FlowValve: Packet Scheduling Offloaded on NP-based SmartNICs

Shaoke Xi
Northeastern University, China
Zhejiang University
shaokexi@zju.edu.cn

Fuliang Li*
Northeastern University, China
lifuliang@cse.neu.edu.cn

Xingwei Wang*
Northeastern University, China
wangxw@mail.neu.edu.cn

***Abstract*—Enforcing scheduling policies at end-hosts with software schedulers suffers from high CPU consumption, low throughput, and inaccuracy. Offloading scheduling functions to the network interface card (NIC) provides a promising direction to address these problems. However, existing efforts in scheduling offloading suffer from inflexible on-NIC packet schedulers, which cannot execute complex hierarchies of network policies. In this paper, we present FlowValve, the first parallel packet scheduler for Network Processor (NP)-based SmartNICs that offloads critical functions of Linux traffic control, including packet classifying and scheduling. The key insight behind FlowValve is to abstract inherent queues attached to the NIC interface (wire side) as a single FIFO queue and perform specialized tail drop to mix the FIFO queue with expected flow proportions. FlowValve takes advantage of on-chip multi-core parallelism and hardware accelerations to produce high throughput. Meanwhile, it substantially reduces CPU and memory burdens on end-hosts. We prototype FlowValve on a Netronome Agilio SmartNIC and demonstrate its effectiveness against non-offloaded kernel schedulers and DPDK QoS Scheduler. We find that FlowValve outperforms both in accurately enforcing network policies while driving line rate performance (i.e., 40Gbps), which contributes to saving at least two CPU cores.**

I. Introduction

In cloud data centers, traditional traffic scheduling on network devices (e.g., switches and routers) are moving toward network edges (e.g., end-hosts) [1]–[3]. This trend arises from network traffic isolation requirements between competing tenants. As data center operators continue upgrading server NICs from 40Gbps to 100Gbps, enforcing scheduling policies at end servers with software systems suffers from high CPU consumption, low throughput, and inaccuracy [4]. An ideal solution would be offloading packet scheduling onto NICs, which reduces CPU workload meanwhile benefits from hardware line rate performance [5].

Recently, SoC (System on a Chip) SmartNICs gain significant popularity in the data center network for offloading data-centric computations at server sides. These computations are characterized by stateful processing of data streams at high rate. SoC SmartNICs utilize purpose-built processors along with various domain-specific accelerators to boost packet processing while maintaining reasonable flexibility. Representative products of this kind includes Netronome SmartNICs [6] with NPs (Network Processors), Fungible data processing units [7] with MIPS64 cores, etc.

Research works have shown considerable performance gains of offloading applications to SmartNICs, which mainly benefits from their increasing computation power and highly customizable features [8]–[12]. However, there is still a lack of explorations for packet scheduling offloading. The inflexible on-NIC packet scheduler is a primary impediment to enforce hierarchies of complex network policies. The state-of-the-art NIC design [13] adopts a programmable packet scheduler to overcome this difficulty. However, realizing it in ASICs likely takes years. So we wonder whether it is possible to implement scheduling algorithms in existing SmartNIC products.

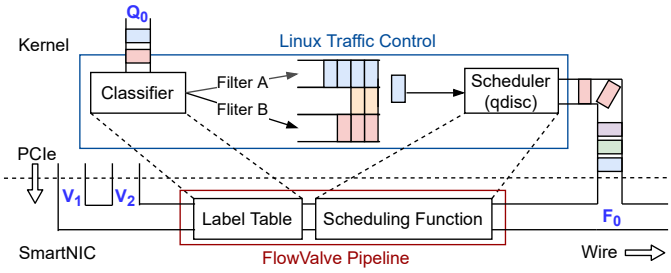


Fig. 1: FlowValve offloads kernel schedulers onto SmartNICs.

Unfortunately, we find the answer varies depending on the target SmartNIC architecture. Different architectures lead to different instruction sets and distinct performance models. So that a cross-platform scheduling paradigm is very difficult to design. Our positive answer of the above question is mainly based on the observations of NP-based SmartNICs. Solutions to other platforms are beyond the scope of this paper.

Observation 1: Specialized programmable cores on NP can do rather complex computation and enable us to develop scheduling functions in a software manner. This eliminates difficulties in designing complicated circuits for stateful processing, which is common in scheduling algorithms [14].

Observation 2: The multi-core architecture and numerous on-chip accelerators boost packet processing efficiency. For example, Exact Match Flow Cache on Netronome Agilio uses dedicated lookup engines to search the cached flow actions, which enlarges the corresponding kernel implementation [15] by 10 times. Generally, making good use of multi-core parallelism and hardware accelerations can guarantee high throughput (e.g., 40Gbps or more) while substantially reducing

CPU and memory burdens on end-hosts.

