field Value	Buffer ID	Priority
VLAN Tag	0x 00 15 25	3	1
VLAN Tag	00 45 25	6	0
...	...	...	...

Table 3
Buffer Parameter Table(BPT) for sample packet of Figure 5

Buffer ID	Size	RX	TX
1	2048	True	False
5	3072	False	True
...	...	...	...

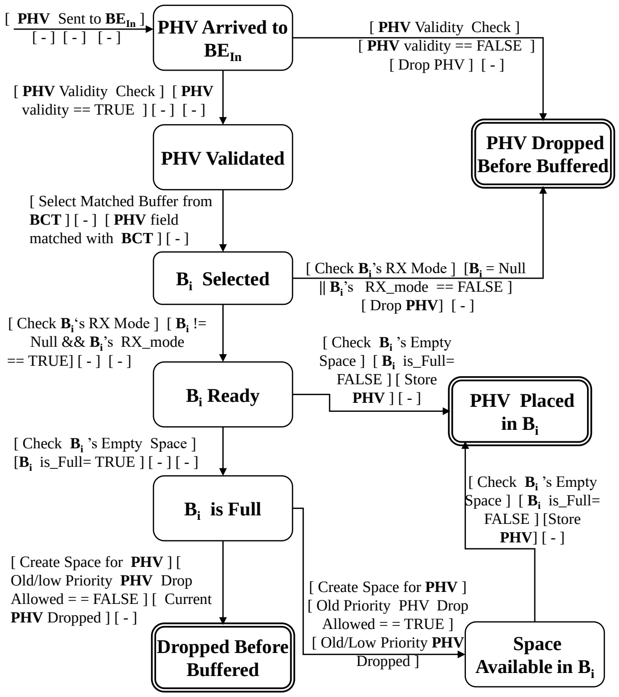

[PHV Sent to BE_In] [-] [-] [-]
PHV Arrived to BE_In
[PHV Validity Check] [PHV validity == FALSE] [Drop PHV] [-]
[PHV Validity Check] [PHV validity == TRUE] [-] [-]
PHV Validated
PHV Dropped Before Buffered
[Select Matched Buffer from BCT] [-] [PHV field matched with BCT] [-]
B_i Selected
[Check B_i's RX Mode] [B_i = Null || B_i's RX_mode == FALSE] [Drop PHV] [-]
[Check B_i's RX Mode] [B_i != Null && B_i's RX_mode == TRUE] [-] [-]
[Check B_i 's Empty Space] [B_i is_Full= FALSE] [Store PHV] [-]
B_i Ready
PHV Placed in B_i
[Check B_i 's Empty Space] [B_i is_Full= TRUE] [-] [-]
B_i is Full
[Check B_i 's Empty Space] [B_i is_Full= FALSE] [Store PHV] [-]
[Create Space for PHV] [Old/low Priority PHV Drop Allowed = = FALSE] [Current PHV Dropped] [-]
[Create Space for PHV] [Old Priority PHV Drop Allowed = = TRUE] [Old/Low Priority PHV Dropped]
Dropped Before Buffered
Space Available in B_i

Figure 8: Buffer Receiver Thread EFSM

Receiver thread of buffer engine can be expressed as

$$f_{BE_{receiver}} : PHVS \rightarrow B_{set}$$
$$f_{BE_{receiver}}(PHV, Null, \{BPT, BCT\}) = B_{set}; \ (PHV \ inserted \ in \ B_{set})$$

and sender thread as(assuming a round-robin order for selecting next buffer from where next PHV will be peeked)

$$f_{BE_{sender}} : B_{set} \rightarrow PHVS$$
$$f_{BE_{sender}}(Null, Null, Null) = PHV$$

CTP features: a) buffer size & number controlling: number of buffers and corresponding size are programmable or

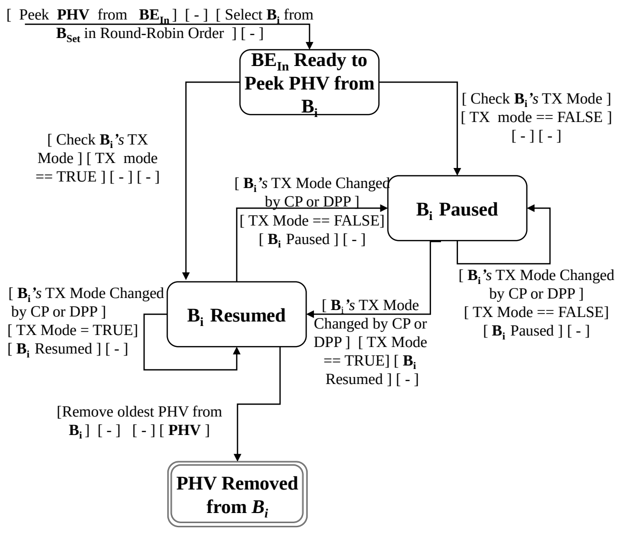

[Peek PHV from BE$_{In}$] [-] [Select B$_i$ from B$_{Set}$ in Round-Robin Order] [-]
BE$_{In}$ Ready to Peek PHV from B$_i$
[Check B$_i$'s TX Mode] [TX mode == FALSE] [-] [-]
[Check B$_i$'s TX Mode] [TX mode == TRUE] [-] [-]
[B$_i$'s TX Mode Changed by CP or DPP] [TX Mode == FALSE] [B$_i$ Paused] [-]
B$_i$ Paused
[B$_i$'s TX Mode Changed by CP or DPP] [TX Mode == FALSE] [B$_i$ Paused] [-]
[B$_i$'s TX Mode Changed by CP or DPP] [TX Mode = TRUE] [B$_i$ Resumed] [-]
B$_i$ Resumed
[B$_i$'s TX Mode Changed by CP or DPP] [TX Mode == TRUE] [B$_i$ Resumed] [-]
[Remove oldest PHV from B$_i$] [-] [-] [PHV]
PHV Removed from B_i

Figure 9: Buffer Sender Thread EFSM

not b) BCT creation: is BCT fixed or can be declared(size and definition) in compile time c) global access of buffer property: can other components of AVS access buffer properties? This is important because of based on buffer depth queuing model works. Moreover buffer sizing is extremely important for congestion control and traffic burst controlling. If we want to implement any protocol that wants to uses these information for traffic controlling then cross component access of these information are necessary. Because, the algorithm may contain logic to use buffer engine depth of buffer in ingress match action unit.

RTC features: a) BPT modifiability: can CP manipulate buffer size through BPT. Ex. assume a switch that has allocated a certain size for IPv4 flows. Now If it wants to handle more large size burst of IPv4 traffic it have to increase the buffer size b) BCT modifiability: can CP add, modify or delete entry in BCT table.

6.4. Ingress Match Action Unit(MAU_{In})

This component can be considered as the computational unit of a data plane device. What is cpu in a server, match-action unit can be considered as that unit for programmable data plane devices. Increasingly, more complex and generalized processing units (FPGA, CPU, GPU) are being proposed as the main computational unit for PDP devices. but match-action base hardware are still dominating. In this work, we have expressed the generic packet header based computations in match-action semantics. hardware implementations may use CPU/GPU/RMT circuit for implementing the actual logic. But the concept is generalized for the component.

In this component, values of PHV field or any other data derived from them are matched with either a) control plane configured data or b) data collected by data plane itself. Based on matching result different actions are executed. Control plane configured data are kept in a table like data structure. Control plane can store, modify or delete data

Table 4
A sample MAT for *'Proto Type'* field in sample packet of Figure 5

Match Type	Values to be Matched	Action[s]
Exact	4	Increase IPv4 counter
Exact	6	Drop Packet
...	...	...

from these tables at runtime through southbound API. Data plane's collected(or derived) data can be of 3 types a) stateful information about a flow(meter, register, counter etc.) b) stateless metadata about a packet and c) any constant value supplied at compile time in DPP.

Programmability and performance of a switch mostly depends on the set of actions it allows programmer to use. Actions can be of different types : **stateless** - only access and modify current PHV fields, **stateful** - access and update previously stored data about a flow and use them to to update PHV.

Different data plane programming language may represent match action semantics in different syntax. But fundamentally, match-action block requires 4 information, a) name/id of the PHV field which will be matched/compared b) control or data plane supplied value, against which PHV field values are matched or compared c) matching/comparison method (exact, ternary, <, > , != etc.) d) one or more action(action block) to execute based on comparison result.

Without loosing generality, here we assumed that, storing computational logic for packet processing can be represented as graph. In current literature, generic data structure for storing processing logic information is termed as $Match-Action-Table(MAT)$ [7]. Processing logic for a match-action-unit can be represented as a match-action graph (G^{MAU}_{In}), where each node(N^i) represents match-action logic for a specific field in PHV and edges are control flow. Let, a protocol field $p\_f \in PHV$, it's value in PHV is $p\_f.value$ and data structure for storing it's match-action information is MAT_{p_f}. $MAT_{p\_f} = \{match\,type \times values\,to\,be\,matched\,with\,p\_f \times action[s]\}$. Corresponding N^i is a tuple (p_f, MAT_{p_f}). Result of match-action processing for each field can be stored in stateful memory in case of stateful operations. Or they can simply update some fields in PHV(ex. updating destination port according to routing table). An example MAT for *'Proto Type'* field of sample packet (Figure 5) is presented in Table 4.

Let, MAT_SET_{In} = Set of MAT for all the nodes(N^i) in G^{MAU}_{In}. Formally ,

$$f_{MAU_{In}} : PHVS \rightarrow PHVS$$
$$f_{MAU_{In}}(PHV, G^{MAU}_{In}, MAT\_SET_{In}) = PHV^*$$

Work-flow of ingress match action unit(MAU_{In}) is presented in Fig. 10. The same work-flow is also applicable for egress match action unit(MAU_{In}) described in section 6.8.

