A 400 Gb/s Carrier-Class SDN White-Box Design and Demonstration: The Bitstream Approach

Aniruddha Kushwaha, Sidharth Sharma, Naveen Bazard, Tamal Das, and Ashwin Gumaste

***Abstract*—Software defined networks (SDNs) could be a game changer for next generation provider networks. OpenFlow (OF)—the dominant SDN protocol, is rigid in its south bound interface—any new protocol field that the hardware must support, must await complete OF standardization. In contrast, OF alternatives such as protocol oblivious forwarding and forwarding and control element separation have simpler schemes for insertion of new protocol identifiers. Even with these there is an inherent limitation on network hardware—the tables must support specific formats and configuration at each node as per protocol semantics. We ask the question—*can we design an open system (white-box) – one that is carrier-class, yet able to meet the requirements of any protocol forwarding/action with a minimal set of data-plane functions*. We propose bitstream, a low-latency, source-routing based scheme that can support addition of new protocols, be compatible with existing protocols, and facilitate a minimum semantic set for acting on a packet. A prototype is built to show bitstream working. The controller architecture is detailed from a provider perspective, as to how it can be integrated in a provider network using YANG models. The hardware architecture is also presented, showing the functioning of a bitstream capable 400 Gb/s whitebox. The issue of protocol processing optimization is considered and its impact on service latency is shown. The results from the test-bed validate the carrier-class features of the bitstream model.**

***Index Terms*—Carrier-class networks, SDN, whiteboxes.**

I. Introduction

SERVICE provider concerns over ever-increasing CapEx and OpEx in a frequent bandwidth-multiplying Internet economy combined with near-flat revenues has led to a broad consensus around the growth of Software Defined Networking (SDN) [1]. SDN has the potential to do away with complex specialized network equipment and replace these with centrally programmable "whiteboxes" [2]. Most of SDN deployments so far have been in enterprise-class networks [3], in campuses [4] and within data-center premises [5]. Providers are reluctant to large-scale SDN deployment due to unavailability of carrier-class large whiteboxes. Even without a killer application, an SDN whitebox can potentially justify the investment on account of plausible savings vis-à-vis contemporary non-agile network equipment [6]. One approach towards an SDN network is to use current network gear and make it compatible with an SDN controller. Another approach is to inculcate a whitebox based solution [7].

Among others, there are two key obstacles that can impact next generation SDN deployment: (1) The SDN protocol of choice (OF [8]) is rather rigid. While OF can be programmed to just about any user-oriented application, it eventually functions on 40-odd protocol identifiers in its controller's south bound interface (SBI), which means that any new protocol that the hardware has to support has to be first routed through a standardization exercise before making into a provider's network. (2) Many vendors have developed their own controller – one that provides a programmable north bound interface (NBI) [9], [10] but severely restricts the south bound interface (SBI) to a vendor-specific forwarding plane.

Providers' desire the full Operations, Administration, Management and Provisioning (OAM&P) features. The introduction of SDN would imply that SDN technologies adhere to the OAM&P service requirements. Current discussions around OAM&P support in SDN are only in the nascent stages and would likely have an impact on assimilation of SDN in provider networks. In particular, providers seek per-service monitoring support and ability to restore service post-a-failure within 50 ms for various kinds of failures as well as end-to-end deterministic delay that facilitates service guarantees and Quality of Experience to the end-user.

In this paper, we propose a solution that removes the limitations on the SBI match-identifier fields, while preserving the programmability in the NBI of the controller. In addition, our conceptual solution called *bitstream* is able to provide carrier-class attributes such as 50 ms restoration, per-service handling capability and deterministic delay. A key contribution of this work is that we build a carrier-class white-box that is capable of performing in provider networks while adhering to SDN concepts, particularly being able to meet a wide spectrum of protocol requirements. This flexibility in protocol support is crucial to achieve programmability in the NBI. We argue that the white-box can perform any programmable task using an SDN controller [11]. Any kind of service can be set up using a modeling language such as YANG [12], [13]. The novelty of the whitebox is the compliance to the bitstream protocol, which in some sense is a generalization of the OpenFlow protocol. In order for the whitebox to function in a provider domain we have developed custom hardware that comprises of FPGAs, framer chips and IO ports along with peripheral electronics such as

Manuscript received November 25, 2017; revised February 19, 2018; accepted March 12, 2018. Date of publication March 26, 2018; date of current version June 15, 2018. *(Corresponding author: Ashwin Gumaste.)*

The authors are with the Department of Computer Science and Engineering, Indian Institute of Technology, Bombay, Mumbai 400076, India (e-mail: anikush88@yahoo.com; sidharth@cse.iitb.ac.in; bazard.naveen@gmail.com; tamaldas@cse.iitb.ac.in; ashwing@ieee.org).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/JLT.2018.2819179

DDR, TCAM memories and processing units. To meet provider requirements in the field, we have also developed a network management system (NMS) that facilitates data collection and event gathering [14], service design features and provisioning aspects as well as serves as a conduit to third-party controllers and potentially opens out to programmers and developers.

The bitstream concept is based on a virtual topology superimposing schema that uses the SDN premise of control and data-plane separation to build a virtual topology that is embedded on a physical topology. The central idea is as follows: a controller abstracts each physical node in a network as an auxiliary *node forwarding graph* (NFG), whose every vertex can be traversed using a single bit. This means that no vertex in the NFG would subtend more than 3 undirected edges. End-to-end paths are now manifested at the controller by conjoining the vertices in the NFGs along the shortest path for each node in the actual topology. The resulting source-routed information is called a bitstream snippet and is stored in MPLS-like labels. The idea is to map all sorts of forwarding protocols (such as IPv4, IPv6, MAC, VLAN, port-based, MPLS, CTAGs/STAGs, etc.) to a homogenous source-routed bitstream snippet (BS).

We argue that such an approach is not just beneficial from the perspective of simplicity, but also solves our two issues of scalability in terms of new protocol adoption as well as carrier-class support. An important advantage of our scheme is overall reduction in latency. We show that for an h-node path, the latency is reduced to approximately $1/\ (h-1)$ of the latency as compared to any other forwarding scheme, such as OF, MPLS etc.

In Section II we discuss prior work that is related to the bitstream contribution. Section III illustrates the principal contribution of this paper – the bitstream scheme. Sections IV and V delve into the hardware and software implementations of our approach. Section VI shows how packet processing occurs using two diverse illustrations. Section VII is an optimization model that computes how to place processing nodes in the network. Section VIII shows simulations and measurements from a testbed that was built from a 400 Gbps capable bitstream hardware.

II. Related Work

One of the first approaches of data-control plane separation is the ForCES work [15], [16]. ForCES provides a generic framework for data-plane-control-plane separation. Importantly ForCES defines a logic and forwarding block (LFB) class library that shows how the data-plane can be implemented to facilitate forwarding for specific protocol requests. This generic partitioning of the data-plane for specific protocol requirements is an inspiration for our architecture. While we do not define specific LFB structures, we argue that using the bitstream concept we are able to better partition the data-plane leading to determinism in the network. Our one-size fits all schema is more open than ForCES for protocol support.

OpenFlow (OF) is by far the leader of the controller protocols pack. While OF is extremely flexible with the NBI of a controller, the SBI supports a fixed set of protocol match identifiers. In fact, as compared to OF1.3 and 1.4 the set of match identifiers has been increased in OF1.5. The problem with this approach is that for each new protocol to be added to the SBI of the OF controller, we have to wait for a two-year round of standardization. Recent examples of VXLAN [17] and NVGRE [18] are testimonial to this delay. Our approach is protocol-agnostic. We argue that any new protocol can readily be added to the bitstream supporting protocol suite. Further such upgrades can be distributed (node-specific) and *in situ* (without affecting the data-plane traffic). A second advantage of our approach over OF is that our focus is particularly provider-oriented and we build upon the concept of provisioned services. Though OF is restricted in the number of fields that its SBI can process, in a way this is important as it also implies good interoperability with gear from various vendors as well as backward compatibility with already installed gear.

Protocol Oblivious Forwarding [19] (POF) relaxes the specific requirements of SBI data-structures used in OF by proposing a reduced set of actions to be performed on a packet, in addition to an offset value that would define the protocol being worked upon in the packet header. The main difference between POF and our scheme is that in POF the hardware must be designed to support specific protocol actions, while in our scheme, the hardware is generic and not protocol-specific. Further, we argue that our scheme is more efficient than POF from a delay perspective. Once a packet is encoded with a bitstream at the edge of a network the processing turns out to be much simpler than POF.

In [20] the authors propose manipulating OpenFlow to add a source-routing tag that facilitates source-routing within a data-center for up to 256 ports. Multiple such tags can be inserted for scalability. The work is specific to data-centers and focuses only on forwarding, implying that other technologies are needed for achieving full network function. Our work is beyond the realm of data-centers and can be expanded to any protocol (see Appendix B) and any network topology.

In [21] the authors explore source-routing in WANs and illustrate the concerns with source routing from a protocol overhead perspective. Our approach of using network forwarding graphs at nodes alleviates this concern.

Omnipresent Ethernet or OE [22], [23] is somewhat similar to our approach from the data-plane perspective and is predecessor to this work. OE is purely built using a Carrier Ethernet stack and does not require an SDN controller. We do use the carrier-class data plane features of Omnipresent Ethernet while facilitating programmability and opening up the data-plane to a much larger spectrum of protocols. While OE facilitates IP, MAC and port based services, it cannot provision layer 4 and other such services which require processing beyond mere forwarding.

Segment routing [24] is a flexible way of doing source routing where the source node chooses a path and encapsulates the path information in a packet header as a list of segments. A segment is a sub-path along a route between any two nodes. Each segment is associated with a segment identifier. The segment identifiers are distributed to all the nodes present in the network using IGP. Segment identifiers could be node identifiers or adjacency identifiers. Node identifiers are associated with a node (router)

and are unique throughout the network. Adjacency identifiers are local to a node and represent the interfaces of the node. Bitstream uses segment identifiers similar to adjacency identifiers for specifying the source routed path. Unlike segment routing which requires IGP-like-protocols, our approach is completely protocol agnostic. The other major differentiator is that members of a bitstream snippet have spatial meaning (ports), while a segment ID is a generically allocated number.

Segment routing in conjunction with SDN has been considered in [25], [26]. The approach is to use populate MPLS forwarding tables through a controller. Our approach is more advanced as we support forwarding based on many different identifiers (not just layer 2.5 labels).

BIER [27], [28] is a segment routing-based scheme where a unique node identifier is assigned to each node in the network. The ingress node encodes the packet with the BIER header that contains a bitstring. The bitstring has a bit corresponding to an egress node in the network. For any flow, this scheme imposes the requirement of a bit forwarding table in addition to match-tables. Our solution neither requires any match at the intermediate nodes nor does it require any additional table.

A shorter version of this work with limited results and without the software and hardware details or analytical justification is presented in [29].

