Arbitrary Packet Matching in OpenFlow

Simon Jouet, Richard Cziva and Dimitrios P. Pezaros
School of Computing Science, University of Glasgow, Glasgow, G12 8QQ, Scotland
{s.jouet.1, r.cziva.1}@research.gla.ac.uk, dimitrios.pezaros@glasgow.ac.uk

***Abstract*—OpenFlow has emerged as the *de facto* control protocol to implement Software-Defined Networking (SDN). In its current form, the protocol specifies a set of fields on which it matches packets to perform actions, such as forwarding, discarding or modifying specific protocol header fields at a switch. The number of match fields has increased with every version of the protocol to extend matching capabilities, however, it is still not flexible enough to match on arbitrary packet fields which limits innovation and new protocol development with OpenFlow.**

In this paper, we argue that a fully flexible match structure is superior to continuously extending the number of fields to match upon. We use Berkeley Packet Filters (BPF) for packet classification to provide a protocol-independent, flexible alternative to today's OpenFlow fixed match fields. We have implemented a prototype system and evaluated the performance of the proposed match scheme, with a focus on the time it takes to execute and the memory required to store different match filter specifications. Our prototype implementation demonstrates that line-rate arbitrary packet classification can be achieved with complex BPF programs.

I. Introduction

Software-Defined Networking (SDN) is an emerging paradigm to centralise the control plane of large communication networks and separate it from the data plane responsible for packet processing and forwarding. SDN makes it easier to introduce new abstractions, simplifies network management, and facilitates network evolution and innovation. It has been used for traffic engineering, QoS enforcement and network virtualization. OpenFlow [1] is the most popular and widely deployed realisation of SDN, and has become the *de facto* standard due to its open-source implementation, well-defined API and standardised protocol. OpenFlow is designed to use hardware components of existing switches such as the Ternary Content Addressable Memory (TCAM) to provide fast packet classification in the data path and alleviate slow CPU-based packet classification [2]. As a result of the open protocol design, reference implementation and backward compatibility to legacy hardware, network vendors have quickly adopted, deployed and maintained OpenFlow-enabled switches.

The first production-ready release of OpenFlow was able to match on 12 fields of the packet header from L1 to L4, while the most recent version of the protocol (1.5) can match on 44 fields. Table I shows each major revision of the OpenFlow protocol alongside the number of supported fields, the maximum number of fields a single flow can match on (referred as depth), and the maximum number of bits a single flow entry requires. The large increase in size between versions 1.1 and 1.2 is related to the addition of IPv6 support. This growth over time shows that the number of supported fields is not likely to slow down as network requirements evolve to support new protocols, such as, e.g., GRE, VxLAN, STT and OTV, as well as operator-specific protocols and encapsulations. However, expanding OpenFlow to support more fields requires TCAM size in switches to grow accordingly, or otherwise incur a significant reduction in the number of flows installed at the switch at any given time. On a standard OpenFlow Top-of-Rack switch, such as, e.g., a Pronto 3290 (FireBolt 3 switching ASIC) running OpenFlow 1.0, around 2000 flows can be simultaneously inserted [3]. If the same switch was to support OpenFlow 1.5 and following the growth shown in Table I, only approx. 700 flow entries could be accommodated.

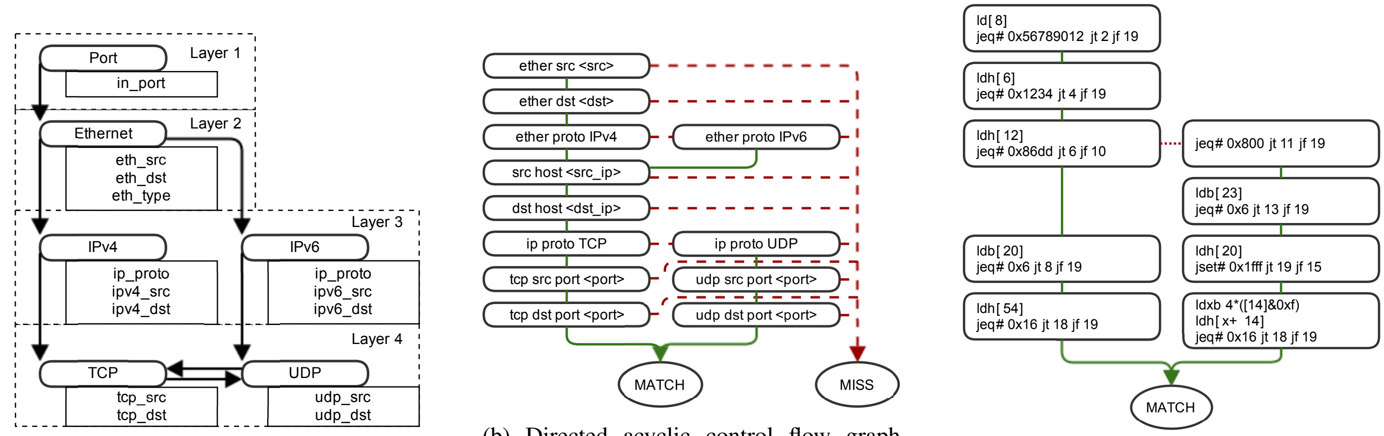
TABLE I: Number of fields supported by OpenFlow for each protocol revision and associated storage required for individual flow entries. The depth range corresponds to the minimum core set of fields to a complete implementation including optional fields.