CTP features: a) data type support in MAT: data types (int, float, bit pattern, string etc.) allowed for lookup in MAT

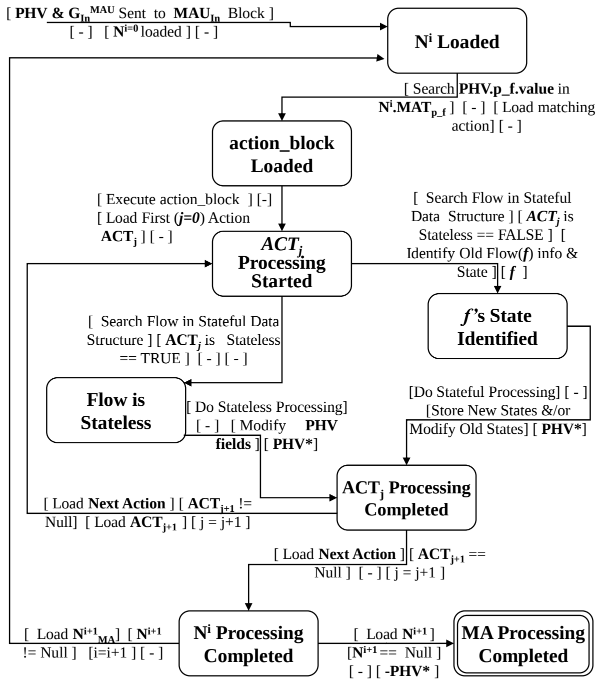

[PHV & G_In^MAU Sent to MAU_In Block] [-] [N^i=0 loaded] [-]
N^i Loaded
[Search PHV.p_f.value in N^i.MAT_p_f] [-] [Load matching action] [-]
action_block Loaded
[Execute action_block] [-] [Load First (j=0) Action ACT_j] [-]
ACT_j Processing Started
[Search Flow in Stateful Data Structure] [ACT_j is Stateless == FALSE] [Identify Old Flow(f) info & State] [f]
f's State Identified
[Search Flow in Stateful Data Structure] [ACT_j is Stateless == TRUE] [-] [-]
Flow is Stateless
[Do Stateless Processing] [-] [Modify PHV fields] [PHV*]
[Do Stateful Processing] [-] [Store New States &/or Modify Old States] [PHV*]
ACT_j Processing Completed
[Load Next Action] [ACT_j+1 != Null] [Load ACT_j+1] [j = j+1]
[Load Next Action] [ACT_j+1 == Null] [-] [j = j+1]
N^i Processing Completed
[Load N^i+1_MA] [N^i+1 != Null] [i=i+1] [-]
[Load N^i+1] [N^i+1 == Null] [-] [-PHV*]
MA Processing Completed

Figure 10: Ingress Match Action EFSM

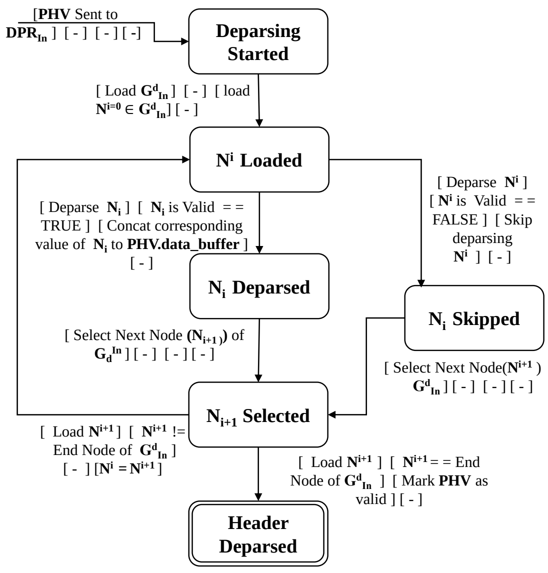

[PHV Sent to DPR_In] [-] [-] [-]
Deparsing Started
[Load G^d_In] [-] [load N^i=0 ∈ G^d_In] [-]
N^i Loaded
[Deparse N_i] [N_i is Valid == TRUE] [Concat corresponding value of N_i to PHV.data_buffer] [-]
N_i Deparsed
[Deparse N^i] [N^i is Valid == FALSE] [Skip deparsing N^i] [-]
N_i Skipped
[Select Next Node (N_i+1) of G_d^In] [-] [-] [-]
[Select Next Node(N^i+1) of G^d_In] [-] [-] [-]
N_i+1 Selected
[Load N^i+1] [N^i+1 != End Node of G^d_In] [-] [N^i = N^i+1]
[Load N^i+1] [N^i+1 == End Node of G^d_In] [Mark PHV as valid] [-]
Header Deparsed

Figure 11: Ingress Deparser EFSM

b) matching type support in MAT: type of look ups (longest prefix match, ternary, exact, range etc.) are supported c) MAT size & number controlling: MAT size are fixed length or their size can be declared in compile time d) availability of stateful action: is it possible to maintain state information about the flows e) availability of stateful data structure: what are the data structures for keeping state information about the flows (register, counter etc.). Custom data structure can be created or not f) state sharing among different components: are the flow states shareable among different components of AVS. g) MAT modifiability from DPP: is the MAT tables modifiable from inside the data plane program. This is necessary for advanced and faster decision making in data plane [68]. h) available action types: different type of actions can be done on the fields of a PHV.

RTC features: a) MAT controlling: can the CP add/ remove/update match-action tables entries b) stateful data access from CP: can the CP read/write stateful data of a flow from/to the DP.

6.5. Ingress Deparser(DPR_{In})

This component deserializes a PHV. All the fields of a PHV may not be necessary for emitting to next component. Some may be dropped or some may be included more than once. Moreover all the header fields declared in PHV may not be valid for all the packets. In deparser definition, it is defined which of the valid header fields will be included in the outgoing packet. Besides the PHV fields, constant data also can be included in the packet. Deparser definition is transformed to a graph($G_d{}^{In}$) in compilation stage. In this graph each node represents a field of PHV or any arbitrary data to be emitted. DPR_{in} checks validity of each node and concatenates the field data in **data_buffer**(a per packet buffer for storing bits). This also inherently defines each fields relative position in the packet to be emitted. After deparsing all the nodes of $G_d{}^{In}$, payload is concatenated to the **data_buffer**. Work-flow of an ingress deparser unit is represented as EFSM in Figure 11.

$$f_{DPR_{in}} : PHVS \rightarrow PHVS$$
$$f_{DPR_{in}}(PHV, G_d^{In}, Null) = PHV^*$$

CTP features: a) maximum length limit: maximum length of a packet that can be created through the deparser unit b) data type support: data types that can be emitted by the deparser.

RTC features: a) G_d^{In} modifiability: can CP modify G_d^{In} at run time.

6.6. Buffer and Replication Engine(BRE)

BRE is the bridge between ingress and egress stage. It contains buffer for each egress port. These buffers can be just simple FIFO buffer or fully programmable buffer (sec. 6.3). But header/metadata based buffer assignment of programmable buffer (BE_{In}^2 section 6.3) is not needed here, because we assume BRE buffers will be reserved as per port resource. Moreover, adding extra programmable circuit increases the packet processing delay.

$$BRE.B_{set} = \bigcup_{i=1}^{n} B_i \text{ , } n = total\ number\ of\ egress\ port$$

For a PHV, egress port selection is done in ingress match action unit. After ingress deparser, the PHV may be destined toward either a single egress port(**unicast**) or more than one egress port (**manycast - multicast, broadcast**).

- For unicast port, PHV is simply moved to egress port's buffer with all of its metadata.
- For manycast port, control plane needs to configure group membership of the ports through *'Manycast Group Table(MGT)'*. BRE will make necessary number of copies of the PHV for each member of the group. Those PHV are placed to buffer of relevant egress ports.

Insertion of PHV to port buffer (B_p) are handled by receiver thread of buffer. Sender thread of buffer removes PHV from port buffer(B_p) and either drops or passes to egress stage. Work-flow of these 2 threads are same as of ingress buffer engine discussed in section 6.3.

Formally

$$f_{BRE} : PHVS \rightarrow PHVS^n$$

$$f_{BRE}(PHV, Null, MGT) = n\,copies\,of\,PHV$$

Here, n = total number of egress port in a manycast group. For unicast packets n = 1.

CTP features: a) all of the CTP features of BE_{in} (except BCT)

RTC features: a) all of the RTC features of BE_{in} (except BCT configuration) b) Manycast group membership table configuration: can CP modify manycast group and their members at runtime.

6.7. Egress Parser(PR_E)

After PHV are removed from BRE, they are sent to egress parser. Structurally it is same as ingress parser. Only difference is, after finishing egress parsing of packet header, instead of sending to a buffer, PHVs are sent to egress match-action unit.

$$f_{PR_E} : BS \rightarrow PHVS$$

$$f_{PR_E}(PHV.data\_buffer, G_E^p, Null) = PHV^*$$

6.8. Egress Match Action Unit(MAU_E)

Same as ingress match action. Only restriction is, egress port can not be changed in this stage. Formally ,

$$f_{MAU_E} : PHVS \rightarrow PHVS$$
$$f_{MAU_E}(PHV, G_E^{MAU}, MAT\_SET_E) = PHV^*$$

6.9. Egress Deparser(DPR_E)

Same as ingress deparser. Only difference is, after egress deparsing a PHV is sent to scheduler instead of buffer and replication engine.

$$f_{DPR_E} : PHVS \rightarrow PHVS$$
$$f_{DPR_E}(PHV, G_E^d, Null) = PHV^*$$

6.10. Scheduler(S)

After egress processing is complete, a PHV is passed to Scheduler(S) for transmission to next hop. Scheduling defines **order** and **time** [63] of a PHV's transmission. To achieve these goals, PHV is needed to be stored in specified order (according to scheduling algorithm) inside an appropriate scheduler data structure (***SDS***) (i.e. Queue, collection of queue, tree etc.). Different hardware implementation can use different data structure. From an abstract point of view, ***SDS*** maintains an ordered PHV Set (Section 5.1.5). From ***SDS*** PHV's are selected according to scheduling algorithm for transmission at appropriate time. As there is no universal scheduling algorithm [64] to match all kind of application goals, there is no common implementation of scheduler unit also.

To create a common and generalized abstract model, 2 abstract interfaces are assumed in conjunction with SDS. Different scheduling algorithm will implement these 2 interfaces differently based on data structures provided by the hardware. CP can adjust behavior of S through configuring scheduling algorithm parameters(S_{param}). Example: table for configuring weights of weighted fair queuing algorithm. These 2 abstract interface are following:

- **Insert(PHV)**: Implementation of this interface calculates the relative order of a PHV inside ***SDS*** and places it in corresponding position. Let, $SchedulingOrder$ is the metadata field that contains the PHV's order in SDS. Now assume, PHV_SET' is a set of PHV ordered by $SchedulingOrder$ (in reality this order may depend in more than one field) stored in **SDS**.

  $$PHV\_SET' = \{PHV_1, PHV_2, PHV_3, .. \\ ...., PHV_n\}$$

  Finding a new PHV(PHV_{new})'s scheduling order requires computation of $PHV_{new}.SchedulingOrder$ from following types of data.

  - value in PHV_{new}'s header fields(H_f) and/or metadata (M_f)
  - stateful information about the flow, computed and stored by switch
  - CP configured parameter such as weight for a weighted fair queuing scheduling

  Insert interface provides a total order (Section 5.1.5) on ($PHV\_SET, <$) based on $SchedulingOrder$ where,

  $$PHV\_SET = PHV\_SET' \cup \{PHV_{new}\}$$
  $$(PHV\_SET, <) = \{PHV_1, PHV_2, PHV_p, \\ .....PHV_{new}, PHV_q, ..., PHV_n\}$$

  Therefore, insert interface can be functionally represented as

  $$f_{S^{insert}} : PHVS \rightarrow Ordered\ PHV\ Set(SDS, \leq)$$
  $$f_{S^{insert}}(PHV, Insert\ Interface\ Implementation,$$

$S_{param}) = PHV\_SET$

Computing $PHV_{new}.SchedulingOrder$ can be done either in match-action stage and result can be carried to scheduler unit through PHV. Or also can be computed in scheduler unit alone. After that, scheduler will insert the packet at appropriate location in the data structure. In case of storage shortage a low priority packet can be dropped or the packet in consideration itself can be dropped depending on scheduling algorithm.

- **Remove**(): This interface's implementation picks the next PHV to be transmitted from SDS and decides when the packet will be actually transmitted through $Port_{egress}$.

  $f_{S_{remove}} : PHV\_SET \rightarrow PHVS$
  $f_{S_{remove}}(Null, Remove\ Interface\ Implemen$
  $tation, S_{param}) = PHV$

CTP features: a) custom insert and remove interface: is it possible to provide implementation of these 2 interfaces for custom scheduling algorithm b) custom SDS creation: custom SDS can be created for complex scheduling algorithm or not c) cross component access: can other components in the pipeline access SDS (ex. clear a queue based on certain event detected at match action unit) or it's properties (occupancy rate, priority etc.).

