BPFabric: Data Plane Programmability for Software Defined Networks

Simon Jouet
School of Computing Science
University of Glasgow
Glasgow, G12 8QQ, Scotland
simon.jouet@glasgow.ac.uk

Dimitrios P. Pezaros
School of Computing Science
University of Glasgow
Glasgow, G12 8QQ, Scotland
dimitrios.pezaros@glasgow.ac.uk

ABSTRACT

In its current form, OpenFlow, the *de facto* implementation of SDN, separates the network's control and data planes allowing a central controller to alter the match-action pipeline using a limited set of fields and actions. To support new protocols, forwarding logic, telemetry, monitoring or even middlebox-like functions the currently available programmability in SDN is insufficient.

In this paper, we introduce *BPFabric*, a platform, protocol, and language-independent architecture to centrally program and monitor the data plane. BPFabric leverages eBPF, a platform and protocol independent instruction set to define the packet processing and forwarding functionality of the data plane. We introduce a control plane API that allows data plane functions to be deployed on-the-fly, reporting events of interest and exposing network internal state to the centralised controller.

We present a raw socket and DPDK implementation of the design, the former for large-scale experimentation using environment such as Mininet and the latter for high-performance low-latency deployments. We show through examples that functions unrealisable in OpenFlow can leverage this flexibility while achieving similar or better performance to today's static design.

CCS Concepts

•Networks → Programmable networks; In-network processing; Data center networks; Programming interfaces;

1. INTRODUCTION

Software-Defined Networking (SDN) has emerged over the last decade as a new paradigm to centralise the network's control plane and separate it from the underlying data plane. Through this physical separation and the resulting ability to programmatically control the devices, research in areas such as traffic routing and engineering, quality of service enforcement (QoS) and network virtualisation have evolved rapidly. OpenFlow has become the *de facto* realisation of SDN by providing an open-source and vendor-agnostic protocol allowing the control plane to manage hardware and software switches from different vendors [14].

OpenFlow has evolved significantly between its first release in 2008 and the current 1.5 specification released in December 2014. It evolved from a single match table with 10 match fields to multiple stage table matching and ingress and egress pipeline separation with 44 different match fields. The number of match fields have been continuously updated to add support for new fields, such as IPv6 addresses, MPLS and VLAN headers. The growth over time of supported match fields is unlikely to slow down as network requirements continuously evolve and support for upcoming and operator-specific protocols is required. This constant evolution of the OpenFlow specifications to cope with the demand for new match fields highlights its limitations regarding protocol support, match-actions and overall future-proofness. These limitations render seemingly simple data plane functions impractical or unfeasible.

In order for the network fabric to evolve in conjunction with the rest of the infrastructure, we argue for a network-specific platform, protocol and language-independent instruction set to express the per-switch data plane function. Each function should define how to parse the network protocols t0o perform traditional routing and forwarding as well as more complex tasks currently not possible in OpenFlow including lightweight middlebox-like functions such as load balancing, anomaly detection, encryption or protocol offloading. With the large variety of possible switches that are currently deployed, from traditional high-density switches, to programmable Network Processors (NPU) or Field Programmable Gate Array (FPGA), and with the recent emergence of software switches in virtual networks, it is necessary for the proposed design to be target independent while considering the inherent requirements of switches such as high throughput, low latency and low jitter. The proposed protocol independent language should be expressive enough to allow for a large number of functions to be implemented but should consider the limitations of such devices in a way to allow hardware implementations.

In this paper we propose BPFabric, a protocol, platform and language independent SDN environment to implement arbitrary high-performance data plane functions. Through a constrained High Level Language (HLL), lightweight functions can be designed to perform a wide range of services such as statistic gathering and reporting, packet tracing,

ANCS, May 18–19, 2017, Beijing, China
© 2017 ACM. ISBN .
DOI:

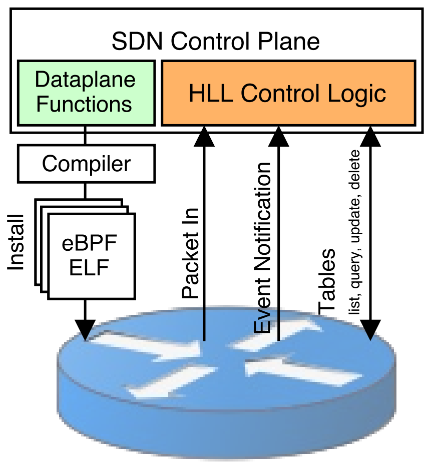


Figure 1: BPFabric controller-to-switch architecture

network telemetry, load balancing and anomaly detection. Once defined, these arbitrary functions are compiled to the eBPF instruction set to allow arbitrary hardware or software targets to be used while capturing the realtime constraints inherent to such networking devices. In addition we introduce a southbound API, to centrally orchestrate, manage, monitor the network behaviour and deploy new data plane functions at runtime. We also provide and open source two implementations of BPFabric switches and multiple examples of data plane functions.

The remainder of the paper is structured as follows. In section 2, we present the design of the proposed architecture in a top-to-bottom approach. In section 3 we show the design of the switch architecture proposed. In section 4, we detail the two software switch implementations and their respective use as well as the central controller. In section 5, we showcase the use of BPFabric for the implementation of forwarding, telemetry and anomaly detection data plane functions. Section 6 discusses related work and section 7 concludes the paper.

2. ARCHITECTURE OVERVIEW

In this section, we take a top-to-bottom approach explaining the design of BPFabric, how it is differs from normal switch design and explain the design decisions that have been made. The proposed architecture is explained from the user-level definition of the network function, to the compilation into the network specific instruction set, deployment of the function in the data plane, and the orchestration of such infrastructure through the control plane.