OF Version	Release date	Match fields	Depth	Size (bits)
<1.0	Mar 2008	10	10	248
1.0	Dec 2009	12	12	264
1.1	Feb 2011	15	15	320
1.2	Dec 2011	36	9–18	603
1.3	Jun 2012	40	9–22	701
1.4	Oct 2013	41	9–23	709
1.5	Dec 2014	44	10–26	773

OpenFlow has been designed for innovation in network protocols [1] with a limited number of supported fields. However, even if kept up to date, it cannot satisfy the flexibility requirements of a clean-slate protocol design [4]. The current approach to match packets on other than OpenFlow fields is to do it at the OpenFlow controller, resulting in additional latency, network traffic and controller load. This problem of fixed match fields has been acknowledged by the introduction of OpenFlow eXtended Match fields (OXM) in version 1.2, and the addition of the packet type awareness in the pipeline of 1.5. Another limitation of the current matching, is the inability to do logical operations other than equality such as "greater than" and "lower than", necessary for range matching. We argue that packet matching should be designed independently of any protocol implementation, and allow the control plane to specify the matching process through a set of platform-independent instructions designed to match packets at every layer. Through such instruction set, the execution of the matching could be left as an implementation detail relying on software optimisations (such as Just In Time (JIT) compilation) or hardware acceleration using, e.g., FPGAs or ASICs.

In 1992, McCanne and Jacobson defined the Berkeley Packet Filter (BPF) [5], a widely-used approach for packet filtering within the Linux kernel[1], subsequently used by *tcpdump* and *libpcap*. Even though now over 20 years old, BPF

[1] https://www.kernel.org/doc/Documentation/networking/filter.txt



```
ld[ 8]
jeq# 0x56789012  jt 2 jf 19
ldh[ 6]
jeq# 0x1234  jt 4 jf 19
ldh[ 12]
jeq# 0x86dd  jt 6 jf 10
jeq# 0x800  jt 11  jf 19
ldb[ 23]
jeq# 0x6 jt 13 jf 19
ldb[ 20]
jeq# 0x6 jt 8 jf 19
ldh[ 20]
jset# 0x1fff jt 19 jf 15
ldh[ 54]
jeq# 0x16 jt 18 jf 19
ldxb 4*([14]&0xf)
ldh[ x+  14]
jeq# 0x16 jt 18 jf 19
```

(a) Mandatory match fields required by OpenFlow 1.3 represented as a tree. A depth of 9 for L1-L4 matching can be seen, in relation to the depth column of table I.

(b) Directed acyclic control flow graph (CFG) representation of OF 1.3 match fields. Conditional jumps are shown as a solid arrow on true and dashed on false.

(c) Example of a BPF program matching on source mac address, IPv4 and IPv6 packets with a TCP source port of 22. The paths leading to a *MISS* have been omitted for clarity.

Fig. 1: OpenFlow 1.3 field specification and the associated transformation into a BPF program.

is still maintained in the kernel and doesn't show any signs of deprecation. On the contrary, the recent integration of extended BPF (eBPF) in Linux kernel 3.18 extends its use on non-networking parts of the kernel. An often forgotten fact is that BPF was defined as a "pseudo-machine" (currently called virtual-machine) with a goal to provide protocol independence through a simple but general instruction set designed for fast interpretation and execution.

In this paper, we present arbitrary matching for OpenFlow switches. We use BPF as the underlying packet matching instruction set and define a new OXM to match packets using BPF at the switches. We show that our proposed match reduces the number of flow entries, allows matching on fields and protocols not supported by current OpenFlow, and mitigates the use of the SDN controller for the classification of packets unmatchable by the switch. We present a prototype implementation using a software switch and finally highlight the feasibility of BPF matching for line-rate processing.

