# P4 language extensions for stateful packet processing

Angelo Tulumello
University of Rome Tor Vergata, Italy.

***Abstract*—The P4 language is the *de facto* standard to define how programmable switches must process packets. P4 is extensively employed in datacenter scenarios as it permits to support a wide set of network applications. However, P4 does not provide a clear description of stateful processing, especially on handling per-flow states. This paper extends both the P4 language and the P4 software switch (bmv2) with novel stateful primitives with a clear representation of per-flow states. Furthermore, we exploit these new functionalities by implementing a datacenter network function that implements a scalable tunneling mechanism. The developed network function differentiates the top talker flows directly handled by the switch and the other flows that are instead managed in a slow path involving specific network devices with large connection tables.**

## I. Introduction

In the last years, data plane programmability gained much attention both in academia and in the industry. Recently [1] [2] have been introduced to provide the ability to reprogram the dataplane specifying packet protocol parsing and reconfigurable match/action tables. The P4 language [1] became the de-facto standard to program the behavior of switching ASICs pipelines, reaching the technological and industrial maturity by providing line-rate packet processing at Terabit speeds.

A further step to enhance data plane programmability is the addition of primitives that permit the implementation of stateful functions that need to maintain per-flow states inside the fast path [3]–[5]. Moreover, many recent research works [6]–[9] increasingly employ heavy per-flow states in the switching devices and SmartNICs. The detailed specification of the set of stateful primitives able to support a meaningful set of use cases requires a thorough review and evaluation of the design challenges introduced by the continuously growing network application requirements. Although the P4 language permits to define an array of registers to store data, a clear representation of per-flow states and other stateful primitives is not covered yet.

In this paper, we propose a set of stateful extensions to the P4 language to provide a clear representation of per-flow states to implement stateful network functions in the dataplane. The remainder of this paper is structured as follows. We present the stateful extensions to P4 in section II by analyzing the four main additions to the P4 language: (i) the stateful table and the per-flow state machines, responsible for maintaining and processing per-flow states; (ii) global register stacks, a new data structure implementing a global stack of registers; (iii) stateful per-flow timers, i.e. the implementation of programmable timers that can be configured by a stateful table; (iv) rate meters, to support the computation of per-flow rates to be used as states, e.g. to take decisions based on them; (v) minor extensions. In section III we present the description and implementation of a Flow Caching use case that exploits the stateful primitives described in section II. Such an application implements a flow caching mechanism that forwards the packets of top talker flows directly in the dataplane acting as a cache. A DEMO of this use case has been presented in [10].

## II. P4 stateful extensions

In this section, we present the main P4 extensions realized to describe the stateful functionalities and their implementation in BMv2.

### A. Stateful Table and Per-Flow State Machines

The Stateful Table is the core of the stateful packet processing as it represents the unit containing: (i) the specification of a *flow* with flow keys, i.e. how to extract a flow from a set of packet header fields; (ii) the description of per-flow states as a list of registers that can only be accessed by the specified flow; (iii) a graph which represents the processing to handle the state transitions. In standard P4, this level of flow isolation cannot be clearly rendered, since the registers are global and accessible to the whole pipeline.

**Stateful Table**. A Stateful Table is an exact match unit that is invoked by the *.apply()* method and is responsible for (i) retrieving the programmable flow context associated to a key extracted from packets and (ii) executing a per flow extended finite state machine (EFSM) defined in the related State Graph component (which may result in both packet modifications and flow context updates).

```
stateful_table table_0 {
        flow_key[0] = {<FIELD_LIST_0>};
        flow_key[1] = {<FIELD_LIST_1>};
        ...
        flow_ctx = ctx_0( (bit<8>) state_size);
        graph = graph_0(<GRAPH_PARAMETERS>);
        timer = timer_0(timer_granularity, <
             GRAPH_PARAMETERS>);
        idle_timeout = TIMEOUT;
        eviction_policy = CACHE_POLICY;
        size = TABLE_SIZE;
    }
```

Listing 1. stateful_table definition in P4

In Listing 1 the P4 code that defines the `stateful_table` is presented. The format is similar to the definition of standard P4 tables, but contains the following additions:

- the *flow_key[i]* keywords represents the flow keys as a list of fields (packet header fields or metadata);
- the *flow_ctx* is the reference to the Flow Context, defined as a `struct` of fields, as happens for metadata headers in standard P4. The `state_size` parameter configures the bitwidth of the `flow_state` field, always present in the Flow Context structure;
- the *graph* takes the reference to the control block implementing the processing to be applied in the stateful stage for data packets;
- the *timer* takes the reference to the control block implementing the processing to be applied in the stateful stage for timer events, treated as special packets;
- the *idle_timeout* is a numerical parameter that configures the ageing time for each entry
- the *eviction_policy* configures the eviction policy algorithm to be used for the insertion of stateful table entries in the case of stateful table is full;
- finally the *size* configures the table size in terms of number of entries

**Flow Context structure.** As mentioned above, the Flow Context is a data structure similar to the header and metadata fields. The only difference is that it takes a mandatory parameter that has to be provided to configure the state size. In fact, the “state” field is hidden in the P4 code but always present in the Flow Context data structure.

```
state_context ctx_0(bit<8> state_size) {
  bit<48> timestamp;
  bit<32> bursts_number;
  . . .
}
```

Listing 2. Flow Context definition in P4

Being a metadata header, the *state_context* struct is inserted inside the *header_type* and *header* arrays. The former contains the description of the structure of the header, i.e. the list of fields with their respective bit widths while the latter configures the instantiation in the headers stack of a header type.

In the implementation, when the stateful table is executed, this header is set to “valid” and invalidated at the end of the stateful processing (end of the state graph).

**State Graph control block.**

The state graph contains the Extended Finite State Machine implementation of the Stateful Table. It can be seen as a special P4 control block that takes the following arguments:

- the *state_context* (mandatory), i.e. the flow context extracted from the lookup of the key in the packet;
- the *parsed_headers_t* (mandatory), i.e. the parsed headers from the packet;
- the *local_metadata_t* (optional);
- the *standard_metadata_t* (mandatory);
- the *queue_metadata_t* (optional), i.e. the metadata header that we added to have the queue depth information for all the egress queues in the ingress pipeline;
- other parameters (optional) like global register arrays, meters and stacks.

```
state_graph graph_0(state_context flow_ctx,
                        parsed_headers_t hdr,
                      local_metadata_t local_metadata,
                      standard_metadata_t
                           standard_metadata,
                      register<bit<16>> reg_1,
                      stack<bit<16>> stack_1,
                        ...) {

  state start {
          . . .
          transition s1;
      }

  state s1 { . . . }

  state s2 { . . . }
 }
```

Listing 3. State Graph definition in P4

The state keyword is similar to the one used in the P4 parsers. Logically it is equivalent to the condition

$$if(flow\_ctx.state == START)\{...\}$$

, and in JSON it is actually compiled as that conditional. The transition keyword is also similar to the keyword used to change states in the P4 parser. Logically it is equivalent to the statement $flow\_ctx.state = S1$. With respect to the parser, the stateful graph is not a directed graph, so transitions can also occur in previous states. As a note, if we think to the parser, the notation could be misleading. A packet can be in one state at a time. When a state transition is executed, the flow state is updated at the end of the state machine execution.

**Stateful Table *apply()* method.** As the standard P4 tables, also stateful tables are executed in the pipeline with the .apply() method. The difference is that the .apply() method for stateful table can take an argument to select the flow key to use for lookup. An example of the .apply() method usage is displayed in Listing 4.

```
apply {
        if (standard_metadata.ingress_port == LAN) {
            stage_0.apply(0);
        } else {
            stage_0.apply(1);
        }
    }
}
```

Listing 4. Table Apply example in P4

## B. Global Register stacks

Global Stacks are similar to P4 Register Arrays. Indeed, a global stack is a data structure that supports also the push and pop operations on an array of registers. This feature is important for many use cases that require to select values from a pool of resources. For instance, a Network Address Translation (NAT) application would need to assign L4 ports from a set of allowed ports to be used for the translation. With a register stack, a new flow can remove a value from the ports pool, such that subsequent flows cannot use the same port value.

The instantiation of the Stack occurs by providing the size of the data that will be stored in the stack registers and the stack size. It implements two methods to insert and pick up values from the stack.