2.1 Switch to Controller Interactions

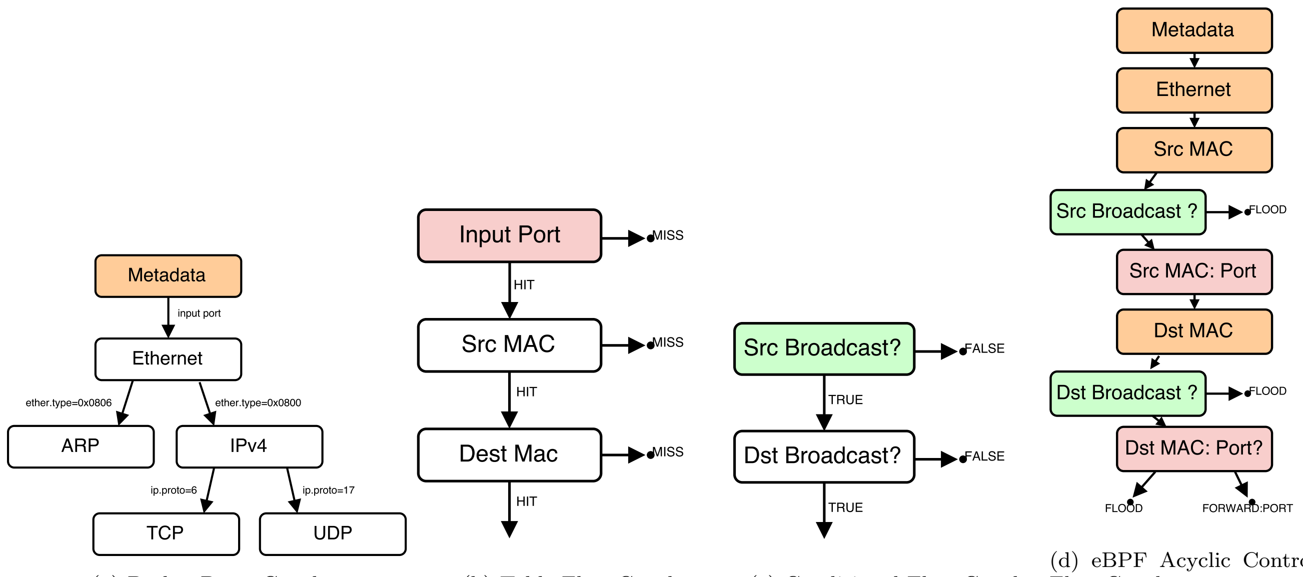
Figure 1 presents a simplified view of the communication between the controller and a single switch within the infrastructure. At this high level representation, the architecture is similar to an OpenFlow network, the centralised controller maintains connections with the switches in the network to configure the data plane through a well-defined API. Through the southbound API, the controller and the agents communicate to allow platform and protocol independent programs to be deployed, events to be generated, and each switch-internal state to be queried and updated. The top-to-bottom logic to create and use distinct data plane behaviour in the proposed architecture is as follows:

- **Data plane behaviour:** Using a constrained HLL, the per-switch data plane behaviour is defined, dictating the parsing, matching, and processing that is performed on every packet.
- **Behaviour compilation:** The constrained HLL definition is compiled to a platform and protocol independent instruction set.
- **Controller Install Request:** Using the southbound interface, a controller program can deploy the data plane functions to the switches in the network.
- **Agent Function Loader:** The agent running on the switch receives the function, processes it into a suitable format for the target (hardware or software), and allocates the necessary match tables to construct a packet processing pipeline.
- **On Ingress Packet:** Every ingress packet goes through the processing pipeline, querying and updating tables, raising notifications to the controller and finally making a forwarding decision.
- **Forwarding decision:** Based on the pipeline output, the packet is forwarded to the relevant output port, sent to the controller, dropped or flooded to all other ports.
- **Controller operation:** On packet notification from the switches, the controller is delegated the responsibility to decide the action to perform.

Using this approach, the controller can deploy new user-defined functions into the data plane of the devices in the fabric, ranging from traditional routing and forwarding functions to more complex middlebox-like functions. The critical difference in this approach compared to OpenFlow is that the pipeline and tables or not simply updated to achieve the desired function but the entire pipeline with the associated tables is restructured when a new data plane function is deployed.

2.2 Dataplane behaviour definition

In BPFabric, the first step is to define the behaviour of each individual switch within the infrastructure and what actions should be performed once a packet is received on a port. This pipeline being executed for every packet in the data plane must be fast to provide line-rate performance and provide guarantees on the execution time. Hence, the behaviour definition should be able to capture the realtime requirements of network switches. The protocol and platform instruction set for the data plane definition must exhibit some properties in order to allow for a wide range of deployment across different target implementations. In the proposed design, the switch either software or hardware, is a substrate optimized for executing a packet processing pipeline but without any predefined behaviour. The behaviour of each switch is then defined by an instruction set allowing access to packet data, protocol headers and restricted branching operations to implement arbitrary functions. To reflect the realtime constraints of a network device, the instruction set should be constrained to an acyclic



(a) Packet Parse Graph (b) Table Flow Graph (c) Conditional Flow Graph (d) eBPF Acyclic Control Flow Graph

Figure 2: L2/L3 eBPF Acyclic Control Flow Graph decomposed into the underlying packet parse graph, table flow graph and conditional flow graph.

control flow graph allowing for a deterministic bounds on the execution time to be computed. The instruction set and compiler should also include security and safety mechanisms to prevent new attack vectors, by preventing at compile time and verifying at runtime memory accesses and function calls.

2.2.1 *eBPF*

In order for data plane functions to be seamlessly deployed across heterogeneous hardware devices and software switches, it is necessary for the compiled function to be platform independent. In 1992, McCane and Jacobson defined the Berkeley Packet Filter (BPF)[1][13], a widely-used instruction set specifically designed for packet filtering. cBPF has been widely deployed in a large number of packet processing libraries such as libpcap used by programs like tcpdump or wireshark. Even though cBPF is over 20 years old, it is still widely used and recent improvements with extended BPF (eBPF) have been integrated in the mainstream Linux kernel (3.18+). Some of the most notable differences between cBPF and eBPF is the move from 32 bits to 64 bits, an increase from 2 to 16 registers, the support for table operations and function calls. [3]

An often forgotten fact is that cBPF was defined in 1992 as a "pseudo-machine", nowadays called virtual-machine, with the main goal to provide protocol and platform independence through a simple instruction set. To allow BPF programs to be executed at a low cost, the instruction set was defined to closely match the instructions of a register machine making interpretation and translation fast and straightforward. BPF does not have any knowledge of the different network protocols and packet structure, and parsing of each packet is performed by loading some of the packet header fields into registers and performing the necessary comparisons. As a result of this simple load and compare approach, the BPF program can be decomposed into a parse graph expressing how the header fields should be extracted (figure 2a), a table flow graph expressing the match table sequence (figure 2b), and a conditional flow graph expressing the algebraic boolean comparisons (figure 2c). One of the main differences between cBPF and any generic CPU instruction set has been to prevent backward jumps in the execution, thus preventing loops and resulting in a non Turing-complete system. This critical difference allows for a fully deterministic execution-time of a program, unlike a traditional instruction set, making it suitable for real-time processing in a bounded execution time environment. As a consequence of this forward-only execution, the BPF functions can be synthesized into an Acyclic Control Flow Graph (aCFG), shown in figure 2d, by combining the underlying parse, table, and conditional graphs.

In the proposed architecture, platform and protocol independence are critical factors to allow a single program to define the function of a software or hardware network device. In this paper, we propose to use eBPF as the instruction set for the compiled data plane functions as it provides protocol independence allowing arbitrary protocols to be parsed and a set of tables to maintain state and perform match-action operations.