The remainder of the paper is structured as follows. Section II explains the relationship between BPF and OpenFlow match fields. Section III shows how BPF can be used for OpenFlow matching and presents the extension to OpenFlow match fields we used. Section IV gives an overview of our prototype system consisting of a high-level match language, an OpenFlow controller, and a software switch. Section V evaluates the feasibility of BPF for line-rate matching considering memory requirements and processing speed. Finally, section VI concludes the paper.

II. BPF for OpenFlow

Since 1992, the BPF-based packet filters have become essential to build network services. BPF has been used for traffic monitoring [6], network engineering [7] and intrusion detection [8]. The most popular use-case today is in *tcpdump* where filter expressions are compiled to BPF bytecode and executed in BPF engines such as in *libpcap*. In the last 15 years, many extensions to BPF have been released and used. For example, xPF [9] provides packet filtering for low-cost, high-speed networking, while BPF+ [10] extends BPF to a general packet filter framework consisting of a filter program translator, a byte code optimizer, a byte code safety verifier and a just-in-time assembler for increased performance. Swift [11] provides dynamic, high-speed packet filtering. To the best of our knowledge, we are the first to use BPF for packet matching in OpenFlow networks.

One of the original design criteria of BPF has been to only allow forward jumps, resulting in a non Turing-complete language that can be synthesized into a directed acyclic Control Flow Graph (CFG). Limiting to only forward jumps, prevents loops and therefore allows a worst-case execution time to be calculated, necessary for realtime packet filtering and classification. Furthermore, CFGs can be directly mapped to code for a register machine suitable for modern processors and allowing a single parse per layer, therefore reducing the number of memory accesses. Capping memory accesses is in itself important since they can be the limiting factor on how fast packets can be matched [12]. Therefore, with its realtime memory and processing properties as well as protocol and platform independence, BPF is a good fit for packet classification. However, current implementations of BPF are designed for packet filtering at an end-host rather than on network devices and therefore match on layer L2 and above (not L1), preventing a full translation from OpenFlow match fields to BPF.

Figure 1a shows the graph of all mandatory match fields between L1 and L4 defined in OpenFlow 1.3 specification and necessary for any OpenFlow compliant switch implementation. This graph also shows the mutual exclusion between IPv4 and IPv6 header fields depending on the Ethernet type. As this figure only shows the mandatory fields, it matches the minimum depth presented in Table I. In order to translate

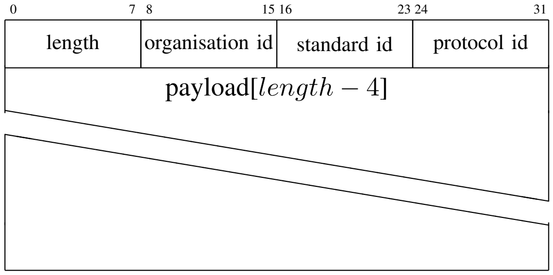


Fig. 2: Proposed Layer 1 structure containing its overall length, identifiers (organisation, standard and protocol id) and payload.

the OpenFlow match fields into a BPF program, the simplest approach is to convert the match fields from Figure 1a to a CFG structure that can be subsequently transformed to BPF. Figure 1b shows the translation from OpenFlow to a CFG using tcpdump's BPF filter syntax[2] with each node representing a packet field and each edge a transition, depending on the comparison result. The two leaf nodes *MISS* and *MATCH* represent the return state of the program. Finally, in Figure 1c we show a BPF program matching on all packets from Ethernet source 12:34:56:78:90:12, with IPv4 or IPv6 at L3, TCP at L4 and a source port of 22.

To implement the match shown in Figure 1c using OpenFlow, two separate rules are required: one for IPv4 and one for IPv6, instead of the single BPF rule. Furthermore, if we wanted to match traffic on, e.g., all system ports (all source or destination ports between 0 and 1024), a single BPF match using *jlt 1024* for the source and destination ports would be required, instead of 2048 individual OpenFlow flow entries (1024 matching on source and 1024 matching on destination ports). Requiring so many flow entries would use most of the entries in a traditional ToR switch preventing any additional rules. Finally, if one needed to match TTL-expiring packets in order to, e.g., detect loops in the network, standard OpenFlow match fields are not an option, since there is no match in the protocol for this particular field. However, a single BPF filter containing only 6 instructions can be used.