RTC features: a) insert and remove interface parameter modifiability: can the control plane configure parameters for controlling behavior of insert and remove functions b) SDS property access: can control plane access SDS properties c) S_{param} controlling: can CP control scheduling algorithm parameters at run time.

6.11. Egress Port($Port_E$)

After a PHV is selected for emitting, ***egress port($Port_E$)*** transmits the content of data_buffer. How a hardware level frame is created and transmitted is is out of the scope of our discussion. Like ingress ports, egress ports also do not have any kind of programmability in terms of CTP and RTC features.

$f_{Port_{In}} : PHVS \rightarrow BS$
$f_{Port_{In}}(PHV, Null, Null) = PKT,$

6.12. Special features required for debug and development

For intelligent network it is also important to enable other component control the scheduler. For example a special packet can be sent from neighbor switch or special packet from an end host to delete all packets of a flow . Based on that special packet the switch may need to delete a whole bunch of packet from the scheduler data structure. For RED implementation, egress match action block needs to know the queue occupancy situation for in general self learning network we need to enable changing every portion of DPP at run time. and run time load reload. But for dev, debug and performance measurement we need to access the time a packet spend between 2 standard states exposed by the abstract model. It is not possible to interrupt and monitor each of the operation on packet inside switch, because of speed loss.

Table 5
Eq. 1 Based Programmability Comparison Matrix

$E_1 = \mathcal{I},\ E_2 = \mathcal{O},\ E_3 = Conf_{param},\ E_4 = ProcLogic$					
		Features			
Component	Products	E_1	E_2	E_3	E_4
Parser	FlexPipe	NA	1	NA	1
	Arista 7170	NA	2	NA	2
	Agilio Cx	NA	2	NA	2
	PSA	NA	2	NA	2
	T-switch	NA	0	0	0
Ingress Buffer Engine	FlexPipe	1	1	1	0
	Arista 7170	2	2	1	0
	Agilio Cx	2	2	2	2
	PSA	NA	2	NA	NA
	T-switch	0	0	0	0
Ingress Match Action Unit	FlexPipe	1	1	1	1
	Arista 7170	2	2	2	3
	Agilio Cx	2	2	2	3
	PSA	2	2	2	3
	T-switch	0	0	0	0
Deparser	FlexPipe	1	2	NA	1
	Arista 7170	2	2	NA	2
	Agilio Cx	2	2	NA	2
	PSA	2	3	NA	2
	T-switch	0	3	NA	0
Buffer & Replication Engine (BRE)	FlexPipe	1	1	1	0
	Arista 7170	2	2	1	0
	Agilio Cx	2	2	2	2
	PSA	2	2	1	0
	T-switch	0	0	0	0
Scheduler	FlexPipe	1	1	1	1
	Arista 7170	2	2	1	1
	Agilio Cx	2	2	2	3
	PSA	2	2	1	1
	T-switch	0	0	0	0

7. Programmability Feature Comparison

Though in network computation gained significant momentum, but all the programmable dataplane devices are not able to do all kinds of computation. Nowadays, programmable core switches add ref to tofino 2 can achieve maximum speed of 12.8 Tb/s. But limited memory size, limited set of actions, limited operations per packet makes them inappropriate for complex processing. On the other hand, smartnic based packet processing in slow path (cpu based processing) provides opportunity for more complex computation. But achieving high speed line rate in such environments still remains a big issue. Therefore, to understand a programmable switchâĂŹs applicability in a certain scenario, it is important to understand what kind of programmability is supported by that device. Deriving formal relation between what set of programmability features can support a specific class of algorithms in data plane can be a promising research direction. Until now, there is no comprehensive formula or empirical list of programmability features to determine whether a device is suitable for a task in data plane.

Programmable data plane devices enables complex computation on packets in data plane. But limited memory, limited set of actions and requirement of maintaining line rate makes core switches unsuitable for complex algorithm. On the other hand, smart-NIC based packet processing in slow path (cpu based processing) provides opportunity for more complex computation. But achieving high speed line rate in such environments still remains a big issue. Therefore, to understand a programmable switch's applicability in a certain scenario, it is important to understand what kind of programmability is supported by that device. Deriving formal relation between what set of programmability features can support a specific class of algorithms in data plane can be a promising research direction.

To understand what kind of algorithms can be implemented in data plane with available programmable switches in the current market, we have selected the following representative platforms. **T-switch** represents most common category of traditional L2/L3 switches. **FlexPipe**[19] is Intel's Openflow supported programmable switching platform. It is significant as one of the early generation programmable switch of recent times. **Tofino**[54] is the most prominent and commercially successful programmable switching chip based on RMT architecture. It is being used by various switch vendors [53]. It's various features are protected under barefoot non disclosure agreement. For comparison, we have chosen Arista 7170 [52] switching platform. Which is based on Tofino chip and supports $P4_{16}$. **Netronome Agilio Cx** [51] (with 4000/6000 family NFP) is selected as a smnart-NIC based programmable switching platform. It also supports $P4_{16}$. It can be programmed using micro-C and P4. **PSA** is the most matured switch architecture by P4 community and it is based on $P4_{16}$. ***AVS*** is also inspired by PSA. Until now, there is no hardware that follows full specification of PSA. But any architecture can be simulated in software level. Bmv2[14] is the reference implementation of P4, which can simulate various switch architectures including PSA. PSA is still not fully supported in bmv2. For comparison purpose, we assumed that PSA will support $P4_{16}$ and it will run on bmv2 simulation platform. Though PSA is still not realized, we believe comparison of existing hardware architecture with PSA (on bmv2) can give a clear picture of current market scenario.

In table x we have listed few important types of algorithms, what are their crucial tasks that needs support from programmable switches (not achievable with traditional switches) and whether these tasks are supported by selected set of devices. From this table it is clear that, all the devices in the market are not equally programmable. To better understand their programmability level, we have scored those selected platforms in table y considering AVS components as base. Table y, gives a 2-dimensional comparison among the selected platforms. Firstly, each component of AVS provides a specific type of programmablity in data plane. Therefore comparing any platform on the basis of whether equivalent AVS components exists or not says whether equivalent functionality can be achieved in a platform or not. Secondly, eqn. 1 gives a functional structure for each of the component. Scoring the platforms based on parameters ($\mathcal{I}$, $\mathcal{O}$, $ProcLogic$ and $Conf_{param}$) of this equation gives a fine grained view of each platform's progammability level.

This comparison tells, whether equivalent functionality of a AVS component is achievable in a switching platform and what is the level of programmability achievable in the platform.

In recent time, programmability has become a buzzword in data plane hardware domain. But yet no agreeable matrix to compare different devices have emerged. To understand capability of devices, they need to be analyzed with a well defined set of features. ***AVS*** can be an ideal platform as it hides low level hardware details and provides a uniform abstraction. Functional representation[eqn. 1] of ***AVS*** components provides a coarse grained view of the component's programmability level. On the other hand RTC and CTP features provide a fine grained view. In this section, few of the major programmable data plane platforms are compared based on these features. We have chosen one traditional fixed protocol stack switch, one non RMT based smartNIC, one RMT based core switch, one RMT based smartNIC and a software implementation of P4. These platforms are:

T-switch represents most common category of traditional L2/L3 switches. These devices do not have any kind of programmability. They are optimized for executing layer 2/3 functionality of OSI layers. They only supports IPv4 or IPv6 forwarding and/or routing.

FlexPipe[19] is intel's programmable switching platform. It provides programmability based on selected set of L2-L4 header fields from TCP/IP stack. As a SDN switch it supports OpenFlow protocol. Though it is not as modular like as AVS design, but it contains the concept of hardware slice. These slices are chained together for a single packet processor. Flexpipe doesn't provide arbitrarily new protocol support in data plane, still it is significant as it can be considered as one of the early generation programmable switch of recent times.

Tofino[54] is the most prominent and commercially successful programmable switching chip based on RMT architecture. It is being used by various core switch and smart-Nic vendors [53, 52]. It can be programmed using P4 or P4+Micro-C(Mellanox smartnic) based on the platform it is being used. A lot of it's features are protected under barefoot non disclousure agreement(PNDA). Hence, all of it's RTC and CTP features can not be accessed. For comparison, we have chosen Arista 7170 [52] which is a Tofino chip based solution.

Agilio Cx[51]: This is a selected as a smart-NIC based solution. We have selected Netronome Agilio Cx [51] (with 4000/6000 family NFP).

Arista 7170 and Agillio CX, first one is a core switch and 2nd one is a smartNIC. Both one supports P4v16. In case of any features of these solutions are protected under non disclosure agreement, we have marked them as **PNDA** (Protected under Non Disclosure agreement) in comparison matrix.

PSA is the most matured switch architecture by P4 community. ***AVS*** is also inspired by PSA. Untill now, there is no hardware that follows full specification of PSA. But any architecture can be simulated in software level. Bmv2[14] is the reference implementation of the most prominent data-plane programming language P4, which can simulate various switch architectures. PSA is still not fully supported in bmv2. For comparison purpose, we assumed that PSA will support P4v16 version and it will run on bmv2 platform. Though PSA is still not realized, we believe comparison of existing hardware with PSA(on bmv2) can give a clear picture of current market scenario.

Due to non availability of enough architectural documentation and standard instruction set, we have followed which version of P4 these devices supports. For example, a device may not support parsing variable length array of bits. But using syntactic sugar and translating varbit(variable number of bit [18]) parse instruction to multiple single bit parse instruction and loop a vendor can support varbit parsing. Lack of information about actual hardware architecture makes the comparison a hard task. As our main target is to understanding programmability level of these devices, we have not differentiated between native hardware support and through syntactic sugar support of a feature. In a programmable data plane device, all the components of ***AVS*** may not be present. We considered if comparable functionality exists in the system or not.

To understand capability of devices, they need to be analyzed with a well defined set of features. ***AVS*** can be an ideal platform as it hides low level hardware details and provides a uniform abstraction. Functional representation[eqn. 1] of ***AVS*** components provides a coarse grained view of the component's programmability level. On the other hand RTC and CTP features provide a fine grained view.

In table x we have listed few important types of algorithms, what are their crucial tasks that needs support of programmable switches (not achievable with traditional switches) and whether these tasks are supported by selected set of devices. From this table it is clear that, all the devices in the market are not equally programmable. To better understand their programmability level, we have scored those selected devices in table y. Scoring system is mentioned also. For brevity and space details comparison chart is not included here. It can be found in our website.

7.1. Programmability Matrix

Fundamental elements of eqn. 1 are: $\mathcal{I}$, $\mathcal{O}$, $ProcLogic$ and $Conf_{param}$. In Table 5, a score matrix is presented to compare programmability level of selected 5 products. In all the comparisons, ingress and egress ports are not included as they don't have any programmability feature. On the other hand, in both ingress and egress stage; parser, match-action unit and deparser are same. Hence they are mentioned only once for comparison(subscript '*' is used). In the comparison, Traditional fixed protocol switch(**T-switch**) always scored ***'0'***, as they don't fulfill the requirements of being programmable discussed in section 3.

Details of the scoring system is following:

FOR $\mathcal{I}, \mathcal{O}$ and $Conf_{param}$: all of these 3 represents some abstract data types.