III. Concept of Bitstream

A network graph $G(V, E)$ of a set of V vertices and E edges is abstracted to an auxiliary collection of *node forwarding graphs* $NFG(\overline{V}, \bar{E})$, where a node in the NFG is denoted by $\{\overline{V_i}\}$ and is a k-dimensional one-hop representation of a source-routed path through the actual node V_i in G. This implies that for each $V_i \in G$, $\exists\{\overline{V_i}\} : V_a \in \{\overline{V_i}\} \in \{0, 1\}$. This implies that a node in the physical topology is represented by a set of 1×2 nodes in the NFG denoting the source-routed path that is to be traversed by the forwarding-plane. Further we say that $V_a = 1$, if the path through V_a has to take a right-turn (from the base of the 1×2 tree rooted at V_a), or $V_a = 0$, if the path through V_a has to take a left-turn beginning at the base of the 1×2 tree rooted at V_a. Note that for every service passing through the same physical node V_i, the value of V_a and the set $\{\overline{V_l}\}$ may be unique, i.e., if V_a is stationed in a path, then for an East-to-West service the value of $V_a = 1$, i.e., in this case the service needs to pass through V_a and not be dropped at V_a. Conversely, if the service is West-to-East, then the value of V_a for that service would be 0. This shows that V_a is direction and service specific. The physical node V_i may have several $V_a s$ as part of its NFG representation, resulting in a *bitstream* to traverse through V_i (Fig. 1). If the degree of V_i (cardinality) is D_i, then for traversing through V_i there can be at the most $k = (D - 1)!$ bitstreams in the NFG. However, we allow for $k > (D - 1)!$ to account for additional service specific functions which are beyond pure forwarding (such as label-swap, BGP-peering, ACL, etc.) (see Fig. 1). Shown in Fig. 1 is a physical network (bottom) that is converted into a NFG supporting network for only forwarding (in the middle) and is further enhanced with specific action implementation at the NFG level (at the top). The topmost implementation in Fig. 1

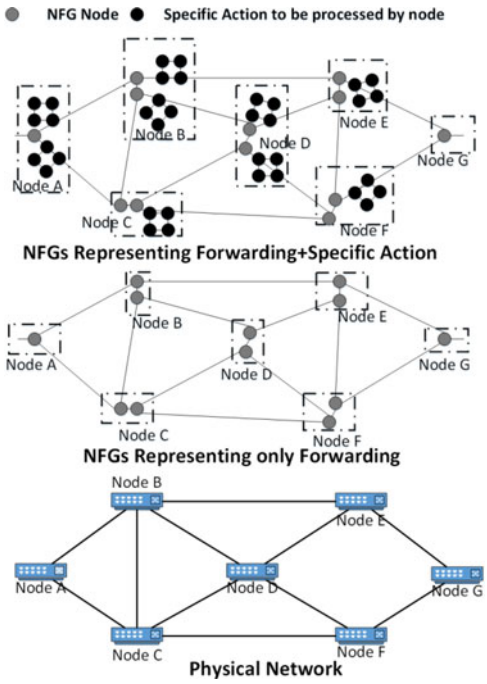

Fig. 1. Physical Network and its NFG representation.

essentially shows the journey of a packet through forwarding and specific action (the dark-black nodes are used for processing functions such as TTL decrement, packet reordering etc., while the grey nodes are used for only forwarding).

A. Bitstream Protocol

We term a network that supports bitstream as a *closed domain*. A closed domain is typically a provider network. The closed domain is governed by one or more controllers. An incoming service is encoded by a *bitstream snippet* (BS) while entering the closed domain. The bitstream snippet for a service is generated as follows:

1) The controller computes the shortest path that suffices for all the parameters for the service in G.
2) The controller then creates the NFG equivalent of each node along the path and conjoins each NFG equivalent stream to generate a bitstream snippet (BS). The BS is essentially source-routed information for carrying the packet from ingress to egress. If the service requires actions apart from pure forwarding such as swap, or sorting, then the controller sets few additional bits in the bitstream snippet. For pure forwarding (switching/routing) the BS is enough to carry the packet to the destination.
3) The controller then creates entries for the table in the parser at the ingress node identifying the service parameter to be measured (such as IPv4, IPv6, TCP port number, or any other combination).

We define two broad service types: (1) provisioned service and (2) impromptu service.

Provisioned service: A provisioned service is one in which the controller has created entries in the forwarding table at the ingress node ahead in time and only thereafter can the data flow.

For example, L2/L3VPN services and Internet leased-lines both of which together constitute bulk of enterprise traffic.

Impromptu service: An impromptu service is one in which the host sends data without prior intimation of the controller, similar to a host attached to a LAN. In this (typically enterprise) setting, traffic from a host is sent unsolicited to a node (edge of the closed domain). If there is a match between any of the service attributes with an entry in the preconfigured table, the packet is thereafter encoded with the BS, else the packet is either routed to a default port or dropped.

B. Bitstream Node Architecture

A node in the closed domain is assumed to have SDN forwarding capabilities, specifically by a table that can be configured by a controller through a control state machine (CSM). The CSM resides in the whitebox hardware and interacts with the controller's SBI. The tables in a bitstream capable node is based on the Reconfigurable Match Table [30] principle. When a packet enters a node, it is classified at a parser as "marked" or "unmarked" based on the Ethertype of the packet. An unmarked packet is one that comes from outside the closed domain and has no BS in it. A marked packet comes from another bitstream supporting node.

For an unmarked packet, the parser sends the packet to a match-table. The match-table may have a match for this packet or may drop the packet or send it to a default port. If a match occurs, then the packet is marked by adding the BS label after the L2 header. This label is of variable length, of which the first 2-bytes constitute a custom Ethertype tag (different tags for unicast and multicast service). The remaining bits constitute the BS and contain a pointer and a set of NFGs.

A node of k-dimensions requires up to $4(\log_2(k)-2)+n$ bits to traverse through it. In these, $2(\log_2(k)-1)$ bits are needed for nascent forwarding; n additional bits are required for specific service parameters (such as ACL, packet ordering, packet modifications etc.) that the node must process in addition to forwarding (Fig. 1); and another $2(\log_2(k)-1)$ for going through the forwarding portion a second time (after service related action). The pointer informs a node in G as to where to start counting the $4(\log_2(k)-2)+n$ bits in BS for forwarding. In the event of only forwarding, we need just $2(\log_2(k)-1)$ bits. The packet after the BS addition is forwarded to either an `action` section or a `switch` section of the bitstream engine (whose functioning is detailed in the Section IV) based on the additional service-specifying bits.

For a marked packet, i.e., one which already has a BS inserted in it, the packet is sent to the parser. The parser extracts the relevant $4(\log_2(k)-2)+n$ bits and routes the packet to the `action`/`switch` section of the bitstream engine.

If any of the n additional bits are set, then the packet is sent to the "`action`" section of bitstream engine. The `action` section invokes a separate hierarchy for dealing with the packet. The `action` section is where all actions apart from forwarding are performed such as: packet reordering, TTL decrement, or any other experimental actions that may be introduced. Each of the n bits correspond to a particular action that needs to be performed. The `action` section after processing the packet, sends it back to the `switch` section for forwarding to the requisite port (based on $2(\log_2(k)-1)$ bits). The pointer is then incremented by up to $4(\log_2(k)-2)+n$ bits at the output port. For forwarding the packet, a hardware encoded logic is used in the `switch` section, which has an output port (one of the VOQ buffers) assigned for each combination of the $4(\log_2(k)-2)$ bits. We again note that once a packet is marked, a node only considers the BS for forwarding or for any other protocol function to be implemented and does not further rely table lookups.

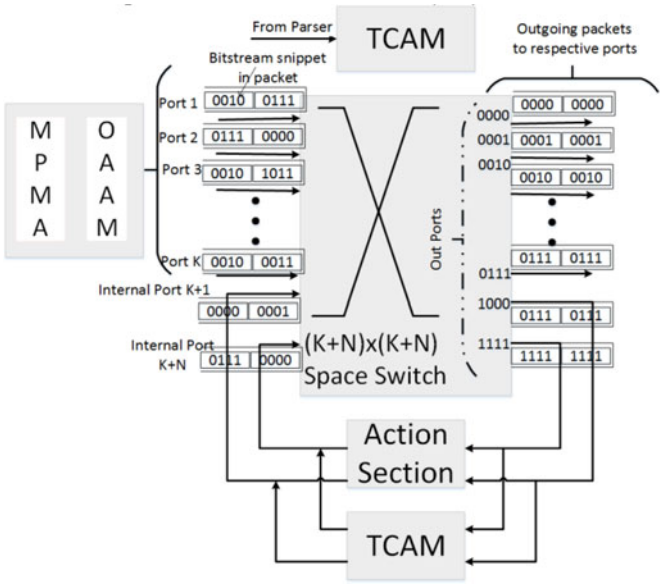

Fig. 2. Data plane architecture of a bitstream switch.

Shown in Fig. 2 is a $k \times k$ switch that supports bitstream. In this figure, packets arrive at ports and are parsed prior to arrival. Packets are first encoded with the BS and then sent to the switching fabric. If a packet requires only a forwarding operation, then it is directly routed to the destination port based on $2(\log_2(k)-1)$ bits. If, however a packet is to be further processed or manipulated then it is forwarded by the switch to the action section, which can be reached through one of the n extra ports. The packet after the action section is sent back to the switch which now sends it to a corresponding egress port. The journey of a packet through the action and switch together is defined by $4(\log_2(k)-2)+n$ bits in the BS. Typically, k can be of the range 12-64 ports depending on the line rate on a per-card basis. In our case, we facilitate a 33-port switch in the front plane and 66 ports in the backplane.

C. Multicast Handling

Multicast is handled different from unicast. A bitstream snippet for multicast is generated in a different manner. The label of a multicast packet is differentiated by a custom Ethertype tag. The node receiving the packet now knows that the packet is of multicast type. The node checks for the pointer-value and then extracts D bits (D is the degree of the node), which denote a multicast vector for forwarding the packet to the appropriate multicast ports. The protection aspect of multicast requires that

the controller find a node-and-edge-disjoint tree. This problem is shown to be NP-hard [31] and not handled in this paper. An elegant solution with some relaxations is shown in [32] for handling multicast. Multicast is implemented similar to the OE scheme and BIER schemes [27]. Ports are identified for multicast and port-vectors that are node specific are created. These port-vectors are conjoined to create the bitstream snippet. The port-vectors in our scheme are similar to port-masks in the BIER scheme (which is entirely layer 2.5/3 multicast), whereas our scheme could be implemented for multicast at any layer.

D. Carrier Class Support

we define carrier-class support through features such as: 50 ms restoration post a node/edge failure; service provisioning; service monitoring and deterministic latency. The bitstream scheme facilitates the former through the proven method of incorporating IEEE802.1ag (or ITU.T Y.1761) connectivity and fault management standard [33]. For each provisioned service, we mark the edge nodes of the service as management end-points (MEPs). MEPs exchange bidirectional connectivity check messages (CCMs) every 10 ms. It is essential that this "control plane" follows the same path as the data-plane and is distinguished by its unique QoS value. Loss of 3 consecutive CCMs triggers the destination to begin accepting traffic on the protection path. This sort of protection is $1 + 1$ (always ON) type and could potentially be replaced with less aggressive schemes. The adoption of the 802.1 ag standard facilitates monitoring, path continuity check, etc. Deterministic latency is achieved through simplistic forwarding as explained in Section VII.

E. Backward Compatibility and Interoperability

The closed domain interacts with hosts and other networks as a pure L2 network within which users can provision any service. A flow can be made to pass (as a transparent service) through the closed domain or as a specific protocol compliant service depending on the provisioning status. The closed domain accepts any traffic that supports Ethernet interfaces. It is pertinent to note that a closed domain can be a single node supporting the bitstream protocol or a country-wide WAN.