III. Design

In this Section, we define a L1 structure that can be prepended to the normal L2 packet for arbitrary packet classification using a BPF program. Then, we present the design of an experimental OXM field for OpenFlow used by the controller to send BPF programs to switches.

A. *L1 matching with BPF*

BPF was originally designed for packet filtering on a single interface, with only the packet content passed to the BPF engine. Therefore, no information is available about the packets' logical or physical input ports (L1). To allow packet classification using BPF and the translation from OpenFlow matches, L1 information must be readily accessible by the instruction set. The main challenge at this stage is to allow L1 matching to be arbitrary the same way L2+ is. Without an arbitrary L1 matching structure, the same limitation as OpenFlow's will emerge requiring constant updates to support emerging protocols and requirements.

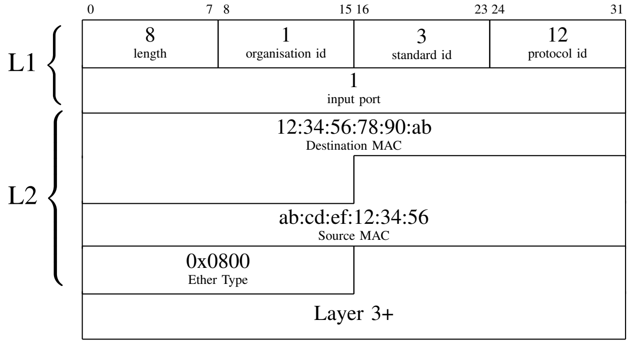
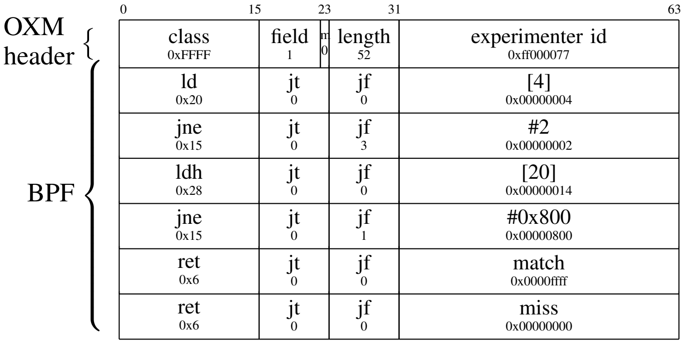


Fig. 3: Example packet structure passed in the BPF engine. The packet arrived on port 1 over a 1000BASE-T link.

To use L1 information with BPF, we have considered two options: the first one is to use the extended addressing mode of *ld* and *ldx* instructions; and the second is to use the normal addressing scheme for L1+ layers instead of L2+. Using the extended addressing mode, BPF can load additional information to the accumulator or index register of the engine. BPF extensions have been used in the Linux kernel to provide additional information to the filter such as the length of the packet, processor identifier, and the interface index. This approach is suitable for the Linux kernel since accessing L1 information for packet filtering is unlikely as a single filter cannot match packets from multiple interfaces. However, for a packet classifier, the interface which the packet arrived at as well as other L1 information are necessary to establish a forwarding decision. For instance, in a learning switch, the source MAC address is associated with the switch's physical input port. Therefore, L1 information is as critical as the layers above and using two different addressing modes for L1 and L2+ is counter-intuitive. Our proposed approach is to pass to the BPF engine not only the L2+ packet but all the layers including L1, allowing the same instruction set and engine to be used for packet classification.

Since there is no formal specification of the L1 layer, a proposed format is shown in Figure 2. This structure allows fast BPF execution by aligning the header on 32 bits (the size of a word in BPF) as well as providing the length. The length is the number of bytes of the L1 structure and can be loaded in the index register using *ldxb 0*, and therefore allow the L2 layer to be directly accessed using the index register as the offset. The identifiers for L1 have been split to an *organisation*, *standard*, and *protocol* identifier, each being a single byte wide. The structure and length of the L1 payload are associated to the identifier and relate to the information available at the particular physical layer. Since most of the widely used L1 protocols have been specified by the IEEE 802 LAN/MAN Standards Committee (e.g., IEEE 802.3x and 802.11x), our design choice was to map IEEE 802 Standards with organisation id of 1. Similarly, other organisation ids could be associated with, e.g., the International Organization for Standardization (ISO), the International Telecommunication Union (ITU), etc. Subsequently, the standard id refers to a standard within the organisation such as, e.g., 3 for IEEE 802.3 Ethernet, 11898 for

