FlowBlaze:
stateful packet processing in hardware

Paper #130

May 31, 2018

1 Introduction

This Appendix contains additional details about the FlowBlaze system and its evaluation.

- Sec. 2 explains the rationale behind the qualitative evaluation of existing systems as presented in the paper;
- Sec. 4 describes additional details about the hardware implementation;
- Sec. 6 describes the public traffic traces used in the evaluation of FlowBlaze;
- Sec. 3 describes the implemented network functions;
- Sec. 5 provides some more details on how FlowBlaze is programmed.

2 Existing systems

This sections explains the qualitative evaluation we provide in the paper's Table 1, which we replicate here for convenience.

	High Perf	**State Scal**	**Ease**	**Expresiv**
General programming frameworks				
HDL	√	√	×	√
HLS [**?**]	×	√	√	√
ClickNP [10]	-	√	-	√
Match-action abstractions				
P4 [5]	-	-	√	-
Domino [13]	√	×	-	√
OpenState [4]	√	√	√	×
FAST [12]	×	√	√	×

Table 1: Qualitative comparison of stateful abstractions. A dash means a requirement is only partly met.

HDLs can express *any* function, since they are very flexible, and the programmer has full control of the system to provide both performance and state scalability. However, the programmer has to be a hardware expert, therefore R3 is unmet.

HLS can be as simple as porting existing high-level code and "recompile" it for a hardware target. However often this leads to poor performance. Therefore, hardware targeted code optimization is required. This requires rethinking the system with the hardware platform constraints and features in mind. Thus, either R1 or R3 are unmet. We report only R1 in the table since we look at the qualitative evaluation from the perspective of a programmer that has little hardware expertise.

ClickNP can provide both high-performance and ease of use, if the function can be fully described only with existing Click's elements. Unfortunately, Click elements are usually capturing functionality that is very much network protocol related (e.g., IP checksum or TTL), but does not provide primitive actions that can be flexibility re-combined. Thus, when Click elements cannot capture the required function, new elements have to be designed using HLS, which incurs the already mentioned issues. For this reason R1 and R3 are only partially met.

P4 cannot describe stateful functions out of the box, but only relies on platform dependent features (also called 'externs' in P4 terminology). As such, R1, R2 and R4 all depend by the selected platform and not by P4 itself.

Domino meet both R1 and R4 since it compiles programs that run at line rate and can use a restricted C version to describe stateful functions. However, Domino does not provide implementations for a hash-table that handles collisions, which limits scalability of the per-flow state memory (R2). Also, the programmer functions may be rejected, requiring sessions of trial and error, which complicate the design process. For this reason R3 is only partially met.

OpenState meets all the requirements but R4, since it uses regular FSM, which for instance do not support arithmetic operations on the kept state. Also, it suffers from state explosion as described in the paper.

FAST has similar constraints, furthermore we are not aware of any high-performance hardware implementation that shows the feasibility of their architecture in hardware, for which we consider R1 unmet.

3 Use Cases Description

This section provides a brief description of the use cases implemented with FlowBlaze and mentioned in NSDI 2019 spring submission #130.

By no means this document should be considered as a blue-print to fully implement production-ready use cases, nor it works as technical documentation for the mentioned implementations. Rather, it should help the reader in understanding how the EFSM-based abstraction is used in FlowBlaze.

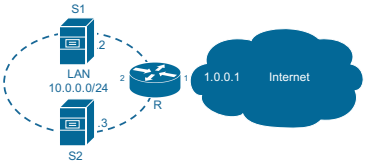
The use cases can be combined by means of sequential composition. In such case, one can use information from a previous EFSM (element) by writing such information in a packet's metadata. A next element can read the metadata to use the information.

The use cases are summarized in the following table:

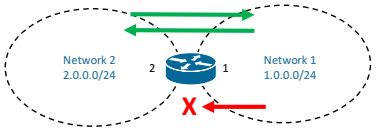
Use case	Description	Entries	Registers
Server Load Balancer	Selects a server for a new flow and remembers that selection for the following packets.	2,2	0+1,1
UDP Stateful Firewall	Tracks connection state and only allows flows in one direction if a corresponding flow in the opposite direction was first seen	5	0
Port Knocking Firewall	Stateful firewall application that allows communication only if a sequence (of length L) of packets addressed to an ordered list of predefined closed ports is received	$L+2$	0
Flowlet load balancer	Distributed flowlet load balancing with in-band network telemetry. In this paper we consider a 3:2 leaf:spine ratio	2, 4, 9	1, 0+2, 0+4
Traffic policer	Traffic policer based on per flow sliding windows	4	$2+2$
Big Flow Detector	Simple application that marks flows after a given threshold of sent packets	3	1
SYN flood Detection and Mitigation	Identifies hosts that generate an unexpected amount of traffic (threshold based)	4	1
Switch Paxos (Coordinator)	Porting on FlowBlaze of the P4-based implementation of the Paxos protocol	1	$0+1$
Switch Paxos (Acceptor)	Porting on FlowBlaze of the P4-based implementation of the Paxos protocol	3	$3+1$
Dynamic NAT	Assigns a port number to new connections and performs the corresponding packet header rewriting	3, 4	1+1, 2
TCP optimistic ACK detection	Monitors that the TCP segments carry a correct ACK number	7,6	7,2
TCP super spreader detection	Monitor the number of TCP connections initiated by each host	8	1
vEPC's subscriber's quota verification	Implementation of the monitoring and accounting function of a 4G Evolved Packet Core architecture	9	1
In-network KVS cache	Offloads a Key-Value Store application, e.g., memcached, working as an in-network cache	6	2

