Virtual Queues for P4: A Poor Man's Programmable Traffic Manager

Hasanin Harkous[†], Chrysa Papagianni*, Koen De Schepper[†], Michael Jarschel[†], Marinos Dimolianis**, Rastin Pries[†]

[†]*Nokia Bell Labs*, firstname.lastname@nokia.com

**University of Amsterdam*, c.papagianni@uva.nl

***National Technical University of Athens*, mdimolianis@netmode.ntua.gr

***Abstract*—The advent of programmable network switch ASICs and recent developments on other programmable data planes (NPUs, FPGAs) drive the renewed interest in network data plane programmability. The P4 language has emerged as a strong candidate to describe a protocol independent datapath pipeline. With its supported architectures, the P4 language provides an excellent way to define the packet processing and forwarding behavior, while leaving other networking components such as the traffic management engine, to non-programmable fixed function elements, based on the capabilities of most programmable devices. However, network flexibility is essential to meet the Quality of Service (QoS) requirements of traffic flows. Thus, enabling programmable control for fixed-function elements like traffic management is crucial.**

Towards that end we propose the use of virtual queues in the P4 pipeline, investigate the application of virtual queue-based traffic management, and portability of the approach using different P4 programmable targets. Specifically, we focus on virtual queue based Active Queue Management (AQM) for congestion policing and meeting the latency targets of distinct network slices. The solution is compared to P4 built-in functionality for bandwidth management using meters, proving also that the additional dimensions of control are achieved without compromising the processing complexity of the solution.

***Index Terms*—Software Defined Networking, Programmable Data Plane, P4, Traffic Management, Performance Evaluation, SmartNIC.**

I. Introduction

Centralization of the network's control plane, through software defined networking, is an advantage for applications where changes in the forwarding state do not have strict real-time requirements and depend on the global network state. However, functionalities pertaining to traffic management, such as rate control and Active Queue Management (AQM) to reduce network congestion and/or scheduling to provide Quality of Service (QoS) and fairness etc., require local state information. In such cases, the support of programmable data planes can minimize the impact of latency and overhead caused by the intervention of a controller.

Data plane programmability implies the switch capability to expose the packet processing logic to the control plane to support systematic, fast and comprehensive reconfiguration [1]. Exploiting advances in networking, including hardware architectures (e.g., match tables [2], the P4 programming language [3], etc.), network devices may be reprogrammed in the field to parse custom protocols and execute custom functionality. However, while the P4 language and supported architectures provide an excellent way to define the packet processing and forwarding behavior, most programmable devices still have limited, non-programmable traffic management capabilities. The P4 open-source community has started working towards defining a programmable traffic manager, encompassing functionality such as packet scheduling, shaping, AQM, etc.

Making the traffic management logic programmable can provide significant benefits in terms of meeting QoS requirements (e.g. low latency communications), reduce the control load on the SDN controller(s) and corresponding network overhead, and ensure the required network flexibility through the introduction of primitives beyond forwarding. Furthermore, portability of such functions between the networking hardware and software stacks running in the cloud should be assured, to enhance network service agility.

To support Quality of Experience (QoE) for applications that make use of high link capacities (if available) while they can adapt to lower rates but still need low latency (e.g., AR/VR, Cloud Gaming), it is important that the network reports capacity limits before queuing latency starts to build up or packets get dropped. Optimized AQM and congestion control are required to manage queuing latency in bottleneck links. Due to architectural limitations, many existing P4 targets cannot make the queue delay accessible in the egress pipeline. Further more, the P4 specification provides a standard method for bandwidth management using packet classification and metering [4]. However, meters are only exposed to P4 programmers as an extern function that can be used to call metering built-in functionality, which may not be available in all P4 targets, as in the case of NetFPGA-SUME [5], or may have proprietary behavior or limited functionality.

Motivated by the above challenges, to add missing or replace existing functionality enabling programmable traffic management and to enhance its portability between targets, we propose the use of virtual queues in the P4 pipeline. Virtual queue-based traffic management schemes (e.g., AQM [6], scheduling [7] etc.,) have been widely employed for legacy switching devices. In this study, we propose their use for different P4 programmable targets, enabling distinct rate limiting per flow or set of flows and active virtual queue management, while exploiting traffic prioritization capabilities and real queue latency, when/if such capabilities are provided by the target. Assuming that different sets of flows correspond to different slices, customization can be performed on a per slice basis. The proposed data plane solutions are implemented using standard parsing and match action pipeline mechanisms that were defined for programmable forwarding, leveraging commonly supported P4 constructs (i.e., registers). We investigate the portability of the approach, by applying it to two different P4 targets: the BMv2 software switch [8] and the Netronome SmartNIC [9]. Finally, we assess the packet processing efficiency of the SmartNIC implementation, following the methodology presented in [10], [11].

The remainder of the paper is organized as follows. Section II provides an overview of related work. Section III provides background information on the Netronome SmartNIC and the P4 language. Section IV describes the design and implementation of the proposed approach and in Section V, we evaluate its efficiency. Finally, in Section VI, we highlight our conclusions and discuss directions for future work.

II. Related Work

Previous research on extending programmability to the data plane [12] stressed the importance of customizing data plane algorithms, such as scheduling and queuing management strategies, to application requirements.

Sharma et.al. [13] proposed a mechanism called Approximate Fair Queuing, prioritizing packets in order to achieve shorter flow completion times, designed to run on programmable switches; namely (i) a hardware prototype based on a Cavium network processor and (ii) a programmable switch implementation using P4. In a follow up work [14], they proposed a programmable calendar queue using either data-plane primitives or control plane commands to dynamically modify the schedule status of queues. Cascone et al. [15] introduce Fair Dynamic Priority Assignment, a design for a packet forwarding pipeline to enforce approximate fair bandwidth sharing, using primitives common in data plane abstractions such as P4 and OpenFlow. Sivaraman et al. in [16] propose a programmable scheduler that can implement variants of priority scheduling and ideal fair queuing at line rate using a Push-In-First-Out (PIFO) priority queue. PIFO allows packets to be enqueued into an arbitrary location in the queue (thus enabling programmable packet scheduling), but only dequeued from the head. Authors in [17] introduce S-PIFO, an approximation of PIFO queues by FIFO queues. Shrivastav [18] proposes a Push-In-Extract-Out (PIEO) data structure to express buffer management policies. PIEO maintains an ordered list of elements as PIFO, but PIEO allows dequeue from arbitrary positions in the list by supporting a programmable predicate-based filtering at dequeue. Other work from Mittal et.al. [19] shows that the classical Least Slack Time First (LSTF) algorithm comes close to being a universal scheduling function. Shrivastav claims in [18] that LSTF has the same limitations as PIFO, as it uses a priority queue abstraction at its core.

Focusing on queue management, Sivaraman et al. [12] argue that there is no “one-size-fits-all” algorithm by analyzing different AQM approaches. They enable programmability at the data plane by adding an FPGA to the fast path of a hardware switch using an interface to packet queues, implementing CODEL and RED as a proof of concept. Kundel et al. [20] [21] demonstrated that is possible to implement such algorithms for P4 programmable data planes, illustrating P4 capabilities and constraints. In [22], a PI2 implementation in P4 is provided, showcasing ways to overcome P4 limitations towards the development of AQM algorithms. In [23] authors present an implementation of activity-based congestion management using P4, introducing additional target-specific externs for floating point operations to support rate measurement, activity computation, activity averaging and computation of drop threshold. Finally, in [24] PIE and RED AQM schemes are implemented within the P4 context. All these approaches enhance queue utilization within common network infrastructures (links) but they do not fully provide per slice bandwidth and delay guarantees. Authors in [25] use built-in P4 meters and priority scheduling for bandwidth management per slice, without addressing delay requirements. In our case, we are able to support customizable congestion control capabilities for elastic traffic, ensuring both rate and delay limits per slice.