- **NA**- Programmability is not applicable here. Example: for all of the selected hardware input to ingress parser is a set of bits($PHV.data\_buffer$). Prograambility is not needed for input to an ingress parser.
- **0**- The component is non programmable(ex. T-switch)
- **1**- Selectable from a set of predefined data types. Example: Flexpipe [19] hardware can only parse selected set of L2-L4 header fields(specific pins are mapped to selected header fields). Hence programmability of Flexpipe is limited to these selected set of header fields.
- **2**- New abstract data type(i.e. struct. class etc.) can be created. Example: Tofino chip support P4 v16. In a P4 program, new abstract data types(struct) can be created from few primitive data types(int, char, bit, varbit etc.).

FOR $ProcLogic$:

- **NA**- Programmability is not applicable here. Example: for all of the selected hardware input to ingress parser
- **0**- Fixed ASIC. Not programmable.
- **1**- Only selectable from a set of pre-implemented logic /algorithm.
- **2**- New algorithms($ProcLogic$) can be implemented but stateful actions (any action that can fetch and store a flow state) not supported
- **3**- New algorithms($ProcLogic$) can be implemented with stateful actions support

Table 6
Parser Programmability Features Comparison

		FlexPipe	Arista 7170	Agilio Cx	PSA
CTP Features	Max. length limit	88 Bytes	PNDA	PNDA	No mentioned limit
	Supported Data types	Only standard L2-L4 header types	All P4v16 data types. Bit,varbit, int, string, structure, enum. But no float	Same as Arista 7170	Same as Arista 7170
	Granularity of field parsing	only pre-specified header fields	single bit to variable length bit	Same as Arista 7170	Same as Arista 7170
RTC Features	G_*^p modifiability	No	No	No	No

In future we will also include whether a component's program logic can be turing complete or what kind of programmaing language they support.

In following subsection, CTP and RTC features of the each AVS components in selected 4 platforms are presented in tabular form. Traditional fixed protocol switch(T-switch) has not been compared here, as they have no programmability feature in terms of our analysis(section 3).

7.2. Parser(PR_*) Features Comparison

Table 6 contains CTP and RTC features comparison for a generic parser component. This is applicable for both ingress parser(PR_{In}) and egrees parser(PR_E). As the input of parser is a set of bits($PHV.data\_buffer$), it requires no programmability in terms of $\mathcal{I}$. Logic($ProcLogic$) for parsing a PKT is also static at run time in selected devices. Hence, both $\mathcal{I}$ and $Conf_{param}$ are marked as **NA** in the comparison matrix. For the parsing logic, we have not distinguished whether a hardware supports complex/generalized programming constructs like recursion, loop etc.

7.3. Ingress Buffer Engine(BE_{In}) Features Comparison

Table 7 contains CTP and RTC features comparison for a ingress buffer engine. In PSA specification [31], ingress buffer engine is not mentioned as component. Therefore all the scores for ingress buffer engine in PSA is scores as **NA** in comparison matrix(Table 6). None of the compared devices mentioned about providing algorithm($ProcLogic$ for Ingress Buffer Engine) for accessing elements from buffer. Hence all the devices scored 0(except Agilio Cx) in the comparison matrix. But we believe in future, parallel packet processing pipeline will be a real important issue, which is already evident in server based networking. As a server based solution **AgilioCx** [51] scores higher in terms of programmability for ingress buffer engine.

7.4. Match Action Unit(MAU_*) Features Comparison

Table 8 contains CTP and RTC features comparison for a generic *'match action unit'*. This comparison is applicable for both ingress and egress match-action-unit. Here, we have not taken into account various complex programming construct like recursion, loop, etc. We believe, in future data plane devices will be more rich in terms on supported actions. More level of scoring can be added after 0-3 range.

7.5. Deparser(DPR_*) Features Comparison

Table 9 contains CTP and RTC features comparison for a generic deparser component. Like parser feature comparison(Table 6), this comparison is also applicable for both *'ingress deparser'*(section 6.5) and *'egreess deparser'*(section 6.9).

7.6. Buffer and Replication Engine(BRE) Features Comparison

BRE contains a generic buffer engine. Hence it inherits the comparison presented in Table 7 and requires extra feature of *'manycast group membership table'*. Table 11 contains CTP and RTC features comparison for *'Buffer and Replication Engine(BRE)'*(section 6.6). Flexpipe doesn't have explicit component for handling replication. On the other hand cpu based datapath offloading can be used for complex processing in AgilioCx smartnic. Some data plane devices do not have explicit component that can be mapped to BRE. If a system for controlling buffer exists we have used it's property for comparison in both Ingress buffer engine and buffer and replication Engine.

7.7. Scheduler(S) Features Comparison

Table 10 contains CTP and RTC features comparison for a generic *'Scheduler'* component. Current days programmable data plane devices do not provide any solid programmability for packet scheduling. In those devices, a specific algorithm can be selected from a pre-implemented set of scheduling algorithms. As we assusmed PSA [31] will be implemented on bmv2 [14] simulator, hence different scheduling algorithm can be implemented for PSA. But PSA specification provides no clear direction about scheduling algorithms. Therefore we considered it has no programmability for packet scheduling.

Table 7
Ingress Buffer Engine Programmability Features Comparison

		FlexPipe	Arista 7170	Agilio Cx	PSA
CTP Features	buffer size & number controlling	Not programmable	Yes	Yes	No
	BCT creation	No	Partial. Because multiple buffer can be maintained for different traffic classes and buffer size are programmable. Not mentions about arbitrary protocol field based buffer	Yes. CPU based DPP enables dynamic control	No
	Global access of buffer property	No	No	Yes. CPU can access and control various buffer information	No
RTC Features	BPT modifiability	Only buffer modes can be changed	Partial. Only buffer memory sharing among ports are allowed	Yes. eBPF, C, P4 based data path offload enables CPU based dynamic control	No
	BCT modifiability	No. But traffic policer available	No	Yes. eBPF, C, P4 based data path offload enables CPU based dynamic control	No

8. Motivating Use Cases

Abstraction plays very important role in SDN [33]. Abstractions used in SDN are hierarchical in nature and used in different layers [9]. Hardware abstraction layer is placed over programmable hardware layer and provides a uniform view to other layers (*'Device and resource Abstraction Layer (DAL)'* [33]). As an instance of hardware abstraction layer, ***AVS*** hides low level hardware complexity and it can provide a uniform view of hardware layer to both control plane and data plane application. Hence its use cases can be found in every aspect of a truly programmable data plane device. We are mentioning few important use cases here.

8.1. Modular Architecture Design & Development

Modular architecture design is the goal of good design because of various benefits it brings. Specially it allows independent development of components and building more complex constructs based on them. Majority of the programmable data plane architecture in literature are expressed in informal language. This makes modular design harder for 2 reasons: a) lack of clear description about the role of a component b) lack of clear boundary between 2 components. These shortcomings create bottleneck in independent design and optimization of components. It also brings disadvantages in designing new pipeline based on reuse of those components.

In ingress buffer egnien we need to say that, intentioanlly we are keeping buffer as a single header field thing. If necessary we can do this as a multi header match action based. But then it needs to be implemented as mat unit where buffer storing and retriving is an actions.

Consider the case of ***Ingress Buffer Engine (*** BE_{In}***)*** and ***Ingress Match Action Unit(*** MAU_{In} ***)t***. As both have match-action semantics, BE_{In} can be designed as a part of MAU_{In}. After header matching, storing in and removing packet from buffer can be designed as one of the actions of MAU_{In}. Consider a smart-NIC based packet processing architecture, where packets are moved to userspace and processed in CPU and/or GPU. So the MAU_{In} is implemented on CPU and/or GPU . GPUs perform better in parallel and batch processing, where buffering is a per packet processing task not best suitable for GPU. Implementing buffer engine as part of match-action unit using GPU needs costly data transfer to and from memory to GPU. This data movement often loses the GPU performance gain. Hence it is better to implement buffer engine as a separate component. If buffer engine functionality is merged with MAU_{In} optimal hardware performance can not be achieved in this case. To overcome these kind of issues, clear definition of a componentâĂŹs structure and role is necessary. ***AVS*** defines the role of each component through extended finite state machine. ***AVS*** also clearly identifies boundary of each component by defining their input and output (eq. 1). It removes ambiguity in component design and how they can be mapped to heterogeneous hardware.

Equation (1) provides a structured framework for each component. It clearly defines the input, output, packet processing logic and parameters for how to configure run-time behavior of a component through control plane ($Conf_{param}$). Communication among the components is only through passing parameters. It removes control dependency among com-

Table 8
MAT Programmability Features Comparison

		FlexPipe	Arista 7170	Agilio Cx	PSA
CTP Features	Data type support in MAT	Selected L2-L4 header types of TCP/IP stack	All P4v16 data types. Bit,varbit, int, string, structure, enum. But no float	Same as Arista7170	Same as Arista7170
	Matching type support in MAT	Longest prefix(lpm), exact, range based, binary search tree matching	All P4v16 match types. Exact, ternary, lpm , range	Same as Arista7170	Same as Arista7170
	MAT size & number controlling	24570 36 bit CAM, 65536 32 bit BST	PNDA	PNDA	No limit
	Availability of stateful action	Only selected set of L2-L4 actions available	Yes	Yes	Yes
	Availability of stateful data structure	Only predefined statistical data collection available & modifiable MAC table	Yes	Yes	Yes
	Flow state sharing among different components	Actions are chained. Hence flow state is shared	No	No	No
	MAT modifiability from DPP	No	No	No	No
	Available action types	Predefined set of actions	Header & metadata modification, arithmatic operation, vendor specific P4 *'extern'* [20]	CPU based dynamic operations possible on various fields	All P4 based actions & open to support hardware specific *'extern'*[20]
RTC Features	MAT controlling from CP	Yes	Yes	Yes	Yes
	Stateful data access from CP	Read only	Yes	Yes	Yes

Table 9
Deparser(Ingress & Egress) Programmability Features

		FlexPipe	Arista 7170	Agilio Cx	PSA
CTP Features	Maximum length limit	112 Bytes	Jumbo L2 frame support (9.6KB)	jumbo L2 frame support (9.6KB)	No limit
	Supported Data types	Only standard L2-L4 header types	All P4v16 data types. Bit,varbit, int, string, structure, enum. But no float	Same as Arista 7170	Same as Arista 7170
RTC Features	G_*^D modifiability	No	No	No	No

ponents. This gives several advantages: a) each component can be designed, developed and optimized independently. b) Task of any component can be assigned to heterogeneous hardware at run time c) new kind of packet processing pipeline can be built by composing existing components.