Use case 1: Server Load Balancer

In this use case, we implement a load balancer function that assigns TCP connections to a set of web servers in a private LAN, in a round-robin fashion. Directing traffic to a given web server is as easy as configuring a static NAT rule. Nonetheless, the complexity of the use case is in the necessity of keeping track of two different states. A global state is used to store a counter value that is checked for each new flow received by the switch and it is used to enforce the round-robin selection. The second one is a per-flow state that binds each established flow to one of the available destination servers and assure the forwarding consistency. That is, the destination web server is selected when the first connection's packet is received, and all the remaining packets for that flow should be forwarded to that same web server. The actual implementation refers to the topology in Figure 1a and consists of the following FlowBlaze elements.



(a) Usa case 1 reference topology



(b) Usa case 2 reference topology

Figure 1

Element 0 (Stateless)

#	inport	ip.src	ip.dst	tcp.sport	tcp.dport	pkt actions
1	2	10.0.0.2	*	80	*	GOTO 3
2	2	10.0.0.3	*	80	*	GOTO 3
3	1	*	1.0.0.1	*	80	GOTO 1

Element 1
Flowkey = ip.src, tcp.src
G0: counter

#	state	state actions	pkt actions
1	0	counter += 1; STATE = 1 (20 s);	metadata = counter; GOTO 2;
2	1	*	GOTO 2;

Element 2
Flowkey = ip.src, tcp.src
R0: value of the counter for a particular flow

#	state	state actions	pkt actions
1	0	counter = metadata; STATE = 1 (20 s);	GOTO 3;
2	1	*	metadata = counter; GOTO 3;

Element 3 (Stateless)

#	inport	ip.src	metadata	pkt actions
1	1	*	*0	ip.dst = 10.0.0.2; OUTPUT(2);
2	1	*	*1	ip.dst = 10.0.0.3; OUTPUT(2);
3	2	10.0.0.2	*	ip.src = 1.0.0.1: OUTPUT(1);
4	2	10.0.0.3	*	ip.src = 1.0.0.1; OUTPUT(1);

Use case 2: UDP Stateful firewall

This use case implements a stateful firewall that allows bidirectional communication between two networks only if initiated from one of the two sides. This is a typical use case for stateful

firewalls like Linux netfilter/iptables for isolating DMZ networks and ”protected” networks. In this case, we implement a UDP stateful firewall, that only checks for bidirectional connectivity. A more complex implementation with TCP connection tracking is presented in the paper. In iptables, assuming that the communication is allowed from a network behind eth1 to a network behind eth2 only if initiated by a host behind eth2, this would be realized by the following rules:

```
iptables -A FORWARD -i eth2 -o eth1 -j ACCEPT
iptables -A FORWARD -i eth1 -o eth2 -m state --state ESTABLISHED -j ACCEPT
```

In the implementation described below (which refers to the topology in Figure 1b), the packets in state 0 coming from port 1 are dropped. Packets from port 1 to port 2 are allowed only if associated to a flow initiated from port 2 to port 1.

Element 0
Flow key = biflow(5-tuple)

#	state	inport	state actions	pkt actions
1	0	1		GOTO 1;
2	0	2	STATE = 1 (20 s)	GOTO 1;
3	1	2	STATE = 1 (20 s)	GOTO 1;
4	1	1	STATE = 2 (20 s)	metadata = 1; GOTO 1;
5	2	*		metadata = 1; GOTO 1;

Element 1 (Stateless)