III. Background

A. The P4 Programming Language

P4 [3] is a declarative language for programming protocol-independent packet processors. It is domain-specific with constructs (e.g., headers, parser, etc.) optimized for writing packet forwarding functions. Using P4, developers can uniformly program data plane pipelines based on a match/action architecture, for a variety of targets (ASICs, CPUs, FPGAs etc.). The execution of a P4 program follows an abstract forwarding model with distinct phases: parsing, ingress processing, replication and queuing, egress processing, and deparsing [2]. The behavior for each phase is defined by the declarations in the P4 program. State during execution includes information from packet headers, metadata provided by the device or computed by the program and state kept in counters, registers, and meters [26]. While the P4 language is target-independent, i.e., it abstracts from the specific hardware characteristics of the switch, a P4 compiler translates P4 programs into the instruction set of the packet processor [27]. The current specification of the language (P4_16), introduced the concept of the P4 architecture that defines the P4-programmable blocks of a target and their data plane interfaces. Along with the corresponding P4 compiler, it enables programming the P4 target.

The Portable Switch Architecture (PSA) [28] defines a generic switch architecture with a library of types and P4_16 externs for frequently used stateful memory resources. At the moment only a partial implementation of the PSA is available, therefore the v1model architecture [29], shown in Fig. 1, is still widely used by the reference open-source BMv2 software

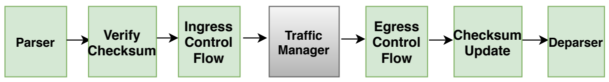


Fig. 1: P4 target architecture (v1model).

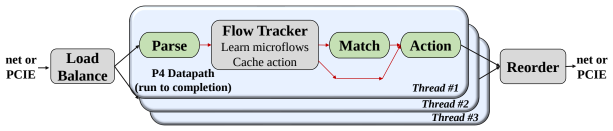


Fig. 2: P4 pipeline in Netronome SmartNICs [31].

switch. Focusing on keeping state, meters are used to record statistics and the state of a flow. This information can be used to drop/mark packets according to burstiness and bit rate criteria. P4_16 supports the "two rate Three Color Marker" (trTCM) [30] for packet classification. The trTCM meters a packet flow and marks its packets based on Peak Information Rate (PIR) and Committed Information Rate (CIR), and their associated burst sizes, to either green, yellow, or red. Using the color, the P4 program can implement drop or mark actions. It is up to the programmer to implement such behavior.

B. Netronome SmartNIC

The Netronome SmartNIC is a Network Flow Processor (NFP) that can be programmed via P4. Fig. 2 shows a schematic of the Netronome SmartNIC's processing pipeline. It is equipped with tens of multi-threaded purpose-built cores that enable high parallelism. Moreover, it utilizes a hierarchical transactional memory and built-in accelerators to boost the packet processing performance. The card also has built-in functionalities such as hashing, stateful memories, and meters, which can be called using P4 extern functions. The SmartNIC's meters have one rate/burst pair of limits and thus can report only two output states for a packet flow. Additionally, the card includes a Flow Tracker mechanism that tracks sessions by learning microflows and caching actions to speed up classification.

The Netronome SmartNIC adopts the v1model architecture as a P4 programming model. However, both ingress and egress pipeline are before the actual queuing and scheduling functions, such that the egress processing has no visibility on actual queuing delays. The back-end compiler of this SmartNIC translates a P4 program to C implementation of the datapath, which is then used to create the firmware for the SmartNIC. Further details about Netronome SmartNIC's hardware design and programmability can be found in [31].

IV. Programmable Traffic Management

In this section, we describe the design and P4 implementation of the proposed virtual queue based approach for programmable traffic management per slice. Then, we discuss the advantages of this approach for rate limiting compared to using P4 meters. Finally, we comment on the portability issues when running these designs on different P4 devices.

A. Design

The proposed P4 programmable traffic management design enables customizing the characteristics of the traffic associated with different slices. In this context, we make sure that the proposed design also satisfies the following requirements:

1) **Traffic Customization**: Managing the traffic characteristics per slice should be possible, so rate limit and maximum tolerable latency can be configured.
2) **State Isolation**: Slices should be independently managed/controlled and customized to meet their performance requirements. Thus, state information per slice should be preserved at the data plane.
3) **Performance Isolation**: Performance isolation implies that service-level key performance indicators should be met per slice, independently of congestion and/or performance levels of other slices sharing the same infrastructure.

The solution comprises a *Traffic Classifier* and a *Virtual Queue based Traffic Manager*. Their design is described hereafter.

1) Traffic Classifier: Match-action tables are the mechanisms used for performing packet processing. We use a match-action table to perform per-flow traffic classification, assuming that the controller allocates a unique local id per slice called *Data Plane Slice ID* (*DP_SID*). For example the VLAN tag of incoming traffic can be used as *DP_SID* similar to [32]. The *DP_SID* is read from the table and stored in the packet's user-defined metadata, allowing the data plane to apply local decisions on policies (i.e. QoS).

2) Virtual Queue based Traffic Management: A vQueue (virtual Queue) is a mechanism used to model the length, or as in our case the sojourn latency, of the queue, as if the packets arriving at the real queue were served by a link with a capacity lower than the actual capacity of the link. It does not hold any packet data. It is a number, incremented as packets arrive and decremented according to the model. The latency of the vQueue can be used, e.g., to drive an AQM scheme, replacing the same metric from the real queue. A description of the structure and its use for an exemplary AQM is provided in [33].

As a design option, the vQueue is implemented using commonly supported P4 constructs (i.e., registers), so that it can be easily ported to different P4 targets. Furthermore, associating a network slice to a vQueue implemented using registers, allows us to keep network slice state information. The control plane can also access these registers at runtime, enabling monitoring and management on a per slice basis. As the use of custom (e.g., multi-dimensional) data structures for indexing such registers is not supported, we employ the local *DP_SID*, to allocate memory for a specific slice and its corresponding port in a network element.

In the example used in this paper, each vQueue is associated with a network slice as color-coded in Fig. 3. Therefore, by dropping or marking excess traffic, virtual queues and thus network slices can be individually rate limited, while a distinct AQM can be applied to each vQueue according to the requirements of each network service. In Fig. 3, we illustrate a possible use of different queue management schemes and transport layer congestion control schemes per slice to address the different requirements of the network services and applications they run, ensuring local queuing latency limits. Furthermore, we use the scheduling capabilities of the non-programmable Traffic Manager - when available - to ensure performance isolation. Each slice can be mapped to a priority

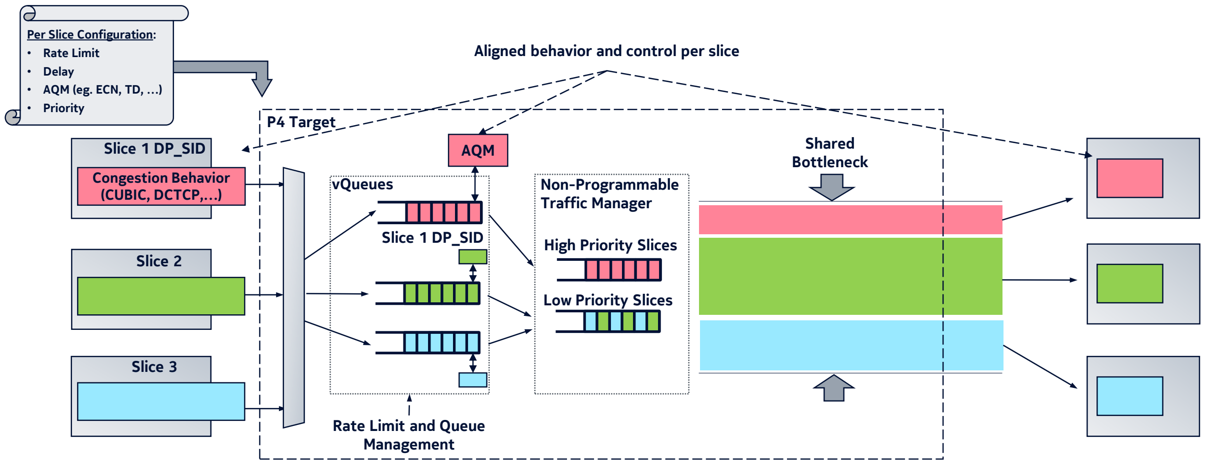


Fig. 3: Virtual queues and slicing at the data plane.

derived according to its application requirements. We assume network slice priorities are calculated and pushed top-down by the control- to the data plane. As depicted in Fig. 3, delay-sensitive traffic from *Slice 1* can be prioritized to traffic from *Slice 2* and *Slice 3* and forwarded with minimum delay. The need to apply different priorities or the opportunity to share priorities will depend on how slice rate limits relate to the worst case real bandwidth limit available left and latency caused by other higher and equal priority slices. How the control plane will assign these is out of scope of this paper, but could use the equations defined in Section V-B.