IV. Bitstream Hardware: Parser, Tables, Switch Fabric

The bitstream hardware consist of multiple match-tables and logical modules. We now describe the detailed hardware whose implementation is shown in Fig. 2.

Logical blocks within the bitstream hardware are shown in Fig. 3. Each IO port is connected to: a *multi-stage parser and match action* module (MPMA), an *output apply action module* (OAAM), *bitstream engine* (BE) and *virtual output queues* (VOQs). The switch fabric module and match-tables are common across all the ports.

A. Match Table

Each entry in a match-table contains the following fields as shown in Fig. 4.

a) *Key:* Matched against the fields provided by the parser.

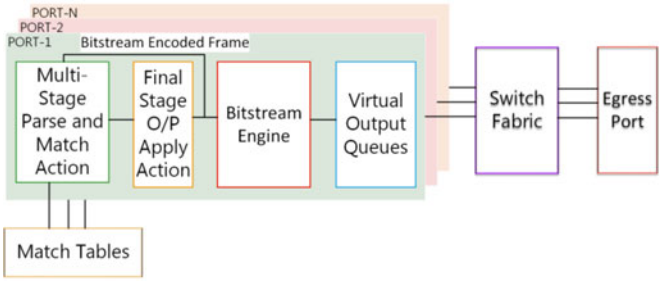

Fig. 3. Bitstream Hardware Logical modules.

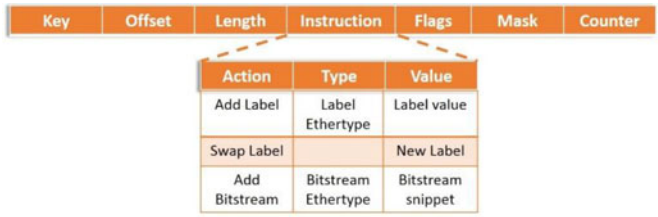

Fig. 4. Match table structure.

b) *Offset:* Indicates the location in the packet for the parser to extract data or to insert/modify a field.

c) *Length:* Provides the length of the field (protocol identifier or bitstream snippet) to be extracted from the packet or to be inserted into the packet.

d) *Instruction:* Provides the action or routing information details required for packet processing at a node. The instruction field is further divided into three sub-fields: (a) *Action:* provides the action information that needs to be executed on the packet, i.e., push or pop or swap a label/tag, decrement TTL, set VLAN, drop packet, add bitstream snippet, etc. (b) *Type:* has information pertaining to the Ethertype which is useful in case the `action` instruction requires to add tags i.e VLANs, MPLS, PBB, etc. (c) *Value:* has information corresponding to the action to be executed, i.e., for adding a label, it has a label value; for swapping labels, it has a new label value to be swapped; for bitstream addition, it has a bitstream snippet for the default and protection route along with QoS value.

e) *Flags:* are provided to process the table information in the parser stages correctly. We define three types of flags: (a) SET_OLI: decides if subsequent parsing of the packet in parser stages is required; (b) Apply_Action: informs whether the instruction field contains a valid action that needs to be executed on the packet; (c) Apply_Key: allows the parser to directly set the "value" field of the instruction as a key for the next stage match-action.

f) *Mask:* The mask is used when a partial match of the key is sufficient to get an instruction i.e., subnet mask for IP. This allows the match-action logic to initiate a match against a key only for the masked bits of the field.

The Offset, Length and Instruction (OLI) fields together constitute an OLI value, which is used for processing a packet in the parser in the bitstream hardware. The OLI value is generated and written in match-tables by the controller.

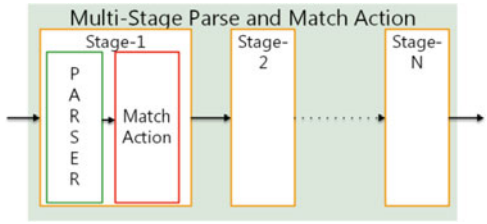

Fig. 5. Multistage parse and match action.

B. Multi-Stage Parser and Match Action (MPMA)

Every incoming packet is initially processed by the MPMA. The MPMA is divided into multiple stages (Fig. 5), where each stage has a parser and a match-action logic. The parser first checks if a packet is marked i.e., contains a BS. If the packet is marked, then the parser extracts the relevant $2(\log_2(k)-1)+n$ bits based on the pointer and forwards the packet to the BE. If the packet is unmarked, then the parser extracts the match fields based on the OLI value in the subsequent stages. The parser in the 1st stage is set by default to extract the source and destination MAC address and corresponding Ethertype field. The parser logic in the 1st stage is programmable by the bitstream controller to extract any arbitrary field from the packet for 1st stage match-action logic. The parser in subsequent stages are programmed based on the OLI value returned from the match-table of previous stages. This programmability gives flexibility to the bitstream hardware to parse and process all types of packets. Based on the OLI value, the parser extracts the respective fields from the packet and forwards the extracted fields to the Match-Action logic, to be used as a *key* for matching in the table. Once a match is found a new OLI value and flags are obtained for the next stage. If the *SET_OLI* flag is true, then the parser of the next stage is programmed based on the obtained OLI from the table. If the *SET_OLI* flag is false, the instruction field returned from the table consists of an action-set or routing information. If the *Apply_Action* field is true, then the instruction field returned by the table consists an action-set information which needs to be executed. If the *Apply_Key* flag is true, then the parser directly applies the information retrieved from the instruction field of the table as a key for match-action in the next stage.

Example: If a match-table returns the instruction to add an MPLS label (value 0x1234) and the *Apply_Key* is set as true. Then the MPLS label (with value 0x1234) will be applied as a key for obtaining a match lookup in the table in the next stage. Once all the parsing of a packet is done, a metadata is generated that constitutes key/OLI and instruction information received in the parsing stage. This metadata along with the packet is forwarded to the OAAM module for further processing. The final action is the addition of the BS in the packet leading to successful forwarding through the node.

C. Output Apply Action Module (OAAM)

This module receives the metadata and the packet from the MPMA. The metadata consists of a set of actions (such as decrement TTL, Push/Pop tags, Swap label, add BS etc.) and routing information in the form of a BS received from the match-tables. The OAAM module processes and applies these set of actions and marks the packet by embedding the BS. After all the actions are completed, the packet is forwarded to the bitstream engine.

D. Bitstream Engine (BE)

This module is responsible for the processing of all the data packets as well as control packets (packets that are exchanged between the controller and the bitstream switch i.e., `ping`, `echo`, *Config* etc.). BE consist of the CSM, action and switch sections. The `CSM` section is used for the interaction with the SDN controller. The `Action`-section is required to perform operations that cannot be performed simply by match-action in the previous stages (such as reordering of packets, etc.). The specific action to be undertaken is decided based on the n-additional bits of the BS relevant to that node. The `switch`-section is responsible for processing all the marked packets. The `switch`-section identifies the egress port for the packet by following the $2(\log_2(k)-1)$ bits from the valid pointer location of a BS as described in Section III. Once the `switch`-section identifies a valid physical egress port, the used bits are then invalidated by incrementing the pointer of the BS by $2(\log_2(k)-1)+n$ number of bits that were used in identifying the physical port and action section processing. This allows the `switch`-section to take a forwarding decision based on the unused/valid bits of the BS. The BE also checks the status of the egress port after identifying a physical port as to whether the egress port is within the closed domain. In case the port is outside of the closed domain, it strips the custom Ethertype and BS from the packet. This process facilitates interoperability with conventional networks.

E. Switching Module

consists of the switch-fabric and an arbiter for each egress port for packet scheduling from the VOQs and is coded using 6×1 multiplexers in VHDL. Based on the buffer occupancy in the VOQ, an egress port arbiter selects the appropriate queue to transfer a packet.

V. Bitstream Controller

We now discuss the software aspect of the bitstream framework, in particular the controller. The bitstream controller provides functionality to capture and translate user requests into the flows/services that are mapped to network resources in the closed domain. This translation is done either by using the pre-existing set of protocols (such as IPv4, IPv6, VLAN, MPLS etc.) or by user-defined protocols/services. New/existing protocols are defined in the service templates' repository of the controller. The controller also allows a user to define the set of policies related to the physical resource mapping and routing of the service. The controller keeps tracking the status and monitors network health by periodically exchanging control messages with bitstream nodes in the closed domain. Key control messages for the SBI are listed in Table I.

Each control message is classified by assigning a unique *opcode*, which is used by the controller and the bitstream node to differentiate the control message.

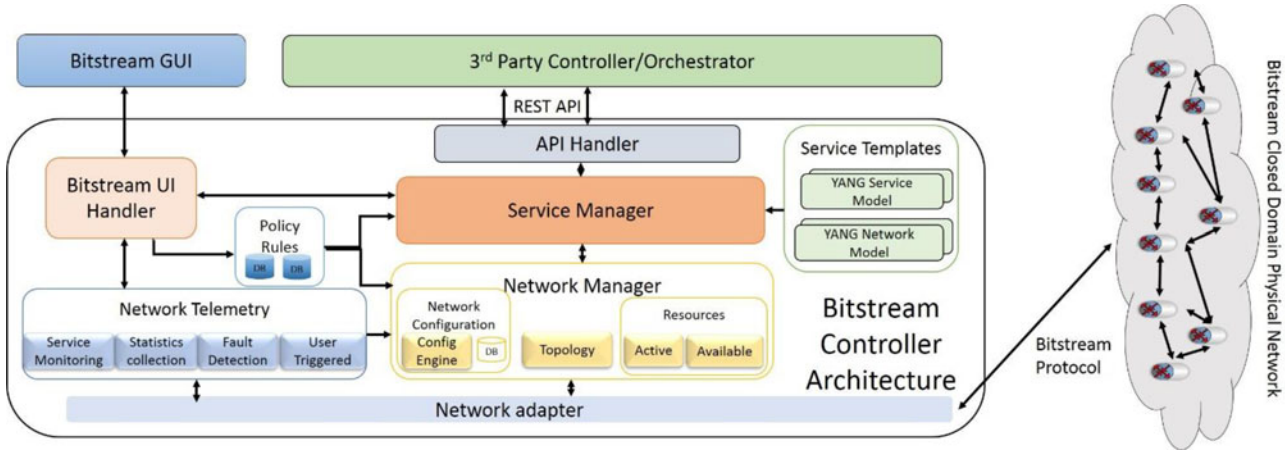

Fig. 6. Bitstream controller architecture.

TABLE I
KEY CONTROL MESSAGES

Ping	is initiated by the controller. A bitstream node broadcasts this message after receiving it from the controller or other bitstream nodes
Echo	is sent by each bitstream node to the controller as a response of a *Ping Message*. An *echo message* conveys hardware capabilities, such as ports, connections, cards etc. In addition this message also carries port status and information of the connected neighboring nodes.
Config	is sent by the controller for writing the match/lookup rules into the match-tables of the bitstream hardware
Config-ack	is sent by the bitstream node in response to a *Config Message,* when match/lookup rules are successfully written in the match-table
Config-nack	is sent by the bitstream node in response to a *Config* message when match/lookup rules cannot be written in the match-table
Connectivity Check	is sent by a service ingress bitstream node to a service egress bitstream node for fault detection.
Monitor	is sent by the controller to get the statistics of a service/node
Monitor Reply	is sent as a response to the *Monitor Message.* This message contains the statistical information about a parameter described in the monitor message. The reply may include packet drop count, latency of a service etc. A bitstream node periodically sends this message to the controller.