#	metadata	inport	pkt actions
1	0	1	DROP;
2	1	1	OUTPUT(2);
3	*	2	OUTPUT(1);

Use case 3: Port Knocking Firewall

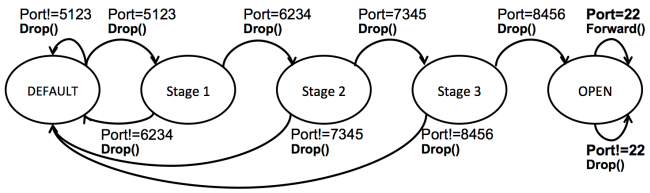
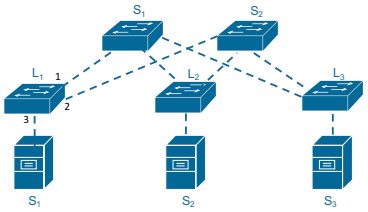
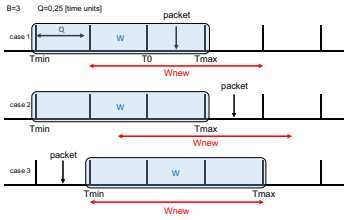


Figure 2: Port Knocking mealy machine

This use case implements a port knocking firewall, a well-known method for opening a port on a firewall. An IP host that wants to establish a connection (say an SSH session, i.e., port 22) delivers a sequence of packets addressed to an ordered list of pre-defined closed ports, say ports 5123, 6234, 7345 and 8456. Once the exact sequence of packets is received, the firewall opens port 22 for the considered host. Before this stage, all packets (including the knocking ones) are dropped. This example can be easily implemented with the Mealy Machine illustrated in figure 2. Starting from a DEFAULT state, 0, each correctly knocked port will cause a transition to a



(a) Distributed Load Balancing use case topology



(b) Traffic Policer approach

Figure 3

series of three intermediate states, 1, 2 and 3, until a final OPEN state, 4, is reached. Any knock on a different port will reset the state to DEFAULT. When in the OPEN state, only packets addressed to port 22 will be forwarded; all remaining packets will be dropped, but without resetting the state. This use case is implemented by the following elements.

Element 0 (Stateless)
This element is an arbitrary stateless pre-filtering stage that decides which traffic goes to the port knocking stage.

Element 1
Flow key: ip.src

#	state	tcp.dport	state actions	pkt actions
1	0	5123	STATE = 1 (1 s);	DROP;
2	1	6234	STATE = 2 (1 s);	DROP;
3	2	7345	STATE = 3 (1 s);	DROP;
4	3	8456	STATE = 4 (60 s);	DROP;
5	4	22	*	GOTO 2;
6	*	*	STATE = 0;	DROP;

Element 2 (Stateless)
Element 2 is an arbitrary final stateless forwarding stage

Use case 4: Distributed load balancing

This use case is an implementation of a distributed load balancing algorithm similar to the one proposed in [1] and designed for multi-rooted topologies like “leaf-spine” and “fat-tree”, widely used in data-centers. Every leaf node dynamically evaluates the best next hop for each destination in order to forward each incoming flow through the current best path. The best path can be verified with different strategies: static per flow, dynamic per flow, per flowlet (like in CONGA). The path utilization metric is derived by all switches by synchronising each other’s with special signalling messages (namely probes). Probes are propagated through the core network in order to monitor the links’ utilization. In our case, they are piggybacked on data traffic and probe

generation frequency depends on the number of flow packets. The following FMA elements implements switch L1 in the topology show in Figure 3a.
The EWMA primitive is an instruction executed by the ALUs, which approximates an exponential weighted moving average of the function parameter (in this case the packet len). In other words, the EWMA instruction returns as output an estimation of the traffic rate on a given input port.

Element 0 (Stateless)

#	inport	eth.dst	eth.type	pkt actions
1	1	*	MPLS	GOTO 3
2	2	*	MPLS	GOTO 3
3	1	*	IPv4	GOTO 1
4	2	*	IPv4	GOTO 1
5	3	leaf2	*	metadata=2; GOTO 3
6	3	leaf3	*	metadata=3; GOTO 3

Element 1
Flow key: inport
R0: counter
C0: $R0 == probeperiod$

#	conditions	state actions	pkt actions
1	counter == probe_period	counter = 0;	PUSH_MPLS; metadata = 1; GOTO 2;
2	counter != probe_period	counter += 1;	GOTO 2;

Element 2
Flow key: -
G0: EWMA estimate relative to port 1
G2: EWMA estimate relative to port 2

#	inport	metadata	state actions	pkt actions
1	1	1	G0 = ewma(pkt.len);	metadata=inport-1$\|\|G0\|\|G1$; GOTO 4;
2	2	1	G1 = ewma(pkt.len);	metadata=inport-1$\|\|G0\|\|G1$; GOTO 4;
3	1	0	G0 = ewma(pkt.len);	OUTPUT(3);
4	2	0	G1 = ewma(pkt.len);	OUTPUT(3);