B. Implementation

We apply traffic management rules for each slice, matching the corresponding *DP_SID* carried by every packet in the user metadata. With the use of the vQueue in the *queue_manage* action (see Listing 1), we can apply rate limiting and queue management per slice, according to the type and version of the transport protocol used and application requirements. We exemplify our approach using two basic schemes: Tail Drop and Explicit Congestion Notification (ECN) step AQM to support classic (e.g., TCP Cubic) and Scalable (e.g., Data Centre TCP [34]) congestion controls.

For efficiency, we use a single table to perform per-flow traffic classification along with implementing the standard IPv4 forwarding functionality (L3Fwd). The L3Fwd match-action table is part of ingress processing of the P4 processing pipeline, forwarding the packet to the appropriate egress port and, in this case, also injecting the *DP_SID* in the packet's user-defined metadata. In our exemplary implementation, a slice is a set of flows having the same destination IP address.

The vQueue per slice is described in Listing 1. For the sake of brevity, we assume equal-sized packets and use a packet transmission time parameter to define the rate. The algorithm can easily be updated to use byte transmission time instead, taking packet size into account. It is implemented using (i) the *slice_ts* register that stores the global timestamp of the previous packet for the slice at the control block (lines 7-13) and (ii) the *delay* register holding the slice's virtual queue size (line 19). Depending on the time that has elapsed, since the previous packet of the vQueue (slice) was processed, we determine how long the current packet has to wait in the virtual queue (lines 20-24). We consider that the *C_DELAY* for each virtual queue is the maximum tolerated delay in msec as prescribed by the slice's Service Level Agreement (SLA). This delay, along with the average packet size and the rate limit (maximum throughput) given in the SLA, is used to derive the burst limit. If the size of the vQueue, augmented by the transmission delay of the packet (*T_DELAY*), exceeds the burst size (*C_DELAY*) then the packet is dropped, else the vQueue size is incremented by the transmission delay of the packet (lines 25-29). In any case, the delay register is updated accordingly (line 30).

Assuming the use of TCP congestion control systems, if the TCP version at flow level supports Explicit Congestion Notification (ECN) functionality and the drop burst limit (*C_DELAY*) has not been reached, the packet is ECN-marked when the virtual queue delay exceeds the marking burst limit *M_DELAY* limit, indicating congestion (lines 32-34). For more complex schemes, each AQM can be a separate action.

While testing our initial P4 code, several target specific modifications were required. The queue admission control should be done at ingress, before enqueuing packets. However, due to some limitations of the SmartNIC target, meters do not work in the ingress. As queues are located after the egress pipeline, metering functionality can safely be executed in the egress pipeline, but it breaks the generality of the P4 program. In the BMv2 case, a P4 program can access information about the real queue in the egress pipeline, as part of the packet metadata. Hence, we have the opportunity to also check real queuing delay and to decide upon dropping or marking a packet. The additional match at egress can be avoided by adding the relevant parameters received at ingress in packet metadata fields. Finally, for targets that support multi-threading, concurrent execution by threads that manipulate the same memory locations can cause inconsistency issues. The P4_16 language provides the @atomic feature which can be used for instructing the compiler to execute atomically a code block.

```
 1 action queue_manage(bit<64> T_DELAY, bit<64> C_DELAY, bit
       <64> M_DELAY){
 2   bit<64> delay=0;                             //Delay
       reported
 3   meta.ts=hdr.intrinsic_metadata.current_global_timestamp;
 4   bit<64> c_ts = meta.ts;
 5   bit<64> p_ts;
 6   bit<64> delta = 0;
 7   @atomic {                                    //Update
       timestamps
 8     slice_ts.READ_REG(p_ts,meta.slice_id);
 9     slice_ts.WRITE_REG(meta.slice_id, c_ts);
10   }
11   if ((p_ts==0) || (p_ts>c_ts)) {              //For reordered
        packets
12      p_ts = c_ts;
13   }
14   delta = c_ts-p_ts;
15   if (delta >= 3294967296) {                   //Wrap up
       timestamps
16      delta=delta-3294967296;
17   }
18   @atomic {                                    //Update delay
19      slice_delay.READ_REG(delay,meta.slice_id);
20     if (delta > delay) {
21       delay = 0;
22     } else {
23      delay = delay - delta;
24     }
25     if (delay + T_DELAY > C_DELAY) {
26      meta.DropFlag = 1;                         //Drop Packet
27     } else {
28      delay = delay + T_DELAY;
29     }
30     slice_delay.WRITE_REG(meta.slice_id,delay);
31   }
32   if ((meta.flag == 0) && (hdr.ipv4.ecn != 0) && (delay >
       M_DELAY)) {
33      hdr.ipv4.ecn = 3;                         //Mark Packet
34   }
35 }
```

Listing 1: Queue Management P4 Action (excerpt).

C. Advantages compared to P4 meters

Off the shelf P4 meters can be used for bandwidth management [25]. In terms of throughput, the use of P4 meters and optionally priority schedulers for bandwidth management should be sufficient. However, to support delay requirements for elastic congestion-controlled traffic, we need to employ additional traffic management schemes (e.g., AQMs, etc.) to cope with congestion. Yet, the traffic management logic in forwarding devices is not programmable.

Virtual queues are used not only to enable programmable active (virtual) queue management and regulate the load on the actual queue(s), but also as a slicing abstraction supporting the implementation of stateful data plane algorithms on a per slice basis, enabling at the same time the easy integration of new traffic management approaches.

As already mentioned, metering built-in functionality may not be available in all P4 targets, as in the case of NetFPGA-SUME [5], or may have proprietary behavior or limited functionality. It also hides the state variables (current queue/burst size) which might be useful for an active queue management (AQM) implementation. Due to architectural limitations or design, many existing P4 targets such as the SmartNICs do not provide queuing delay information in the egress pipeline.

D. On the Portability of P4 Implementations

We implemented the proposed vQueue-based approach and a baseline solution using indirect P4 meters, for the BMv2 software switch and the Netronome SmartNIC. In the following, we summarize the lessons learned from this exercise.

- P4 registers, used in the vQueue implementation, are not synchronized with the in-hardware flow cache on the SmartNIC. This leads to a flaw in the design logic unless we disable the *cache-flow* option on the SmartNIC, even though this may impact the performance of the card.
- Multi-thread processing on the SmartNIC leads to a lack of synchronization between registers read/write operations. This issue was solved by forcing read and write register operations as "atomic operations" (lines 7, 18).
- There is a difference in the time-stamping mechanisms between the two targets. As the 64bit time in the SmartNIC is represented by two 32bit fields (seconds and nanoseconds) we need to subtract $2^{32} - 10^9$ whenever one second is exceeded (lines 15-16).
- The SmartNIC supports the v1model architecture. However, its queues are located after the egress pipeline, hence some of the standard metadata fields that report the queue occupancy are missing. For this reason, we can only monitor the virtual queue (see Section V).
- Assigning different queue priorities on SmartNIC is currently not well supported [35].
- The SmartNIC behavior is undefined when meters are executed in the ingress pipeline. Thus, we had to apply traffic policing (using the queue_manage action or meters execution) at the egress pipeline.
- Meters on the SmartNIC have only one threshold. Two meters have been used in tandem to implement the trTCM.

V. Evaluation and Analysis

In this section, we focus on a set of representative experiments, which illustrate the level of flexibility and portability of our proposed solution, using two different existing P4 targets: the BMv2 software switch and the Netronome SmartNIC. We validate and compare the performance of our approach, denoted as *vQueue*, to the baseline approach using P4 *Meters* [25]. Specifically, in Subsection V-A we validate the effectiveness of the proposed approaches in controlling throughput and delay per slices, ensuring performance isolation. The operational limits and trade-offs of the proposed vQueue approach are investigated in Subsection V-B. Finally, in Subsection V-C, we compare the processing efficiency of the vQueue implementation to that of the Meters, in the SmartNIC setup.