Controller architecture: The architecture of the Bitstream controller and its functional modules are shown in Fig. 6. Different modules of the controller are now described.

A. Network Manager

The network manager module maintains the current state of the closed domain network by storing information pertaining to available resources (nodes and their capabilities) and provisioned services in a database. This module discovers the network topology and capabilities of the nodes by using *ping* and *echo* messages, and subsequently forms a node-adjacency matrix. This adjacency matrix is used to create a network graph representing the real-time network topology. Based on the extracted information, the network manager also updates its resource inventory in its database. The network manager uses this network topology and resource inventory information to map the service configuration request received from the service manager. Once the configuration request is successfully mapped by the network manager to the physical network, node configuration is initiated by the network configuration submodule using a *config*. A node responds to the *config* by a *config-ack* if the node is configured successfully, else the node responds with a *config-nack*.

In the case that the network manager receives a *config-nack*, then the *config* message is resent. This process is repeated till a time-out occurs. On reaching a time-out, all nodes related to the unsuccessful service request are rolled back to their previous state by using the earlier configuration stored in the network manager database.

B. Bitstream Handler

The bitstream handler module classifies the requests originating from the bitstream GUI application and forwards them to an appropriate controller module. There are three types of requests a) service configuration; b) service monitoring; and c) policy updates. Based on the request type from the GUI application, the bitstream handler module forwards the request to Service manager, Network Telemetry or Policy rules module.

C. Service Manager

The service manager handles all the requests related to a service configuration. A user can request a service configuration either by using a GUI or possibly through a REST API. For this paper, we define two types of service requests: a) predefined service request such as MAC, IP, CTAG/STAG based; and b) user-defined service request in which a user can specify any protocol. A service request contains the service parameters provided by the user at the time of request such as the source/destination address, protocol, bandwidth requirement, QoS and additional service requirements etc. The service manager interprets the

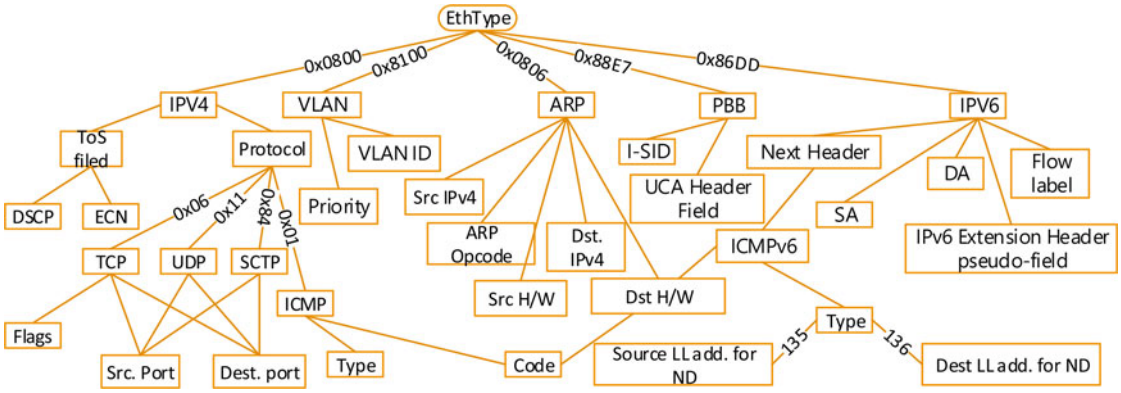

Fig. 7. Parse tree for popular existing protocols.

service requests using the parse tree in conjunction with the available service templates in the repository (stored as YANG models), and extracts the service parameters required for configuration. Once all the required parameters are available, the service manager attempts to find edge-and-node-disjoint routes for the primary and protection paths. Assuming that such routes exist, the service manager prepares the NFGs for both the paths. These NFGs along with the extra bits for the additional protocol processing are conjoined together to obtain the bitstream snippet (BS). Once all the service parameters along with the BS are available for configuration, this information is passed to the Network Manager module for the hardware configuration.

For example, let us assume that the service manager receives a service request that requires forwarding based on a TCP flag value. In this case, the service manager uses the parsing tree to extract the service parameters (i.e., protocol identifiers and their respective OLI) by traversing the tree from its root. Service templates are used to map these identifiers and service parameters to create this service. A parse tree representing the popular existing protocols is shown in Fig. 7. The service manager traverses the following path in Fig. 7: Ethertype $\rightarrow$ IPv4 $\rightarrow$ IP Protocol $\rightarrow$ TCP $\rightarrow$ TCP-flag. A service corresponding to this path requires that the packet has an Ethertype of 0×0800, a specific IP address with the protocol field in the IP header to be 0×06 and a specified TCP flag. The bitstream hardware should match all these fields for the specified service request. The required OLI and other service parameters are provided to the network manager for writing the service configuration in the match-table as shown in Fig. 8.

D. Network Telemetry

This module monitors and gathers performance metrics from the closed domain. For service monitoring and fault detection, this module creates MEPs at service ingress and egress nodes. The MEPs periodically exchange connectivity check messages (CCM). Loss of three consecutive CCMs results in an alarm and switchover from the work to the protection path. Subsequently, an alert is sent to the GUI for updating the palette and database. A user can trigger real-time performance monitoring (i.e., latency, jitter, throughput and packet-loss) for any specific service through the bitstream GUI. This module also gathers node statistics by sending the *monitor* messages periodically to all the nodes in the closed domain. After receiving a *monitor* message, the node replies with the *monitor reply* message that contains statistics related to the service such as packet-count, dropped packets, latency, throughput, etc. This module maintains a repository of such statistics and events for analysis.

E. Bitstream GUI

The Bitstream GUI interacts with all other modules through the bitstream handler and displays the required information to the user. A user gets the complete view of the closed domain topology in the GUI. The GUI is shown in the evaluation section (Section VIII) and is presented in Fig. 15. The GUI facilitates a user to select a service request either by using a set of predefined protocol-based services (i.e., MAC, IP etc.) or by selecting any new protocol-based service (given that the service template for a new service is defined and available to the service manager). The bitstream GUI also allows a user to define policy rules such as routing decisions, resource allocation etc. that must be followed during service configuration. The bitstream GUI also interacts with the Network Telemetry module to get the near real-time statistics of the network.

F. Service Templates

This module keeps the repository of all the service and network model templates. These templates are defined using YANG models [12], [13]. Any new service or new protocol is first defined and added in this module.

YANG model example: Appendix A shows a YANG model for an IPv6 (point-to-point) leased-line service. The model initially defines MAC, IPv4 and IPv6 addresses types. After defining the required protocols, the client address, (which can be of type MAC, IPv4 or IPv6 is defined) followed by the uniqueness of the bitstream closed domain – NFGs are defined. The NFG definition is important since primary and protection paths for the service are concatenated from the NFGs along the path. After defining the required parameters, the actual service definition is presented in the service container. The service container has an address container to store the source and destination client addresses (expressed as source and destination leaves). The service container also includes a statistics container that depicts the

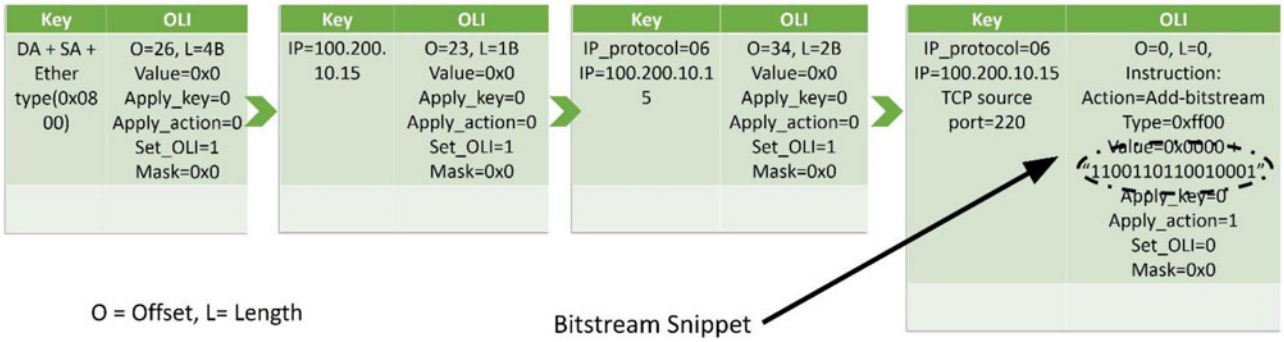

Fig. 8. Multi-stage Match action in table.

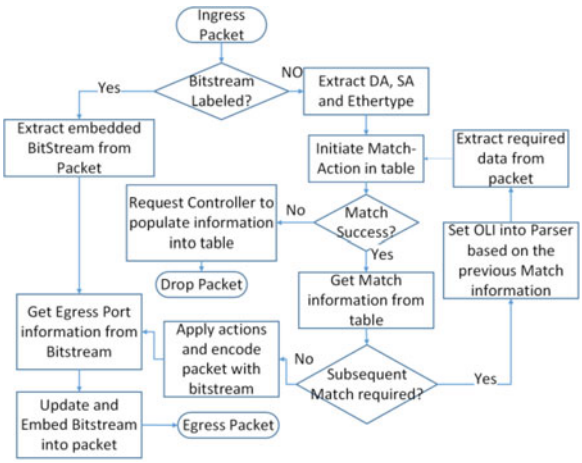

Fig. 9. Packet flow chart.

parameters supplied by the user in order to configure the service. These parameters are defined by: bandwidth, CBS and the QoS leaves. The service container also stores the path information in the container path. The path container includes primary and protection path containers that consist of the leaf-list of the NFGs.

VI. Packet Processing and Illustrations

In Fig. 9 is a flowchart for packet processing followed by the bitstream hardware. An incoming packet is checked if it is marked or unmarked with a BS. A marked packet is directly forwarded to the bitstream engine for further processing. This direct forwarding to a bitstream engine benefits the packet to achieve low-latency. If a packet is unmarked, it is processed by the MPMA module and matched sequentially against multiple tables resulting in the generation of a BS. Although sequential matching adds to the delay at the ingress node, this is a one-time effort and subsequently BS based forwarding results in overall lower latency.

Appendix B shows detailed treatment in the bitstream framework of all protocols that OpenFlow can support. We now consider two examples for protocol processing.

Illustration-1: Consider as an example, L4 processing in the bitstream hardware. We assume the parser of the 1st stage is programmed to extract `Destination MAC (DMAC)`, `Source MAC (SMAC)` and `Ethertype` and assume that there exists a table entry for the `Ethertype = 0x0800`, `IP = 100.200.10.15`, `TCP_source port = 220` (Fig. 8). In the first stage, `Ethertype 0x0800` is matched and OLI corresponding to IPv4 address is obtained. In the second stage, the extracted IPv4 address (`100.200.10.15`) is matched and OLI for IPv4 protocol type is obtained. In the third stage, IPv4 address along with protocol type (`0x06`) is matched and an OLI for TCP source port is obtained. In the fourth stage the IPv4 address along with IPv4 protocol type and TCP source port are matched and finally the BS is obtained. After obtaining the bitstream snippet, all the set of instructions along with BS are forwarded as metadata to the OAAM. The OAAM applies this set of instructions on the packet and adds the bitstream snippet to the packet. The packet is marked by setting custom `Ethertype = 0xFF00`. This marked packet is then forwarded to the BE. The BE identifies the egress port using the embedded pointer and bitstream information from the packet. Note that any forwarding rule or ACL processing (table-lookup, TTL, swap, etc.) happens only once during the packets journey through a bitstream network, and thereafter at other hops the packet only undergoes NFG based forwarding (this limitation of processing only once can easily be relaxed in future hardware versions).