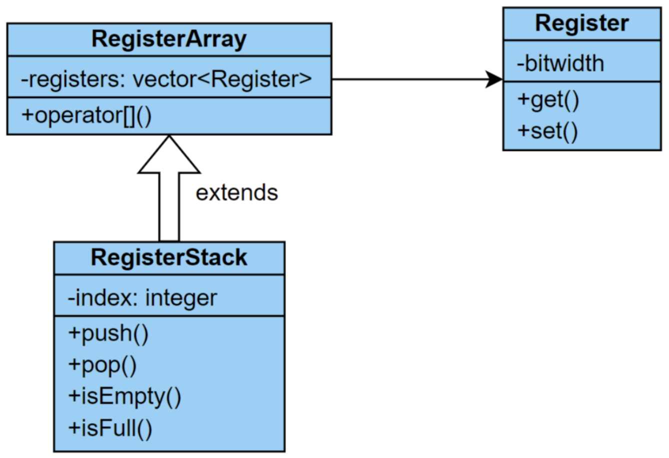

Fig. 1. Global Registers structure

- The **push** operation inserts a value of type T inside the stack, if the stack is not full;
- The **pop** operation picks a value of type T from the stack, if the stack is not empty.

The management of the stack capacity is up to the programmer. If one of the above operations fail, the stack state would not be changed. The push and pop operations were added as new P4 primitives. The *isEmpty()* and *isFull()* methods are not exposed to the programmer and are respectively called in pop and push operations, preventing the pop from an empty stack and the push in a full stack.

A simple class diagram of register stacks implementation is displayed in Fig. 1.

```
//object
extern stack<T> {
     stack(bit<32> size);

     void push(in T value);
     void pop(out T value);
}

//instance
stack<bit<16>>(65536) ports;
```

Listing 5. Global stack definition in P4

## C. Stateful per-flow timers

Asynchronous events are an important feature to implement complex Network Functions like L4 protocols and proxies as well as functions that need to pace the rate of packets. With stateful per-flow timers, we define an interface to handle asynchronous events at a per-flow granularity.

The Timer Graph control block has the same structure of the State Graph control block. Basically, the only thing that changes is the number and type of parameters to feed into the control block:

- the *state_context* (mandatory), as in the state graph;
- the *standard_metadata_t* (mandatory);
- the *timer_metadata_t* (mandatory), representing the timer metadata header that will be enabled to timer packets (containing the current tick number);
- the *queue_metadata_t* (optional), i.e. the metadata header that we added to have the queue depth information for all the egress queues in the ingress pipeline;
- other objects (optional) like global register arrays, meters and stacks.

Then the *state/transition* mechanism is the same as in the state graph and should explicit all the states indicated in the state graph, related to the stateful table.

```
state_timer timer_0(state_context flow_ctx,
                      standard_metadata_t
                         standard_metadata,
                   timer_metadata_t timer_metadata,
                   queue_metadata_t queue_metadata,
                      register<bit<16>> reg_1,
                       ...)  {

 state start { . . . }

 state s1 { . . . }

 state s2 { . . . }
}
```

Listing 6. Timers State Graph definition in P4

Programmable per-flow timers are implemented as synchronous events triggered by an external timer thread. It is up to the stateful table the storage of the timeouts, that at each synchronous "tick" event must check whether a timer has expired or not.

```
if (timer.current_tick >= configured_timeout) {
   // timeout handling
}
```

Listing 7. Timeout handling

A per-flow timer instance is created for each stateful table that makes use of this service and the events generated by the timer instance are redirected to the stateful table, i.e. timer events cannot be propagated in the pipeline.

In the BMv2 implementation, we added a new thread for each stateful table that requests programmable timers. The timer thread operations are rather simple:

- It sleeps for a time equal to the granularity configured by the programmer;
- After the sleeping time, it constructs a "fake" packet and enables the timer metadata header field, setting the ticks value to the current tick;
- The packet is enqueued in the ingress pipeline thread. To discriminate it with a real packet, a reserved ingress port is assigned to "fake" packets representing a timer event.

In the ingress thread, we have to route the timer packet to the stateful table owning that specific timer thread. This is done by checking that the ingress port of the packet corresponds to the special port reserved to timer packets. We then retrieve the correct stateful table reference and trigger the execution of the stateful graph.