A. Rate and Latency Management

Evaluation Environment and Reporting. The testbed used in this set of experiments consists of three Linux machines which serve as traffic client, server, and the host for the P4 target, as shown in Fig. 4. The client and the server are configured with macvlan to provide traffic isolation at the hosts. We

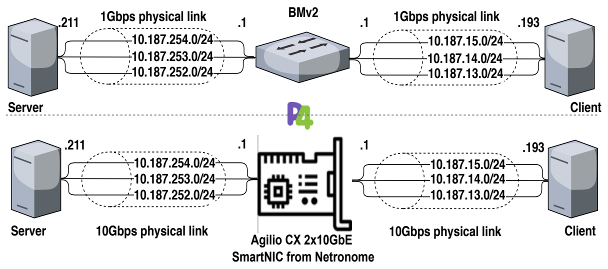


Fig. 4: Testbed for rate and latency management evaluation.

TABLE I: Traffic per slice.

Slice / Parameter	DC		Enterprise		Peering	
	BMv2	SNIC	BMv2	SNIC	BMv2	SNIC
RTT (msec)	5		10		30	
TCP Flavor	DCTCP		Cubic		Cubic	
TCP flows (#)	10	100	1	10	10	100

TABLE II: Configuration per slice.

Slice / Parameter	DC		Enterprise		Peering	
	BMv2	SNIC	BMv2	SNIC	BMv2	SNIC
Rate limit (Mbps)	48	480	12	120	240	2400
Burst limit (msec)	20		10		30	
(pkts)	80	800	10	100	600	6000
QM Scheme	ECN_Step		TD		TD	
Target delay (msec)	5		-		-	

employ two experimentation setups, utilizing either the BMv2 switch or the Netronome SmartNIC. In the first case, we use the three hosts running Ubuntu (16.04 for the switch with Linux Kernel version 4.4.0) each with different Intel CPU (i7-4770 @ 4x3.40GHz, Pentium D @ 2.8GHz and i5-4590 @ 4x3.30Ghz). The machines are equipped with at least two 1GbE NICs with MTU set to 1500 bytes. In the Netronome SmartNIC setup, we use hosts running 18.04 Ubuntu based on Linux Kernel version 4.19.0, each with 16 cores (dual-socket Intel Xeon CPU E5-2630 v3 @ 2.40GHz), 64GB of 2133 MHz DDR4 memory and equipped with 82599ES 10-Gigabit Ethernet network interface card.

With regards to reporting, the identification field in the IPv4 header has been re-purposed to report measurement results. This is done solely for validating the Proof of Concept (by generating the corresponding graphs) and is not required in a real-world deployment of the proposed approach. When BMv2 switch is used as the target, the 6 least significant bits store the packet's virtual queuing delay in ms, 6 bits report the packet's real queuing delay in ms (up to 63ms), while the 4 most significant bits are used to report the number of dropped packets in the first next non-dropped packet of the same slice. Due to size restrictions, up to 15 drops can be reported. Any more dropped packets will be carried over to the next packet, making the maximum reportable drop rate = 15/16 = 94%. When virtual queues are not utilized all 12 least significant bits store the packet's real queuing delay in ms (up to 4s). However, when the SmartNIC is used as the target, the real queuing delay can not be reported as discussed in Subsection IV-D. Therefore, we use the 16 bits identification field to report the virtual queuing delay (in ms) using the least 9 significant bits, and the number of dropped packets using the remaining upper 7 bits.

Network Slice description, traffic & configuration. We assume 3 network slices with different requirements based on their use, that is (i) data center interconnection [*DC slice*], (ii) enterprise WAN connectivity [*Enterprise slice*] and (iii) peering between two virtual network operators [*Peering slice*]. The traffic properties per slice (number of TCP flows and RTT) used in the experiments are provided in Table I. The greedy TCP traffic is only limited by the selected congestion control scheme and its interaction with drops and marks by the configured AQMs in the slice. The congestion control is set to either Cubic or Data Center TCP (DCTCP) on the client(s) and server(s) before starting these applications. The configuration parameters required for the three slices when running on the BMv2 switch and the SmartNIC are provided in Table II. To match the throughput capabilities of the forwarding devices, which are relatively restricted especially in the case of the BMv2 reference software switch[1], we have down-scaled the aggregated slices' rate limit (approximately threefold decrease for the BMv2 switch and the SmartNIC whose line rates are 1Gbps and 10Gbps respectively). Initially, a rate and target delay (burst limit) per slice are arbitrarily set. The burst limit is set as the delay a burst of packets would cause to a real queue served by the specified rate limit. We assume that slice *DC* has the most stringent delay and loss requirements. Thus, DCTCP with an immediate ECN step [34] active queue management algorithm is used. We use ECN packet marking to control the delay of the virtual queue to a 5ms target and avoid loss, by allowing a larger (exceptional) burst limit before packets are dropped. The other two slices use only their drop based burst limit, effectively implementing a delay-based TailDrop (TD) (virtual) queue.

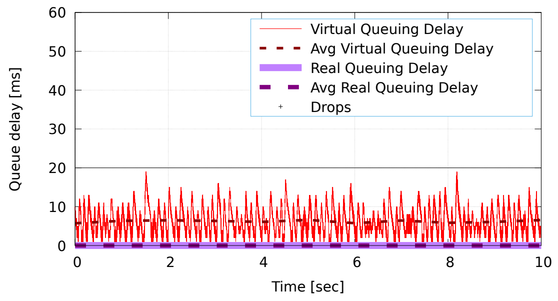
Experiments, Measurements & Evaluation Metrics. The 1st experiment (Subsection V-A1) evaluates the efficiency of the proposed approach using *vQueues* for policing and AQM and the system's interaction with TCP flows on the **BMv2 switch**. The results are compared to the *Meter*-based implementation. We operate the 3 slices simultaneously over the shared non-congested link. All Traffic is sent to the same physical queue at the switch's egress port where the size is set high enough (e.g., 200,000 packets) to ensure that congestion control is only performed by the queue management algorithm at the virtual queues.

We measure the throughput and delay over time for the virtual queue per slice and the physical queue. Measurements are taken after 50 seconds from the start of the experiment, to capture only the steady-state and span over an interval of 250 seconds. We capture the packets at the outgoing switch-to-client interface. The throughput plots are averages over one-second intervals of the per-packet reported size. We measure the queuing delay per packet and zoom in the plot (10s period) to make the TCP variations visible. We also plot the Cumulative Distribution Function (CDF) of the queuing delay. The queuing delay CDFs use the per-packet queue delays to count in bin-sizes of 1024μs, ranging from 0ms to 30ms.

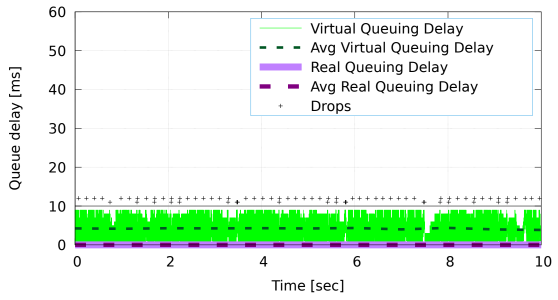
The 2nd experiment (Subsection V-A2) is carried out to evaluate the performance of the *vQueue* management mechanism when running on the P4 programmable **Netronome SmartNIC**. The same measurement procedure is applied in this experiment. However, due to the SmartNIC architecture/implementation, where the physical queue succeeds the ingress/egress pipeline, we do not report the packet delay in the physical queue. The experiment validates the portability of the mechanism, running on a hardware target.

To further investigate the potential benefits of the proposed implementation for slice performance isolation (i.e., meeting

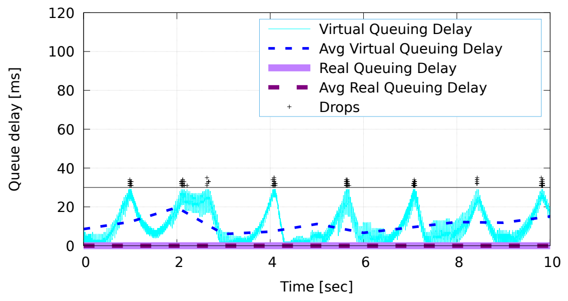
[1]https://github.com/p4lang/behavioral-model/blob/main/docs/performance.md



(a) DC slice.



(b) Enterprise slice.



(c) Peering slice.

Fig. 5: BMv2 - vQueue delay and packet drop results.