Observation 3: NIC virtualization techniques (i.e., SR-IOV) combined with offloaded scheduling functions naturally support multi-queue feature, which is lacking for most software schedulers [16], [17]. As shown in Figure 1, transmitting packets of different apps or tenants to the NIC through separated virtual function ports (i.e., V_1,V_2) removes the necessity of employing a central queue (i.e., Q_0) to enforce network policies. So that the offloaded scheduler can be more efficient and scalable [18].

In this paper, we present FlowValve, a parallel packet scheduler on NP-based SmartNICs. The key insight behind FlowValve is to abstract inherent queues attached to the NIC wire interface as a single FIFO queue (i.e., F_0 in Figure 1) and perform specialized tail drop to mix the FIFO queue with expected flow proportions. We start from offloading packet classifiers and queueing disciplines (qdiscs) of Linux traffic control [19], e.g., Priority Qdisc (PRIO) [20] and Hierarchy Token Bucket (HTB) [21]. However, the offloading process is non-trivial. First, running unoptimized scheduling algorithms on multi-core NPs results in extremely low throughput. Therefore, we develop parallel scheduling algorithms along with new data structures for optimizations. Second, the run-to-completion processing model and inherent queueing system on NPs make traffic shaping expensive. Traffic shaping requires to buffer and resend packets at appropriate times. FlowValve does not directly perform traffic shaping. Instead, it emulates shaping by dropping packets, which it predicts would also be dropped by a hypothetical traffic shaper. FlowValve enforces rate control with hierarchical token buckets. It also supports chaining offloaded qdiscs by performing runtime rate estimations to keep adjusting token fill rate for the buckets of different traffic classes. Evaluations on a real NP-based SmartNIC show that FlowValve can almost achieve the same rate conformance as a real traffic shaper. FlowValve is open source and available at [22].

In summary, our main contributions are:

- We explore the capability of NP-based SmartNICs and propose the first feasible approach to offload end-host packet scheduling on this platform.
- We develop parallel scheduling algorithms and key data structures to offload two queueing disciplines of Linux traffic control, i.e., PRIO and HTB.
- We implement FlowValve prototype on a Netronome Agilio CX 40GbE SmartNIC. Experiments show that FlowValve accurately enforces network policies while driving line rate performance (i.e., 40Gbps). This contributes to saving at least two CPU cores.

II. Motivation

We demonstrate the inefficiency of software schedulers and the inflexibility of on-NIC schedulers to motivate the need of offloaded scheduling. As the example shown in Figure 2, when a server hosts multiple virtual machines, their outbound traffic should be disciplined to prevent one tenant from occupying too much bandwidth. Similarly, applications which are running in the guest machines (e.g., KVS and ML in vm1) also require appropriate network policies to guarantee their performance. As for crucial management softwares running in the host OS (e.g., NC), it is necessary to reserve enough bandwidth for their communication channels to lessen packet drop.

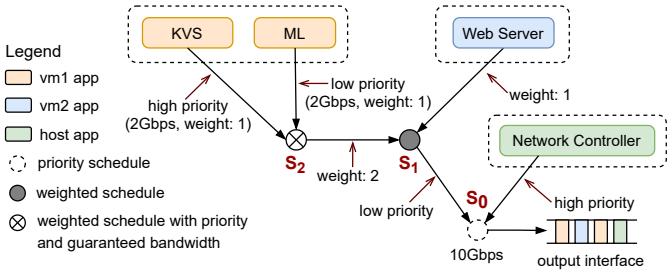


Fig. 2: Motivation example. Guest OS vm1 runs a key-value store (KVS) and a machine learning service (ML). Guest OS vm2 runs a web server (WS). A network controller (NC) runs on the host OS. These apps together share 10Gbps egress bandwidth.

More specifically, an administrator may want to specify QoS policies as follows. First, traffic from NC has the highest priority. Second, traffic from vm1 and vm2 proportionally shares the rest of link bandwidth. Vm1 (KVS and ML) gets two thirds and vm2 (WS) gets one third. Third, in vm1, KVS is assigned a higher priority than ML. However, to prevent ML from starvation, it has the guaranteed bandwidth of 2Gbps. So that at least 2Gbps bandwidth should be available to ML when the total is more than 4Gbps. Otherwise, ML and KVS share the bandwidth proportionally to their weights (1:1).