Consider a smart-NIC based scenario where the datapath is moved to userspace using DPDK. Also assume, the data plane prorgam (**DPP**) is assigned to do both IPv4/6 packet processing and running an algorithm for accelerating ML algorithms through in network aggregation technique [60]. Aggregation is basically mathematical processing and computationally expensive which can be executed in parallel fashion. Let, IPv4/6 packet processing is expressed as MAT_{IP} and aggregation based packet processing logic is expressed as MAT_{AGG}. Now, to make packet aggregation faster, MAT_{AGG} is assigned to be executed on GPU. In the other hand, IPv4/6 packet processing (MAT_{IP}) tasks are executed on CPU. Also assume, CPU can process 1 IPv4/6 packet in 1 cycle

Table 10
Scheduler Programmability Features(check which hardware have selectable scheduling algo

		FlexPipe	Arista 7170	Agilio Cx	PSA
CTP Features	Custom insert & remove interface(new scheduling algorithm implementation)	No. Only pre-implemented 3 algorithms are supported	Same as Flexpipe	CPU based custom datapath can support complex algorithm	No. Specification [31] avoids programmability for scheduler
	Custom SDS creation	No. Predefined fixed data structure only	Same as FlexPipe	Same as FlexPipe	Same as FlexPipe
	Cross component access of scheduler properties	No	No	CPU based datapath program can support it	No
RTC Features	Insert & remove interface parameter configurability	No	No	Yes	No
	SDS property access	No	No	Yes	No
	S_{Param} controlling	Yes	Yes	Yes	Yes

Table 11
BRE Programmability Features

		FlexPipe	Arista 7170	Agilio Cx	PSA
All CTP & RTC Features of Buffer Engine(except BCT)					
RTC Features	Manycast group membership table configuration	Yes	Yes	Yes	Yes

where as GPU can aggregate 10 packets in a single cycle and produce result of aggregation as one packet. As CPU and GPU run in different speed, synchronizing and scheduling the packet processing over them is very important. Without a clear definition and structure of MAT_{IP} and MAT_{AGG}, synchronizing packet processing over CPU and GPU is not possible. Through expressing each component in a structured fashion ***AVS*** (eq. 1) enables synchronization packet processing over heterogeneous hardware.

8.2. Testing & Verification

Earlier data plane devices were mainly designed for executing network protocol. But in programmable data plane devices, more and more application layers tasks are pushed toward data plane. These devices can be loaded with new data plane program at any period of their life time. Testing them or validating some property at run time over these programs are very important. But testing programs and verifying invariant over heterogeneous hardware architecture increases the cost and complexity.

Consider the scenario of figure x, 2 hosts (h1 & h2) are involved in a client-server communication with Server (S) through Switch (S1). At a certain time H1 got disconnected due to H1-S1 link failure. Identifying the link failure switch S1 stores all packets directed for H1 either in ingress buffer or port buffer(buffer engine) and counts these number of packets. When this count crosses a certain threshold (Th), S1 sends a notification to controller. Upon receiving this notification, controller initiates migration of the whole network (migration including switch state a packet stored in buffer (ref to lime)) from one data center to another. To test this kind of behavior of a DPP, a common abstraction layer is mandatory. Otherwise, if a switch vendor does not support ingress buffer engine and another vendor does not support port buffer (in BRE), then this kind of program will not work across different data center. Also to write test cases, a common set of packet processing state among switches are required. Without a common abstraction layer, how various hardware vendors implements packet processing logic can differ. But if the DPP developer gets a uniform view provided of the hardware,then the same program need to be tested only once for all hardware. AVS can provide these advantages, as it has an abstraction layer and expresses the components workflow using EFSM. Test cases can be developed based on the states represented in EFSM of AVS components.

pre condition: assume BRE has a confuguration for host-tobuffer store, so src mac to buffer mapping in BCT trigger: buffer depth > th post condition: send notification to controller

Now, at first we present few motivating scenarios to explain usage of ***AVS*** components. Then we present scenarios where AVS is useful as an abstraction layer.

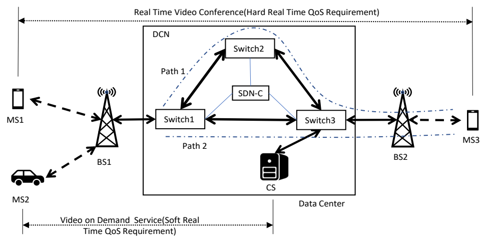

Real Time Video Conference(Hard Real Time QoS Requirement)
DCN
Switch2
Path 1
SDN-C
MS1
Switch1
Switch3
MS3
Path 2
BS1
BS2
CS
MS2
Data Center
Video on Demand Service(Soft Real Time QoS Requirement)

Figure 12: Network at Time T1

In this section we will consider a 5G deployment scenario. Following are few related terms used in this section: **Mobile Station(MS)**: Any connected device that can change location. Example: a cellphone or a self driving car which is using 5G network service. **Base Station(BS)**: Main radio unit in an area. Depending on scenario may or may not contain MEC. **Data center (DC)**: Core data center of the operator where C-RAN is running. The data center network (DCN) is controlled by SDN controller (either centralized or distributed). We assume, all switches are programmable . **SDN-C**: Centralized control plane application of DCN. **Content Server (CS)**: Any server running some application (example: video/audio/file server etc) inside or outside the data center. In our scenario(Fig : reference), we assume Mobile Stations are communicating to Content Server (CS). Mobile Stations are connected to Base Stations and BSs are connected to the DC. We analyze the network's condition at 2 separate time instance, T1 and T2 (T1 < T2).

Consider the network of Figure 12. At time **T1**, three 5G mobile stations **MS1, MS2, MS3** are connected to operator's core network(running in data center DCN) through 2 base stations: **(BS1 and BS2)**. DCN is running equal cost multipath routing at each hop.

8.3. Bypassing Protocol Standardization

One of the main goal of SDN is to generate global view of the network in control plane. For that, switches need to provide various telemetry [30, 73] data to control plane. Assume switches of scenario in Figure 12 can provide queue occupancy rate, hop by hop delay etc. Other switches or SDN controller will use these data to determine optimal route. In figure 12, **Switch1** directly connects to wireless medium, hence information about wireless signal strength, noise etc. are more relevant here. On the other hand, **Switch2** is core data center switch. Here hop by hop delay, queue occupancy etc. is more relevant. With increase of computational power inside the switches, more complex matrices can be computed in data plane and send to control plane. Without data plane programmability, switch vendor need to follow standardized protocols. Protocol standardization is a lengthy path which may take few years. But programmable parser and deparser gives data plane the ability to interpret and create packet of arbitrary protocol. These obsoletes the requirement of protocol standardization by community. Using these capabilities switches can go bewyond their usual L2 header based processsing. Assume, with programmable parsing, ***Switch1-3*** can identify video packets which can be used for calculating jitter of video communication between **MS1** and **MS3**. Based on that switches may need to execute complex event based processing [42]. Parser(section 6.5), deparser(section 6.5) and match-action unit(section 6.4) of both ingress and egress stage cover this programmabilities in ***AVS***. Moreover, representing their work flow through EFSM enables to exposes state in a hardware independent manner. Testing frameworks can exploit those states for generating [55, 70] test cases and automatic testing for data plane programs.

8.4. QoS Scheduling

Now assume in Figure 12, **MS1** is involved in a real time video conference with **MS3** and **MS2** is sending a large file **MS3**. These two flow can be categorized as: **Category-1** traffic(packet loss acceptable but low delay required) from **MS1** to **MS3** and **category-2** traffic(packet loss unacceptable but low delay acceptable) from **MS2** to **MS3**, at first reaches to **Switch1**. From here traffic can flow through any one or both of the 2 paths shown in Figure 12. Hence, **Switch1** needs to schedule packets among next 2 nodes **Switch2** and **Switch3** with different weight. If **Switch1** wants to maintain multiple queue for multiple traffic classes or the network operator want to follow revenue model based on traffic queues [58] then situation become even more complex. This requires custom algorithm for scheduling. Current generation programmable switches implement fixed algorithm for queue management and scheduling (Table 10) , which is inadequate [64]. ***AVS*** includes abstraction for customized data structure(SDS), scheduling algorithm(*insert* & *remove* interface)(section 6.10), and parameters for configuring scheduling algorithm(S_{Param})(section 6.10).

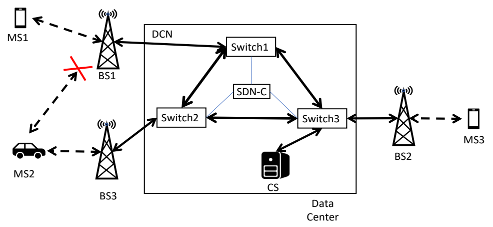


Figure 13: Network at Time T2

8.5. Mobility Support

Now consider the scenario of Figure 13. While file transfer was going on, **MS2** started moving and got disconnected from **BS1**. And after a certain time interval **(ΔT)**, **MS2** reconnected to the operator's network through **BS3** at time **T2 (T1 <T2)**. Now the possible paths from **MS2** to **MS3** have changed. While **MS2** was disconnected from network, on the fly packets were traveling through different nodes.

Those nodes were still unaware of the fact that, **MS2** has reconnected through new path. Moreover, on the fly packets have **MS2**'s old location as source/destination in header. As a result, those packets follow old path and reaches to **Switch1**. They find **MS2** is not connected through the path. As a result packet loss and out-of-order delivery of packets starts happening. This severely impacts TCP connection and network performance. As host application running on **MS2** is agnostic of underlying network connectivity, it can not signal the network about it's mobility. Network itself should intelligently identify the mobility of end host and needs to take measure. In the scenario of Figure 13, **MS2** reconnects to the network at time **T2**, instead of dropping the packets traveling through the data plane , network should do following things a) hold the newly generated packets at new path b) reroute all the old packets to new destination c) after delivering all the old packet, resume the flows which were paused at new path at step **a**. Current generation programmable switches do not provide that. To augment, ***AVS*** included ingress buffer engine(BE_{In}) (section 6.3) in the pipeline with these capabilities.

Now we focus on advantages of using ***AVS*** as an hardware abstraction layer. We discuss few of relevant technology's current scenario and how ***AVS*** can help .

8.6. Network Function Modeling

Eq. (1) enables representation of each components of ***AVS*** in a uniform and hardware independent manner. Composing f_c for all the components and concatenating them represents ***AVS*** as a transfer function(τ) [39]. Compiler translates this functional representation to hardware instruction. Any network function(NF) programmed on ***AVS*** can be expressed as τ, where $\mathcal{I}$, $\mathcal{O}$, $Conf_{Param}$, $Proc_{Logic}$ are different for each NF . Each link of a network transfers a PKT from one hop to another. Link also can be represented as a function(Γ) of same structure.

$$\Gamma : \mathcal{BS} \rightarrow \mathcal{BS}$$
$$\Gamma(PKT, Medium\ Access\ Logic, Conf_{param}) = PKT$$

Γ can be considered as the functional representation of data link layer. Depending on medium, corresponding $Medium\ Access\ Logic$,(i.e. ring/mesh/star networks, CSMA, CSMA/CD etc.) and $Conf_{param}$(i.e. persistent level for CSMA) can be different. Based on these, a PKT's transmission and propagation delay can be different. These delays can be derived from time stamp(passed as H_f) at sender node and receiver node. Again applying τ and Γ a whole network topology can be represented as a topology function(ψ) [39]. These functional representation are based on abstraction layer and they are hardware independent representation with well defined parameters for each component. This kind of representation provides various benefits. Few of them are

- **Network algebra**: Network algebra is an old and deeply studied topics. With rise of programmable data plane device, it is more relevant and applicable for today's network. Expressing network functions and networks in a functional paradigm allows usage of formal methods of network algebra [2] [39].
- **Network Delay Modeling**: Time-stamping [49] packet in data plane is a strong concept with many usage [50]. But maintaining time-stamp only at entry and exit point of a switch can't provide deep information about various type of delay. Time-stamp can be maintained at entry($time\text{-}stamp_C^{Entry}$) and exit($time\text{-}stamp_C^{Exit}$) point of each component in packet metadata($PKT_{Metadata}$). Delay inside a component(C) can be calculated as

  $$D_C = time\text{-}stamp_C^{Exit} - time\text{-}stamp_C^{Entry}$$

  Thus, fine grained information about various delay can be collected. One of the most simple but powerful model of delay is Network Delay. It can be computed as following

  $$\begin{aligned}
  &Network\ Delay\ = \\
  &Queuing\ Delay\ +\ Processing\ Delay + \\
  &(Transmission\ Delay\ +\ Propagation\ Delay) \\
  &= D_{BE_{In}} + (D_{PR_{In}} + D_{MAU_{In}} + D_{DPR_{In}} \\
  &+ D_{BRE} + D_{PR_E} + D_{MAU_E} + D_{DPR_E} \\
  &+ D_S) + D_{PR_{In}} + D_\Gamma
  \end{aligned}$$

  Complex data plane processing with time stamp can be used for further complex measurement of various types of delay categorizated by source [8].