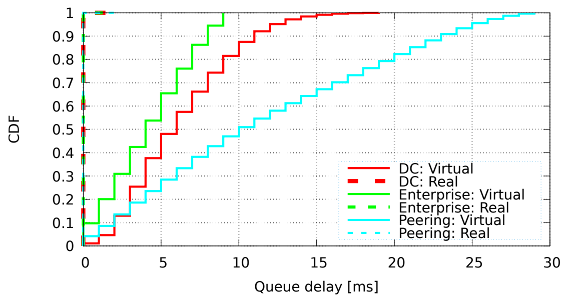


Fig. 6: BMv2 - vQueue CDF of delay.

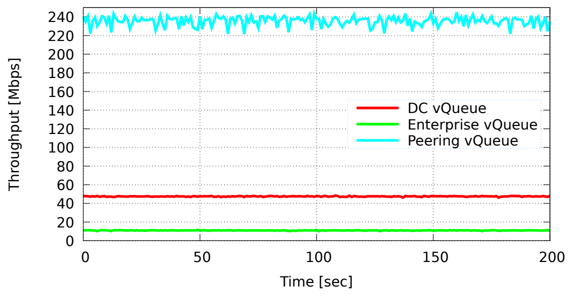


Fig. 7: BMv2 - vQueue throughput results.

throughput and latency bounds per slice), we repeat the experiment, operating however the three slices simultaneously over a shared bottleneck. To this end we rate limit the link to 285Mbps, just below the aggregated capacity of the three slices at 300Mbps. Furthermore, apart from the vQueue-based traffic management, we also exploit the capabilities of the target's non programmable traffic manager. Specifically, we employ traffic prioritization using two priority queues, assigning the traffic of the DC and Enterprise slices to the highest priority one. This 3^{rd} experiment (Subsection V-A3) is conducted only using the BMv2 target as real queuing delay cannot be measured and configuring queue priorities with P4 is not well supported in the SmartNIC.

1) Experiment 1- BMv2 software switch: The results of vQueue implementation running on the BMv2 switch are presented in Figs. 5-7, with plots showing the (i) per packet and average delay in the virtual and the real (switch egress port) queue as well as packet drops (zoomed in for the first 10 seconds of the experiment), (ii) the delay CDFs for the duration of the experiment and (iii) average throughput. Fig. 8 shows the measured throughput for all slices when using the standard P4 packet classifier/meters (baseline).

The gauged TCP throughputs, for the three slices in Fig. 7 confirm the effectiveness of the policing approach using vQueues; the results are similar to the baseline in Fig. 8 where with the use of P4 meters we are limiting the per slice rate according to their configurations. Moreover, the corresponding delay CDFs, show that for a non-congested physical link, the real queuing delay for all slices sharing that link is in the order of microseconds, while the virtual queue delay is kept below the corresponding burst limit and the hard rate limits are obeyed. This signifies that the proposed vQueue implementation is able to control the behavior and limits of each slice and match its requirements, in terms of throughput as well as drop/target delay (when applicable). The latter is not possible using the off-the-shelf P4 meters.

Under close inspection of the vQueue results, in Fig. 5a, we see that the DCTCP flows (slice DC) are being controlled at the target rate and around the 5ms marking threshold (Table II). As packets are marked and not discarded, the

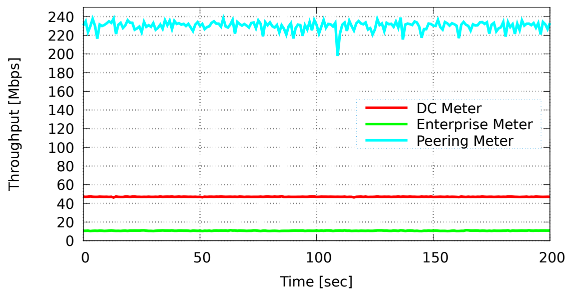


Fig. 8: BMv2 - Meter throughput results.

DCTCP flows average virtual queue delay can exceed the 5ms marking threshold. The extra 15ms virtual queue size is required to minimize packet drops, which sporadically still happens when this threshold is exceeded. The higher delay variation in the virtual queue is due to the fast integration function of the virtual queue and the delayed mark response from the TCP senders.

With regards to the Cubic TCP flow of enterprise slice in Fig. 5b we see that due to the non-shaped (non-ACK-paced) traffic, the full congestion window of the single flow is transmitted repetitively at line rate, resulting in a very bursty virtual queuing delay. Because the virtual transmission time of one packet is 1ms on the virtual rate limit of 12 Mbps, only bursts of about 10 packets per RTT (10 ms) are allowed. This reduces the actual rate to only slightly above 10 Mbps.

In Fig. 5c, we also observe the Cubic TCP behavior (in Reno mode) for the flows of the peering slice. Throughput variations can be attributed to the abrupt 30% Cubic backoff on loss and slow increase. A high level of jitter is noted due to line-rate bursts at congestion window size, but in this case, it is not limited by the burst limit of the virtual queue. Even with 10 flows we observe a high level of synchronization. This causes frequent episodes of slight under-utilization, visible in the throughput and CDF plot. Note that the real queuing delay of the three slices is always in the order of microseconds, and hence close to 0 ms.

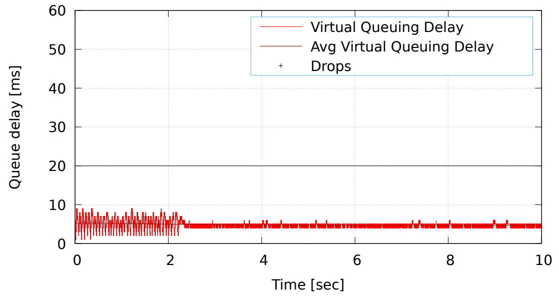
As a last note, the results reveal the efficiency of the proposed approach to support alternative traffic management schemes per slice, adapted to the TCP flavor used and application needs. Obviously, slice (state and performance) isolation can be supported when sharing a real queue and capacity, given that both are dimensioned sufficiently large to accommodate the slices.

2) Experiment 2- Netronome SmartNIC: Similar to the BMv2 experiment, the results of the vQueue implementation are presented in Figs. 9-11, while Fig. 12 depicts the throughput for the three slices when using the standard P4 packet classifier/meter implementation for the SmartNIC.

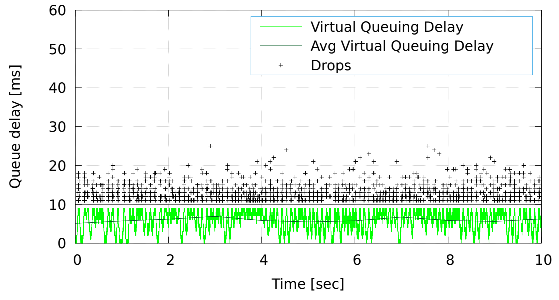
Overall, we see a similar trend to the BMv2 results at a larger scale (x10 flows). For the three slices in Fig. 11 the throughput results are similar to the baseline implementation in Fig. 12. Although in the SmartNIC setup we can only report the delay of the virtual queues, the virtual queue delay for all three slices is kept below the corresponding burst limit and the hard rate limits are obeyed.

In more details, the DCTCP flows (slice DC) in Fig. 9a are fluctuating around the 5ms marking threshold, similar to BMv2. In this case, however, DCTCP traffic fills the virtual queue, since congestion control (via marking) takes a few RTT to respond and reduce the rate to the target value. After approximately two seconds the synchronized TCP oscillation, manifested as the delay variation and the typical on/off marking pattern, is smoothed out as on/off marking gets broken due to micro marking bursts spread more evenly over time.

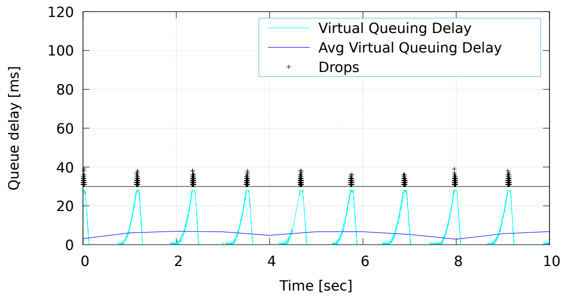
Regarding the Enterprise slice in Fig. 9b, the tenfold increase in the number of packets dropped is due to the tenfold increase in the number of flows. We also observe a better utilization of the slice capacity, indicating no TCP flow synchronization.