A. Software Schedulers

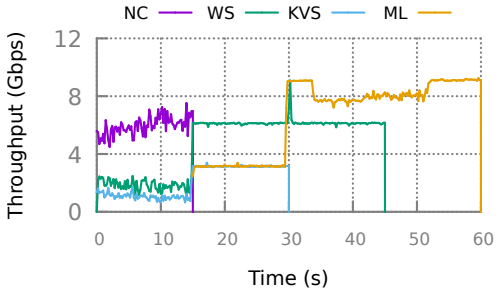


Fig. 3: Enforce network policies in the motivation example with Linux traffic control.

We demonstrate the inefficiency of kernel schedulers with an experiment on a 10Gbps link. The policies are specified as shown in Figure 2. The root qdsic is PRIO and we direct the traffic of NC to PRIO's band 0. We further attach HTB qdisc to PRIO's band 2 to serve the traffic from WS, KVS, and ML. More detailed information of our testbed settings is reported in Section V. We observe poor rate conformance as shown in Figure 3. First, the ingress flow rate of NC is 10Gbps. However, it is forced to drop packets from 0s to 15s

although its packets are directed to the highest priority queue. Second, we specify the ceiling rate of HTB's root class to 10Gbps. This should throttle the traffic rate under 10Gbps. However, the total bandwidth consumption from 15s to 45s is approximate 12Gbps. Third, HTB ignores our priority setting between KVS and ML. It equally divides bandwidth between them from 15s to 30s. However, this should only happen when their total bandwidth (current 6Gbps) is less than 4Gbps. Previous work [23] studies the inaccuracy of kernel schedulers. It discovers that the phenomenon is mainly caused by the global locking overhead upon each packet enqueue. Another disadvantage of software locks is high CPU consumption rate [4]. We also evaluate DPDK QoS scheduler [24] under the same setup. It improves the overall throughput meanwhile offering good rate conformance. We show that the consumed CPU cores of both schedulers could be saved by offloading scheduling functions on SmartNICs (Section V-B).

B. NIC Schedulers

Unfortunately, today's NIC queueing systems cannot support complicated QoS policies. For example, most NICs have multiple FIFO queues controlled by a round-robin scheduler, which only provides per queue fairness. Some powerful NICs integrate dedicated traffic manager to buffer and schedule packets according to predefined schemes [6]. For example, a typical traffic manager can organize queues with fixed levels of hierarchy. In each layer, queues are configured with priorities. Higher priority queues are strictly preferred. Queues of the same priority are served in a weighted round-robin manner. Enforcing conditional network policies on these queues (e.g., specify guaranteed bandwidth for ML and KVS) requires a runtime configuration, which is merely supported in today's products [25].

III. Problem Statement

In this section, we first overview the workflow of kernel classful schedulers. Then, we introduce the architecture of a typical NP-based SmartNIC to elaborate on the challenges brought by multi-core parallelism. Finally, we sketch out how FlowValve conducts calculations to overcome these difficulties and precisely simulate a variety of queuing procedures.

A. Classful Packet Scheduling

Classful packet scheduling in kernel is enforced with a classifier, multiple queues, and a scheduler (qdisc), as illustrated in Figure 1. Classes are defined by user policies and correspond to separated queues. The queue serves aggregated flows over the same conducted policy. Egress packets first match against filter rules to be classified into queues. Meanwhile, a scheduler serves these queues to send packets. For example, PRIO and HTB are both classful schedulers. PRIO always first sends packets from the highest priority non-empty queue. HTB uses a tree of token buckets to support fine-grained traffic control. It has two features: 1) traffic of each class is throttled at the leaf token bucket and 2) token insufficient classes can borrow from those sufficient ones, however, tokens are preferentially shared among sibling classes [21]. HTB is a commonly used traffic shaper and large-scale deployed in production networks [26].

B. NP-based SmartNICs

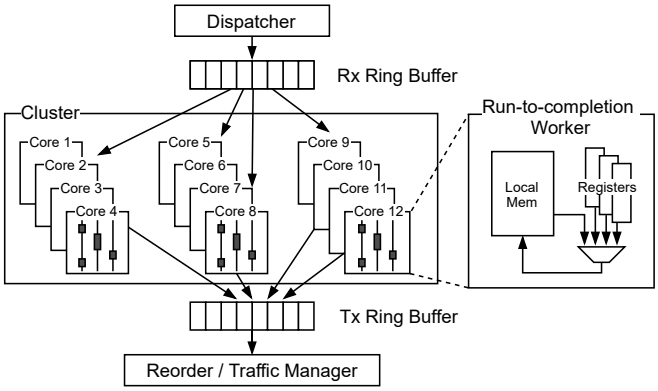


Fig. 4: An example of NP-based SmartNIC architecture. Packets are received from PCIe and forwarded to the wire.