[2] http://www.tcpdump.org/manpages/pcap-filter.7.txt



OXM header	class 0xFFFF	field 1	m 0	length 52	experimenter id 0xff000077
BPF	ld 0x20	jt 0		jf 0	[4] 0x00000004
	jne 0x15	jt 0		jf 3	#2 0x00000002
	ldh 0x28	jt 0		jf 0	[20] 0x00000014
	jne 0x15	jt 0		jf 1	#0x800 0x00000800
	ret 0x6	jt 0		jf 0	match 0x0000ffff
	ret 0x6	jt 0		jf 0	miss 0x00000000

Fig. 4: Example BPF experimenter OXM

ISO CAN bus, or ITU 992.1 for ADSL. Finally, the protocol id would be a variant such as, e.g., *an* for 10GBASE-T (802.3an) or B for ADSL over ISDN (xDSL Annex B).

In OpenFlow, the only mandatory L1 field is the switch input port. This is logical for wired Ethernet, but it is not suitable in a wireless environment where one might be interested in, e.g., a particular channel. Using the payload field of the L1 structure, either the input port for Ethernet or a channel number for WiFi can be matched upon depending on the identifiers (organisation, standard and protocol ids). Using these 3 identifiers, it is possible for a BPF program to match on all Ethernet or WiFi packets by loading and comparing bytes 2 (organisation id) and 3 (standard id). Using the protocol id, it is possible to match on specific link types, e.g., a 10GBASE-T link or ranges of protocol revisions such as 802.3-2015 protocol amendments (802.3bk/bj/bm). An example packet structure passed on to the BPF engine is shown in Figure 3. The example shows a 1000BASE-T (802.3ab) frame received on port 1 from source ab:cd:ef:12:34:56 to 12:34:56:78:90:ab carrying an IPv4 payload.

B. OpenFlow OXM

From OpenFlow 1.2, the protocol can be extended to support additional matching capabilities. A Type-Length-Value (TLV) structure, called OpenFlow eXtensible Match (OXM), can be used to identify a specific field using a class and field identifier as well as a value to match against and optional bitmask. This approach provides a consistent way to use the OpenFlow protocol between switches with different supported match fields. Until version 1.2, the match structure had to be modified both in the switch and controller. If either one had a modified structure, the protocol at both ends would be different and therefore incompatible. OXM has been used for example in SensorFlow [13] to transfer Concatenated Attribute-Value (CAV) pairs for sensor nodes and for IPv6 support [14].

Our OXM field has been designed and implemented to transfer a compiled, platform-independent BPF program from the controller to the switch. The structure of the field is shown in Figure 4, containing an OXM header and 6 BPF instructions as the payload. A single BPF instruction takes 8 bytes: 2 bytes for the opcode, 1 byte for each *true* and *false* jump pointers, and 4 bytes for a generic multi-use argument. Since the length of an OXM payload in the current OpenFlow specification is stored in a byte and includes the size of the experimenter header $(255 - 4)$, our OXM field has a fixed size of 248 bytes which can hold a BPF program of up to 31 instructions (as a comparison, the BPF program shown in Figure 1c consists of 20 instructions). This limitation of 255 bytes is for a single oxm_field within an ofp_match structure. Therefore, using the current OpenFlow protocol, it is possible to send larger BPF programs by splitting them into fragments the size of a single match field. The OpenFlow specification states that for a given oxm_class, oxm_field, and oxm_hasmask, the length must be constant to simplify parsing in software. In order to support this, the payload sent containing the BPF instructions is padded with zeros up to 248 bytes.

IV. Prototype implementation

In this Section, we present a prototype implementation of the proposed system. The prototype is publicly available from our GitHub repositories[3][4][5].

A. Match syntax

BPF programs can be written using an assembly-like language as shown in Figure 4 with tools such as bpf_asm included in the Linux kernel's networking tools. However, writing a filter this way can be complicated and requires a deep understanding of the packet structure at each layer. Therefore, these filters are usually compiled from a higher, more natural language, such as the filter syntax used by tcpdump or WireShark. For instance, using tcpdump, the filter "ip src 10.0.0.1" is compiled into a BPF machine language containing instructions similar to Figure 1c.

Since we added L1 information to the packet structure for classification purposes, existing filter compilers must be modified to offset the load operations accessing packet content by the length of the L1 header. For instance, in tcpdump where packets are filtered at L2, a BPF program matching on the source MAC must read bytes 6 to 11. With the L1 header, reading the source MAC address of a packet carried over an Ethernet link can be done by reading bytes 8 to 13 (shifted by 8, 4 for the header and 4 for the input port). We have written a tool on top of tcpdump to automatically shift the memory location of the load instructions and to extend the syntax to add l1_length, organisation_id, standard_id, protocol_id and input_port. This tool is also able to translate OpenFlow match rules to a filter using tcpdump syntax and subsequently compile it into a BPF program. This translation and compilation process follows the three subfigures of Figure 1.