2.2.2 *High Level Language Definition*

eBPF provides both the platform and protocol independence required, however, it is an instruction set modeled around a small set of instructions and registers. Similar to any instruction set, such as, e.g., the x86_64 general purpose instruction set, expressing a particular behaviour directly with the instruction set or its closest representation assembly is a complex task, requiring long development effort, prone to errors and potentially less efficient than compiler-generated code.

A constrained high-level programming language (HLL) should therefore be used to define the behaviour of the data

[1]BPF has been retrospectively called cBPF to differentiate it from eBPF. For the remaining of this paper BPF will refer to cBPF and eBPF.

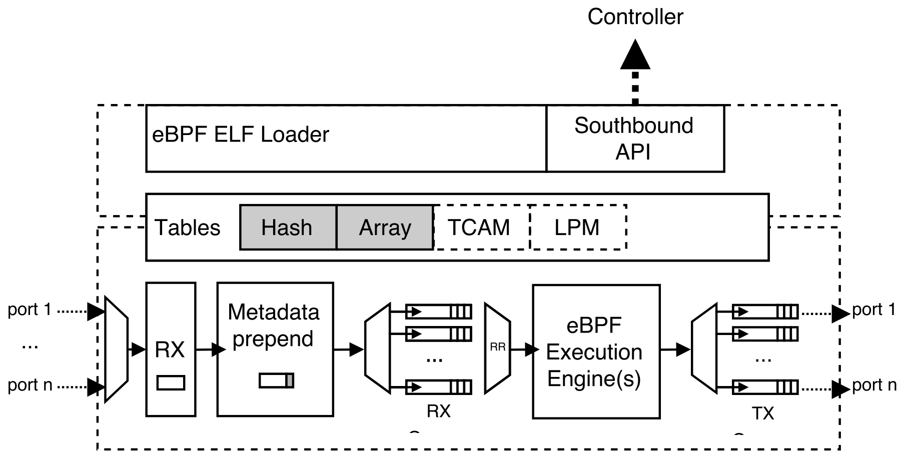


Figure 3: A simplified diagram of the switch architecture with the separation of logic between the control plane hosting the agent and the dataplane processing packets arriving at the ingress.

plane. Currently, the most common programming language for eBPF is a restricted subset of C, due to its very close ties to the Linux kernel. The restricted subset C exclude most external libraries, system calls and most pointer arithmetic while providing few built-in functions for table definition and operations. As of LLVM 3.7, the ebpf backend was added to the compiler to allow straightforward compilation from purpose-made C source code to an eBPF Executable and Linkable Format (ELF) file. A Domain Specific Language (DSL) can also be envisaged to better match the behaviour of the data plane, such as, e.g., P4 [5], a DSL specifically designed for protocol-independent packet processors. Existing efforts within the iovisor open source project have shown promising results in that direction and have implemented a partial P4-to-ebpf compiler [2].

2.2.3 *ELF*

Most data plane functions will contain one or multiple tables to perform match-action pipeline operations and to some network function state across packets. A traditional hardware switch contains hardware tables that can perform queries and updates at line-rate and often have a relatively small amount of general purpose memory. Therefore, to truly achieve platform independence, the table requirements should be expressed by the data plane behaviour and allocated by the device itself based on the available resources.

A data plane function compiled to eBPF will contain the compiled eBPF instructions defining the packet processing pipeline, as well as some additional metadata for each table specifying the table unique identifier, type, size of keys, size of values and maximum number of entries. The resulting ELF file allows true platform independence regardless of hardware or software switch implementation. It defines the eBPF instructions defining the processing pipeline, which maps should be allocated, and how the allocated tables should be linked to the packet processing pipeline. Using this approach the creation and allocation of the tables is delegated to the individual devices, allowing hardware or software tables to be used depending on the platform and resource availability.

3. SWITCH DESIGN

Figure 3 provides an architectural overview of the control and data plane within the switch. Following the SDN approach made popular by OpenFlow, the device is designed to be “dumb”, or more specifically, it does not contain any internal logic (e.g., self-learning, peering, discovery, etc.). As a result, the control plane of each device can be kept lightweight, hosting only an agent akin to the OpenFlow agent, responsible for interfacing the underlying data plane with the central controller through the southbound API. Using this approach, the behaviour of a switch is only defined by the explicitly deployed eBPF pipeline or through requests issued by the controller.

A critical component within the control plane of the network device is the eBPF ELF Loader that is responsible for allocating the eBPF tables required by the pipeline and finally transforming the eBPF bytecode into a fast and usable format suitable for the device. The eBPF loader is device-specific and therefore should perform the operations that are the most suitable for the target device. This design choice is significantly different from the one taken by P4 in which the data plane function must be compiled independently for each target device. By delegating the loading of the bytecode to the agent, the controller can be kept platform independent, not requiring to know in advance the devices that will be managed and allowing precompiled functions to be distributed. We can leverage the simplicity and platform independence of the instruction set as well as the aCFG execution model to transform the packet processing pipeline into a format optimized for the target switch. The agent loader can Just-in-Time (JIT) compile the byte code, or perform more complex tasks such as translating to specific Network Processing Unit (NPU) platform, synthesize the aCFG graph into an FPGA pipeline or into a match-action pipeline suitable for RMT[6].

The eBPF loader should also include a verifier that checks the validity, security and performance of the eBPF program being loaded. The verifier can be used to statically and deterministically verify if a program terminates correctly, the memory accesses are bound to the address space allocated, loops are not present, and the maximum depth of the execution path to calculate the worst-case execution time of the

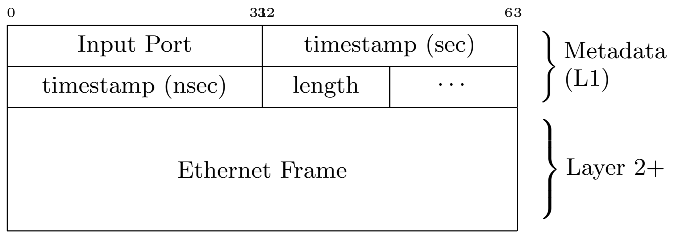


Figure 4: Packet format within the dataplane. Metadata information is appended to the frame received.

program. Using a verification stage within the loader prevents the introduction of malicious or erroneous data plane functions in the network. The verifier is out-of-scope of this paper, however more details on the implementation and the simplicity of performing these checks can be observed in the Linux kernel eBPF implementation.