SmartNICs open programming interfaces to developers for rapid innovation in supporting new network functions. NP-based SmartNICs utilize network processors to provide programmability. However, packet processing in NICs needs to meet a stringent time budget. So that network processors tailor specialized architectures and programming models to meet the performance goal. Figure 4 briefly shows the architecture of a Netronome SmartNIC. For the sake of space, we omit some accelerators (e.g., DMA Engine) and shared memory units (e.g., external DRAM).

The network processor has a large number of independent processing cores (also named micro-engines), which are grouped into several clusters. Ingress packets are distributed to one cluster by load-balancing module and waited to be pulled by the available cores. The processing core is further threaded (i.e., 4 or 8 threads). Threads share registers and local memory as well as executing the same program code. Programmers can develop NIC applications with high-level languages (e.g., P4 or Micro-C) following a run-to-completion software model. More specifically, a worker core is assigned to process one packet from the beginning to the end, including parsing, classifying, modifying, forwarding, etc. Finally, the reorder system sends packets out roughly according to their incoming sequences. Meanwhile, a manager core (other than worker cores) collects freed buffers and re-links them to the buffer lists for new incoming packets.

This architecture is inherently suitable for parallel packet processing, especially when there is little or no intervention between calculations performed by independent cores. We define such computation tasks as stateless processing. For example, the calculation of TCP checksum only involves a single packet. Conducting this calculation on a number of cores simultaneously without synchronization does not compromise the validity. In contrast, stateful processing cannot be performed precisely in the absence of synchronization. For

example, if there is a flow counter of value C read by two cores simultaneously, each core increases it by one locally and writes the value of $C + 1$ back to the variable memory. Then the result is imprecise because two packet passing only results in one counter increment. This could happen on some network processor systems that lack flow-to-core affinity. In order to improve statistics precision, we need a synchronization mechanism to protect the counter increment procedure, e.g., by locks. However, this inevitably lowers performance because one processing core must wait until the other core finishes using the shared variables. Unfortunately, scheduling algorithms desire much more complicated stateful processing than the counter example. They need to decide whether and when to forward a packet based on the global traffic status. We further elaborate on the challenges of offloading a classful packet scheduler.

C. Challenges

Offloading scheduling algorithms onto NPs is challenging. To deliver high performance, packet processing on NPs is highly optimized by taking advantage of parallel processing. Unfortunately, this fundamentally restricts offloaded choices and offloading implementations, which makes end-hosts qdiscs hard to migrate. We take the priority qdisc (PRIO) as an example.

Challenge 1: Multi-core parallelism. The kernel PRIO runs on a single CPU core and enqueues one packet at a time sequentially. If we follow the same paradigm on NPs, we would select one core to perform the scheduling task and leave the other cores conducting computation tasks. However, this is not scalable because as the traffic volume increases, the selected core should always provide the same throughput as the rest of cores amount to. Now the challenge is to develop parallel scheduling algorithms accordingly to prevent any single core from becoming a bottleneck. Simply running a scheduling function on each core is not enough. We must guarantee that the additional scheduling mechanism does not affect line rate performance. For example, if we have 50 cores simultaneously processing packets with different priorities, the overhead of strictly sorting those packets through complex inter-core communication is unacceptable. We need to reduce inter-core collaboration as much as possible and make the most of processing operations stateless.

Challenge 2: Constrained buffer management and packet queueing. Ideally, PRIO qdisc should be able to control the sending orders of buffered packets. However, the way network processors recirculates buffers is not customizable. As illustrated in Figure 4, packets are first copied from the receive (Rx) ring buffer to some fast memories near processing cores (e.g., local memory), which accelerates manipulations on packet headers. After processing, packets are copied to the shared transmit (Tx) ring buffer and fed into multiple FIFO queues in the traffic manager. This results in packets of all classes mixed in the Tx buffer and treated equally upon egress. Without scheduling, large flows of lower priority classes may easily overwhelm small flows of higher priority ones. Since there is no intermediate queues per traffic class, the challenge is how to avoid congestion on egress by handling packets on their way into Tx buffers.

D. Methodologies

To overcome the above challenges, FlowValve implements parallel scheduling by plugging a specific *scheduling* function into each core's processing routine. Under the hood, this function abstracts the Tx buffer (and hardware queues in the traffic manager) as a FIFO queue and performs specialized tail drop to avoid congestion. Unlike common tail drop, FlowValve prejudges which packet would cause buffer overflow to its belonged traffic class. Then it explicitly drops this packet in advance. In this way, FlowValve assigns buffers conceptually.