Illustration-2: Assume that the VXLAN protocol is to be added to the existing SBI protocol suite. To this end, the controller maps ingress/egress MAC addresses to the *VXLAN tunnel endpoint* (VTEP) and writes the rules in the table with appropriate OLI values. The controller programs the parser at the ingress node to extract the MAC address from the packet. The extracted MAC address is matched in the table and an OLI is received with add instructions to add the outer L2 header information at an offset = 0x00 in the packet. In the second stage, the outer MAC is used as a key to match against the table and the next OLI is received with add instruction to add the outer IP header information at an offset = 0x0d, indicating the 14th byte position in the packet. In a similar way, a UDP header and the VXLAN header are also added. At the end, the bitstream header is added in the packet. This whole process is done only at the ingress node. It can be concluded that the bitstream hardware is oblivious to a protocol. The controller simply manipulates the table with corresponding identifiers resulting in forwarding and specific processing.

TABLE II
PARAMETERS AND DECISION VARIABLES

$G(V,E)$	Network graph of set of V nodes and set of E edges
C_j	Bandwidth capacity of node $j: j \in V$
D_j	Processing capacity of node $j: j \in V$
P	Set of protocols $\{P_1, P_2, \ldots, P_y\}$
$\overline{P}$	Set of protocols $\{P_x, P_{x1}, \ldots, P_y\}$, where $x < y$
$T_{abkm}^{p_n}$	m^{th} instance of traffic request on the k^{th} path between node a and b for protocol $p_n \in P$
PM_{ab}^{k}	Set of nodes on k^{th} path between node a and b
d_n	Processing delay of protocol $p_n \in P$
$\theta_{abkm}^{p_n}$	$\begin{cases} 1, & \text{if } T_{abkm}^{p_n} \text{is provisioned,} \\ 0, & \text{otherwise} \end{cases}$
$\alpha_{abkm}^{p_n j}$	$\begin{cases} 1, & \text{if } T_{abkm}^{p_n} \text{is provisioned and } p_n \text{ is processed at node j,} \\ 0, & \text{otherwise} \end{cases}$
β_{ab}^{k}	$\begin{cases} 1, & \text{if path } PM_{ab}^{k} \text{is chosen,} \\ 0, & \text{otherwise.} \end{cases}$

VII. OPTIMIZATION MODEL AND CLASSIFYING PERFORMANCE

An important question of consideration is to compute which protocols should be processed at which nodes and how to distribute such logic for a particular network and given traffic. Our goal is to process protocols that require processing that is beyond just forwarding (such as TTL should be done at the end of a path). The processing may have to be done once or multiple times for a particular service, and it may have an order of precedence (swap followed by decrement TTL). Essentially this means that there is code developed to process a protocol in the `action` section of a BS-enabled device, but since there are a large number of protocols, we want to compute where to place such code in a network. From a system design perspective, the challenge is that the processing unit size and memory in a node are also constrained. Hence based on the network topology, work and protection requirements and a given traffic profile, we must distribute processing code across the network such that service parameters (latency) are satisfied. This leads to a constrained optimization model. The goal of the model is to minimize overall delay across all the services in the network. For the optimal placement of processing code that is to be distributed across a closed domain, we assume that we are given a network, a set of y protocols, bandwidth and processing capacities at nodes and the observed latency for processing a particular protocol. Optimization parameters are in Table II.

Objective function: Our objective is to minimize the overall delay across all the services in the closed domain.

$$\min\left[\Sigma_{p_n}\Sigma_{a,b,k,m}\theta_{abkm}^{p_n}\cdot d_n\right]$$

Subject to the following constraints:

Traffic provisioning constraint: The following constraint requires that every traffic request that we consider as part of the optimization model is provisioned.

$$\theta_{abkm}^{p_n}.T_{abkm}^{p_n} \geq 0,\ \forall a,b,k,m,p_n$$

Capacity constraint: Each BS supporting device has fixed bandwidth for processing protocols and fixed processing power. The below constraint states that the sum of all bandwidth allocated to traffic at a node for processing as well as the sum of all processing power must be less than the capacity of the node.

$$\Sigma T_{abkm}^{p_n}.\alpha_{abkm}^{p_n j} \leq C_j\ \forall a,b,k,m,p_n,j$$

$$\Sigma T_{abkm}^{p_n}.\alpha_{abkm}^{p_n j} \leq D_j\ \forall a,b,k,m,p_n,j$$

Delay constraint: The traffic should be provisioned in a way that the individual traffic delay guarantees are met. Hence,

$$\Sigma_{p_n\in P}\Sigma_{j\in PM_{ab}^{k}}\alpha_{abkm}^{p_n j}\cdot d_n \leq \Delta_n, \forall a,b,k,m$$

Protocol processing constraint: The below constraint guarantees that each traffic request that is provisioned is always assigned to at least one node that processes the traffic request. Hence, we have,

$$\Sigma_{\substack{\forall a,b,k,m \\ j\in PM_{ab}^{k}}}\alpha_{abkm}^{p_n j} \geq 1,\ \forall p_n \in P$$

$$\Sigma_{j\in PM_{ab}^{k}}\alpha_{abkm}^{p_n j} + \beta_{ab}^{k} \geq 2,\ \forall a,b,k,m,p_n$$

The second equation above guarantees that processing for each traffic request occurs on the selected path.

Protocol precedence:

$$\arg_j \alpha_{abkm}^{p_n j} < \arg_j \alpha_{abkm}^{p_n j'}, \forall j,j' \in PM_{ab}^{k} \forall a,b,k,m,p_n$$

The above constraint facilitates preordering of protocols that should be processed per service.

Evaluation: The above constrained optimization model was developed as an integer linear program in Matlab using the linprog module and solved over a 30-node network (typical metro environment). The topology is a randomly generated mesh network with average degree of connectivity 3.4. Each node was assumed to be a 400 Gbps cross-connect with 2 ms of buffers per 10 Gbps port and 4x2.5 GHz duo processors that provided for the node `action` section, in addition to 8 Gbit of RAM. Two types of protocols were assumed: a group of 30 base protocols (those that required mere forwarding of packets based on some protocol identifier – such as IP/MAC/port etc.) and a group of 12-processing protocols (those that required processing beyond forwarding of packets). Flows were randomly generated in integer values in increments of 10 Mbps from 10 Mbps to 10 Gbps. Load was computed as the ratio aggregate total traffic in the network divided by average hop count to the total maximum traffic possible in the network.

Shown in Fig. 10 is the delay profile of the optimization model for various number of protocols that are processed (on average) at a node from the processing group. In this case,

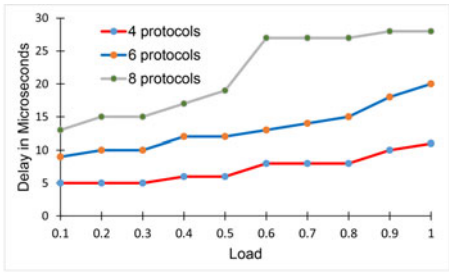

Fig. 10. Delay vs load as a function of number of protocols processed at a node.

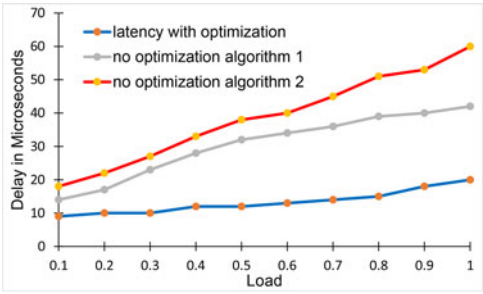

Fig. 11. Comparison of the optimization approach with two efficient algorithms.

on average it takes 46 cycles of duration 8-ns and standard deviation of 4 cycles for processing of packets from the group. The counterintuitive observation is that the system balances itself out – beyond a certain number of protocols, there is not much change in latency. It can hence be said that there for a particular switch size with given processing capacity, and given network topology, there is a maximum number of protocols that can be processed by the network. This is an important result as it shows the limitations of routing and need for load balancing.

Shown in Fig. 11 is the comparison of the optimization model with two efficient and biased algorithms. The first algorithm (#1) is a planning and placement algorithm that uses a branch and bound technique to distribute processing modules among nodes in the network. The algorithm initially gives equal processing capability at every node in the network for each protocol. Subsequently, the algorithm reduces or enhances the processing capabilities on a per-node/per protocol basis till it reaches 'its' own best possible solution – this is not the optimal. It stores the recently achieved best solution and then continues to find another solution. If another better solution is found, then that is replaced with the best solution. The number of times this process is carried out is $\log(T^n_{abkm})$ (a practical bound). The second algorithm (#2) is a random fit (with determined sizing). It randomly assigns processing capabilities to nodes initially and then dimensions these processing capabilities on the intensity of traffic request through the node. We observe in Fig. 11 that the random fit and the branch-and-bound technique perform somewhat similar – which is not expected. The optimization results give us the lowest bound. The optimization technique is NP-hard – can be reduced to multi-dimensional bin-packing [34]. For low-loads the random algorithm with determined sizing performs similar to the branch-and-bound, and its complexity is low (of the order of $O(T^n_{abkm})$ for V_a). The other key take-away is that the theoretical bound (expressed by the optimization technique) is almost flat. We have verified this for load between 0.6-0.8 for a large ~400 node network.

Classifying performance: We now present a short analytical model that compares latency in the bitstream network to a conventional openflow network. For this comparison, we define the variables in Table III:

Then, for an h node path, the average delay (excluding propagation delay) for the bitstream network is:

$$\Delta_{\mathrm{B}} = \delta_t^{BS} + \delta_{en}^{BS} + (h-2)\cdot\left(\delta_{fwd}^{BS}\right) + \delta_{process}^{BS}.$$

TABLE III
ANALYTICAL VARIABLES

δ_t^{BS}	Avg. delay for a table match at an ingress node in the bitstream scheme
δ_{en}^{BS}	Avg. time required to encode a packet with a bitstream snippet
$\delta_{process}^{BS}$	Time required to process a packet when pure forwarding is not to be followed (i.e. for swapping, decrementing TTL, etc.)
δ_{fwd}^{BS}	Time required for pure forwarding in bitstream through a switch
δ_{conv}	Time required for conventional routing/forwarding.

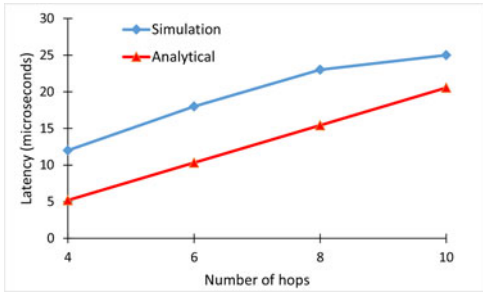

Fig. 12. Simulation and analytical comparison.

The corresponding delay for forwarding in an SDN scheme is: $\Delta_{\mathrm{conv}} = (h-1).\delta_{conv}$.