3.1 Data Plane

The data plane receives packets from the input interfaces and stores them into receive queues with some prepended metadata providing link layer information and a receive timestamp. Figure 4 shows the structure stored in the receive queues with the metadata prepended before the ethernet frame. The reason for prepending the metadata to the packet is to be able to query and compare against some of the fields in the eBPF execution engine. Originally, BPF was designed as a per-socket program and therefore link-layer information were out of scope. With its evolution over time and the requirement for metadata, the Linux kernel has used specific memory addresses to represent the metadata, however we believe that this approach is a workaround and in a switch where the metadata will be required for the majority of functions, prepending the information to the frame itself is the most suitable approach. Using this approach, eBPF can load the metadata information as any other protocol field.

Subsequently, once one of the eBPF execution engines is available, a packet and its associated metadata is retrieved from the receive queues. The order in which the packets are retrieved from the input queues is dependent on the packet scheduling algorithm which is out-of-scope in this work (the existing implementations are using round-robin). The packet retrieved from the queues is passed to the available eBPF execution engine. The engine will pass the packet through the processing pipeline, updating and querying internal state and match-action tables to perform complex functions, and finally returning a forwarding decision for this particular packet.

In BPFabric, we have used a similar approach to OpenFlow in using the port number to represent both physical and virtual ports. Using this approach we can use the pipeline return value to either forward the packet to a physical port on the switch, any allocated virtual port (e.g. tunnels) and reserved ports for flooding, dropping and sending the packet to the controller. In table 1 we list the three reserved values that have been defined in BPFabric. Once a forwarding decision is made, the packet is dropped or enqueued on the relevant transmit queue(s). In either the receive or transmit scenario, a tail-drop mechanism is used in the case of a fully occupied buffer, however this is implementation independent and, similar to the packet scheduler, any suitable mechanism can be used.

Name	**Value**	**Description**
FLOOD	0xFFFFFFFD	Send the packet on all the ports except the input port.
CONTROLLER	0xFFFFFFFE	Send the packet to the control-plane for it to be encapsulated in a packet_in message and sent to the controller.
DROP	0xFFFFFFFF	The packet drop.

Table 1: eBPF return values used for special actions.

3.2 Control Southbound API

Similar to OpenFlow, a controller provides an overarching view and control over the network by maintaining a persistent connection with the controlled switches. In SDN terminology, this communication interface between the controller and the devices has been referred to as the Southbound API with the most well-known implementation being OpenFlow and other alternatives being the Network Configuration Protocol (NETCONF) or Cisco OpFlex. In BPFabric, both in-band and out-of-band control-plane are supported, and whether to use one or the other is at the discretion of the user depending on the infrastructure available, and the reliability and security levels required.

In the proposed architecture, this persistent connection between the controller and the switches is used by the controller to issue synchronous requests to the switches and for the switch to raise events and notifications when the controller must be involved. The first step of the communication is to set up an handshake between the controller and the device to establish the version compatibility of the endpoints, negotiate the supported features and the datapath identifier (identity) of the switch. Once the connection is made, per-switch data plane functions can be installed dynamically. With each data plane pipeline keeping its internal state in a set of tables, the controller is able to request the content of the tables or specific entries allowing the global view to be maintained. If necessary, the controller can also update those entries, if a controller-local decision is made. Finally, the controller is able to craft or resend packets to any connected switch through the southbound API much like the *packet_out* message in OpenFlow.

Using the same communication channel between the agent and the controller, the switches can also emit asynchronous requests to the controller. The protocol in its current format defines two different types of asynchronous requests, the first one is triggered when the data plane program in eBPF delegates the decision-making to the controller logic, similar to a table miss or an explicit forward-to-controller action in OpenFlow. This request contains the packet that triggers the miss as well as the associated metadata and is therefore similar to a *packet_in* message in OpenFlow. The second type of asynchronous message is a notify message that can be triggered by the data plane function to notify the controller of an event that took place without impacting the

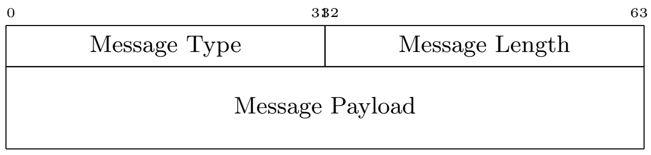


Figure 5: Southbound API message structure

default forwarding mechanism of the packet. Asynchronous notifications are critical to data plane programmability in order to support a wide range of functions such as telemetry, monitoring and reporting necessary to orchestrate today's very large scale networks.

In its current implementation, the southbound API is very straightforward, the TCP transport protocol is used between the controller acting as the server and accepting multiple concurrent connections from the switches acting as clients. Each message contains a short header and the payload of the message, as shown in figure 5. The header is only 64 bits, containing two 32-bit fields, the message identifier (opcode), and the length of the payload to be expected after the header. The payload itself is dependent on the message type and can be any of the message defined in table 2. To follow the aim of the proposed architecture to be platform and language-independent and to avoid the over-complexity of wire protocols such as in OpenFlow, the protocol messages have been defined using Google protocol buffers. In comparison, the OpenFlow protocol was designed from the ground up using a custom binary format and specified using C structures with many hardcoded identifiers, opcodes and packet layout. With the many revisions of the protocol and the new requirements OpenFlow was heavily modified overtime while imposing some level of backward compatibility resulting in many edge cases in dealing with the protocol and difficulties supporting it in other programming languages, as visible in software like the Ryu controller or Open vSwitch.

4. IMPLEMENTATION

4.1 Switch

As part of this work, we have implemented two different versions of BPFabric software switches and are currently working on an hardware implementation using Verilog and targetting the NetFPGA platform[2]. The first implementation uses the Linux raw socket interface to provide in user-space the functionalities described in the previous sections. The second implementation relies on the Intel DPDK framework [1] to perform high-speed and low-latency packet processing.

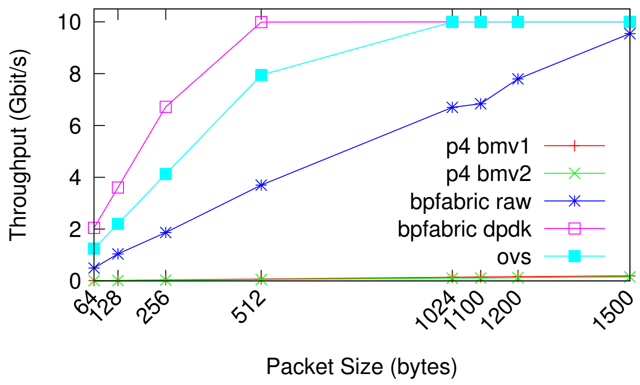
The first implementation has been designed to be used on any conventional Linux machine as a way to quickly test and evaluate data plane functions in eBPF without much focus on performance. The entire switch pipeline to receive, process and transmit the packets is within the same thread limiting the overall performance. However, by using mmap to share the receive and transmit ring buffers between user space and kernel space, the performance is on par or superior to user-space OpenFlow switches [10]. As this implementation is fully in user-space and based on the default Linux kernel socket interface, it can be used in containers and namespaces to allow large scale experiments to be designed on a single machine. For this purpose, we have also added support for BPFabric in Mininet [12], allowing existing network topologies and network scripts to be executed over the widely used Mininet environment.