In case of priority scheduling, two flows (named f_{high} and f_{low}) compete for egress bandwidth. The scheduling function is triggered after each packet arrives at a processing core. Now the core needs to determine whether to discard this packet from the global traffic status. For example, on a 10Gbps link, assume that both f_{low} and f_{high} want to send traffic at 9Gbps. Without scheduling, egress flow rates of f_{high} and f_{low} are both 5Gbps (if their traffic patterns are the same). Because packets of both flows fill the NIC receive buffer on host side at the same speed, their ingress rates are the same. Without interventions on the NIC, they egress bandwidth naturally equals. However, priority scheduling desires that if f_{high} sends at 9Gbps, f_{low} can only get 1Gbps bandwidth. Later as f_{high} decreases its sending rate, f_{low} can accordingly occupy more bandwidth. The difficulty is how to make each processing core, at the time when holding packets of f_{low}, get aware of rate changes of f_{high} to correctly forward or discard packets of f_{low}.

Fortunately, modern network processors can perform accurate rate estimation on the data plane, which helps scheduling algorithms make runtime decisions. More specifically, many processing cores (e.g., $\geq$ 50) can collaborate to estimate flow rate with high sampling frequency. Besides, there are also specific hardware instructions for efficient atomic updates on flow counters/meters, eliminating considerable contention overhead brought by using software locks. With runtime traffic measurement, processing cores can first estimate an instant rate of f_{high}, called R_{high}. Then they can throttle f_{low} to the rate of (10Gbps - R_{high}), reserving enough room in the transmit buffers for packets of f_{high}. We present more detailed algorithms in Section IV-C.

E. Interfaces

FlowValve can fully offload PRIO and HTB meanwhile support qdisc chaining. It provides a shell command-line interface, called *fv*, which inherits the corresponding *tc* command options. For example, to specify the QoS policies in our motivation example, part of the commands go as follows.

```
$ fv qdisc add dev eth0 root handle 1: prio
$ fv qdisc add dev eth0 parent 1:3 handle htb 30: htb
$ ...
$ fv class add dev eth0 parent 30: classid 30:1 htb rate 10gbit
$ ...
$ fv filter ...
```

IV. Design

In this section, we first overview FlowValve workflow. Then, we detail the design of scheduling functions, including key data structures and algorithms. Next, we demonstrate how these functions can be executed parallelly to distribute bandwidth among active flows following their QoS requirements. Finally, we analyse the accuracy of scheduling functions.

A. FlowValve Workflow

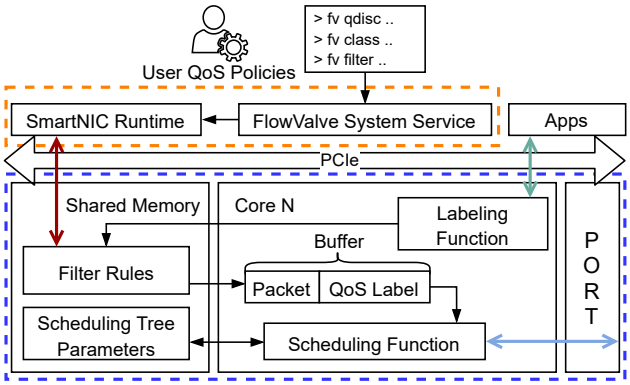


Fig. 5: FlowValve workflow.

FlowValve consists of a system service on host and a processing pipeline on SmartNICs, as shown in Figure 5. We refer to the system service as the front end (i.e., the orange dash line box) and the processing pipeline as the back end (i.e., the blue dash line box). The front end takes in user-specified QoS policies (i.e., *fv* command scripts), which include qdiscs, hierarchy classes, and filter rules. Then, it constructs a scheduling tree and populates configuration parameters and filter rules into the SmartNIC shared memory (i.e., the red arrow). This makes all the NIC processing cores can access the data. An application packet first matches filter rules to be classified (i.e., the green arrow). Then the matched packet gets its QoS labels, which indicate its belonged traffic classes and borrowing permissions. These labels are stored as metadata fields within the packet buffer without requiring additional header space. Later, the scheduling function extracts the QoS labels and updates the scheduling tree accordingly. Upon finishing update, a forwarding decision is made to either drop this packet or transmit it to the wire (i.e., the blue arrow).