The execution of the timer event is applied to each entry in the Flow Table. The processing then is similar to the one applied to normal packets, with the following core difference: the timer packet does not have packet headers parsed, so it is not necessary to lookup the Flow Table (just cycle over all the entries). We then get the reference to the next node in

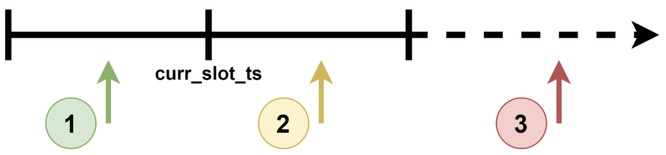

Fig. 2. EWMA time window logic

the stateful timer pipeline, that in most cases would be the conditional checking whether the packet is a timer packet or not. Finally, we process the nodes in the pipeline with the same while loop as in the *Pipeline::apply()* method.

## D. Rate meters

The programmable rate meter has been implemented using the same Meter interface used in P4 standard meters. In the BMv2 extensions, we specialized the meters implementation to support explicit metric calculations like rates. In fact, the standard meters are implemented as two-rate three color policers, giving as output the color (green, yellow, red) of the meter instance at runtime. Our implementation instead permits us to update the meter calculations with an input sample and read the meter current rate, as well as storing the result in a flow register.

For the implementation, we added a primitive to execute the meter and read the current rate that has the prototype showed in listing 8.

```
meter.execute(in bit<32> rate_block_id,
              in T sample,
              out T current_rate);
```

Listing 8. Timeout handling

The *rate_block_id* parameter selects the desired meter in the meter array; the sample parameter represents the input sample to be fed to the meter; the *current_rate* parameter is the output field in which the current rate should be stored.

**EWMA.** We chose to adopt the Exponentially Weighted Moving Average (EWMA) as the algorithm for rate calculation. It exponentially gives less weight to the previous samples. The formula describing the EWMA calculation can be written as follows:

$$rate(t) = sample(t) * a + rate(t - T_{update}) * (1 - a)$$

- *rate(t)* is the EWMA calculation at time t
- *sample(t)* is the current rate sample at time t
- *a* is the decaying factor, comprised in (0,1) and representing the "amount" of the exponential decay of previous samples
- *rate(t-1)* represents the previous EWMA calculation
- *T_update* is the frequency in which we calculate the EWMA

Since the meter is executed by packets, we cannot assure that the ewma calculation is triggered at the desired frequency. To emulate this feature, we opted for a "windowed" version of EWMA. In the meter memory we store:

- the sum of the samples for the current time window
- the "end" time of the current time window
- the ewma calculation result

The algorithm for EWMA calculation is designed as follows. At each execution of the meter we compare the timestamp in which the execution is triggered, with the timestamp of the current time window. We have three cases:

1) The current timestamp is less than the end of the current time window, i.e. the packet arrival occurred within the time window. In this case, we just add the current sample to the samples sum.
2) The current timestamp is greater than the end of the current time window but less than the end of the next window. In this case, we update the EWMA value for the previous window, initialize the sample sum with the current sample and move the "end" time of the current window to the next one.
3) The current timestamp is greater than the end of the next window, i.e. the packet arrived, at least, after one complete window with respect to the timestamp stored for the previous one. In this case, we simplify by setting the EWMA current result to 0 and by re-initializing the memory.

In Figure 2 a visual representation of the above described logic is displayed.

## E. Minor extensions

**Ageing mechanism for stateful table entries.** In BMv2 the ageing mechanism for table entries was already supported. Anyway, such configurable idle timeouts must be configured by the control plane for each entry added to a table. We needed instead the support of an ageing mechanism for each entry added to the stateful table with the same idle timeout value for each new entry added. The ageing mechanism already present in the BMv2 featured a thread for the ageing monitor of the table entries that periodically wipes the aged entries. We reused this thread and we implemented per-flow ageing by setting the time to live to an entry every time it is matched.

**Queue occupancy metadata.** Even if BMv2 already supported some type of queueing metadata, that information is available in the egress pipeline only. To have information about egress queues occupancy also in the ingress pipeline, we added another metadata header to support such information. This metadata header contains the queue occupancy of all the egress queues present in the target.