B. Controller design

We used the Ryu controller [15] as it supports recent OpenFlow specifications (up to OF 1.4) and enables fast prototyping in Python. We have implemented the appropriate converting methods for our OXM field from and to OpenFlow's wire protocol using Python's *pack* and *unpack* methods. A new class called *BPFProgram* has been defined to represent the BPF program in a Python data structure containing tuples of the opcode, two jump pointers (true and false) and the argument. In the example below in Code 1, we show a simple

[3] Openflow to BPF: https://github.com/simon-jouet/ofbpftranslator
[4] Software switch: https://github.com/czivar/ofsoftswitch13
[5] Controller: https://github.com/czivar/ryu

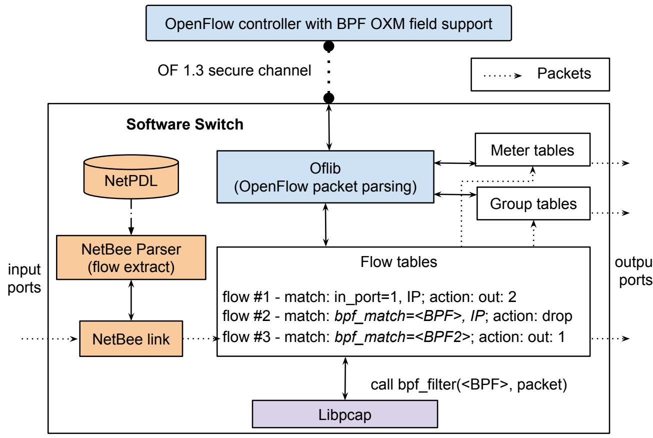

OpenFlow controller with BPF OXM field support
OF 1.3 secure channel
Packets
Software Switch
NetPDL
Oflib
(OpenFlow packet parsing)
Meter tables
Group tables
NetBee Parser
(flow extract)
input
ports
Flow tables
flow #1 - match: in_port=1, IP; action: out: 2
flow #2 - match: bpf_match=<BPF>, IP; action: drop
flow #3 - match: bpf_match=<BPF2>; action: out: 1
output
ports
NetBee link
call bpf_filter(<BPF>, packet)
Libpcap

Fig. 5: Architecture of our BPF-enabled software switch, based on the OpenFlow 1.3 Software Switch (OfSoftSwitch) [16].

BPFProgram that has two match conditions: the packet needs to come from input port 2 (line #2 loads the appropriate word and #3 compares it) and it needs to be an IP packet (line #4 and #5) to reach the match (line #6). This BPFProgram can be set as a parameter to Ryu's *OFPMatch()* structure and installed using a standard OpenFlow-compliant FlowMod message.

```
1  BPFProgram((
2    ( 0x20, 0, 0, 0x00000004 ), # load word
3    ( 0x15, 0, 3, 0x00000002 ), # in_port: 2
4    ( 0x28, 0, 0, 0x00000014 ), # load half word
5    ( 0x15, 0, 1, 0x00000800 ), # IPv4 (0x800)
6    ( 0x6,  0, 0, 0x0000ffff ), # MATCH
7    ( 0x6,  0, 0, 0x00000000 ), # MISS
8  ))
```

Code 1: BPFProgram sent by the Ryu controller. Figure 4 shows the OXM structure of the same BPF program.

C. Switch instrumentation

Our prototype switch is based on Ericson's OpenFlow 1.3 Software Switch [16], an open-source, user-space software switch implemented in C and built on top of Stanford's OpenFlow reference switch. It is designed for fast experimentation purposes with the latest OpenFlow versions. The switch, as seen in Figure 5, has been extended to use BPF filters for packet matching by calling *libpcap*'s user-space *bpf_filter()*. The original switch, as shown in the architecture, relies on NetBee, a powerful packet processing library that parses L2+ packets from input ports defined by NetPDL [17], an XML-based protocol description language.

The first major modification in the switch was altering *oflib*, the library converting to and from OpenFlow's wire protocol to understand and parse our new OXM match field described in Section III-B and store the BPF program at the flow entry's match structure. Also, *Ofdatapath*, the actual implementation of the switching logic has been modified to pass packets to the BPF classifier if they are matched with a flow that contains a BPF match. A BPF engine takes a BPF program (which can be found in a match structure of the flow entry in our case) and a linear pointer to the packet. Since we require L1 information in the BPF engine (as described in Section III-A) we prepend L1 information to the L2+ packet. Currently, L1 information attached to the packets contains only the input port.