Figure 6 illustrates the scheduling tree update procedure in our motivation example. We simplify the QoS requirements compared to the original version for ease of discussion. FlowValve uses token buckets to enforce rate control. The core idea of the scheduling function is to decide token fill rates for various buckets in each short time interval. For example, the NC flow has a strictly higher priority to consume any amount of available tokens. FlowValve simulates this behavior by measuring NC's token consumption rate and subtracts it from the ceiling rate to get the rest available token rate for the other traffic classes (i.e., S_1's subclasses). This ensures that the bandwidth requirement of NC is always first satisfied. Similarly, weighted scheduling between classes WS and S_2 is implemented by proportionally updating their token rates at each time when replenishing token buckets. In case of a class lacking tokens, FlowValve supports borrowing to maximize bandwidth utilization. The borrowing result depends on user configurations and runtime flow status. For example, in Figure 6(d), the solid shadow area indicates the guaranteed token consumption rate per traffic class. However, ML demands more than its committed share. So it borrows tokens from WS and KVS, which is allowed when tokens of these two classes are unused at the moment. The slash shadow areas illustrate the borrowed parts.

As shown in Figure 5, FlowValve offloads classifying in the *labeling function* and scheduling in the *scheduling function*. The labeling function essentially performs table lookups to match packets against filter rules. These operations are easily implemented with SmartNIC SDK. Therefore, we focus on discussing the idea and algorithms of the scheduling function. Before that, we briefly introduce the scheduling tree data structure.

B. Scheduling Trees

The scheduling tree describes class relationships (e.g., parent or sibling) and bandwidth distribution policies. Each tree node represents a traffic class and contains necessary operating parameters, e.g., token bucket configurations. The leaf class uses tokens to limit flow rate, while the root or interior class uses tokens to measure flow rate. As shown in Figure 5, QoS labels record the operations on the scheduling tree. There are two parts of a QoS label. The first part is the *hierarchy class label*, which records the sequence of traffic classes a packet belongs to. For example, as Figure 6(c) illustrates, packets of the ML service should have the hierarchy class label of $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow$ ML, which directs the scheduling function to update flow status of node S_0, S_1, S_2 and ML correspondingly. The second part is the *borrowing class label*, which indicates borrowing permissions of this flow. For example, the ML class is supposed to borrow bandwidth from classes WS and KVS.

C. Scheduling Functions

In support of NP architecture, our scheduling functions must supplement two lacking features of kernel qdiscs.

Feature 1: Parallel execution. There are two-fold advantages. One is to guarantee the offloading effectiveness. The other is to break the bounding between scheduling policies and specific queues. For example, the kernel HTB qdisc traverses tree nodes to update class info per packet transmission. According to its algorithm, we need a global lock to guarantee consistent updating of the shared scheduling tree on network processors

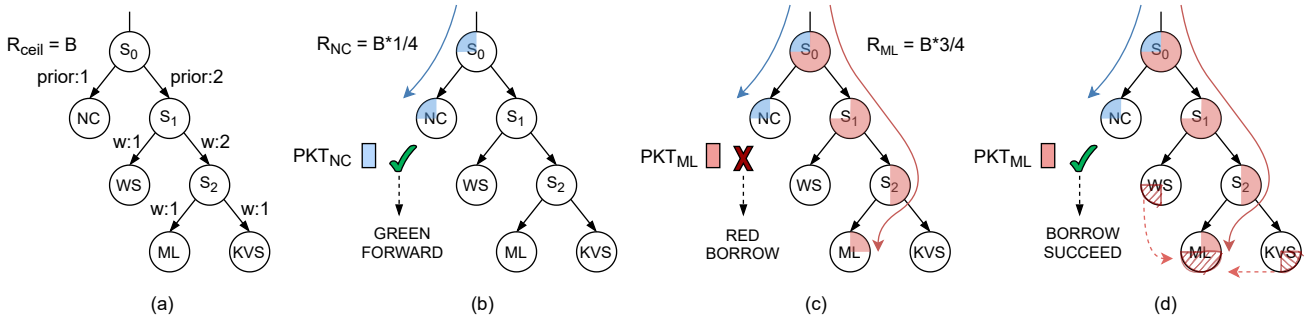


Fig. 6: Scheduling tree update procedure of the motivation example (Section II). (a) Scheduling tree structure. NC is strictly prior. WS, ML, and KVS proportionally share the rest of bandwidth. Their weights are annotated on the edges. (b) The flow of NC occupies one-fourth of the ceiling bandwidth. All its packets are marked green and forwarded straightly. (c) The flow rate of ML is three-fourths of the ceiling bandwidth, which exceeds its guaranteed one-fourth share. Therefore, packets of excess parts are marked red and attempt to borrow bandwidth from other idle classes. (d) Fortunately, they borrow enough bandwidth from the inactive WS and KVS classes. So the packets are marked green again and successfully transmitted to the wire without the worry of causing egress congestion.