(a) DC slice.



(b) Enterprise slice.



(c) Peering slice.

Fig. 9: SmartNIC - vQueue delay and packet drop results.

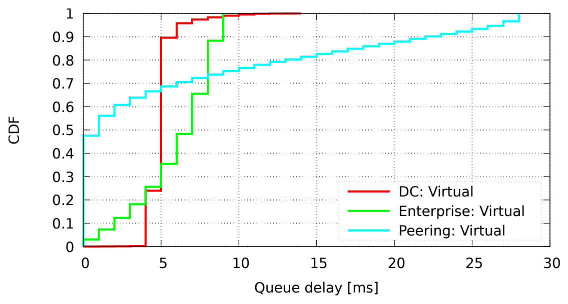


Fig. 10: SmartNIC - vQueue CDF of delay.

For the Peering slice in Fig. 9c, the effects of global synchronization we noted in Fig. 5 for the Cubic TCP flows are amplified when scaling up (x10). This leads to increased under-utilization of the slice capacity, made evident also by the CDF and the throughput plots. Using our vQueue implementation, one could alternatively adopt a random dropping scheme

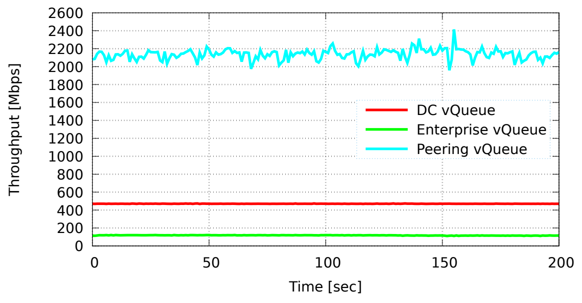


Fig. 11: SmartNIC - vQueue throughput results.

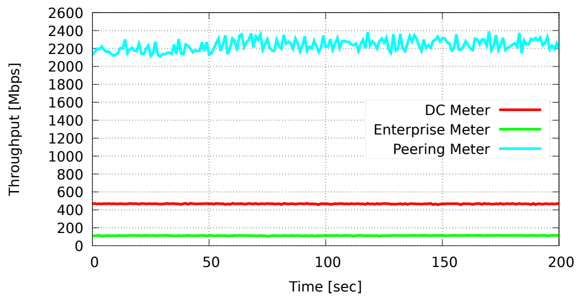


Fig. 12: SmartNIC - Meter throughput results.

such as the RED-like scheme, to break synchronization issues, enforcing a linear increase in the drop probability starting at a minimum delay threshold ending at a maximum threshold where 100% of the packets are dropped.

3) Experiment 3- BMv2 performance isolation with over-utilized link: Performance isolation in the network slicing context implies that service-specific performance requirements are always satisfied on each network slice instance, regardless of the congestion and workloads of other slice instances running over the shared infrastructure. Fig. 13 and Fig. 14 depict the coo responding delay and throughput, using priority queues for both the virtual queue and P4 meters. The DC and Enterprise slice traffic is prioritized - sharing the high priority physical queue of the switch - over the Peering slice traffic. The three slices operate simultaneously, sharing the same constrained link capacity of a single egress port (285 Mbps). As the real rate limit is lower than the sum of the virtual rate limiters, the real queue delays need to be also controlled. To that end, the virtual queue policies we implemented use the

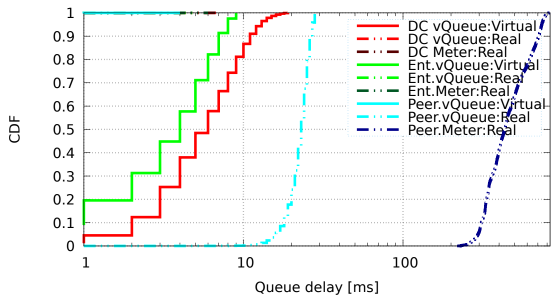


Fig. 13: BMv2 - performance isolation with over-utilized link CDF of delay.

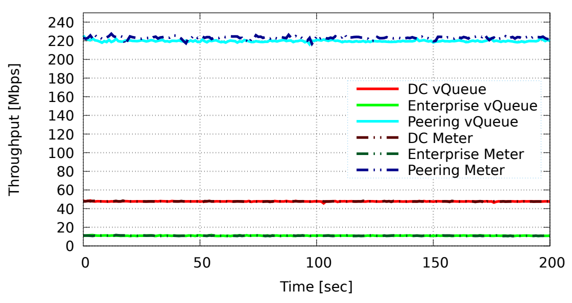


Fig. 14: BMv2 - performance isolation with over-utilized link throughput results.

maximum of the virtual and real queue delay of packets for their drop and mark decisions.

In the vQueue case, we observe that the QoS requirements of the two high priority slices are supported, while we observe a decrease in throughput and an increase in the packets' real queue delay for the low priority one (Peering Slice) which however remains within the pre-defined bound (30 ms). Fig. 13 (CDF) signifies that the Peering slice is controlled by the real queuing delay (Peer.vQueue:Real). The vQueue is not operational anymore, as it is not able to reach the virtual rate limit and hits the burst limit of 30 ms, as long as the higher priority slices are using their maximum throughput levels.

Using the standard P4 meters, throughput-wise, the results are similar as evident in Fig. 14; there is the same notable decrease in throughput for the Peering slice. As expected, no real queuing delay is observed for the DC and Enterprise slices since their traffic is prioritized. However, the increase in the packets' real queuing delay for the low priority traffic is significant (up to 800ms), since it is not controlled as in the vQueues case.

Considering delay as the key performance indicator for performance isolation, the use of the P4-standard meters does not sufficiently isolate the performance of the low priority slice. Meters only support under-provisioned rate limits when latency guarantees are required, unless additional real queue latency configuration and checks are implemented, like in our vQueue implementation. To omit using additional real queue latency controls, we need to manage the capacity margin between the (configured) sum of the slices' (vQueues) throughput and the capacity of the physical link.

B. Guaranteed Performance Targets Using vQueues

As noted in the last experiment in V-A3, namely *Experiment 3- BMv2 performance isolation with over-utilized link*, it is necessary to investigate the capacity margin required between the (configured) sum of the vQueues and the actual link capacity, to guarantee that all slices meet their performance target. Towards that end, we assume the behavior of a Markovian non-preemptive priority queuing system and estimate the corresponding physical link capacity required to support the delay and throughput targets set per slice, under different slice priorities.

Using priority scheduling, the queues of lower-priority slices are effectively blocked during the busy period of higher-

TABLE III: Priority per slice.

Case \ Slice	DC	Enterprise	Peering
A	7	7	7
B	7	6	6
C	7	7	6
D	7	6	5

priority slices. Based on the maximum busy period for every slice $s \in S$ with priority $i \in I$, where $|I|$ is the highest priority, we can calculate the maximum real queuing delay (worst case scenario) for each slice. The maximum busy period t_i^s for every slice s with priority i can be estimated based on its burst limit b_i^s (in bytes), and the residual capacity r_i of the bottleneck for priority i as following:

$$t_i^s = \frac{b_i^s}{r_i} \tag{1}$$

The residual capacity that can be used by every slice s with priority i and rate limit L_i^s, over the bottleneck with nominal capacity C is:

$$r_i = C - \sum_{\forall j \in S} \sum_{k=i+1}^{|I|} L_k^j \tag{2}$$

Based on Eq. (1) and (2), the worst case queuing delay d_i for every slice s with priority i is equal to the aggregate busy period of all higher or equivalent priority slices, including slice s as follows:

$$d_i = \sum_{\forall j \in S} \sum_{k=i}^{|I|} t_k^j \tag{3}$$

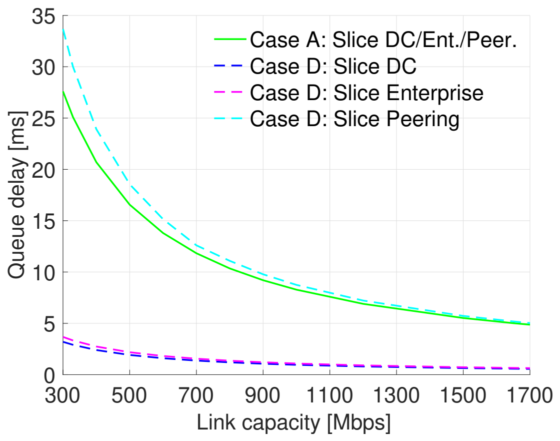
The extra worst case queuing delay $\Delta d_i = d_i - d_{i+1}$ for all slices with priority i is dependent on (i) the residual capacity of the link for that priority r_i and (ii) sum of all burst sizes (derived from the delay burst limits and respective rate limits) of all slices of that priority as follows:

$$\Delta d_i = d_i - d_{i+1} = \frac{\sum_{\forall j \in S} b_i^j}{r_i} \tag{4}$$