Element 3
Flow key: (5-tuple)
G1: utilization on port 1 to destination leaf 2

G2: utilization on port 1 to destination leaf 3
G3: utilization on port 2 to destination leaf 2
G4: utilization on port 2 to destination leaf 3
C0: G1 $\leq$ G3
C1: G2 $\leq$ G4

#	conditions	state	inport	metadata	eth.type	mpls.tc	state actions	pkt actions
1	*	0	1	*	MPLS	2	G1 = mpls.label	*
2	*	0	2	*	MPLS	2	G3 = mpls.label	*
3	*	0	1	*	MPLS	3	G2 = mpls.label	*
4	*	0	2	*	MPLS	3	G4 = mpls.label	*
5	p1u_to_2 $\leq$ p2u_to_2	0	3	2	IPv4	*	STATE = 1 (RTT);	OUTPUT(1);
6	p1u_to_2 $>$ p2u_to_2	0	3	2	IPv4	*	STATE = 2 (RTT);	OUTPUT(2);
7	p1u_to_3 $\leq$ p2u_to_3	0	3	3	IPv4	*	STATE = 1 (RTT);	OUTPUT(1):
8	p1u_to_3 $>$ p2u_to_3	0	3	3	IPv4	*	STATE = 2 (RTT);	OUTPUT(2);
9	*	*	3	*	IPv4	*	*	b[41:42]=STATE; GOTO 4;

Element 4 (Stateless)

#	metadata[0]	metadata[41:42]	pkt actions
1	0	0	GROUP({SET_MPLS(tc=1,label=b[1:20]), OUTPUT(inport) }; {SET_MPLS(tc=1, label=b[21:40]),OUTPUT(2)}); POP_MPLS; OUTPUT(3);
2	1	0	GROUP({SET_MPLS(tc=1,label=b[21:40), OUTPUT(inport) }; {SET_MPLS(tc=1, label=b[1:20]), OUTPUT(1)}); POP_MPLS; OUTPUT(3);
3	*	1	OUTPUT(1)
4	*	2	OUTPUT(2)

Use case 5: Per flow traffic policer

This use case implements a single rate token bucket with burst size B and token rate 1/Q, where Q is the token inter arrival time. Since in the current FlowBlaze architecture the update functions are performed after the condition verification, we cannot update the number of tokens in the bucket based on packet arrival time before evaluating the condition (token availability) for packet forwarding. For this reason, we have implemented an alternative approximated algorithm based on a sliding time window (Figure 3b). For each flow, a time window $W(Tmin - Tmax)$ of length BQ is maintained to represent the availability times of the tokens in the bucket. At each packet arrival, if arrival time $Tnow$ is within W (Case 1), at least one token is available and the bucket is not full, so we shift W by Q to the right and forward packet. If the arrival time is after $Tmax$ (Case 2), the bucket is full, so packet is forwarded and W is moved to the right to reflect that $B-1$ tokens are now available ($Tmin = Tnow - (B-1) * Q$ and $Tmax = Tnow + Q$).

Finally, if the packet is received before $Tmin$ (Case 3), no token is available, therefore W is left unchanged and the packet is dropped. Upon receipt of the first flow packet, we make a state transition in which we initialize the two registers: $Tmin = Tnow - (B-1) * Q$ and

$Tmax = Tnow + Q$ (initialization with full bucket).

Element 0 (Stateless)
This element is an arbitrary stateless pre-filtering stage that decides which traffic is processed to the policer element.

Element 1 Flow key: ip.src
R0: Tmin
R1: Tmax
G0: B*Q
G1: Q
C0: Tnow $>$ R0
C1: Tnow $\leq$ R1

#	conditions	state	state actions	pkt actions
1	Tnow $>$ Tmax	0	Tmin = Tnow - B*Q; Tmax = Tnow + Q; STATE = 1;	GOTO 2;
2	Tmin $<$ Tnow && Tnow $\leq$ Tmax	1	Tmin += Q; Tmax += Q; STATE = 1;	GOTO 2;
3	Tnow $>$ Tmax	1	Tmin = Tnow - B*Q; Tmax = Tnow + Q;	GOTO 2;
4	Tnow $\leq$ Tmin	1		DROP;

Element 2 (Stateless)
Element 2 is an arbitrary final stateless forwarding stage

Use case 6: Big Flow Detector

This application marks packets belonging to a given flow if the flow has generated more than 100 packets.