(Figure 7-②). Unfortunately, this turns packet forwarding single-threaded. Because only one thread enters the critical updating code section while other threads are forced to wait. The network processor loses its parallel advantage, therefore making the offloading ineffective. Besides, HTB only supports a single queue. So that traffic from multiple tenants must first be aggregated before being scheduled. That is due to HTB queues packets before scheduling. In contrast, FlowValve schedules packets before queueing (Feature 2). So that FlowValve does not care how many input queues of the incoming traffic. The upper bound of queue numbers is simply restricted by the maximum number of virtual ports.

Feature 2: Predictable buffering. On-NIC hardware queues do not support user management (Section III-C). Once a packet resides in the buffer, it reserves a small fraction of the future bandwidth. The only way for a custom control is to predict which packet should arrive at the buffer so that its desired bandwidth is appropriately reserved. Fortunately, this prediction can be made through a combination of rate limiting and sharing procedures. The core idea is to continue estimating each flow's instant rate to make up for the lack of queueing information, which is essential in making scheduling decisions.

Now that it is necessary to re-implement algorithms in combination of hierarchical rate limiting and sharing on NPs, we break down the general procedure into three subprocedures. For a packet P, the scheduling function needs to sequentially update all its relevant traffic classes. For example, packet NC_1 needs to update classes S_0 and NC.

Subprocedure 1: Hierarchical rate-limiting. As shown in Figure 7-①, the update procedure without global synchronization on NPs is invalid. The final status of class S_0 is inaccurate due to multi-core data races. However, a valid yet sequential update procedure easily causes low throughput (Figure 7-②). An ideal solution is the parallel scenario (Figure 7-③), where packets NC_1 and ML_1 can simultaneously update classes NC and S_0. Note this scenario is not lock-free. For example, packet

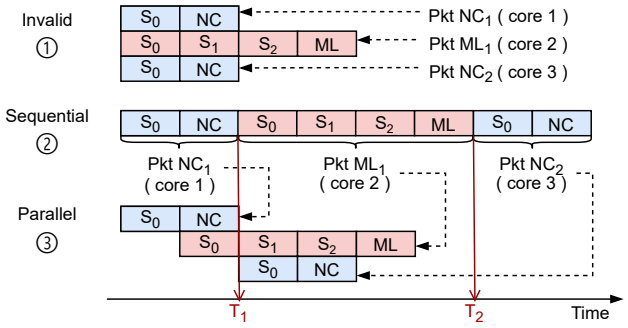


Fig. 7: Different scheduling tree update procedures.

NC_2 must wait ML_1 to release the lock before it can update class S_0. We guard the update code of each class with locks. We prove that our design can achieve hierarchical rate-limiting on NPs. Figure 8 illustrates the rate-limiting procedure for a single traffic class. Later, we extend this basic scene to any number of traffic classes encoded by arbitrary scheduling trees.

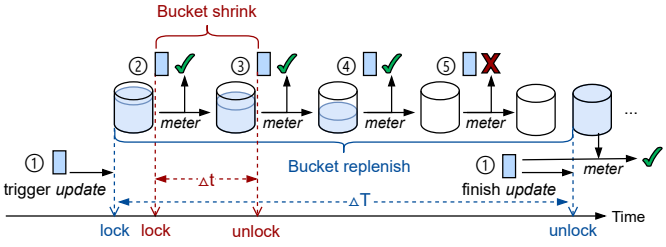


Fig. 8: Single class rate-limiting procedure.

Rate-limiting in Figure 8 is performed by the *meter* and *update* subprocedures. The *meter* implements rfc2697 [27]. It marks every arrival packet P with a color indicator σ. This color result reflects the relation between an approval

bandwidth b and the instant flow rate γ, where

$$\sigma = \begin{cases} \text{red}, & \text{if } \gamma > b, \\ \text{green}, & \text{otherwise.} \end{cases} \tag{1}$$

More specifically, each class is maintaining a token bucket. The meter function checks the number of tokens left in the bucket. If there are enough tokens, packets are marked green (Figure 8-①②③④) and the token number is reduced. Otherwise, packets are marked red (Figure 8-⑤) and the token number is unchanged. The token bucket is replenished regularly by the *update* subprocedure, which is also triggered by packet arrival events (Figure 8-①). Note that the *update* function is not executed on a per-packet basis. In a multi-core environment, only one core executes this procedure at a time. The other cores which process the same flow but do not grab the lock only execute the *meter* function. This does not compromise the validity. Because inside the *update* procedure, the supplement token number is calculated by multiplying the update interval ΔT (annotated in Figure 8) and token rate θ. The update interval is calculated from the recorded timestamps between the current and previous stages. While the token rate is either converted from a user-specified bandwidth or calculated at runtime.