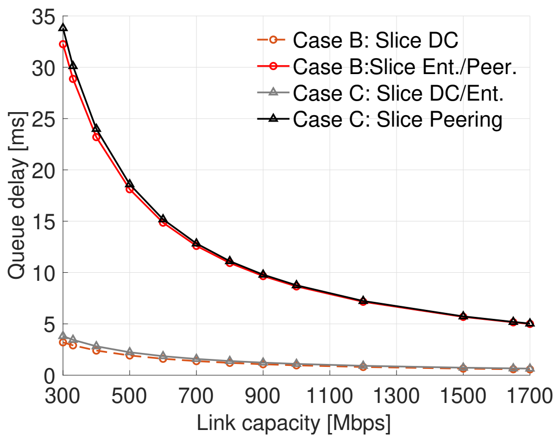
From this reworked equation, we can derive the bound on the maximum extra latency Δd_i for slices of priority i given a residual capacity r_i, or the bound on the minimum residual capacity r_i needed for the given maximum extra latency Δd_i that is allowed for the slices of priority i.

When the real queuing latency cannot be controlled, these bounds will limit the number of slices that can be supported for a given link capacity. On the other hand, when the real queues can be controlled, these bounds provide the worst case rate that a priority can experience.

In Fig. 15 we plot the worst case queuing delay of the three slices used in the experiments above (i.e. slice DC, Enterprise and Peering with target rates of 48 Mbps, 12 Mbps and 240 Mbps respectively), for various rates (bottleneck links),



Cases A and D



Cases B and C

Fig. 15: Worst case delay as a function of link capacity.

with priorities as depicted in Table III. The figure is split to improve readability as the cases B, C and D partially overlap. Case A is similar to the initial slice setup used in the first two experiments of the previous Subsection for BMv2 and SmartNIC evaluations in *Experiment 1* and *Experiment 2* while Case C reflects the slice setup used in the third experiment *Experiment 3* for the over-utilized link evaluation. Case D yields similar results to case C; the DC and Enterprise traffic are both prioritized over Peering traffic. Case B identifies the corresponding limits, when traffic from slice DC is prioritized over the other two.

Graph *Case A: Slice DC/Ent./Peer.* in Fig. 15 depicts the worst case queuing delay for each slice, with no traffic prioritization. We observe that a capacity of 1650 Mbps is required to guarantee a (maximum) 5 ms delay for Slice DC.

The queuing delays for Slices DC, Enterprise and Peering in Case B are denoted as *Case B:Slice DC* and *Case B:Slice Ent./Peer.* respectively. We note that in this case, where DC traffic is prioritized over traffic from the other two slices, the 10 ms delay limit for the Slice Enterprise is met only when the bottleneck rate is higher than 900 Mbps.

In case C, where traffic from Slices DC and Enterprise is prioritized over Peering traffic, the queuing delays are denoted as *Case C:Slice DC/Ent.* and *Case C:Slice Peering* respectively. Similarly, graphs *Case D: Slice DC*, *Case D Slice Enterprise* and *Case D: Slice Peering* depict the worst case queuing delay for each slice, when different priorities are applied between the slices. We verify from these graphs, that both priority policies allow us to support the corresponding maximum burst limits for all slices even in the worst case scenario, as long as the bottleneck is at least equal to 330 Mbps.

Overall, we observe that by using the corresponding maximum burst limits (worst case queuing delay) of the slices, strict slice priority allows us to determine the necessary capacity margin required for the vQueues to operate efficiently under different conditions (e.g., 330 Mbps in case C and D for the specific QoS/rate targets per slice). On that account, an intelligent control plane could adaptively assign slices to different priorities, based on their requirements and the physical link capacity, to meet their KPIs.

C. Packet Processing Latency

Experiment, Measurements & Evaluation Metrics. In the SmartNIC setup, we assess the efficiency of our proposed approach using virtual queues, following the methodology presented in [10] and [11]. To that end, we compare the packet processing latency of the vQueue pipeline to the corresponding processing latency for the pipeline using P4 meters. To identify the extra processing latency needed for meters and vQueue, we measure the latency of a program with only L3 Forwarding (L3Fwd) packet processing, which serves as a baseline.

Specifically, we change the experimentation setup as shown in Fig. 16, using the MoonGen packet generator [36] to benchmark the device-under-test (Netronome SmartNIC). After loading the P4 implementation under test, MoonGen sends 1500 Byte UDP packets at 2.4 Gbps to the Netronome SmartNIC. The packets are processed in the card and then sent back to MoonGen where packet latency is reported.

Additionally, we examine the impact of different actions taken by each P4 implementation on packet processing latency, by configuring the two traffic management implementations at rates equal to 2.3 Gbps and 5 Gbps to be smaller (Mark Action) and greater (No-Mark Action) than the 2.4 Gbps generated traffic rate. We made sure that no packets are dropped by

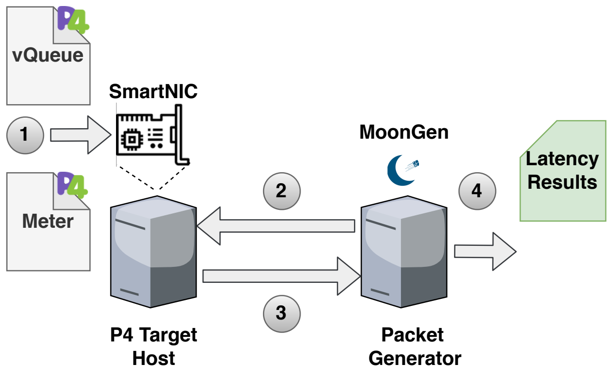


Fig. 16: Testbed for evaluating processing latency.

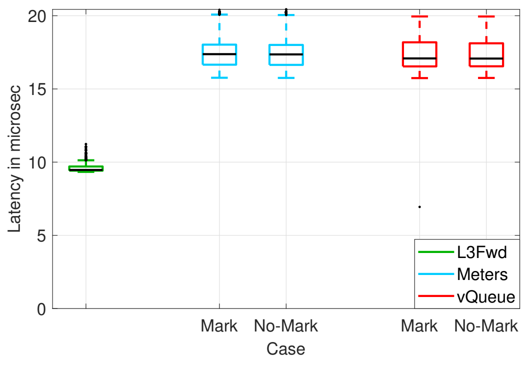


Fig. 17: Packet processing latency.

the traffic management mechanisms as this will disrupt the packet latency measurements. This was achieved by arbitrarily increasing the dropping burst size, and by minimizing the difference between the limited rate in the Mark-Action case, i.e. 2.3 Gbps, and the sent rate, i.e. 2.4 Gbps, to make sure that drop burst threshold is never reached (no packet is dropped).

Packet Processing Latency Results. The box plots of the measured latency, in μs, corresponding to the different P4 implementations with different configurations is shown in Fig. 17. We can observe that the traffic management mechanism implemented in both vQueue and meter implementations contributed an additional 8 μs on top of the L3Fwd baseline pipeline. Additionally, the packet processing load of our proposed vQueue implementation shows a slight advantage compared to that of meter implementation, where the median of the measured latency in the vQueue cases is 0.3 μs less than that measured in the meter cases. Moreover, we can observe that the packet processing latency of the two implementations is invariant whether packets are marked or not. Memory-wise, the reported usage of different memory types in both implementations is approximately the same, except for the usage of Cluster Local Scratch memory (CLS), which is responsible for storing frequently used data, where vQueue implementation requires additional 2% of the available memory compared to meter-based implementation.