8.7. Virtualization, Hypervisor, OS

Virtualization, hypervisor and OS support in programmable data plane devices are still at very primitive stage. Though SDN controllers are sometimes referred as network operating system [32, 4], it barely have the capability of actual data plane resource control. A close look at computer virtualization reveals that, it's success can be largely attributed to an abstraction of the underlying hardware which provides resource slicing and sharing to guest operating systems. And guest OS gets an isolated and complete view of a computer. This well defined abstraction layer enabled innovation in underlying hardware, hypervisor, operating system and all related levels. As a result several key technologies(i.e. virtual machine, containers, hardware assisted virtualization(KVM) etc.) for optimal resource usage have emerged. To replicate same success story in network data plane a similar type of abstraction layer which can provide users a view of isolated data plane device is required. ***AVS*** can play that role as it models the full path of a packet's life cycle inside a programmable switch. Based on this abstraction OS for data plane, hypervisor, containers etc. can be developed.

Slicing & isolation: Most of the existing works on data plane virtualization focuses only on adding extra header for virtualization layer [62, 72, 34, 71]. True hardware level resource sharing is not present. On the other hand, programmable data planes are becoming more computationally capable and complex processing tasks are being pushed to data plane [65, 21]. But still there is no computer program like abstraction is available here. As a result, multiple data plane program can not get view of a completely isolated device. To execute multiple program on same hardware either they have to be pre-compiled to single program [34] or decomposed into restricted set of OpenFlow based instructions [62]. This lack of resource slicing and isolation poses security vulnerability. Because, if one portion of the data plane program is compromised, whole data plane is effected.

Now,consider an imaginary programmable switch with only ingress stage and no egress stage or buffer and replication engine. The ingress stage contains 2 parser each having capacity of parsing 10k packets/second, 1 match unit with 32 match action table each having capacity of storing 40k rules and a deparser with capability of deparsing 20k PHV/second. We want to run 2 different network function on this switch. Any network function can be modeled through AVS with unique ID for each one. Assume the switch hardware components and instruction memory is designed to handle an ID for each unique AVS, and a virtual address translation like scheme for MAT entries. Then the parser, deparser or match-action unit resource can be grouped into 2 part for each of the AVS. AVS ID included in instructions can be used to restrict one AVS from using resource of other one. Moreover, **AVS** has a modular structure, a permission matrix can be maintained for each of the components and their relevant programming features. Based on this matrix role based permission can be maintained for different category of data plane program. This way ***AVS*** can help in providing resource isolation, secured execution and program like abstraction for data plane.

Now, each of the 2 NF of scenario can be modeled through AVS with unique ID for each one. Assume the parser/deparser circuit and instruction memory is designed to handle an ID for each unique AVS, and a virtual address translation like scheme for MAT entries. Then the parser, deparser and match-action unit resource can be grouped into 2 part for each of the AVS. AVS ID included in instructions can be used to restrict one AVS from using resource of other one. Moreover, **AVS** has a modular structure, a permission matrix can be maintained for each of the components and their relevant programming features. Based on this matrix role based permission can be maintained for different category of data plane program. This way ***AVS*** can help in providing resource isolation, secured execution and program like abstraction for data plane.

A common hardware abstraction layer(HAL) which can provide the view of a complete data plane device(over shared resources) to each guest users is the base for creating a program like abstraction or virtual machine/container like technologies. ***AVS*** provides a hardware independent and complete abstract view of a data plane device which will be translated to actual hardware instruction. This makes ***AVS*** a suitable abstraction layer for virtual machine like technologies in data plane.

8.8. Virtual networking

Lack of data plane abstraction and emphasis on rigid protocol set has made network a 2nd class entity in current cloud environment. As a result, most of the existing algorithms for virtual network embedding [24] , VDC embedding [66], VM migration [47] at first embed or migrate VMs then setup overlay network/tunnel to connect VMs. After embedding or migrating VM, they establish end to end link connection. But, these virtual machine's work closely with neighboring vm and switches. As a result it is important to keep network in consideration at first while placing virtual machine. Moreover, rise of in network computation and use of VNF making network more stateful. Consider the **QoS Buffering** examples mentioned in this section. If the operator wants to migrate the whole virtual data center network, only reallocating virtual machine's are not enough. Because on that case, packet residing in the switch buffer will be lost. Hence, data plane devices with their full state along with the VMs are required to be migrated. Because of difference in hardware architectures, this network snapshot should be based on a common and intermediate hardware abstraction. Few works have tried to address these issues [40], but they work on abstractions provided by OpenFlow protocol. These works can not be extended for switches with arbitrary protocol. **AVS** can work as a suitable solution here. ***AVS*** based functional representation of a whole network function implies that, it can be represented independent of hardware with all of it's components. Hence ***AVS*** can be an ideal candidate for representing a switch as a virtual machine like entity for data plane. Emergence of in network computation paradigm will push more complex processing to data plane. On that case, only flow level abstraction provided by OpenFlow or P4 is not sufficient. Taking snapshot of packet processing states inside switch also have to be saved. Representing each components work flow with EFSM allows capturing the data plane states as states of EFSM. A whole network can be transformed to VM like entities with both processing states and relevant data. Then those entities can be migrated to new location seamlessly. This also can help for developing hypervisor for data plane.

For example, assume in the scenario the virtual network created by 2 neteowrk functions are needed to be migrated. So tau is a representation of the hardware. If for buffering scebarion we want to migrate we can migrate tge ingress buffer and bre with buffer state and data structres. Moreover as the work-flow is presented as state machine, the exact sbapshot of yhe state machine can be trabfered and resume the execution in new substrate swutch.switch.

8.9. Machine learning

Functional representation have well defined domain, range and configuration parameter. This can help for creating framework for applying machine learning techniques on PHV space and configuration parameter space. We believe, this can be extremely useful for knowledge defined networking [add ref]. Any ML algorithm that is developed and tested on AVS can be used on all hardware architectures that supports AVS.

moreover for reuse of same algorithm on various hardware a common hardware abstraction layer[ref] is needed. AVS provides that. Because ML algorithm will output optimal parameters for some components. If there is no common abstract model, then we can not use one algorithm for other hardwares.

8.10. Top Level Abstraction

One Big switch is an important concept for SDN scalability and better management[ref]. In this paradigm, whole network is seen as a single big switch. All policy are defined by the user on end to end basis of this big switch. Actual network management is done by controller platform. The allows various advantage [38]: scalability[ref], automatic controller placement in distributed sdn controller environment[ref], SDN rule compression[ref] etc. It's most crucial component is an abstraction layer. ***AVS*** with it's capability of network modeling can be combined with existing works on service chaining[ref] for this kind of platform.Few of the major capability requirements for data plane virtualization are a) facility of resource level sharing and execution on them in isolation b) hypervisor level support for managing resources c) a common abstraction to allow access to resources in shared manner and run data plane program on them d) a program like abstraction for data plane.

Consider 2 data plane devices, one can use re-configurable parser hardware(i.e. core switch) and another can use general purpose CPU for packet parsing(i.e. dpdk with smart-NIC). But for a VM like entity in data plane, programmable components of heterogeneous architectures are needed to be represented in hardware independent manner. **AVS** can provide that. Moreover, such representation can help in developing better programming language and also can enable top down design for newer hardware architectures.

Another shortcomings of current approaches to tag based virtualization layer is that, there is no true resource level sharing and isolation is possible. As a result if one portion of the data plane program is compromised, whole data plane is effected. Which poses tremendous security vulnerability. As, **AVS** provides a modular structure, permission matrix can be maintained for each of the components and their relevant programming features. Recently a lot of application level tasks are pushed to data plane for better performance[heavy hitter,concensus ref]. Emergence of in network computation paradigm will push more complex processing to data plane. On that case, only flow level abstraction provided by OpenFlow or P4 is not sufficient. A generic computer program like abstraction is needed for data plane. But basis of such kind of facility is existence of an abstract representation of data plane device, which can be provided by **AVS**. Besides this a common abstraction layer can also help in design and development of cloud orchestration frameworks[ref]

An abstraction like **AVS** not only provides advantage in run time and deployment. It can also provide significant advantages in development and testing of data plane program. Representing each component's work flow using EFSM can help to build frameworks for program optimization, test case generation, automated testing and verification. Moreover common abstraction layer also can be used as the basis of common debugger for various hardware architectures.

Use of ***AVS*** is not limited to these use cases. Importance of abstraction in computer is immense. In SDN stack abstractions will be used at different layers [9]. These are hierarchical in relation. Upper layer abstractions depends on abstractions provided by lower layers. And hardware abstraction layer is the lowest and closest to hardware abstraction in SDN stack. As a result, abstraction provided by ***AVS*** will find it's use in numerous set of application. In fact, from the architecture provided in [33] it is cleat that, *'Device and resource Abstraction Layer (DAL)'* is placed between programmable switch and both of control and management plane. Therefore as an example of this abstraction layer, in northbound side ***AVS*** can find it's use case in all kind of control & management aspects of SDN. In southbound side it hides low level details of forwarding and operation aspect of a data plane device. Hence it's use cases can be found in every aspect of a truly programmable data plane device.

8.11. Virtualization & New Pipeline Design

Equation (1) provides a structured framework for each component. It clearly defines the input, output, packet processing logic and parameters for how to configure run-time behavior of a component through control plane ($Conf_{param}$). Communication among the components is only through passing parameters. It removes control dependency among components. This gives several advantages: a) AVS gives a hardware independent and portable representation of data plane device which can be used to slicing the hardware layer b) DPPs can be developed based on machine model provided by ***AVS*** c) hypervisor like entities can execute or migrate same DPP to different hardware architecture.

Consider a smart-NIC based scenario where the datapath is moved to userspace using DPDK. Also assume, the data plane prorgam (**DPP**) is assigned to do both IPv4/6 packet processing and running an algorithm for accelerating ML algorithms through in network aggregation technique [60]. Aggregation is basically mathematical processing and computationally expensive which can be executed in parallel fashion. Let, IPv4/6 packet processing is expressed as MAT_{IP} and aggregation based packet processing logic is expressed as MAT_{AGG}. Now, to make packet aggregation faster, MAT_{AGG} is assigned to be executed on GPU. And IPv4/6 packet processing (MAT_{IP}) tasks are executed on CPU. Also assume, CPU can process 1 IPv4/6 packet in 1 cycle where as GPU can aggregate 10 packets in a single cycle and produce result of aggregation as one packet. As CPU and GPU run in different speed, synchronizing and scheduling the packet processing over them is very important. Without a clear definition and structure of MAT_{IP} and MAT_{AGG}, synchronizing packet processing over CPU and GPU is not possible. Using the structured representation provided by ***AVS***, switch hypervisors can assign, synchronize and migrate the DPP over various types of hardwares.