TABLE II: Comparing different match scenarios using OpenFlow match fields and BPF OXM match fields.

	OF 1.3 match fields		**BPF OXM**		
Match	**# Flow entries**	**TCAM space (B)**	**# Flow entries**	**# BPF inst.'s**	**RAM space (B)**
in_port	1	4	1	4	32
all fields	1	88	1	34	272
system ports	2048	8192	1	15	120
in_port exclusion	23	92	1	4	32
TTL expiry	-	-	1	6	48

V. Results

In this Section, we provide details about the performance of our proposed match scheme compared to OpenFlow's standard match fields. We also provide an experimental evaluation of different BPF engines.

A. Flow entries and memory usage

The first two match scenarios in Table II are *in_port* and *all fields* that can be expressed using one flow entry with both matching structures (OpenFlow and BPF OXM). The *in_port* matches packets on a specific input port. As shown, using the OpenFlow standard match field requires only 4 bytes instead of 32 bytes required for the BPF program. This memory difference is a consequence of the arbitrary matching as both the BPF instructions and matching data must be stored. Rule *all fields* uses each available OpenFlow field to match (9 fields, cf. Figure I). As shown, the TCAM space is still significantly lower with OpenFlow match fields, as our BPF match rule needs 34 instructions to perform the same match.

Matches *system ports* and *in_port exclusion* present scenarios when using OpenFlow match fields result in excessive number of flow entries. Rule *system ports* matches on all ports between 0-1024 for either destination or source. Using OpenFlow match fields, 2048 separated entries are required using 8192 bytes of TCAM space. With our BPF OXM fields, a BPF program with 15 instructions using 120 bytes of memory can be used. Rule *in_port exclusion* is a match on every traffic not arriving on a specific port, such as filtering every traffic where input port is not 1. As OpenFlow match fields don't support exclusion, 23 flow entries should be set that match on all other ports on a 24 port switch. Using BPF OXM, one flow entry is enough using 4 instructions. It is clearly shown that range and exclusion matching are extremely inefficient with standard OpenFlow match fields. Finally, the *TTL expiry* match rule mentioned in Section II is an example of the variety of match scenarios that cannot be expressed with current OpenFlow match fields. To match on the IP-TTL field, our BPF OXM can be used with 6 instructions taking 48 bytes of memory space.

Even when the memory requirement of BPF programs is higher than an equivalent OpenFlow match, a simple high-speed RAM can be used instead of a TCAM as there is no

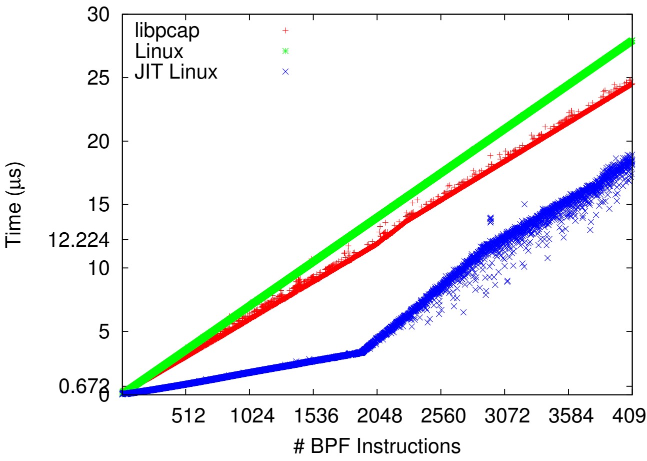


Fig. 6: Performance of different BPF engines.

need for ternary logic. TCAMs have various drawbacks, they require more transistors than traditional RAM and therefore are expensive, requiring an excessive chip area, a lot of power to operate and become slower as they get larger.

B. BPF engines

In order to use BPF programs for packet classification, it is necessary for a switch implementation to be able to execute them at line-rate. Figure 6, shows the performance benchmark of three different and widely used BPF engines on a commodity machine. The performance of each engine is evaluated as the time taken to execute BPF programs with increasing number of instructions. Both the libpcap and Linux engines are interpreters, with libpcap slightly outperforming the kernel interpreter. The BPF JIT compiler added in as part of the Kernel 3.0 significantly outperforms both interpreters.