Although we are able to show the use-cases for BPFabric, user-space software switches are designed for experimentation as due to the networking stack overhead they are unable to cope with the line-rate speed of production networks. Therefore, we also provide a second, high-performance software-switch implementation, using Intel DPDK, a set of libraries for fast packet processing in user space bypassing the traditional kernel networking stack. This implementation relies on the Intel DPDK poll-mode drivers and therefore can be used with a limited set of physical NICs and few virtualized NICs such as in Xen, QEMU, VMWare ESXi, and Amazon ENA. Using the DPDK threading model, we are able to instantiate a single eBPF execution engine per core and by using DPDK memory ring-buffers with Linux hugepages, we can have transmit and receive queues without pagefaults. By using the libraries provided by DPDK and the concurrent execution engine, we are able to fully utilise the resources of the machine.

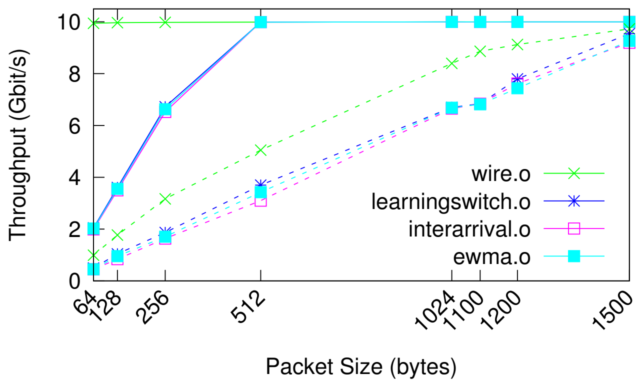
Type	Direction	Description
Hello	S $\leftrightarrow$ C	Handshake message between switch and controller
Install	C $\rightarrow$ S	Install eBPF ELF on the switch
Packet In	S $\rightarrow$ C	Packet sent from a switch to the controller
Packet Out	C $\rightarrow$ S	Send a packet from the controller to a switch's output port
Tables List	C $\rightarrow$ S	List all the instantiated tables
Table List	C $\rightarrow$ S	List the content of the table specified
Table Entry Get	C $\rightarrow$ S	Get a single entry from the table specified
Table Entry Insert	C $\rightarrow$ S	Update or Insert an entry with the key and value provided
Table Entry Delete	C $\rightarrow$ S	Remove an entry from the table specified
Notify	S $\rightarrow$ C	Asynchronous notification of an event with user defined payload

Table 2: Supported messages over the southbound API between the controller and the controlled switches

[2]The complete source code of the switch implementation and the examples described in this paper is available on GitHub at https://github.com/UofG-netlab/BPFabric



(a) Performance comparison of P4 behaviour models, OpenvSwitch and BPFabric for a layer 2 learning switch.



(b) Performance comparison of example programs between Raw socket and DPDK implementation.

Figure 6: Packet processing behaviour of BPFabric implementations

To highlight the clear separation between the control and data planes, both switch implementations execute the same control plane agent but each run their individual data-plane for receiving, processing and forwarding the packets: Linux raw socket for the first one and Intel DPDK for the second one.

In both switch implementations we have used the userspace JIT compiler ubpf as the eBPF engine [4]. ubpf is a just-in-time compiler for eBPF to x86_64 architecture, it was originally designed as an Apache-licenced library for executing eBPF programs and has now been included as part of the IOvisor project [2]. Our implementation uses a modified version of ubpf that allows the eBPF tables to be allocated if they have not been created and relocate them before the code is JIT compiled.

In figure 6, we highlight the performance of BPFabric between existing software switches (fig. 6a) and the performance achievable with some example use cases (fig. 6b. These experiments have all been run on a high-end modern commodity machine, running Linux kernel 4.4 with an Intel X710 4x10G Network card. In figure 6a we compare the achievable throughput for a comparable learning switch implementation in P4 and BPFabric as well as the default learning behaviour of Open vSwitch. In these experiments, the P4 behavioural models version 1 and 2 (bmv1, bmv2) never exceed the throughput of 200 Mbit/s, the raw socket implementation of BPFabric performs linearly based on the packet size, and the DPDK implementation of BPFabric performs faster than kernel-based Open vSwitch, reaching 10G line-rate for 512 bytes packets. It is worth noting that the P4 behavioural model was never optimized for speed and therefore the low speed observed can be expected. In figure 6b, we compare the performance of a wire implementation simply forwarding packets between two ports, the learning switch, as well as packet inter arrival and ewma examples both performing per-packet processing and then learning switch behaviour. The solid lines represent the DPDK implementation and the dashed line the raw socket implementation. These experiments highlight that an application without table lookup can achieve line-rate even for the smallest packet size and that table operations are the most costly operation which is to be expected in a software implementation.

4.2 Controller

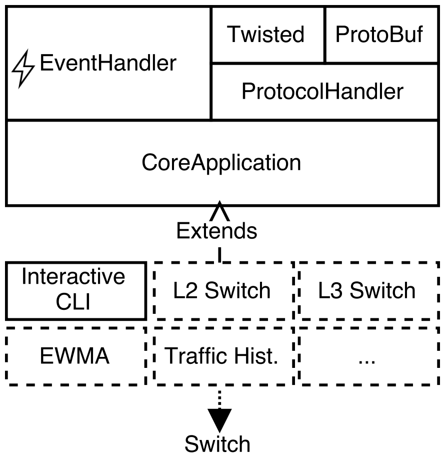


Figure 7: BPFabric controller architecture

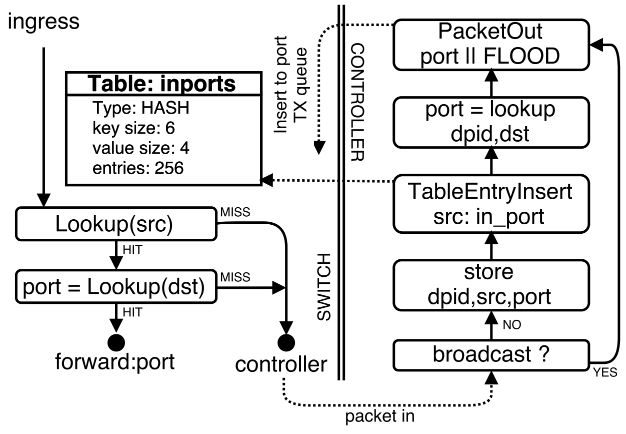
The central controller was implemented using Python for simplicity and to allow new controller functions as seen with POX and Ryu for OpenFlow to be quickly designed. The architecture of the controller is shown in figure 7. New controller centric function can be designed to interact with the switches at runtime, to install new functions if necessary or monitor the internal state of the devices. The controller is also responsible for monitoring and processing the asynchronous notifications raised by the data plane functions. As these notifications are application specific custom controller applications must be developed to handle the events and perform the necessary actions. As shown in the figure 7 controller applications such as EWMA and Traffic Histogram have been developed to display in realtime the current state of the network based on the notifications raised by some of the dataplane functions.