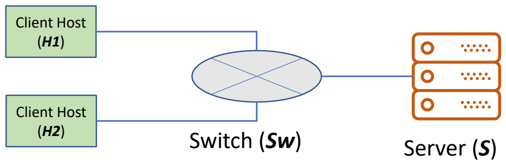


Figure 14: Server-client communication example

8.12. Testing & Verification

Earlier data plane devices were mainly designed for executing network protocol. But in programmable data plane devices, more and more application layers tasks are pushed toward data plane. These devices can be loaded with nserverew data plane program at any period of their life time. Testing them or validating some property at run time over these programs are very important. Without a common abstraction layer and workflow, testing programs and verifying properties over heterogeneous hardware architecture increases the cost and complexity. To write test cases, a common set of packet processing state among switches are required. Without a common abstraction layer and a common set of packet processing states , how various hardware vendors implements packet processing logic can differ.

Consider server-client scenario of figure 14. At a certain time ***H1*** got disconnected from server (***S***) due to ***H1-Sw*** link failure. Detecting the link failure switch (***Sw***) stores all packets directed for ***H1*** in B_i of ingress buffer engine (section 6.3) . When B_i becomes full, ***Sw*** sends a notification to controller. Upon receiving this notification, controller initiates migration of the whole network (migration including switch state a packet stored in buffer [40]) from one data center to another. Clearly, this kind of DPP depends on the ***"B_i is Full"*** state of buffer receiver thread (sec. 6.3, fig. 8). For testing them and ensuring uniform behavior, switch vendor of both the data center need to support ingress buffer engine with common packet processing states. Use of ***AVS*** as an abstraction layer provides a common abstract switch over heterogeneous architecture and DPP developer gets a uniform view of the hardware. Moreover ***AVS*** expresses behavior of the components through EFSM. Common abstraction of the components and common set of packet processing states of the components can be leveraged together to write portable code and testing them.

9. Conclusion

In recent time, programmable data plane has attracted significant attention of both academic and industry community. Despite being an active research area of great importance, it lacks of a hardware abstraction layer. An abstract model that can map all possible use cases and future architectures is practically impossible. Goodness of a model is determined by the extent of supported architectures, extendibility, use cases and higher level constructs that can be build upon it. In this work, we attempted to address these issues by proposing **AVS**. We have avoided concurrent packet processing in the design. Our work on representing components of data plane devices in a well defined functional structure and their work flow in EFSM can provide a strong base for optimizing, bench marking and comparing different programmable data plane devices. We invite research community to investigate how to improve this formal structure and develop frameworks for using it. Though we have analyzed the programmability features of components and compared them, but it is in a coarse grained manner. Programmability features analysis in a more fine grained manner can be of great importance. Besides this, any kind of formal analysis on CTP and RTC features can also be a good research scope.

References

[1] Abhashkumar, A., Lee, J., Tourrilhes, J., Banerjee, S., Wu, W., Kang, J.M., Akella, A., 2017. P5: Policy-driven optimization of p4 pipeline, in: Proceedings of the Symposium on SDN Research, ACM, New York, NY, USA. pp. 136–142. URL: http://doi.acm.org/10.1145/3050220.3050235, doi:10.1145/3050220.3050235.

[2] Anderson, C.J., Foster, N., Guha, A., Jeannin, J.B., Kozen, D., Schlesinger, C., Walker, D., 2014. Netkat: Semantic foundations for networks. SIGPLAN Not. 49, 113–126. URL: http://doi.acm.org/10.1145/2578855.2535862, doi:10.1145/2578855.2535862.

[3] Benson, T.A., 2019. In-network compute: Considered armed and dangerous, in: Proceedings of the Workshop on Hot Topics in Operating Systems, ACM, New York, NY, USA. pp. 216–224. URL: http://doi.acm.org/10.1145/3317550.3321436, doi:10.1145/3317550.3321436.

[4] Berde, P., Gerola, M., Hart, J., Higuchi, Y., Kobayashi, M., Koide, T., Lantz, B., O'Connor, B., Radoslavov, P., Snow, W., Parulkar, G., 2014. Onos: Towards an open, distributed sdn os, in: Proceedings of the Third Workshop on Hot Topics in Software Defined Networking, ACM, New York, NY, USA. pp. 1–6. URL: http://doi.acm.org/10.1145/2620728.2620744, doi:10.1145/2620728.2620744.

[5] Bifulco, R., Rétvári, G., 2018. A survey on the programmable data plane: Abstractions architectures and open problems, in: Proc. IEEE HPSR, pp. 1–7.

[6] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., Walker, D., 2014. P4: Programming protocol-independent packet processors. SIGCOMM Comput. Commun. Rev. 44, 87–95. URL: http://doi.acm.org/10.1145/2656877.2656890, doi:10.1145/2656877.2656890.

[7] Bosshart, P., Gibb, G., Kim, H.S., Varghese, G., McKeown, N., Izzard, M., Mujica, F., Horowitz, M., 2013. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn, in: Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM, ACM, New York, NY, USA. pp. 99–110. URL: http://doi.acm.org/10.1145/2486001.2486011, doi:10.1145/2486001.2486011.

[8] Briscoe, B., Brunstrom, A., Petlund, A., Hayes, D., Ros, D., Tsang, I., Gjessing, S., Fairhurst, G., Griwodz, C., Welzl, M., 2016. Reducing internet latency: A survey of techniques and their merits. IEEE Communications Surveys Tutorials 18, 2149–2196. doi:10.1109/COMST.2014.2375213.

[9] Casado, M., Foster, N., Guha, A., 2014. Abstractions for software-defined networks. Commun. ACM 57, 86–95. URL: http://doi.acm.org/10.1145/2661061.2661063, doi:10.1145/2661061.2661063.

[10] Casado, M., Koponen, T., Ramanathan, R., Shenker, S., 2010. Virtualizing the network forwarding plane, in: Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow, ACM. p. 8.

[11] Choi, S., Burkov, B., Eckert, A., Fang, T., Kazemkhani, S., Sherwood, R., Zhang, Y., Zeng, H., 2018. Fboss: building switch software at scale, in: Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, ACM. pp. 342–356.

[12] Choi, S., Long, X., Shahbaz, M., Booth, S., Keep, A., Marshall, J., Kim, C., 2017. Pvpp: A programmable vector packet processor, in: Proceedings of the Symposium on SDN Research, ACM. pp. 197–

198.

[13] Chole, S., Fingerhut, A., Ma, S., Sivaraman, A., Vargaftik, S., Berger, A., Mendelson, G., Alizadeh, M., Chuang, S.T., Keslassy, I., Orda, A., Edsall, T., 2017. drmt: Disaggregated programmable switching, in: Proceedings of the Conference of the ACM Special Interest Group on Data Communication, ACM, New York, NY, USA. pp. 1–14. URL: http://doi.acm.org/10.1145/3098822.3098823, doi:10.1145/3098822.3098823.

[14] Consortium, P.L., 2017a. Behavioral model version 2. URL: https://github.com/p4lang/behavioral-model.

[15] Consortium, P.L., Dec 21, 2016. Switchsai. URL: https://github.com/p4lang/switch/tree/master/switchsai.

[16] Consortium, T.P.L., . Behavioral model targets. URL: https://github.com/p4lang/behavioral-model/tree/master/targets.

[17] Consortium, T.P.L., 2017b. The p4 language specification version 1.0.4. URL: https://p4.org/p4-spec/p4-14/v1.0.4/tex/p4.pdf.

[18] Consortium, T.P.L., 2018. P4 16 language specification:version 1.1.0. URL: https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.pdf.

[19] Corporation, N.D.I., 2017. Intel ethernet switch fm5000/fm6000(datasheet). URL: https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/ethernet-switch-fm5000-fm6000-datasheet.pdf.

[20] da Silva, J.S., Boyer, F., Chiquette, L., Langlois, J.M.P., 2018. Extern objects in p4: an rohc header compression scheme case study, in: 2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft), pp. 517–522. doi:10.1109/NETSOFT.2018.8460108.

[21] Dang, H.T., Sciascia, D., Canini, M., Pedone, F., Soulé, R., 2015. Netpaxos: Consensus at network speed, in: Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research, ACM, New York, NY, USA. pp. 5:1–5:7. URL: http://doi.acm.org/10.1145/2774993.2774999, doi:10.1145/2774993.2774999.

[22] Dang, H.T., Wang, H., Jepsen, T., Brebner, G., Kim, C., Rexford, J., Soulé, R., Weatherspoon, H., 2017. Whippersnapper: A p4 language benchmark suite, in: Proceedings of the Symposium on SDN Research, ACM, New York, NY, USA. pp. 95–101. URL: http://doi.acm.org/10.1145/3050220.3050231, doi:10.1145/3050220.3050231.

[23] Feamster, N., Rexford, J., Zegura, E., 2014. The road to sdn: An intellectual history of programmable networks. SIGCOMM Comput. Commun. Rev. 44, 87–98. URL: http://doi.acm.org/10.1145/2602204.2602219, doi:10.1145/2602204.2602219.

[24] Fischer, A., Botero, J.F., Beck, M.T., de Meer, H., Hesselbach, X., 2013. Virtual network embedding: A survey. IEEE Communications Surveys Tutorials 15, 1888–1906. doi:10.1109/SURV.2013.013013.00155.

[25] Foundation, L., 2018. Data plane development kit. URL: https://www.dpdk.org/.

[26] FOUNDATION, O., a. Open data plane (odp). URL: https://opendataplane.org/.

[27] FOUNDATION, O., b. Vector packet processing (vpp). URL: https://fd.io/technology/.

[28] Foundation, O.N., 2015. Openflow switch specification: Version 1.3.5. URL: https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.3.5.pdf.

[29] Gibb, G., Varghese, G., Horowitz, M., McKeown, N., 2013. Design principles for packet parsers, in: Architectures for Networking and Communications Systems, pp. 13–24. doi:10.1109/ANCS.2013.6665172.

[30] Group, T.P.A.W., 2018a. In-band network telemetry (int) dataplane specification. URL: https://github.com/p4lang/p4-applications/blob/master/docs/INT.pdf.

[31] Group, T.P.A.W., 2018b. P416 portable switch architecture (psa)(working draft). URL: https://p4.org/p4-spec/docs/PSA.html.

[32] GUDE, N., 2008. Nox : towards an operating system for networks. ACM SIGCOMM Computer Communication Review 38, 105–110. URL: https://ci.nii.ac.jp/naid/10028095244/en/.

[33] Haleplidis, E., Pentikousis, K., Denazis, S., Salim, J.H., Meyer, D., Koufopavlou, O., 2015. Software-Defined Networking (SDN): Layers and Architecture Terminology. RFC 7426. URL: https://rfc-editor.org/rfc/rfc7426.txt, doi:10.17487/RFC7426.