**Flow context and global storage create/read/update/delete from control plane.** We added the support for CRUD queries in the Flow Context of a stateful table in the *simple_switch_CLI* control plane tool.

## III. Use Case: Flow Cache

Figure 3 depicts the reference networking topology for the proposed use case. The datacenter hosts a number of tenants' Virtual Machines in Host Servers (HS) connected to Top of Rack switches in a VXLAN domain. A programmable switch (SW1) interconnects the HSes to some bare-metal applications (BMA) in an IP domain. Typically the BMAs implement

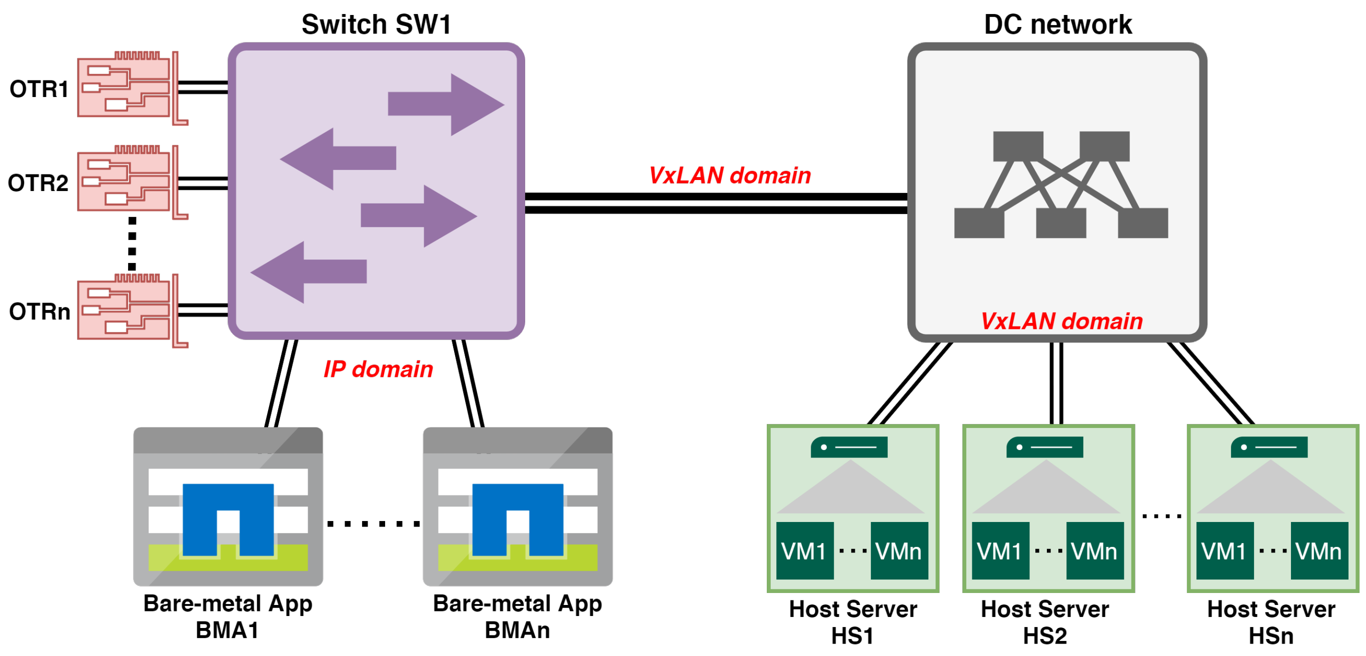

Fig. 3. Flow Cache Scenario

functions that are provided to the customer VMs as additional applications such as databases or distributed filesystems.

The communication from the VMs to the BMAs requires to decapsulate the packets in the VXLAN domain and forward the inner IP packet to the correct BMA. The return path should re-encapsulate the packet and bind the correct VXLAN tag to the incoming packet. This binding is maintained in a huge overlay VXLAN forwarding table that cannot be fully stored in the limited memory of the programmable switch.