An alternative implementation of the vQueue using one register and one atomic section did not improve the performance, as accessing a register was faster than the extra processing required. Depending on the target and register constraints, it might be possible to further optimize the algorithm striking the right balance between resource usage and processing latency, of course again at the cost of generalization.

VI. Conclusion

Programmable data planes support network flexibility, promoting rapid development and prototyping cycles. However, most programmable devices still typically have non-programmable traffic managers. In this paper, we propose the use of virtual queues in the P4 pipeline and the application of programmable virtual queue-based traffic management for different P4 programmable targets. We show that the use of the virtual queues not only enables bandwidth management per slice as P4 meters do, it also allows us to ensure delay bounds per slice, employing virtual queue-based AQM. Essentially, we are taking a step towards fully customizable data plane slices. We validate the performance of the proposed approach focusing on elastic congestion-controlled traffic and further investigate the correlation between the shared link and network slices' capacity required for the proposed approach using vQueues to operate efficiently. We further look into the portability of the approach, using the reference P4 software switch and an NPU based hardware target. The proposed implementation can be an alternative for built-in metering implementations which, if present in a P4 hardware target, require additional effort to assure portability. We showcase that our design has a similar rate limiting performance, full access to state information, and a slightly lower processing delay compared to P4 Meters available on a P4 target. In the future, we plan to evaluate other hardware targets and investigate programmable approaches for inter-slice scheduling, to provide a fully programmable traffic management system.

Acknowledgment

This work has been partially funded by the EU H2020 5Growth Project (grant no. 856709).

References

[1] R. Bifulco and G. Rétvári, "A survey on the programmable data plane: Abstractions architectures and open problems," in *Proc. IEEE HPSR*, 2018, pp. 1–7.

[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," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 99–110.

[3] 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, pp. 87–95, Jul. 2014. [Online]. Available: http://doi.acm.org/10.1145/2656877.2656890

[4] P4-Consortium. (2016) P4 16 Language Specification. https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html. Accessed: 2020-06-25.

[5] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore, "NetFPGA SUME: Toward 100 Gbps as research commodity," *IEEE micro*, vol. 34, no. 5, pp. 32–41, 2014.

[6] S. S. Kunniyur and R. Srikant, "An Adaptive Virtual Queue (AVQ) algorithm for active queue management," *IEEE/ACM Transactions on networking*, vol. 12, no. 2, pp. 286–299, 2004.

[7] A. Eryilmaz and R. Srikant, "Fair resource allocation in wireless networks using queue-length-based scheduling and congestion control," *IEEE/ACM transactions on networking*, vol. 15, no. 6, pp. 1333–1344, 2007.

[8] P4-consortium. (2020) Behavioral Model (bmv2). https://github.com/p4lang/p4c/blob/master/p4include/v1model.p4. Accessed: 2020-08-01.

[9] Netronome. (2020) Netronome Agilio SmartNics. https://www.netronome.com/products/smartnic/overview/. Accessed: 2020-08-01.

[10] H. Harkous, M. Jarschel, M. He, R. Priest, and W. Kellerer, "Towards understanding the performance of p4 programmable hardware," in *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE, 2019, pp. 1–6.

[11] H. Harkous, M. Jarschel, M. He, R. Pries, and W. Kellerer, "P8: P4 with predictable packet processing performance," *IEEE Transactions on Network and Service Management*, 2020.

[12] A. Sivaraman, K. Winstein, S. Subramanian, and H. Balakrishnan, "No silver bullet: extending SDN to the data plane," in *Proceedings of the Twelfth ACM Workshop on Hot Topics in networks*. ACM, 2013, p. 19.

[13] N. K. Sharma, M. Liu, K. Atreya, and A. Krishnamurthy, "Approximating Fair Queueing on Reconfigurable Switches," in *USENIX Symposium on Networked Systems Design and Implementation*, 2018, pp. 1–16.

[14] N. K. Sharma, C. Zhao, M. Liu, P. G. Kannan, C. Kim, A. Krishnamurthy, and A. Sivaraman, "Programmable calendar queues for high-speed packet scheduling," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020, pp. 685–699.

[15] C. Cascone, N. Bonelli, L. Bianchi, A. Capone, and B. Sansò, "Towards approximate fair bandwidth sharing via dynamic priority queuing," in *Local and Metropolitan Area Networks (LANMAN), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 1–6.

[16] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown, "Programmable packet scheduling at line rate," in *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 2016, pp. 44–57.

[17] A. G. Alcoz, A. Dietmüller, and L. Vanbever, "SP-PIFO: approximating push-in first-out behaviors using strict-priority queues," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020, pp. 59–76.

[18] V. Shrivastav, "Fast, scalable, and programmable packet scheduler in hardware," in *Proceedings of the ACM Special Interest Group on Data Communication*, 2019, pp. 367–379.

[19] R. Mittal, R. Agarwal, S. Ratnasamy, and S. Shenker, "Universal packet scheduling," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016, pp. 501–521.

[20] R. Kundel, J. Blendin, T. Viernickel, B. Koldehofe, and R. Steinmetz, "P4-CoDel: Active Queue Management in Programmable Data Planes," in *Proceedings of the IEEE 2018 Conference on Network Functions Virtualization and Sofwtare Defined Networks*. IEEE, 2018, pp. 27–29.

[21] R. Kundel, A. Rizk, J. Blendin, B. Koldehofe, R. Hark, and R. Steinmetz. (2020) P4-codel: Experiences on programmable data plane hardware.

[22] C. Papagianni and K. De Schepper, "PI2 for P4: An active queue management scheme for programmable data planes," in *Proceedings of the 15th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 84–86. [Online]. Available: https://doi.org/10.1145/3360468.3368189

[23] M. Menth, H. Mostafaei, D. Merling, and M. Häberle, "Implementation and Evaluation of Activity-Based Congestion Management Using P4 (P4-ABC)," *Future Internet*, vol. 11, no. 7, p. 159, 2019.

[24] S. Laki, P. Vörös, and F. Fejes, "Towards an AQM Evaluation Testbed with P4 and DPDK," in *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos*, ser. SIGCOMM Posters and Demos '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 148–150. [Online]. Available: https://doi.org/10.1145/3342280.3342340

[25] Y.-W. Chen, L.-H. Yen, W.-C. Wang, C.-A. Chuang, Y.-S. Liu, and C.-C. Tseng, "P4-enabled bandwidth management," in *2019 20th Asia-Pacific Network Operations and Management Symposium (APNOMS)*. IEEE, 2019, pp. 1–5.

[26] J. Liu, W. Hallahan, C. Schlesinger, M. Sharif, J. Lee, R. Soulé, H. Wang, C. Caşcaval, N. McKeown, and N. Foster, "p4v: Practical verification for programmable data planes," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 2018, pp. 490–503.

[27] A. Sivaraman, C. Kim, R. Krishnamoorthy, A. Dixit, and M. Budiu, "DC.p4: Programming the forwarding plane of a data-center switch," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. ACM, 2015, p. 2.

[28] T. P. A. W. Group. (2018, November) P416 portable switch architecture v1.1. [Online]. Available: https://p4.org/p4-spec/docs/PSA-v1.1.0.html

[29] P4-consortium. (2020) V1 Model. https://github.com/p4lang/behavioral-model. Accessed: 2020-08-01.

[30] J. Heinanen and R. Guerin. (1999) RFC2698: A two rate three color marker.

[31] Netronome. (2017) Mapping P4 to SmartNICs. https://p4.org/assets/p4_d2_2017_nfp_architecture.pdf. Accessed: 2020-05-25.

[32] 5Growth-consortium. (2020) Initial implementation of 5G End-to-End Service Platform. https://5growth.eu/deliverables/.

[33] B. Briscoe, "The native AQM for L4S traffic," *arXiv preprint arXiv:1904.07079*, 2019.

[34] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center TCP (DCTCP)," *ACM SIGCOMM computer communication review*, vol. 41, no. 4, pp. 63–74, 2011.

[35] Netronome. (2020) Traffic class configuration. https://groups.google.com/g/open-nfp/c/kNqO8mSTupE. Accessed: 2020-08-01.

[36] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "Moongen: A scriptable high-speed packet generator," in *Proceedings of the Internet Measurement Conference (IMC)*. ACM, 2015, pp. 275–287.