Element 0
Flow key: 5-tuple
R0: packet counter
C0: R0 $\geq$ 100

#	conditions	state	state actions	pkt actions
1	pkt_cnt $<$ 100	0	pkt_cnt += 1;	TX;
2	pkt_cnt $\geq$ 100	0	pkt_cnt += 1; STATE = 1 (10 s);	ip.tos = 1; TX;
3	*	1		ip.tos = 1; TX;

Use case 7: SYN flood Detection and Mitigation

This use case realizes a simple SYN flood detection and mitigation. This application counts the number of TCP SYN packets sent by each host, and drops such packets if a given threshold of SYNs per second is reached. Since we are interested in counting the TCP SYNs per host, the flow key definition for this element is the IP source address. The register R0 is used to save the counter of TCP SYNs for each flow, i.e., for each host. A hard timeout of 1 second brings the state machine back to the wait state (i.e., deleting the corresponding flow context's entry). After the mitigation phase (60 seconds of ban) the host is allowed to transmit packets again.

Element 0 (Stateless)
This element is an arbitrary stateless pre-filtering stage that decides which portion of traffic goes into the flood detection element (e.g. SYN packets from a given input port).

Element 1
Flow key: ip.src
R0: SYN counter
C0: R0 $>$ threshold

#	conditions	state	state actions	pkt actions
1	*	0	SYN_cnt += 1; STATE = 1 (1 s);	TX;
2	SYN_cnt $\leq$ threshold	1	SYN_cnt += 1;	TX;
3	SYN_cnt $>$ threshold	1	STATE = 2 (60 s);	DROP;
4	*	2		DROP;

Use case 8: Switch Paxos

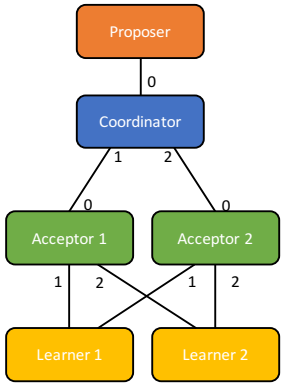


Figure 4: Switch Paxos reference topology

This use case implements the switch paxos mechanism as described in [8]. The reference topology is shown in Figure 4. As in the original work, only coordinators and acceptors are implement in programmable data plane.

Coordinator
Flow key: -
G0: Paxos instance number (initialized at 0)

#	inport	state actions	pkt actions

1	0	paxos_inst_num += 1;	paxos.type = 2A;
			paxos.swid = paxos_inst_num;
			FLOOD;

Acceptor
Flow key: paxos.inst
G0: switch ID
R0: round
R1: vround
R2: value
C0: paxos.rnd $>$ R0

#	inport	conditions	paxos.type	state	state actions	pkt actions

1	0	*	1A	0	round = paxos.round;
					vround = paxos.vround;
					value = paxos.value;
					STATE = 1;
						paxos.type = 1B;
						paxos.swid = sw_id;
						FLOOD;

2	0	*	1A	1	round = paxos.round;	paxos.type = 1B;
						paxos.swid = sw_id;
						paxos.vround = vround;
						paxos.value = value;
						FLOOD;

3	0	paxos.rnd $>$ round	2A	*	round = paxos.round;
					vround = paxos.round;
					value = paxos.value;
						paxos.type = 2B;
						paxos.swid = sw_id;
						FLOOD;

Use case 9: Dynamic NAT

Assigns a source port number to new connections and performs the corresponding packet header rewriting.

Element 0
First element is used to generate/retrieve the NATted port, port is stored inside the metadata and the packet is passed to the second stage.

Flow key: 5-tuple
R0: assigned source port
G0: next source port

#	state	inport	state actions	pkt actions

1	0	internal	assigned_src_prt=next_src_prt;
			next_src_prt += 1;
			STATE = 1;
				metadata = assigned_src_prt;
				GOTO 1;

2	0	external		metadata = tcp.dst; GOTO 1;
3	1	internal	*	metadata = assigned_src_prt; GOTO 1;

Element 1
NAT operations

Flow key: metadata
R0: internal IP src address
R1: internal TCP src port

#	state	inport	state actions	pkt actions

1	0	internal	int_ip = ip.dst;
			int_port = tcp.src;
			STATE = 1;
				ip.src = public_ip;
				tcp.src = metadata;
				GOTO 2;

2	0	external	*	DROP;

3	1	internal	*	ip.src = public_ip;
				tcp.src = metadata;
				GOTO 2;

4	1	external	*	ip.dst = int_ip;
				tcp.dst = int_port;
				GOTO 2;

Element 2 (Stateless)
Stateless MAT forwarding element

Use case 10: TCP optimistic ACK detection

This use case verifies that the TCP segments carry a correct ACK number, preventing an attacker to get data faster by sending optimistic ACKs. We assume that a connection tracking function, like the one presented in the paper, is placed before the element that implements this use case. The connection tracking function ensures that the connection is correctly established and that sequence numbers are in the correct range.