In the figure the time axis has two markers at $0.672\mu s$ and $12.224\mu s$ representing the time at which minimum and maximum size packets must be classified to achieve line-rate on a 1Gbps link. The libpcap interpreter can execute up to 108 and 2090 instructions on minimum and maximum-sized packets, respectively, the Linux interpreter 95 and 1790, and finally 415 and $\approx$ 3100 for the Linux JIT engine. In Table II we have shown that an OpenFlow rule using every field can be translated to only 34 instructions, therefore OpenFlow equivalent matches as well as more complex ones can be achieved without degrading the line-rate performances.

VI. Conclusion

Today's OpenFlow switches use fixed number of fields to match packets with flow entries. Even though the number of matchable fields has increased with every version of the protocol, it still only covers a subset of the packet headers. Current matching is limited to only wildcards or exact match that can result in extremely large number of flow entries required for simple matches such as port ranges. Moreover, matching is limited to the standard Internet protocols (Ethernet, IP, TCP/UDP, etc) and there is no option to use arbitrary protocols or Layer 1 matching.

In this paper, we have argued for adding a platform-independent instruction set to OpenFlow, capable of matching packets on any field, layer or protocol. We proposed to use the Berkeley Packet Filters (BPF) instruction set for packet classification instead of continuously extending match fields. We presented a Layer 1 structure to allow arbitrary physical layer matching and extend OpenFlow to more than Ethernet networks. Finally, we have prototyped the proposed approach on an OpenFlow software switch and evaluated the memory and processing requirements to achieve line-rate classification.

Acknowledgments

This work was supported in part by the UK Engineering and Physical Sciences Research Council (EPSRC) projects EP/L026015/1 and EP/L005255/1. The authors would like to thank Eder Leao Fernandes for his insight on OfSoftSwitch, and Markus Koetter for his work on benchmarking BPF engines.

References

[1] 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.

[2] J. Naous, D. Erickson, G. A. Covington, G. Appenzeller, and N. McKeown, "Implementing an OpenFlow Switch on the NetFPGA Platform," ser. ANCS. ACM, 2008.

[3] Pica8 – Open Networking, "OpenFlow data center, a case study," 2014. [Online]. Available: http://nvirters.org/wp-content/uploads/2014/03/MeetUp0227B.pdf

[4] 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*, Jul. 2014.

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

[6] S. Kornexl, V. Paxson, H. Dreger, A. Feldmann, and R. Sommer, "Building a time machine for efficient recording and retrieval of high-volume network traffic," in *SIGCOMM*. USENIX Association, 2005.

[7] C. Partridge, A. C. Snoeren, W. T. Strayer, B. Schwartz, M. Condell, and I. Castineyra, "Fire: Flexible intra-as routing environment," *JSAC*, 2001.

[8] M. Roesch *et al.*, "Snort: Lightweight Intrusion Detection for Networks," in *LISA*, vol. 99, no. 1, 1999, pp. 229–238.

[9] S. Ioannidis, K. Anagnostakis, J. Ioannidis, and A. Keromytis, "xpf: packet filtering for low-cost network monitoring," in *HPSR*, 2002.

[10] A. Begel, S. McCanne, and S. L. Graham, "Bpf+: Exploiting global data-flow optimization in a generalized packet filter architecture," in *SIGCOMM*, vol. 29, no. 4. ACM, 1999, pp. 123–134.

[11] Z. Wu, M. Xie, and H. Wang, "Swift: A fast dynamic packet filter." in *NSDI*, vol. 8, 2008, pp. 279–292.

[12] W. Jiang, Q. Wang, and V. Prasanna, "Beyond tcams: An sram-based parallel multi-pipeline architecture for terabit ip lookup," in *INFOCOM 2008*, April 2008.

[13] T. Luo, H.-P. Tan, and T. Q. Quek, "Sensor openflow: Enabling software-defined wireless sensor networks," *Communications Letters, IEEE*, vol. 16, no. 11, pp. 1896–1899, 2012.

[14] R. R. Denicol, E. L. Fernandes, C. E. Rothenberg, and Z. L. Kis, "On IPv6 support in OpenFlow via Flexible Match Structures," *OFELIA/CHANGE Summer School*, 2011.

[15] Ryu SDN controller. [Online]. Available: http://osrg.github.io/ryu/

[16] E. L. Fernandes and C. E. Rothenberg, "OpenFlow 1.3 software switch," In SBRC'2014, 2014. [Online]. Available: https://github.com/CPqD/ofsoftswitch13

[17] F. Risso and M. Baldi, "NetPDL: an extensible XML-based language for packet header description," *Computer Networks*, vol. 50, no. 5, 2006.