A possible solution is to forward the packets from the BMAs directed to the VXLAN to an external device, called Overlay Tunnel Router (OTR). The OTR retrieves the overlay VXLAN forwarding information (*i.e.* the binding between the destination IP subnet and the VXLAN tag) in a local database and encapsulates the incoming packet accordingly. Since the OTRs have significant throughput, latency and memory requirements, they are currently implemented using ad-hoc FPGAs.

Since the "overlay routing lookup" is performed on all the packets coming from the BMAs, due to bandwidth requirements, the solution requires to use several OTRs. This increases the cost of the solution and occupies some ports of the switch for the connection to the OTRs.

A possible improvement to this solution is to develop a flow cache mechanism that permits to the switch to automatically learn the binding between the VXLAN tunnel and the destination IP address and store the most frequently queried entries. This permits to limit the queries to the OTRs, thus decreasing the number of required OTR instances, and results in an overall improvement in terms of latency and bandwidth availability. Furthermore, it frees some of the switch ports previously used by the OTRs. Unfortunately, such a solution requires to use several stateful features that are not currently supported in programmable dataplanes. Nevertheless, since the switch internal memory is limited, we implement a cache eviction policy (e.g. Least Frequently Used or Least Recently used) to clean unused entries. In particular, in our solution, packets are processed as follows:

1) The first packet from a VM is forwarded toward the BMA by the stateful switch (SW1).
2) The reply from BMA is received by SW1 that queries the local cache table. Since it is the first packet, the cache is empty and SW1 forwards the packet to the OTR.
3) The OTR encapsulates the packet into the proper VXLAN tunnel and forwards it to SW1. SW1 learns the association between the subnet containing the VM and the VXLAN tunnel exploiting some metadata headers in the data packet added by the OTR.
4) SW1 forwards the encapsulated packet to the VXLAN domain.
5) When BMA sends the second packet of the same flow to SW1, it queries the internal cache entry, retrieves the VXLAN tunnel binding and encapsulates the packet.

The P4 code implementing the described application is displayed in Listing 9, and the full implementation can be found in [11].

To conclude the description of this use case, it is worth discussing the case in which SW1 learns the tunnel while there are further packets that have already been sent to the OTR. If a new packet arrives, it could be sent out before the old packets coming from the OTR are processed, thus resulting in an out of order packet. To avoid this problem, SW1 should continue sending packets of the same flow to the OTR until there are no more packets in flight. This feature can be implemented by keeping track of packets in flight with a register and use the internal cache only when such counter is zero.

```
state_graph graph_0(state_context_t flow_ctx,
     parsed_headers_t hdr,  local_metadata_t
     local_metadata, standard_metadata_t
     standard_metadata) {
 state start {
   //learn if the pkt come from the OTR
   if (standard_metadata.ingress_port == OTR) {
     flow_ctx.vxlan_vni = hdr.vxlan.vni;
     flow_ctx.out_ip_dst = hdr.ipv4.dst_addr;
     flow_ctx.out_ip_src = hdr.ipv4.src_addr;
     flow_ctx.out_udp_src = hdr.udp.src_port;
     standard_metadata.egress_spec = DC;
     if (flow_ctx.in_flight != 0) {
       flow_ctx.in_flight = flow_ctx.in_flight - 1;
     }
     transition learnt;
   } else if (standard_metadata.ingress_port != OTR){
     fwd_to_otr();
     flow_ctx.in_flight = flow_ctx.in_flight + 1;}
 }
 state learnt {
   if (standard_metadata.ingress_port != OTR){
     flow_ctx.in_flight = flow_ctx.in_flight - 1;
     standard_metadata.egress_spec = DC;
   } else if (standard_metadata.ingress_port != OTR){
     if (flow_ctx.in_flight == 0) {
       encap_to_dc_network(flow_ctx);
     } else { //send packets to OTR to avoid OOO pkts
       fwd_to_otr();
       flow_ctx.in_flight = flow_ctx.in_flight + 1;}
   }
 }
}
```

Listing 9. P4 state graph for the flow cache use case

## IV. Discussion

The main contribution of this work focuses on the representation of stateful primitives as an extension to the P4 language, with the goal of proposing a cozy abstraction for stateful Network Functions. Another contribution is the addition of the missing features to the bmv2 P4 executor to assess the proposed extensions from a behavioral point of view. In fact, bmv2 is not meant to be a production grade software switch: it is a tool for developing and testing P4 applications.