We argue that $\delta_{conv} \approx \delta_t^{BS} + \delta_{en}^{BS} + (h-2).\delta_{fwd}^{BS} + \delta_{process}^{B}$ as $\delta_{process}^{BS}$ is negligible on two counts: (a) it is not as much invoked as δ_{fwd}^{BS} and (b) its value by itself is in nanoseconds as it is implemented purely in the hardware `action` section. Further, δ_{fwd}^{BS} is by definition in the nanosecond range as well (as we have to simply act upon a few bits. This leads us to $\Delta_{\mathrm{conv}} \approx (h-2)\Delta_{\mathrm{B}}$.

Shown in Fig. 12 is the comparison of the analytical result developed above and simulation developed in the next section. The analytical and simulation results converge for larger sized networks, though for more practical metro type networks, there is a 30% average error due to dynamic traffic and variation in packet size which is not considered in the analytical model. The simulation model assumes the same parameters for network, load and traffic as shown in the optimization evaluation.

VIII. PROTOTYPE AND SIMULATION

A. Prototype and Evaluation

We built a bitstream capable 400 Gbps hardware prototype as shown in Fig. 13 (top) using a 20-layer fabricated PCB that encompasses two Xilinx Virtex-7 690T-2 FPGAs along with peripheral memories (QDR, TCAM) and a Framer/Mapper chip. The bitstream protocol is coded in VHDL. We also developed a JAVA-based SDN controller to manage the bitstream network. The reason we had to develop our own controller was because of the large spectrum of protocols supported by bitstream hardware – more than any other off-the-shelf controller. The controller can be connected to other controllers using a REST API.

The hardware block diagram is shown in Fig. 13 (bottom). At the heart of the hardware are two Xilinx Virtex 7 690T FP-

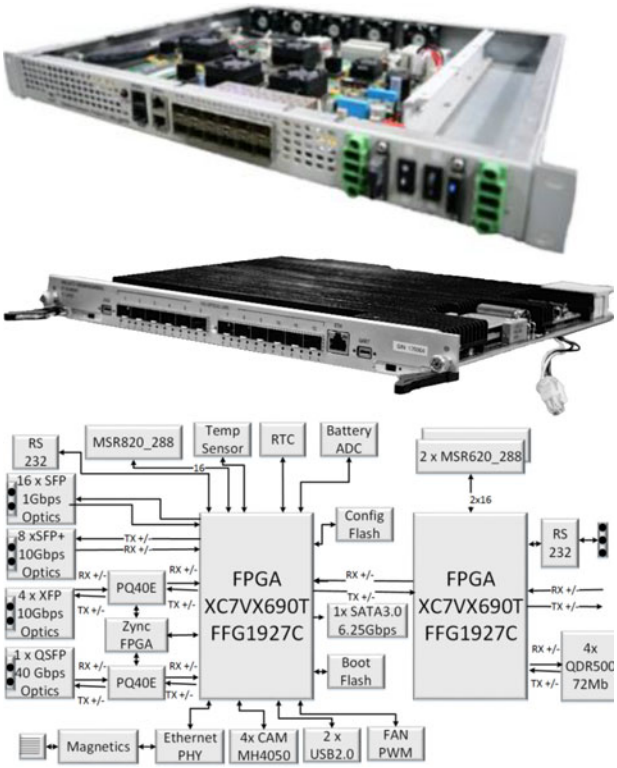

Fig. 13. Bitstream hardware prototype (two variants) (Top) and PCB Schematic and block diagram (Bottom).

GAs. The first FPGA is connected to all the IO ports, while the second FPGA is connected to a passive backplane for scalability. In each card, the first FPGA is used for parsing and local switching, while the second FPGA is used for action, and further switching. Local memories in the form of block RAMs are instantiated as VOQs in each FPGA. The switch-fabric is developed by cascading multiplexers in the FPGAs. The FPGAs are connected to 4 TCAMs. Each TCAM is 576 Mbit in size. TCAMs store flow table entries written by the SDN controller through a VHDL coded CSM. Packets are stored in both on-chip (Block RAMs) and off-chip large bandwidth QDR memories. The first FPGA is connected to an on-board 32-bit 1 GHz (Arm Cortex 9) processor. Each card has 16 $\times$ 1 Gbps, 8 $\times$ 10 Gbps IOs (SFP+), 4 $\times$ 10 Gbps long-reach capable XFP optics and a 40 Gbps port in addition to a fully non-blocking supporting backplane port that is capable of transmitting 352 Gbps in the backplane. The board also houses PQ40E chips SERDES for communication with the 10 Gbps and 40 Gbps ports.

For the purpose of testing, a single card test-bed is setup as shown in Fig. 14. (left). A Viavi ONT-506 [35] series tester is used for packet generation and reception. Performance of the hardware is evaluated based on the number of clock cycles (each of 6.4 nanoseconds) required to complete an instruction. Performance of different instructions are shown in Table IV. Clock cycles taken by an instruction is dependent on the location of the respective fields in the packet. Each IO port receives 8-bytes of packet in each clock-cycle. To reach an offset of the packet, it takes $\lceil offset/8 \rceil$ number of cycles.

An example of creating a managed layer-2 point-to-point service through the Bitstream GUI is shown in Fig. 15. A MAC

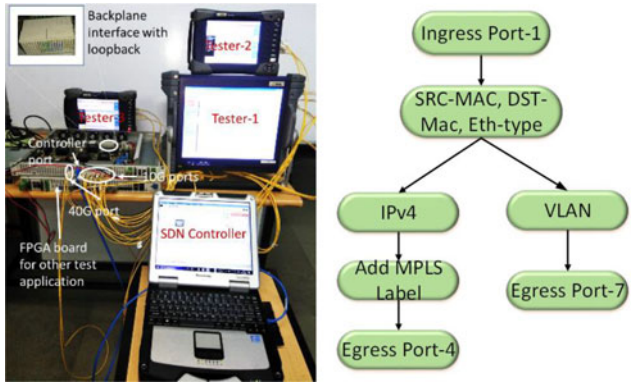

Fig. 14. Experimental setup (left), Service Parse graph (right).

TABLE IV
OPERATIONAL PERFORMANCE

Instruction	# of cycles
Set field	4+offset/8
Mod field	4+offset/8
Write	24
add field	4+offset/8
Delete field	6+offset/8
output	42+Len/2
Offset is the field location in the packet, Len: Length of the Packet in multiple of 8 Bytes.	

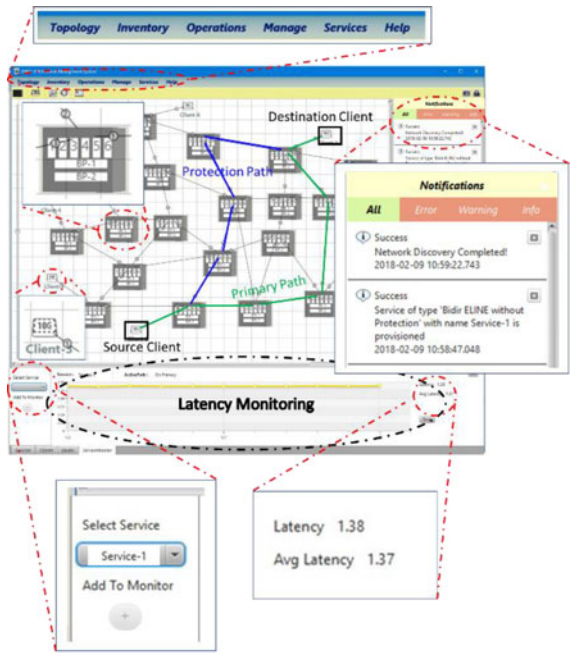

Fig. 15. GUI for BS controller.

based service is requested between client-1 and client-2 (shown as source and destination client nodes in the GUI palette). The controller configures the requested service and displays the provisioned paths in the GUI. The primary path is shown in green and the protection path is shown in blue. A real-time latency monitoring for the provisioned service is displayed at the bottom of the Bitstream GUI (black dashed circle).

For evaluation, two flows are provisioned. One to add an MPLS label in an IPv4 packet and another flow is set for a

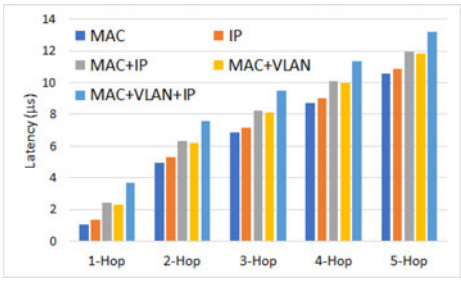

Fig. 16. Experimental measurement of latency as a function of hop-count for a 1 Gbps service.

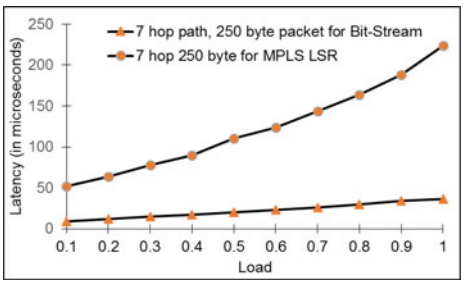

Fig. 17. Latency as a function of load for BS and MPLS LSR.

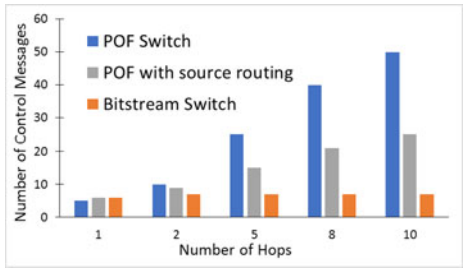

Fig. 18. Effect of hop length on the control packets.

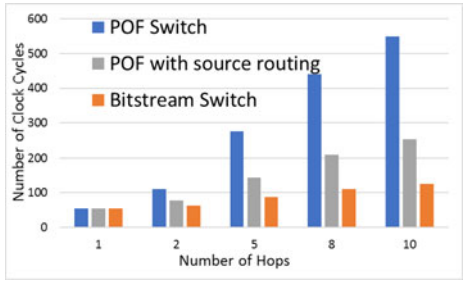

Fig. 19. Latency comparison with different number of hops.

VLAN tagged packet and forwards both flows to specific port. Service parse graph for two flows are shown in Fig. 14 (right). The parser extracts the source address, destination address and Ethertype from an incoming packet and matches these in the flow table. If the Ethertype matches to an IPv4 packet, an MPLS label is added into the packet along with the appropriate BS. If the Ethertype matches for a VLAN tagged packet then only the BS is added in the packet. We measured a throughput of 14.2 MPPS where average packet-size is 80B and latency of 1.7 μs for only the MPLS flow.