Using a simple southbound API and relying on widely used libraries for the design of the controller such as Twisted and Google Protocol Buffers, the code required to interact with the Agent in the switches can be kept simple. The entire implementation of the core is less than 200 lines of

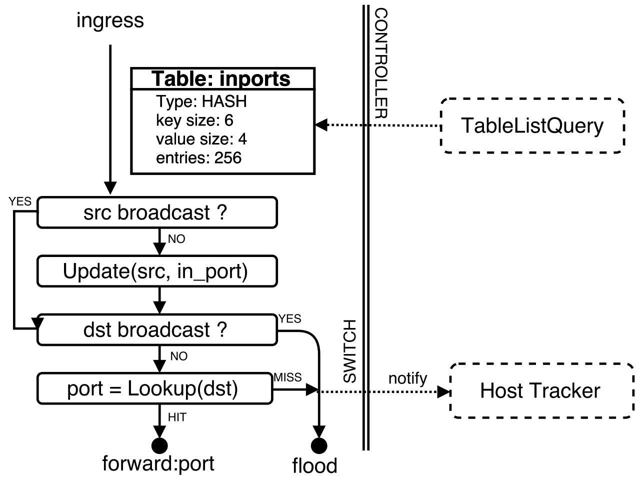
Program	# LoC	# Instr.	# Tables	# Table Lookups	# Table Updates	Table Space (bytes)
Centralized L2 Switch	20	25	1	2	0	$10m$
Learning L2 Switch	22	28	1	1	1	$10m$
Pkt. size distribution	10	44*	1	1	1	$8n$
Pkt. inter arrival time	37	98*	2	3	2	$24 + 8n$
EWMA	24	99*	1	1	1	$32p$

m = number of unique MAC addresses, p = number of ports on the device, n = number of buckets in the histogram
* is including the self-learning switch logic

Table 3: Example programs implemented using BPFabric, with associated complexity, table operations and memory requirements.



(a) SDN centralized learning switch



(b) Legacy self-learning switch with centralized monitoring

Figure 8: Implementations of a learning switch in an SDN centralized manner and in a legacy self-learning approach.

code, and therefore adapting this to any other language is a trivial implementation task.

5. EXAMPLE PROGRAMS

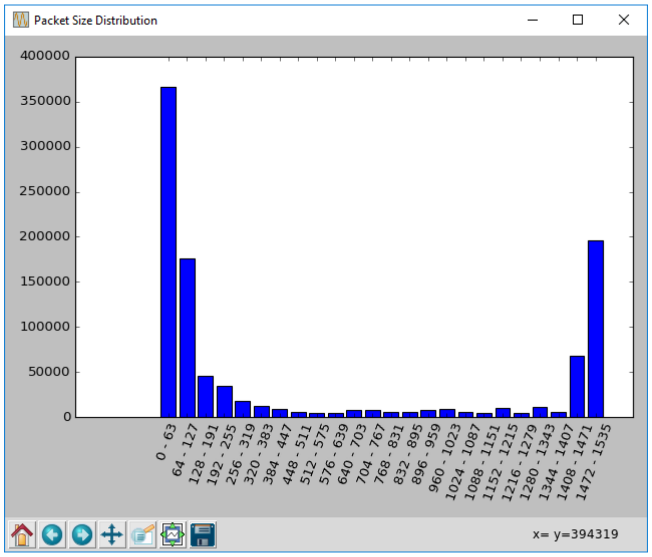
We showcase some example data plane and control plane tasks that can be implemented using BPFabric, highlighting the utility of the proposed framework for a wide range of applications. The following example programs and more are provided with the switch implementations and can be run indifferently on either implementation. In table 3, we highlight the complexity in number of lines of code (C) and resulting number of eBPF instructions, the maximum number of table operations required and the memory requirements for state keeping.

5.1 Layer 2 Learning Switch

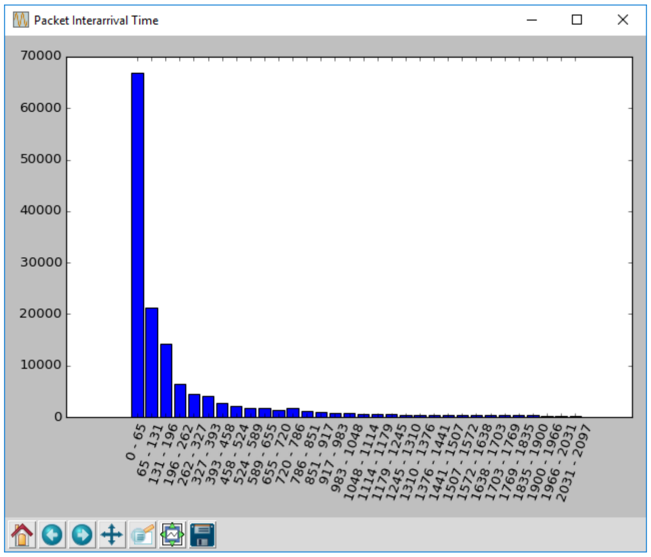
The typical example "hello world" for new SDN or controller implementations is a simple centralized layer 2 learning switch. Showcasing a layer 2 switch is a useful and practical way to demonstrate the ability of the individual switches to delegate packet forwarding responsibility to the controller, make a central decision based on the global view of the network, and update the switch accordingly. In figure 8, we show two different implementations of a learning switch, showcasing both the ability to delegate responsibility to the controller and secondly the ability for data plane functions to self-update tables.

Centralized Learning Switch: The first implementation shown in figure 8a represents the centralized SDN approach, with the switch delegating the entire decision process to the controller. In this scenario, a lookup table is maintained to hold the mapping between an Ethernet MAC address and the input port of the packet. When a packet is received and the source or destination address is not present in the lookup table, the controller is notified. The controller program stores the switch identifier, associated source address and input port in a table, then emits a table insert request to update the switch lookup table. Finally, the controller looks up for the port associated to the destination address and emits a packet out request to the switch specifying either the looked up port or a flood action if the entry was not found.