Maintaining a large number of states in switching ASICs and SmartNICs while conciliating Terabit data rates and limited dataplane memory is a tackling problem in literature. Anyway, a number of research works proposed solutions to address this problem [5] [12]. Another challenge is the management of multiple timer events inside the dataplane while maintaining scalability. In [13] an FPGA implementation of a stateful calendar for asynchronous events is presented, demonstrating the feasibility of hardware timers in SmartNICs.

## V. Conclusion

Although P4 is a very expressive language to perform stateless packet processing in a programmable way, a specific abstraction tailored to handle per-flow states is missing. In this short paper we presented the definition and implementation of a possible way to define P4 language extensions tailored to stateful, per-flow packet processing. Moreover, we have shown a Flow Caching solution to enable the possibility to process directly in the fast path traffic from top talker flows in a datacenter scenario.

## References

[1] 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 Comput. Commun. Rev.*, vol. 44, no. 3, p. 87–95, Jul. 2014. [Online]. Available: https://doi.org/10.1145/2656877.2656890

[2] 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," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, p. 99–110, Aug. 2013. [Online]. Available: https://doi.org/10.1145/2534169.2486011

[3] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "Openstate: Programming platform-independent stateful openflow applications inside the switch," *ACM SIGCOMM Computer Communication Review*, vol. 44, pp. 44–51, 04 2014.

[4] M. Moshref, A. Bhargava, A. Gupta, M. Yu, and R. Govindan, "Flow-level state transition as a new switch primitive for sdn," ser. HotSDN '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 61–66. [Online]. Available: https://doi.org/10.1145/2620728.2620729

[5] S. Pontarelli, R. Bifulco, M. Bonola, C. Cascone, M. Spaziani, V. Bruschi, D. Sanvito, G. Siracusano, A. Capone, M. Honda, F. Huici, and G. Bianchi, "Flowblaze: Stateful packet processing in hardware," in *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'19. USA: USENIX Association, 2019, p. 531–547.

[6] G. Bianchi, M. Welzl, A. Tulumello, F. Gringoli, G. Belocchi, M. Faltelli, and S. Pontarelli, "Xtra: Towards portable transport layer functions," *IEEE Transactions on Network and Service Management*, vol. 16, no. 4, pp. 1507–1521, 2019.

[7] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "Hula: Scalable load balancing using programmable data planes," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2890955.2890968

[8] C. H. Benet, A. J. Kassler, T. Benson, and G. Pongracz, "Mp-hula: Multipath transport aware load balancing using programmable data planes," in *Proceedings of the 2018 Morning Workshop on In-Network Computing*, ser. NetCompute '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 7–13. [Online]. Available: https://doi.org/10.1145/3229591.3229596

[9] M. T. Arashloo, M. Ghobadi, J. Rexford, and D. Walker, "Hotcocoa: Hardware congestion control abstractions," in *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, ser. HotNets-XVI. New York, NY, USA: Association for Computing Machinery, 2017, p. 108–114. [Online]. Available: https://doi.org/10.1145/3152434.3152457

[10] A. e. a. Tulumello. Extending p4 to realize a scalable flow caching mechanism, 2021 p4 workshop 3.0,. [Online]. Available: https://opennetworking.org/wp-content/uploads/2021/05/2021-P4-WS-Angelo-Tulumello-Slides.pdf

[11] P4 flow cache github repository. [Online]. Available: https://github.com/axbryd/p4-flow-cache

[12] L. Zeno, D. R. K. Ports, J. Nelson, and M. Silberstein, "Swishmem: Distributed shared state abstractions for programmable switches," in *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, ser. HotNets '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 160–167. [Online]. Available: https://doi.org/10.1145/3422604.3425946

[13] S. Pontarelli, G. Bianchi, and M. Welzl, "A programmable hardware calendar for high resolution pacing," in *IEEE 19th International Conference on High Performance Switching and Routing, HPSR 2018, Bucharest, Romania, June 18-20, 2018*. IEEE, 2018, pp. 1–6. [Online]. Available: https://doi.org/10.1109/HPSR.2018.8850731