Furthermore, we assume that the SYN-ACK packets are always forwarded to the element described in this section, and that the packet metadata is set to contain the value TCP.SEQ + TCP PAYLOAD LEN for each packet received by the element (but the SYN-ACK).

Element 0
Stateful forwarding element

Flow key: biflow(5-tuple)
R0: A's IP (the connection opening side)
R1: A's expected maximum ack number
R2: B's expected maximum ack number

C0: R0 == ip.src
C1: R2 >= TCP.ack
C2: meta(TCP.seq+LEN) ¿= R1
C3: R1 >= TCP.ack
C4: meta(TCP.seq+LEN) ¿= R2

#	state	conditions	tcp.flags	state actions	pkt actions

1	0	*	SYN/ACK	a_ip = IPdst;
				a_expMaxAck = TCP.ack;
				b_expMaxAck = TCP.seq+1;
				STATE=1
					TX;

2	1	a_ip = IPsrc && b_expMaxAck >= TCP.ACK && meta(SEQ+LEN) >= a_expMaxAck	*	a_expMaxAck = meta(SEQ+LEN)	TX
3	1	a_ip = IPsrc && b_expMaxAck >= TCP.ack && meta(SEQ+LEN) <a_expMaxAck	*		TX
4	1	a_ip != IPsrc && a_expMaxAck >= TCP.ack && meta(SEQ+LEN) >= b_expMaxAck	*	b_expMaxAck = meta(SEQ+LEN)	TX
5	1	a_ip != IPsrc && a_expMaxAck >= TCP.ack && meta(SEQ+LEN) <b_expMaxAck	*		TX
6	1	a_ip == IPsrc && b_expMaxAck <TCP.ack	*	STATE=2	DROP
7	1	a_ip != IPsrc && a_expMaxAck <TCP.ack	*	STATE=2	DROP
8	2	*	*	*	DROP

Use case 11: TCP super-spreader detection

This use case counts the number of opened TCP flows from a source IP address to detect if it is a a super-spreader (i.e. it opens too many connections). If a source is a super-spreader, new connections from that host are dropped.

Element 0 (Stateless)
Stateless MAT forwarding element

#	inport	pkt actions
1	*A*	metadata = ip.src; OUTPUT(B); GOTO 1;
2	*B*	metadata = ip.dst; OUTPUT(A); GOTO 1;

Element 1
Flow key: metadata(metadata)
R0: in-flight SYNs

C0: R0 $<$ threshold

#	state	conditions	tcp.flags	state actions	pkt actions
1	0	in_flight_SYN $<$ threshold	SYN	in_flight_SYN += 1;	TX;
2	0	in_flight_SYN $<$ threshold	FIN	in_flight_SYN -= 1;	TX;
3	0	in_flight_SYN $\geq$ threshold	SYN	in_flight_SYN += 1; STATE = 1;	DROP;
4	0	in_flight_SYN $\geq$ threshold	FIN	in_flight_SYN -= 1;	TX;
5	1	in_flight_SYN $\geq$ threshold	SYN	in_flight_SYN += 1;	DROP;
6	1	in_flight_SYN $\geq$ threshold	FIN	in_flight_SYN -= 1;	TX;
7	1	in_flight_SYN $<$ threshold	SYN	in_flight_SYN += 1; STATE = 0;	TX;
8	1	in_flight_SYN $<$ threshold	FIN	in_flight_SYN -= 1; STATE = 0;	TX;

Use case 12: vEPC's subscriber's quota verification

Implementation of the monitoring and accounting data plane function for a 4G Evolved Packet Core's Packet Gateway.

Element 0 (Stateless)
Stateless MAT forwarding element

#	inport	pkt actions
1	*A*	metadata = ip.src; OUTPUT(B); GOTO 1;
2	*B*	metadata = ip.dst; OUTPUT(A); GOTO 1;

Element 1
Flow key: metadata
R0: credit left
C0: R0 $>$ credit alert threshold
C1: R0 $<$ 0

#	state	conditions	state actions	pkt actions
1	0	credit $>$ alert_thrs && credit ≥ 0	credit -= *pkt.len*;	TX;
2	0	credit $\leq$ alert_thrs && credit ≥ 0	credit -= *pkt.len*; STATE = 1;	TX;
3	0	credit $\leq$ alert_thrs && credit < 0	credit -= *pkt.len*; STATE = 2;	DROP;
4	1	credit $>$ alert_thrs && credit ≥ 0	credit -= *pkt.len*; STATE = 0;	TX;
5	1	credit $\leq$ alert_thrs && credit ≥ 0	credit -= *pkt.len*;	TX;
6	1	credit $\leq$ alert_thrs && credit < 0	credit -= *pkt.len*; STATE = 2;	DROP;
7	2	credit $>$ alert_thrs && credit ≥ 0	credit -= *pkt.len*; STATE = 0;	TX;
8	2	credit $\leq$ alert_thrs && credit ≥ 0	credit -= *pkt.len*; STATE = 1;	TX;
9	2	credit $\leq$ alert_thrs && credit < 0	credit -= *pkt.len*;	DROP;

Use case 13: In-network KVS cache

Offloads a Key-Value Store application, e.g., memcached, working as an in-network cache. In this example keys and values have a fixed size of 8 bytes each.