Self-learning Switch: Secondly, in figure 8b, we demonstrate that a traditional (legacy) self-learning switch can be easily implemented using the data plane ability to self-insert and update entries in the switch tables which is not possible in today's OpenFlow or P4 implementations. For every incoming packet, the lookup table is updated if the source address is unicast and the destination port looked up, if an entry is found the packet is forwarded or otherwise flooded. In this scenario, the controller is not responsible for making the forwarding decision, however, a big difference from a traditional switch is the ability of the controller to be notified if a new entry is added, and the ability to query the content of the tables at any point in time. The support for self-updating tables has been mostly ignored in current SDN



(a) Packet size distribution histogram



(b) Packet interarrival time histogram

Figure 9: Telemetry histograms based on reported values from dedicated data plane functions

implementations: OpenFlow delegates the entire responsibility to the controller, while P4 considers self-updating as a possible future improvement limiting greatly the functions that can be implemented efficiently in the data plane.

5.2 Network Telemetry

Network telemetry is used to verify that the network is behaving as intended and to monitor any changes occurring in the network over time. Through the process of continuously collecting network metrics, the central controller or third-party management applications can gain granular visibility into the network behaviour in order to model and predict future trends. This insight into the network behaviour can be used to adapt the fabric as the demand evolves, migrate applications or virtual machines (VMs) to improve overall network utilisation [9], as well as improve policies to meet with the customer's Service Level Agreements (SLA). Moreover, the telemetry data can be correlated with other logs to identify potential anomalies, security risks and for debugging purpose.

Telemetry support has been present in network devices for a long time through different implementations and providing different levels of insight into the network traffic. The most common implementation of telemetry might be the Simple Network Management Protocol (SNMP) that allows a monitoring station to pull vendor-specific values from the devices. However, with the growth of network sizes and the associated cost-incentive to better utilise the available resources, telemetry is currently being heavily revisited and reimplemented in a push-model where the devices stream events of interest directly to data collection points, reducing network load and volume of data to process.

Using BPFabric, we showcase two telemetry examples using both the pull and push approaches to demonstrate that user-defined telemetry metrics can be defined and collected from the infrastructure. In figure 9 we show two screenshots of the controller realtime visualization of the reported metrics by the switches.

Packet Size Distribution: We have implemented a data plane function to store the per-switch packet size distribution as a histogram (figure 9a. Using an array-type eBPF table to store the buckets for the histogram and the length available in the packet's metadata, we can keep track of the packet distribution over time. Using the table list control message, the controller can query (pull) the current state of the histogram in any switch where this function was deployed. It can provide insight of the nature of the traffic, for instance if the traffic is from real-time applications with overall small packet size or batch-based with MSS-sized packets, as well as determining trend over long period of time to determine network normal behaviour.

Packet Inter Arrival Time: The second telemetry example showcased measures the packet inter arrival time at a particular switch (figure 9b). The function is designed around two tables, one to store the histogram of the inter arrival time and a second one to keep track of the last packet received. At a regular interval, every 2048 packets in this case, the histogram data is pushed to the controller using the API asynchronous notification message. The inter-arrival time of packets and the associated mean and jitter can provide information about the congestion state of the network, the burstiness of the traffic impacting real-time streams or help with traffic classification.

5.3 Lightweight Anomaly Detection

In this example we demonstrate that some functions that have been typically delegated to special middleboxes in the network can be implemented as part of the switching fabric through lightweight middlebox-like packet processing pipelines.

We implement a lightweight anomaly detection algorithm using Exponential Weighted Moving Average (EWMA). EWMA is a statistic to average data over time and reduce the weight of previous measurements the further away in time they are. Using this statistical value on some of the networks metrics provides insight on the normal operating condition. Significant deviation from this normal behaviour can then high-

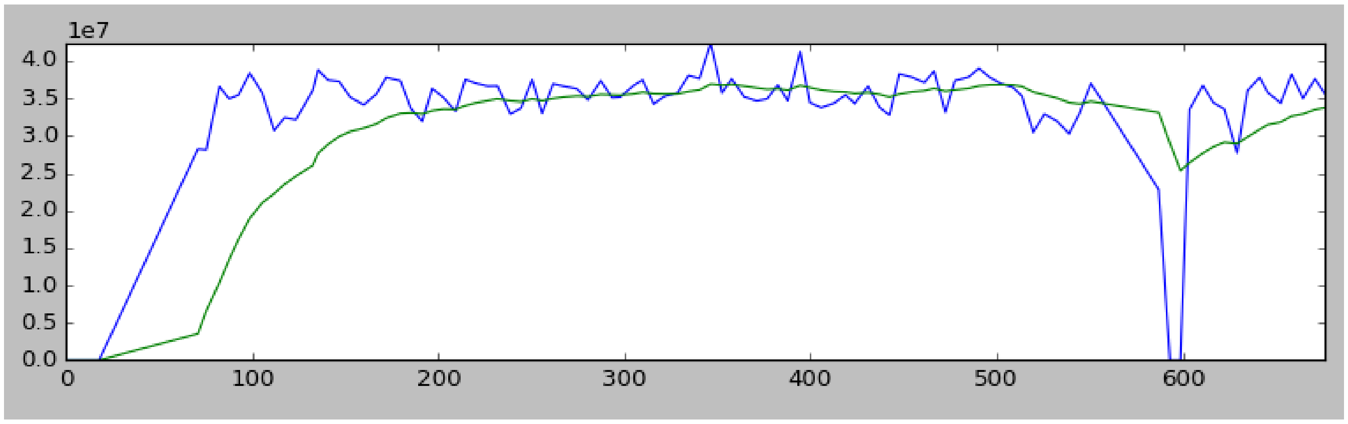


Figure 10: Reported EWMA volume average in bytes (blue) and predicted volume (green) over time by the switch with an introduced anomaly (link failure) at t=600s

light anomalies within the network.

Our implementation computes the EWMA value of the volume of traffic received on every port of the switch for every incoming packet. The information is maintained in an array-type map containing one entry for every port. At a specific time interval, 5 seconds in the current implementation, the EWMA value is calculated and compared against the threshold values. If the computed value is not within the expected bounds, a notification is raised to the controller signaling an anomaly. In figure 10 we show the reported and predicted EWMA values and introduce an anomaly at $t = 600s$.

Using this approach, many lightweight network functions can be implemented, however, the limitations of the device should also be taken into consideration. As described previously in section 2.2.1, in order to provide a realtime execution environment, unbounded or long lasting loops should be forbidden which can be an issue while implementing more complex network functions. Also, switches typically do not have hardware support for floating point arithmetic as it is of no use for network operations and therefore limit the complexity and accuracy of the computations. An example of the latter is the weight (α) in the calculation of the EWMA value that should be between 0 and 1 to specify the importance of the previous measurement on the calculated average. To accommodate for this issue, we have used the value $\alpha = 0.125 = 2^{-3}$ that can be achieved using bitwise shift operations.