Shown in Fig. 16 is the measured latency for the services configured on the bitstream hardware using different combination of protocols as a function of number of hops. The goal of this experiment is to show stability in the performance of the bitstream hardware as we increase the hop-count as well as use different protocols. The goal of this experiment is to show that the delay across multiple hops is deterministic (linear). It can be observed that in case of a single hop with increase in the protocol stack (number of protocols to be processed), the resulting latency increases linearly with the depth of the protocol stack. This is due to the number of successive lookups in the match-table at the ingress node. With an increase in the number of hops, there is a linear increase in latency but the gradient is significantly lower now due to the source routing feature of the bitstream hardware, which avoids any match-table lookups at the nodes other than the ingress node. This result also supports our analytical model presented in Section VII. Note that as compared to Figs. 10–12 the latency values in Figs. 16–19 are lower because of actual measurement. In the simulations we are consistently rounding-off values to the nearest 10-microseconds to conserve processing power required for the simulation. The decrease in latency for MAC+VLAN service as compared to MAC+IP service is due to the added latency by the parser in extracting the required IP field from the packet. As we go for higher-layer protocol extraction, the parser needs to process more number of bytes for extracting the required protocol identifier. For example, to extract a VLAN identifier, the parser needs to process only the initial 16 bytes of data, as the VLAN is at an offset of 12 bytes, i.e., at byte location # 13-16 in the packet. In contrast, for extracting the IP protocol identifier, the parser needs to process up to the 36th byte (byte location #17-36), as the IP header is at an offset of 16 bytes and of length 20 bytes. Hence the time required for MAC+IP is more than the time required for MAC+VLAN.

Shown in Fig. 17 are the observed latency values for the protocol as compared to an MPLS LSR. We built a 7-hop path for our prototype, as well as for a commercially available MPLS LSR (which was configured as a daisy-chain across multiple ports). To build the 7-hop path, we connected ports of the prototype to one-another in a daisy-chain fashion. Packets of 250-byte average size were injected and QoS level was set to 3. For the Bitstream prototype, a VHDL application was developed to measure latency. In case of the commercial LSR, the LSR itself recorded latency. Load was computed as utilization of the data-path. The maximum data-path load was 9.9 Gbps. As we see, the average latency in the bitstream case for a 7-hop path is approximately 1/6th the latency of the MPLS case. More importantly, due to the carrier-class nature of bitstream, the latency profile is deterministic, whereas it is non-linear for higher-loads in the MPLS case.

B. Comparison of Bitstream With POF

We simulated a POF switch and a bitstream hardware using Modelsim SE 10.4 for control and datapath, assuming similar capabilities and considering that the parser and match-action of both takes the same amount of time. POF as opposed to OF is chosen as a comparative protocol due to its performance betterment (scalability) than OF. Further, we determine control

traffic as a key measure of how an SDN would work from scalability perspectives.

We set a flow similar to Illustration-1 (Section VI) on POF and bitstream switches, which requires multiple stages for packet parsing and match-actions. We evaluated the number of control packets exchanged between the controller and the switch for flow setup as shown in Fig. 18. For the POF switch, the number of control packets continues to increase with hop length. This increase in the control packet traffic is due to the fact that each node between the source and destination needs to be configured. As a result, control packets get added at each hop. Bitstream uses source routing for packet forwarding and therefore, bitstream switches require configuration only at the ingress switches of a closed domain. As a result, there is no effect of increasing number of hops on control packets. Though a new version of POF can also support source routing [36] by embedding the routing information into a packet with add field action-set, the control traffic reduces only slightly, and is always significantly more compared to bitstream.

We also compare the ingress-to-egress latency in terms of number of clock-cycles required by a flow as shown in Fig. 19. In case of POF, all switches are configured with flow entries and an ingress packet needs to be parsed and matched at multiple stages of each switch. These multiple stages of parsing and matching keeps adding in clock-cycle count at each switch. It is observed that for a single node, the clock-cycle count is same for POF and Bitstream (Fig. 19). However, for POF switches (with and without source routing) there is a non-linear increase in clock-cycle count as the number of hops increases. For bitstream, there is a minimal increase in clock-cycle count due to processing of $2(\log_2(k) - 1) + n$ bits at the intermediate nodes making the protocol ideal for latency sensitive carrier-class applications.

C. Achieving Service Scalability

OpenFlow 1.5 supports about 40 types of match identifiers on a controller's SBI. These identifiers are matched to tables within an SDN whitebox. In the bitstream concept, we support all of the 40-odd match identifiers and also allow the user to define any new protocol match identifier *without any change required in the hardware*. We show compliance of bitstream with the match identifiers supported by OpenFlow 1.5 in Appendix-B, while examples of new protocol support are already discussed in Section VI.

IX. Summary and Future Work

We present the bitstream scheme as a way to further open SDN towards data-plane programmability. The bitstream scheme uses source routing as an addition to facilitate scalability to be brought into the SDN domain. The scheme facilitates carrier-class communication that is quintessential for SDN adoption in provider networks. A hardware prototype and a controller that supports YANG modeling are developed that can allow implementation in large networks. We show lower and importantly network-topology agnostic latency using bitstream. A comparison of bitstream and another leading SDN protocol POF (which in a limited manner betters OF) is also presented showcasing reduction in control traffic and carrier-class performance. A comparative list of OF and bitstream is developed. The test-bed presents a 400 Gbps white-box and in future it is planned to scale to a terabit whitebox.

Appendix A
YANG Model Example

```
module Bitstream-example{
        namespace "http://Bitstream.example.com/";
        prefix "Bitstream";
        import ietf-yang-types {
           prefix "yang";
        }
        include acme-types;
        organization "IIT Bombay "
        contact
           "IIT Bombay ";
        description
           "For Bitstream "
        revision "2017-10-16" {
           description "Initial revision.";
        }
typedef mac_address {
    type string {
     pattern '[0-9a-fA-F]{2}(:[0-9a-fA-F]{2}){5}';
    }
    description "MAC address ";
}
typedef ipv4_address {
    type string {
     pattern
      '(([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])\.){3}'
     + '([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])'
     + '(%[\p{N}\p{L}]+)?';
    }
    description "IPV4";
  }
typedef ipv6_address {
    type string {
     pattern '((:|[0-9a-fA-F]{0,4}):)([0-9a-fA-F]{0,4}:){0,5}'
          + '((([0-9a-fA-F]{0,4}:)?(:|[0-9a-fA-F]{0,4}))|'
          + '(((25[0-5]|2[0-4][0-9]|[01]?[0-9]?[0-9])\.){3}'
          + '(25[0-5]|2[0-4][0-9]|[01]?[0-9]?[0-9])))'
          + '(%[\p{N}\p{L}]+)?';
     pattern '(([^:]+:){6}(([^:]+:[^:]+)|(.*\..*)))|'
          + '((([^:]+:)*[^:]+)?::(([^:]+:)*[^:]+)?)'
          + '(%.+)?';
    }    description  "IPV6";
}
typedef client_address{
   type union {
      type mac_address;
      type ipv4_address;
      type ipv6_address;
   }
}
typedef NetworkFowordingGraph {
   type string{
      pattern"[0-1]*";
   }  description "NFG for a node";
}
container Service{
   description "Service "
   container ELINE {
      container address{
         leaf source{
            type client_address
         }
         leaf destination{
            type client_address
         }
      }
      container Stats{
         leaf bandwidth{
            type uint16;
            description "bandwidth of service ";
         }
         leaf CBS{
            type uint16;
            description "CBS level of service ";
         }
         leaf QOS{
            type QOS_Level;
            description "Quality of service ";
         }
      }
         container Path{
         container primary_path
            leaf-list NFG  {
               type NetworkFowordingGraph ;
               description "A list of  NFG for nodes";
             }
         container protection_path
            leaf-list NFG  {
               type NetworkFowordingGraph ;
               description "A list of  NFG for nodes along the service path ";
             }
      }
   }
```

APPENDIX B
BITSTREAM AND OF MATCH FIELD COMPLIANCE

OF 1.5.1 Match Field Type	Bitstream matching identifier
OFPXMT_OFB_IN_PORT	Ingress logical port
OFPXMT_OFB_IN_PHY_PORT	Ingress physical port
OFPXMT_OFB_METADATA	Metadata are processed and matched based on table flag
OFPXMT_OFB_ETH_DST	Extracted in stage-1 parser (extraction is programmable by controller)
OFPXMT_OFB_ETH_SRC	Extracted in stage-1 parser (extraction is programmable by controller)
OFPXMT_OFB_ETH_TYPE	Extracted in stage-1 parser (extraction is programmable by controller)
OFPXMT_OFB_VLAN_VID	Extracted by parser programmed by SDN or in later stages based on received OLI value against eth_type=0x8100
OFPXMT_OFB_VLAN_PCP	Extracted by parser in 2nd or later stage based on OLI value received against eth_type=0x8100
OFPXMT_OFB_IP_DSCP	Extracted by parser in 2nd or later stage based on OLI value received against eth_type=0x0800
OFPXMT_OFB_IP_ECN	Extracted by parser in 2nd or later stage based on OLI value received against eth_type=0x0800
OFPXMT_OFB_IP_PROTO	Extracted by parser in 2nd or later stage based on OLI value received against eth_type=0x0800 or 0x86dd
OFPXMT_OFB_IPV4_SRC	Extracted by parser programmed by SDN or in later stages based on received OLI value against eth_type=0x0800
OFPXMT_OFB_IPV4_DST	Extracted by parser programmed by SDN or in later stages based on received OLI value against eth_type=0x0800
OFPXMT_OFB_TCP_SRC	Extracted by parser in 2nd or later stage based on OLI value received against eth_type=0x0800, IP_protocol=0x06
OFPXMT_OFB_TCP_DST	Extracted by parser in 2nd or later stage based on OLI value received against eth_type=0x0800, IP_protocol=0x06
OFPXMT_OFB_UDP_SRC	Extracted by parser in 2nd or later stage based on OLI value received against eth_type=0x0800, IP_protocol=0x11
OFPXMT_OFB_UDP_DST	Extracted by parser in 2nd or later stage based on OLI value received against eth_type=0x0800, IP_protocol=0x11
OFPXMT_OFB_SCTP_SRC	Extracted by parser in 2nd or later stage based on OLI value received against eth_type=0x0800, IP_protocol=0x84
OFPXMT_OFB_SCTP_DST	Extracted by parser in 2nd or later stage based on OLI value received against eth_type=0x0800, IP_protocol=0x84
OFPXMT_OFB_ICMPV4_TYPE	Extracted by parser in 2nd or later stage based on OLI value received against eth_type=0x0800, IP_protocol=0x01
OFPXMT_OFB_ICMPV4_CODE	Extracted by parser in 2nd or later stage based on OLI value received against eth_type=0x0800, IP_protocol=0x01
OFPXMT_OFB_ARP_OP	Extracted by parser in 2nd or later stage based on OLI value received against eth_type=0x0806
OFPXMT_OFB_ARP_SPA	Extracted by parser in 2nd or later stage based on OLI value received against eth_type=0x0806
OFPXMT_OFB_ARP_TPA	Extracted by parser in 2nd or later stage based on OLI value received against eth_type=0x0806
OFPXMT_OFB_ARP_SHA	Extracted by parser in 2nd or later stage based on OLI value received against eth_type=0x0806
OFPXMT_OFB_ARP_THA	Extracted by parser in 2nd or later stage based on OLI value received against eth_type=0x0806
OFPXMT_OFB_IPV6_SRC	Extracted by parser programmed by SDN or in later stages based on received OLI value against eth_type=0x86dd
OFPXMT_OFB_IPV6_DST	Extracted by parser programmed by SDN or in later stages based on received OLI value against eth_type=0x86dd
OFPXMT_OFB_IPV6_FLABEL	Extracted by parser in 2nd or later stage based on OLI value received against eth_type=0x86dd
OFPXMT_OFB_ICMPV6_TYPE	Extracted by parser in 2nd or later stage based on OLI value received against eth_type=0x86dd, Type= 58
OFPXMT_OFB_ICMPV6_CODE	Extracted by parser in 2nd or later stage based on OLI value received against eth_type=0x86dd, Type= 58
OFPXMT_OFB_IPV6_ND_TARGET	Extracted by parser in 2nd or later stage based on OLI value received against eth_type=0x86dd, Type= 0x87 or 0x88
OFPXMT_OFB_IPV6_ND_SLL	Extracted by parser in 2nd or later stage based on OLI value received against eth_type=0x86dd, Type= 0x87
OFPXMT_OFB_IPV6_ND_TLL	Extracted by parser in 2nd or later stage based on OLI value received against eth_type=0x86dd, Type= 0x88
OFPXMT_OFB_MPLS_LABEL	Extracted by parser in 2nd or later stage based on OLI value received against eth_type=0x8847 or 0x8848
OFPXMT_OFB_MPLS_TC	Extracted by parser in 2nd or later stage based on OLI value received against eth_type=0x8847 or 0x8848
OFPXMT_OFP_MPLS_BOS	Extracted by parser in 2nd or later stage based on OLI value received against eth_type=0x8847 or 0x8848
OFPXMT_OFB_PBB_ISID	Extracted by parser in 2nd or later stage based on OLI value received against eth_type=0x88E7
OFPXMT_OFB_TUNNEL_ID	In form of bitstream snippet
OFPXMT_OFB_IPV6_EXTHDR	Extracted by parser in 2nd or later stage based on OLI value received against eth_type=0x86dd
OFPXMT_OFB_PBB_UCA	Extracted by parser in 2nd or later stage based on OLI value received against eth_type=0x88E7
OFPXMT_OFB_TCP_FLAGS	Extracted by parser in 2nd or later stage based on OLI value received against eth_type=0x0800 or0x86dd, IP_prot=0x06
OFPXMT_OFB_ACTSET_OUTPUT	Output port from bitstream snippet
OFPXMT_OFB_PACKET_TYPE	Packet ethertype