Element 0
Flow key: kvs.key[0-63]
R0: kvs.val[0-31]
R1: kvs.val[32-63]

#	state	kvs.type	state actions	pkt actions
1	0	GET	*	TX;
2	0	SET	*	TX;
3	0	RESP	R0 = kvs.val[0-31]; R1 = kvs.val[31-64]; STATE = 1	TX;

4	1	GET	*	kvs.val[0-31] = R0;
				kvs.val[31-64] = R1;
				tmp = mac.src;
				mac.src=mac.dst;
				mac.dst = tmp;
				TX;

5	1	SET	R0 = kvs.val[0-31]; R1 = kvs.val[31-64];	TX;
6	1	RESP	R0 = kvs.val[0-31]; R1 = kvs.val[31-64];	TX;

4 Packet Processing Optimization

The implementation of the EFSM's enabling functions requires the evaluation of both algebraic conditions and of boolean operations. When the number of conditions is bigger than one and involves multiple operands, it becomes challenging to perform such operations within the time budget of a single clock cycle [13]. To minimize the number of cycles required by our pipeline, we decided to re-use the EFSM table for the execution of the enabling functions' boolean operations.

In particular, the conditions block performs the algebraic comparisons between two operands (recall Figure **??**) selected using a crossbar. The output of the comparison is stored in a bit vector $\vec{C}$, where each set bit corresponds to a true value in the corresponding comparison operation. The vector is generated for each packet traversing the pipeline, even if for such packet processing the enabling functions evaluation is not required. The condition vector is then passed to the EFSM table, which can select the required subset of conditions by performing ternary matching operations on it. In other words, the EFSM table is implemented using a TCAM, and the vector $\vec{C}$ is added to the packet header fields and state label that are already contained in an EFSM table's entry. In sum, we optimize packet processing by using a few bits of TCAM space to speed-up and simplify the evaluation of the enabling functions.

5 Programming FlowBlaze

This section provides some more insights about programming a FlowBlaze data plane.

FlowBlaze is executed either by the NIC or by a software forwarder in a server. In both cases, it works as a bump-in-the-wire system, i.e., FlowBlaze processing applies to all the packets flowing in the NIC or in the software forwarder. By NIC, here we assume something like the NetFPGA, but also a SmartNIC such as Netronome or Mellanox processor-based NICs would work. For software forwarder we could have an eBPF/XDP program running in the NIC's software driver, a virtual switch, such as mSwitch or OVS, etc.

In the large majority of the cases, a programmer will find a FlowBlaze pipeline already configured in the NIC or software forwarder. That is, like in an OpenFlow switch, there will be already a parser defined and a set of predefined actions. The programmer has essentially two tasks: (i) provide a flow definition, i.e., to tell how to build the flow key used to lookup in the flow context table; (ii) fill the EFSM table with the transitions and state/packet actions that implement the function. One way to perform such tasks is to extend an OpenFlow-like controller to define the flow key, and where the flow entries written to the forwarder are the EFSM table entries instead of regular MAT entries.

To further clarify how FlowBlaze works, we can have a look at the next listing, which reports a pseudo-eBPF implementation of a software forwarder that implements FlowBlaze. This forwarder would be configured by the Controller to implement the required function.