6. RELATED WORK

The idea of programmable networks emerged from Active Networks focusing in the ability to add arbitrary programs in-band with packets [20, 16]. However most active networking efforts were not widely deployed due to concerns on the performance and security. More recently the concept of active networks was realised in TPP [11], allowing end-hosts and switches to interface with an extremely simple and limited instruction set. Over the last decade, the paradigm of programmable networks shifted from in-band towards a out-of-band controller-centric approach [8], with the now commonly used OpenFlow protocol and its foundation Ethane [14, 7]. This central-controller approach has highlighted the benefits in empowering the network administrator with the ability to partially control the network and emerged as Software-Defined Networking (SDN). Although OpenFlow has been widely successful in showing the advantages of programmability and therefore resulting in a wide number of contributions to networking, its lack of flexibility and future-proofness have shown its limitations.

To provide the required flexibility and high performance in the data plane, chipsets have been designed with different levels of programmability. Intel's FlexPipe architecture supports a programmable packet parser with a 32 stages match-action pipeline through a dedicated microcode [15]. The publication in 2013 by Bosshart et al. of the Reconfigurable Match Table (RMT) architecture for a multi-stage programmable packet processor highlighted that ASICs could be designed to provide the desired programmability without compromising the packet processing rate [6].

The P4 programming language was proposed as a declarative high-level language to describe the data plane packet processing behaviour [5]. From its roots as a HLL for RMT, P4 has been designed to build a match-action pipeline making it unsuitable for architectures with a different paradigm. POF [19] was also suggested as protocol-oblivious approach at a match-action pipeline providing instructions for modifying packets and updating the match tables. POF relies on a search key extracted from header fields to perform table matches and uses the Flow Instruction Set (FIS) to manipulate the packets and associated metadata. PISCES highlights the need for software packet processing functionality implementations in data centers where the number of hypervisors is higher than the number of switches [17]. NetASM [18] provides a simple intermediate representation focusing on the primitive data plane function and high-speed execution, however it does not provide bounds on the execution time due to its Turing-complete instruction set.

In BPFabric, we aim to demonstrate a design of both the network devices as well as the controlling infrastructure where the target goal is a flexible data plane in a SDN environment providing routing, forwardind and middlebox-like functions. For this purpose, we show how a controller-centric architecture that can be used to program the data plane of an arbitrary hardware of software device through a southbound API, using a platform and protocol-independent instruction set.

7. CONCLUSION

OpenFlow, through its vendor-agnostic protocol and open implementation, has been able to show the benefits of SDN by centrally controlling network devices. However, in its current implementation, the programmability is limited, providing a fixed and limited set of headers fields, actions, matching primitives and a very rigid processing pipeline. With this approach, the control plane cannot define the packet processing logic, how and which header fields should be extracted and matched, preventing the support for new network protocols, statistics, monitoring, routing and other middlebox-like functions.

In this paper, we have proposed BPFabric, an architecture that allows the control plane to specify on-the-fly the data plane packet processing pipeline of the switches as well as to query and manipulate the network state directly. Using a platform and protocol independent instruction set, the data plane function can be expressed independently of the target device and without constraints on the HLL while providing performance and safety guarantees.

We demonstrate through examples how current and new data plane functionality can be designed to improve routing and forwarding, provide insight in the network behaviour, and the ability to implement lightweight middlebox-like functions. Through a simple southbound API, we also demonstrate how the controller can react to network events and maintain a global view of the network state. Finally, through two implementations, we provide a large-scale experimentation setup based on Mininet and a high performance switch using Intel DPDK with performance comparable to today's static implementations.

8. ACKNOWLEDGMENTS

The work has been supported in part by the UK Engineering and Physical SciencesResearch Council (EPSRC) projects EP/L026015/1, EP/N033957/1, EP/P004024/1,and EP/L005255/1, and by the European Cooperation in Science and Technology(COST) Action CA 15127: RECODIS âĂŞ Resilient communication services protect-ing end-user applications from disaster-based failures.

9. REFERENCES

[1] Intel DPDK: Data Plane Development Kit. http://www.dpdk.org.

[2] IOvisor Project. https://www.iovisor.org/.

[3] Linux Socket Filtering aka Berkeley Packet Filter (BPF). www.kernel.org/doc/Documentation/networking/filter.txt.

[4] uBPF: Userspace eBPF VM. https://github.com/iovisor/ubpf.

[5] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-independent Packet Processors. *SIGCOMM*, July 2014.

[6] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 99–110, New York, NY, USA, 2013. ACM.

[7] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking control of the enterprise. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '07, pages 1–12, New York, NY, USA, 2007. ACM.

[8] M. Casado, T. Koponen, D. Moon, and S. Shenker. Rethinking packet forwarding hardware. In *In Proc. HotNets*, 2008.

[9] R. Cziva, S. JouÃńt, D. Stapleton, F. P. Tso, and D. P. Pezaros. Sdn-based virtual machine management for cloud data centers. *IEEE Transactions on Network and Service Management*, 13(2):212–225, June 2016.

[10] E. L. Fernandes and C. E. Rothenberg. OpenFlow 1.3 software switch. In SBRC'2014, 2014.

[11] V. Jeyakumar, M. Alizadeh, Y. Geng, C. Kim, and D. Mazières. Millions of little minions: Using packets for low latency network programming and visibility. *SIGCOMM Comput. Commun. Rev.*, 44(4):3–14, Aug. 2014.

[12] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, pages 19:1–19:6, New York, NY, USA, 2010. ACM.

[13] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX*. USENIX, 1993.

[14] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM*, Mar. 2008.

[15] R. Ozdag. Intel ethernet switch fm6000 series - software defined networking. 2012.

[16] B. Schwartz, A. W. Jackson, W. T. Strayer, W. Zhou, R. D. Rockwell, and C. Partridge. Smart packets: Applying active networks to network management. *ACM Trans. Comput. Syst.*, 18(1):67–88, Feb. 2000.

[17] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, and J. Rexford. Pisces: A programmable, protocol-independent software switch. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*, SIGCOMM '16, pages 525–538, New York, NY, USA, 2016. ACM.

[18] M. Shahbaz and N. Feamster. The case for an intermediate representation for programmable data planes. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 3:1–3:6, New York, NY, USA, 2015. ACM.

[19] H. Song. Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 127–132, New York, NY, USA, 2013. ACM.

[20] D. L. Tennenhouse and D. J. Wetherall. Towards an active network architecture. In *Proceedings of the 2002 DARPA Active Networks Conference and Exposition*, DANCE '02, pages 2–, Washington, DC, USA, 2002. IEEE Computer Society.