[34] Hancock, D., van der Merwe, J., 2016. Hyper4: Using p4 to virtualize the programmable data plane, in: Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies, ACM, New York, NY, USA. pp. 35–49. URL: http://doi.acm.org/10.1145/2999572.2999607, doi:10.1145/2999572.2999607.

[35] Høiland-Jørgensen, T., Brouer, J.D., Borkmann, D., Fastabend, J., Herbert, T., Ahern, D., Miller, D., 2018. The express data path: Fast programmable packet processing in the operating system kernel, in: Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies, ACM. pp. 54–66.

[36] Jose, L., Yan, L., Varghese, G., McKeown, N., 2015. Compiling packet programs to reconfigurable switches, in: 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), USENIX Association, Oakland, CA. pp. 103–115. URL: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/jose.

[37] Kaljic, E., Maric, A., Njemcevic, P., Hadzialic, M., 2019. A survey on data plane flexibility and programmability in software-defined networking. IEEE Access 7, 47804–47840.

[38] Kang, N., Liu, Z., Rexford, J., Walker, D., 2013. Optimizing the "one big switch" abstraction in software-defined networks, in: Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies, ACM, New York, NY, USA. pp. 13–24. URL: http://doi.acm.org/10.1145/2535372.2535373, doi:10.1145/2535372.2535373.

[39] Kazemian, P., Varghese, G., McKeown, N., 2012. Header space analysis: Static checking for networks, in: Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12), USENIX, San Jose, CA. pp. 113–126. URL: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/kazemian.

[40] Keller, E., Ghorbani, S., Caesar, M., Rexford, J., 2012. Live migration of an entire network (and its hosts), in: Proceedings of the 11th ACM Workshop on Hot Topics in Networks, ACM, New York, NY, USA. pp. 109–114. URL: http://doi.acm.org/10.1145/2390231.2390250, doi:10.1145/2390231.2390250.

[41] Kogan, K., Menikkumbura, D., Petri, G., Noh, Y., Nikolenko, S., Sirotkin, A., Eugster, P., 2017. A programmable buffer management platform, in: 2017 IEEE 25th International Conference on Network Protocols (ICNP), pp. 1–10. doi:10.1109/ICNP.2017.8117533.

[42] Kohler, T., Mayer, R., Dürr, F., Maaß, M., Bhowmik, S., Rothermel, K., 2018. P4cep: Towards in-network complex event processing, in: Proceedings of the 2018 Morning Workshop on In-Network Computing, ACM, New York, NY, USA. pp. 33–38. URL: http://doi.acm.org/10.1145/3229591.3229593, doi:10.1145/3229591.3229593.

[43] Lin, Y., Kozat, U.C., Kaippallimalil, J., Moradi, M., Soong, A.C., Mao, Z.M., 2018. Pausing and resuming network flows using programmable buffers, in: Proceedings of the Symposium on SDN Research, ACM, New York, NY, USA. pp. 7:1–7:14. URL: http://doi.acm.org/10.1145/3185467.3185473, doi:10.1145/3185467.3185473.

[44] Liu, J., Hallahan, W., Schlesinger, C., Sharif, M., Lee, J., Soulé, R., Wang, H., Caşcaval, C., McKeown, N., Foster, N., 2018. P4v: Practical verification for programmable data planes, in: Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, ACM, New York, NY, USA. pp. 490–503. URL: http://doi.acm.org/10.1145/3230543.3230582, doi:10.1145/3230543.3230582.

[45] Malloy, D., Maltz, D., Williams, C., Manickam, A.S., Daparthi, A., Wichmann, C., Lazar, M., Sane, S., Baldonado, O., Fang, T., Simpkins, A., Gale, B., Cummings, U., Daly, D., Penner, M., Kadosh, M., Baz, I., Raveh, A., 2015. Switch abstraction interface v0.9.2. URL: https://github.com/opencomputeproject/SAI/blob/master/doc/spec.md.

[46] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., Turner, J., 2008. Openflow: Enabling innovation in campus networks. SIGCOMM Comput. Commun. Rev. 38, 69–74. URL: http://doi.acm.org/10.1145/1355734.1355746, doi:10.1145/1355734.1355746.

[47] Medina, V., García, J.M., 2014. A survey of migration mechanisms of virtual machines. ACM Comput. Surv. 46, 30:1–30:33. URL: http://doi.acm.org/10.1145/2492705, doi:10.1145/2492705.

[48] Mittal, R., Agarwal, R., Ratnasamy, S., Shenker, S., 2015. Universal packet scheduling, in: Proceedings of the 14th ACM Workshop on Hot Topics in Networks, ACM, New York, NY, USA. pp. 24:1–24:7. URL: http://doi.acm.org/10.1145/2834050.2834085, doi:10.1145/2834050.2834085.

[49] Mizrahi, T., Moses, Y., 2016. The case for data plane timestamping in sdn, in: 2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), pp. 856–861. doi:10.1109/INFCOMW.2016.7562197.

[50] Mizrahi, T., Yerushalmi, I., Melman, D.T., Browne, R., 2018. Network Service Header (NSH) Context Header Allocation: Timestamp. Internet-Draft draft-mymb-sfc-nsh-allocation-timestamp-05. Internet Engineering Task Force. URL: https://datatracker.ietf.org/doc/html/draft-mymb-sfc-nsh-allocation-timestamp-05. work in Progress.

[51] NETRONOME, 2018. Agilio cx smartnics. URL: https://www.netronome.com/products/agilio-cx.

[52] Networks, A., 2018a. Arista 7170 series. URL: https://www.arista.com/en/products/7170-series.

[53] Networks, B., 2018b. Barefoot partners. URL: https://www.barefootnetworks.com/partners/.

[54] Networks, B., 2019. Barefoot tofino. URL: https://www.barefootnetworks.com/products/brief-tofino.

[55] Nötzli, A., Khan, J., Fingerhut, A., Barrett, C., Athanas, P., 2018. P4pktgen: Automated test case generation for p4 programs, in: Proceedings of the Symposium on SDN Research, ACM, New York, NY, USA. pp. 5:1–5:7. URL: http://doi.acm.org/10.1145/3185467.3185497, doi:10.1145/3185467.3185497.

[56] Patra, P.G., Rothenberg, C.E., Pongracz, G., 2017. Macsad: High performance dataplane applications on the move, in: 2017 IEEE 18th International Conference on High Performance Switching and Routing (HPSR), IEEE. pp. 1–6.

[57] Rim, S.Y., Cui, Z., Qian, L., 2018. High performance packet processor architecture for network virtualization: Programmable packet processor architecture as a data flow machine, in: Proceedings of the 2018 International Conference on Algorithms, Computing and Artificial Intelligence, ACM, New York, NY, USA. pp. 6:1–6:5. URL: http://doi.acm.org/10.1145/3302425.3302435, doi:10.1145/3302425.3302435.

[58] Roberts, J.W., 2004. Internet traffic, qos, and pricing. Proceedings of the IEEE 92, 1389–1399. doi:10.1109/JPROC.2004.832959.

[59] Sapio, A., Abdelaziz, I., Aldilaijan, A., Canini, M., Kalnis, P., 2017. In-network computation is a dumb idea whose time has come, in: Proceedings of the 16th ACM Workshop on Hot Topics in Networks, ACM, New York, NY, USA. pp. 150–156. URL: http://doi.acm.org/10.1145/3152434.3152461, doi:10.1145/3152434.3152461.

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

[61] Shahbaz, M., Choi, S., Pfaff, B., Kim, C., Feamster, N., McKeown, N., Rexford, J., 2016. Pisces: A programmable, protocol-independent software switch, in: Proceedings of the 2016 ACM SIGCOMM Conference, ACM. pp. 525–538.

[62] Sherwood, R., Gibb, G., kiong Yap, K., Casado, M., Mckeown, N., Parulkar, G., 2009. FlowVisor: A Network Virtualization Layer. Technical Report.

[63] Sivaraman, A., Subramanian, S., Agrawal, A., Chole, S., Chuang, S.T., Edsall, T., Alizadeh, M., Katti, S., McKeown, N., Balakrishnan, H., 2015. Towards programmable packet scheduling, in: Proceedings of the 14th ACM Workshop on Hot Topics in Networks, ACM, New York, NY, USA. pp. 23:1–23:7. URL: http://doi.acm.org/10.1145/2834050.2834106, doi:10.1145/2834050.2834106.

[64] Sivaraman, A., Winstein, K., Subramanian, S., Balakrishnan, H., 2013. No silver bullet: Extending sdn to the data plane, in: Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks, ACM, New York, NY, USA. pp. 19:1–19:7. URL: http://doi.acm.org/10.1145/2535771.2535796, doi:10.1145/2535771.2535796.

[65] Sivaraman, V., Narayana, S., Rottenstreich, O., Muthukrishnan, S., Rexford, J., 2017. Heavy-hitter detection entirely in the data plane, in: Proceedings of the Symposium on SDN Research, ACM, New York, NY, USA. pp. 164–176. URL: http://doi.acm.org/10.1145/3050220.3063772, doi:10.1145/3050220.3063772.

[66] Sivaranjani, B., Vijayakumar, P., 2015. A technical survey on various vdc request embedding techniques in virtual data center, in: 2015 National Conference on Parallel Computing Technologies (PARCOMPTECH), pp. 1–6. doi:10.1109/PARCOMPTECH.2015.7084515.

[67] Tu, W., Ruffy, F., Budiu, M., 2018. Linux network programming with p4, in: Linux PlumbersâĂŹ Conference 2018.

[68] Turkovic, B., Kuipers, F., van Adrichem, N., Langendoen, K., 2018. Fast network congestion detection and avoidance using p4, in: Proceedings of the 2018 Workshop on Networking for Emerging Applications and Technologies, ACM, New York, NY, USA. pp. 45–51. URL: http://doi.acm.org/10.1145/3229574.3229581, doi:10.1145/3229574.3229581.

[69] Vörös, P., Horpácsi, D., Kitlei, R., Leskó, D., Tejfel, M., Laki, S., 2018. âĂđt4p4s: A target-independent compiler for protocolindependent packet processorsâĂİ, in: IEEE HPSR, pp. 17–20.

[70] Yao, J., Wang, Z., Yin, X., Shiyz, X., Wu, J., 2014. Formal modeling and systematic black-box testing of sdn data plane, in: 2014 IEEE 22nd International Conference on Network Protocols, pp. 179–190. doi:10.1109/ICNP.2014.37.

[71] Zhang, C., Bi, J., Zhou, Y., Dogar, A.B., Wu, J., 2017. Hyperv: A high performance hypervisor for virtualization of the programmable data plane, in: 2017 26th International Conference on Computer Communication and Networks (ICCCN), pp. 1–9. doi:10.1109/ICCCN.2017.8038396.

[72] Zhang, C., Bi, J., Zhou, Y., Wu, J., 2019. Hypervdp: High-performance virtualization of the programmable data plane. IEEE Journal on Selected Areas in Communications 37, 556–569. doi:10.1109/JSAC.2019.2894308.

[73] Zhu, Y., Kang, N., Cao, J., Greenberg, A., Lu, G., Mahajan, R., Maltz, D., Yuan, L., Zhang, M., Zhao, B.Y., Zheng, H., 2015. Packet-level telemetry in large datacenter networks. SIGCOMM Comput. Commun. Rev. 45, 479–491. URL: http://doi.acm.org/10.1145/2829988.2787483, doi:10.1145/2829988.2787483.