The *process_packet* function receives a buffer containing a packet, and processes it. Lines 5-8 implement packet parsing, to populate protocol header structures that can be then used later by the program logic. Using a subset of the extracted fields, a flow_key can be built in Lines 18-23. Here recall that the flow_key is used to lookup in the flow context table. In Line 26, a port field used for lookup in the EFSM table is instead extracted. Line 32 extracts the flow context (current state plus any flow register) from the flow context table. Notice that the current state, together with the port field, builds the lookup key for the EFSM table. Such lookup is performed in Line 41. As a result, we get the matching entry that contains the state update and packet forwarding actions for this packet. E.g., the next state is set in Line 46, while the packet forwarding actions are applied by the switch statement of Line 49.

```
1  int process_packet(void* data, void* data_end)
2  {
3
4  // Parser
5  [...]
6  This code section is written once, to extract packet headers,
7  or could be automatically generated, e.g., from a P4 program.
```

```
[...]
// End parser

[...]

// Extraction of the flow keys and match fields,
// this part is programmable at runtime in FlowBlaze
flow_ctx_extract:{

        // flow key is 5tuple biflow
        flow_key.ip_proto = ip->protocol;
        flow_key.ip_src = ip->saddr;
        flow_key.ip_dst = ip->daddr;
        flow_key.l4_src = l4->source;
        flow_key.l4_dst = l4->dest;
        biflow(&flow_key);

        // The EFSM table matches also on the ingress port
        EFSM_key.matches.eth_port = ingress_ifindex;
}

// Flow context table lookup
flow_ctx_lookup:{
    // flow_ctx_table is an eBPF map
        flow_leaf = bpf_map_lookup_elem(&flow_ctx_table, &flow_key);
        EFSM_key.state = flow_leaf->state;

        [...]
}

// EFSM table lookup
EFSM_lookup:{

        struct EFSM_table_leaf *EFSM_leaf =
                bpf_map_lookup_elem(&EFSM_table,&EFSM_key);
        [...]

        // update state for this flow EFSM
        next_state = EFSM_leaf->next_state;

        // read and apply state/packet actions
        switch (EFSM_leaf->act[0].identifier){
                        case [...]
                        [...]
                        default:
                                [...]
                }
}

[...]
```

```
58
59 }
```

Of course, the above listing is a simplified version of an actual forwarder, missing a number of required implementation's parts, such as the logic to perform new flow context entry insertions.

6 Traffic Traces

We selected 2 traffic traces from carrier networks (CHI15, MAWI15) and two from university datacenters (UNI1, UNI2). CHI15 is a 1h long trace captured from a backbone link in the US [6]. The packet size has a bimodal distribution, with 30% of packets having sizes below 80B, and 50% close to 1500B. MAWI15 is a trace with 15 minutes of traffic captured from a link between Japan and the US and also showing a bimodal distribution of packet sizes, but this time with 45% of packets being smaller than 80B [11]. We selected these two traces as representative of typical conditions on such links, since the packet size and flows distributions are similar to the majority of the other CAIDA and MAWI traces recently published [7, 9]. UNI1 and UNI2 are taken from an edge switch at two different university datacenters [3, 2]. Both present bimodal distributions for the packet sizes, but UNI1 has 30% of packets of size smaller than 80B, an additional 20% below 300B and the rest above 1400B. Instead, UNI2 has 50% of the packets smaller than 100B and the rest close to 1500B. For all the traces, we measured the number of concurrently active flows (using a 10s idle timeout) and the maximum rate of new flows per ms (see Table **??**).

References

[1] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese. Conga: Distributed congestion-aware load balancing for datacenters. *SIGCOMM Comput. Commun. Rev.*, 44(4):503–514, Aug. 2014.

[2] T. Benson. Data set for IMC 2010 data center measurement. `http://pages.cs.wisc.edu/~tbenson/IMC10_Data.html`.

[3] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *ACM SIGCOMM IMC*, ACM SIGCOMM IMC '10, 2010.

[4] G. Bianchi, M. Bonola, A. Capone, and C. Cascone. Openstate: Programming platform-independent stateful openflow applications inside the switch. *ACM SIGCOMM CCR*, 44(2):44–51, 4 2014.

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

[6] CAIDA. The CAIDA UCSD anonymized internet traces - chicago 2015-02-19. `http://www.caida.org/data/passive/passive_2015_dataset.xml`.

[7] CAIDA. The CAIDA UCSD statistical information for the CAIDA anonymized internet traces. `http://www.caida.org/data/passive/passive_trace_statistics.xml`.

[8] H. T. Dang, M. Canini, F. Pedone, and R. Soulé. Paxos made switch-y. *SIGCOMM Comput. Commun. Rev.*, 46(2):18–24, May 2016.

[9] R. Fontugne, P. Borgnat, P. Abry, and K. Fukuda. MAWILab: Combining diverse anomaly detectors for automated anomaly labeling and performance benchmarking. In *ACM CoNEXT '10*, ACM CoNEXT '10.

[10] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, and P. Cheng. Clicknp: Highly flexible and high-performance network processing with reconfigurable hardware. In *ACM SIGCOMM '16*.

[11] MAWI. MAWILab traffic trace - 2015-07-20. `http://www.fukuda-lab.org/mawilab/v1.1/2015/07/20/20150720.html`.

[12] M. Moshref, A. Bhargava, A. Gupta, M. Yu, and R. Govindan. Flow-level state transition as a new switch primitive for sdn. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, ACM HotSDN '14, pages 61–66. ACM, 2014.

[13] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking. Packet transactions: High-level programming for line-rate switches. In *ACM SIGCOMM '16*, ACM SIGCOMM '16, pages 15–28. ACM, 2016.