REFERENCES

[1] B. Naudts *et al.*, "Techno-economic analysis of software defined networking as architecture for the virtualization of a mobile network," in *Proc. Eur. Workshop Softw. Defined Netw.*, Oct. 2012, pp. 67–72.

[2] B. Nunes *et al.*, "A survey of software-defined networking: Past, present, and future of programmable networks," *IEEE Commun. Surveys Tuts.*, vol. 16, no. 3, pp. 1617–1634, Jul.–Sep. 2014.

[3] S. Jain *et al.*, "B4: Experience with a globally-deployed software defined WAN," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 3–14, 2013.

[4] ONF, "SDN in the campus environment," 2013. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/solution-briefs/sb-enterprise-campus.pdf

[5] A. Singh *et al.*, "Jupiter rising: A decade of clos topologies and centralized control in Google's datacenter network," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 183–197, 2015.

[6] A. Gumaste, S. Sharma, T. Das, and A. Kushwaha, "How Much NFV Should a Provider Adopt?," *J. Lightw. Technol.*, vol. 35, no. 13, pp. 2598–2611, Jul. 2017

[7] AT&T, "AT&T vision alignment challenge technology survey," Dallas, TX, USA, White Paper 2013, pp. 1–22.

[8] N. McKeown *et al.*, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp 69–74, 2008.

[9] Ciena Blue Planet, 2018. [Online]. Available: http://www.blueplanet.com/

[10] Juniper Contrail, 2018. [Online]. Available: http://www.juniper.net/us/en/products-services/sdn/contrail/

[11] M. Masse, *REST API Design Rulebook*. Sebastopol, CA, USA: O'Reilly, 2012.

[12] M. Bjorklund, "YANG - A data modeling language for the network configuration protocol (NETCONF)," RFC 6020, Oct. 2010.

[13] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman, "Network configuration protocol (NETCONF)," RFC 6241, Jun. 2011.

[14] J. Donovan and K. Prabhu, *Building the Network of the Future*. Boca Raton, FL,USA: CRC Press, 2017.

[15] L. Yang, R. Dantu, T. A. Anderson, and R. Gopal, "Forwarding and control element separation (ForCES) framework," IETF, Fremont, CA, USA, RFC 3746, Apr. 2004.

[16] A. Doria *et al.*, "Forwarding and control element separation (ForCES) protocol specification," IETF, Fremont, CA, USA, RFC 5810, Mar. 2010.

[17] M. Mahalingam *et al.*, "Virtual extensible local area network (VXLAN): A framework for overlaying virtualized layer 2 networks over layer 3 networks," No. RFC 7348, 2014.

[18] P. Garg and Y. Wang, "NVGRE: Network virtualization using generic routing encapsulation," No. RFC 7637, 2015.

[19] H. Song, "Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane," in *Proc. ACM SIGCOMM Workshop Hot Topics Softw. Defined Netw.*, Aug. 2013, pp. 127–132.

[20] A. Gumaste *et al.*, "Omnipresent ethernet—Technology choices for future end-to-end networking," *J. Lightw. Technol.*, vol. 28, no. 8, pp. 1261–1277, Apr. 2010.

[21] A. Abujoda, H. Kouchaksaraei, and P Papadimitriou, "SDN based source routing for scalable service chaining in data centers," in *Proc. Int. Conf. Wired/Wireless Internet Commun.*, Apr. 2016, pp. 66–77.

[22] M. Soliman *et al.*, "Exploring source routed forwarding in SDN-Based WANs," in *Proc. IEEE Int. Conf. Commun.*, Sydney, NSW, Australia, 2014, pp. 3070–3075.

[23] S. Bidkar *et al.*, "On the design, implementation, analysis, and prototyping of a 1-μs, energy-efficient, carrier-class optical-ethernet switch router," *J. Lightw. Technol.*, vol. 32, no. 17, pp. 3043–3060, Sep. 2014.

[24] C. Filsfils *et al.*, "Segment routing architecture," IETF Draft-Ietf-Spring-Segment-Routing-01, Cisco Systems, San Jose, CA, USA, Feb. 2015.

[25] L. Davoli *et al.*, "Traffic engineering with segment routing: SDN-based architectural design and open source implementation," in *Proc. Eur. Workshop Softw. Defined Netw.*, 2015, paper. 112.

[26] A. Sgambelluri *et al.*, "Experimental demonstration of segment routing," *J. Lightw. Technol.*, vol. 34, no. 1, pp. 205–212, Aug. 2015

[27] I. J. Wijnands *et al.*, "Multicast using bit index explicit replication," draft-wijnands-bier-architecture-00, Cisco Systems, San Jose, CA, USA, Sep. 2014.

[28] N. Kumar *et al.*, "BIER use cases, " draft-ietf-bier-use-cases-06, Cisco Systems, San Jose, CA, USA, Jan. 2018.
[29] A. Kushwaha, S. Sharma, N. Bazard, and A. Gumaste, "Bitstream: A flexible SDN protocol for service provider networks," in *Proc. IEEE Int. Conf. Commun.*, 2018, to be published.
[30] P. Bosshart *et al.*, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 99–110, Oct. 2013.
[31] M. Médard, S. G. Finn, R. A. Barry, and R. G. Gallager, "Redundant trees for preplanned recovery in arbitrary vertex-redundant or edge redundant graphs," *IEEE/ACM Trans. Netw.*, vol. 7, no. 5, pp. 641–652, Oct. 1999.
[32] C. A. S. Oliveira and M. P. Panos, "A survey of combinatorial optimization problems in multicast routing," *Comput. Oper. Res.*, vol. 32, no. 8, pp. 1953–1981, 2005.
[33] IEEE 802.1ag - Connectivity Fault Management, 2007. [Online]. Available: http://www.ieee802.org/1/pages/802.1ag.html
[34] V. Vazirani, *Approximation Algorithms*. New York, NY, USA: Springer, 2001, ch. 4.
[35] Viavi ONT 506, 2009. [Online]. Available: https://www.microlease.com/eu/products/viavi-formerly-jdsu-/sdh-sonet-ethernet/ont506?basemodelid=10898#overview
[36] S. Li *et al.*, "Protocol oblivious forwarding (POF): Software-defined networking with enhanced programmability," *IEEE Netw.*, vol. 31, no. 2, pp. 58–66, Mar./Apr. 2017.

Aniruddha Kushwaha received the Master's degree in advanced semiconductor electronics from AcSIR, Delhi, India, in 2012. He is currently working toward the Ph.D. degree in the Department of Computer Science and Engineering, Indian Institute of Technology (IIT) Bombay, Mumbai, India. He received the Google India Ph.D. Fellowship-2016. His research interests are in software defined networking, scalable datacenter networks and high speed optical networks.

Sidharth Sharma received the M.Tech. degree in computer science and engineering from the National Institute of Technology, Rourkela, India, in 2013. He is currently working toward the Ph.D. degree in the Department of Computer Science and Engineering, Indian Institute of Technology Bombay, Mumbai, India. His research interests include software-defined networks and network function virtualization.

Naveen Bazard received the Master's degree in computer science and engineering from Indian Institute of Technology, Delhi, India. He is currently working as a Senior Project Software Engineer in Gigabit Networking Laboratory, Indian Institute of Technology Bombay, Mumbai, India. His research interests include software-defined networks and network function virtualization.

Tamal Das received the B.Tech. and M.Tech. degrees from Indian Institute of Technology (IIT) Delhi, Delhi, India, and the Ph.D. degree from IIT Bombay, Mumbai, India. He is a Research Scientist at IIT. Prior to this, he was a Postdoctoral Researcher at TU Braunschweig, Germany. His research interests are in stochastic analysis, telecommunication networks, and network algorithms. He has authored more than 25 high-quality scientific publications, and he has received the IEEE ANTS 2010 Best Paper Award.

Ashwin Gumaste is currently an Associate Professor in the Department of Computer Science and Engineering, Indian Institute of Technology (IIT) Bombay, Mumbai, India. He was the Institute Chair Associate Professor (2012–2015) and the JR Isaac Chair Assistant Professor (2008–2011). From 2008–2010, he was a Visiting Scientist and Scholar at the Massachusetts Institute of Technology (MIT), Cambridge, MA, USA. He has held positions with Fujitsu Laboratories (USA) and has also worked with Cisco Systems. He was a consultant to Nokia Siemens Networks focusing on NGPON2. He has also held short-term positions at Comcast, Lawrence Berkeley National Labs, and Iowa State University. His work on light-trails has been widely referred, deployed, and recognized by both industry and academia. His recent work on Omnipresent Ethernet has been adopted by tier-1 service providers and also resulted in the largest ever acquisition between any IIT and the industry. This has led to a family of transport products under the premise of Carrier Ethernet Switch Routers. Recently he and his team have built a Terabit capable SDN whitebox that is under deployment. He has 24 granted US patents and has published about 175 papers in referred conferences and journals, inclusive of 3 best paper awards. He has also authored three books in broadband networks. For his contributions, he received the DST Swaranajayanti Fellowship in 2013, the Government of India's DAE-SRC Outstanding Research Investigator Award in 2010, the Vikram Sarabhai research award in 2012, the IBM Faculty award in 2012, the National Academy of Sciences in India, NASI-Reliance Industries Platinum Jubilee award 2016, as well as the Indian National Academy of Engineering's (INAE) Young Engineer Award (2010).