The relation between a user-specified bandwidth b (bits/s) and the corresponding token rate θ (bits/cycle) of a specific class C is linear. Assume the frequency of the processing core is f. So that $b_C \times \Delta T = \theta_C \times \Delta T \times f$, which derives

$$\theta_C = \frac{b_C}{f}. \tag{2}$$

For example, in Figure 6(a), the token rate of root class S_0 is $\theta_{S_0} = \frac{B_{S_0}}{f}$. Runtime calculations happen when user policies specify priorities, weights or other conditions in bandwidth distributions.

1) Priority: Priority scheduling assigns the unsatisfied prior class as much bandwidth as it requires. While other less prior classes only get the residual bandwidth. The bandwidth requirement of a specific class C is linearly reflected on its token consumption rate Γ_C. Let L_P denote the token number consumed by forwarding packet P. Then Γ_C can be calculated by counting the amount of tokens consumed by forwarding packets of class C during a short time interval, which is

$$\Gamma_C = \frac{\sum L_P}{\Delta T}, P \in C. \tag{3}$$

The interval ΔT is exactly the aforementioned update interval. So that Γ_C is evaluated at each time when class C's token bucket gets replenished. This makes Γ_C reflect C's bandwidth requirement timely. The rest available token rate is

$$\theta_{rest} = \theta_{parent} - \Gamma_C \tag{4}$$

For example, in Figure 6(a), class NC is prior so we set its token rate to θ_{S_0}. NC's token consumption rate is $\Gamma_{NC} = \frac{\sum L_P}{\Delta T}, P \in NC$. Class S_1 gets the rest token rate except NC's consumption, which is $\theta_{S_1} = \theta_{S_0} - \Gamma_{NC}$.

2) Weight: Weighted scheduling divides the token rate of parent classes to their child classes proportionally. For any parent class with $N(N \geq 1)$ children, let w denote the child's weight, the token rate of the i–th child class is

$$\theta_{child_i} = \theta_{parent} \times w_i, \sum_{i=1}^{N} w_i = 1. \tag{5}$$

For example, in Figure 6(a), the token rate of WS and S_2 is calculated as $\theta_{WS} = \theta_{S_1} \times \frac{1}{3}$ and $\theta_{S_2} = \theta_{S_1} \times \frac{2}{3}$, respectively.

3) Other conditions: More complex rate control conditions can be defined similarly. For example, in Figure 6(a), the user may also want to restrict NC's ceiling bandwidth to $\frac{3}{4}B$ to prevent the potential starvation of S_1. This changes the previous token rate of NC from θ_{S_0} to $\frac{3}{4}\theta_{S_0}$. The calculation of S_1's token rate also changes to $\theta_{S_1} = \Gamma_{NC} < \frac{3}{4}\theta_{S_0} \,?\, \theta_{S_0} - \Gamma_{NC} \,:\, \frac{1}{4}\theta_{S_0}$. We abstract various condition templates and record them in the scheduling function. So that appropriate calculations are selected for concrete user policies.

Single class rate-limiting can be performed with high precision (i.e., Figure 8). The key point of class hierarchy extension is to feed desired token rates for specific classes (e.g., Equation 2, 4, 5). Methods to determine token rates at different tree levels are consistent. However, the convergence speeds may differ due to the asynchronous update procedures happening on different processing cores (i.e., Figure 7). We further analyse the accuracy of the scheduling function in section IV-D.

Subprocedure 2: Bandwidth sharing. Rate-limiting effectively prevents undesired bandwidth occupancy under severe traffic congestion. However, it also prohibits bandwidth sharing due to the simple token-based forwarding behavior. Continue on the previous example in Figure 6(a), class NC cannot use more than $\frac{3}{4}B$ bandwidth even if S_1's reservation is idle. One way to break through this limitation is to implement a complementary borrowing mechanism.

Bandwidth borrowing needs synchronization. A class is only supposed to lend out bandwidth when it has residual part. However, the bandwidth utilization can vary all the time. It is hard for borrowers to find appropriate lenders due to the lack of buffering. Our solution is simple. We explicitly provide indications of bandwidth usage through shadow buckets. The shadow bucket contains unconsumed tokens of a regular traffic class at each *update* epoch. We extend the idea of Equation 4 to calculate the lendable token rate as

$$\theta_{lendable} = \theta_C - \Gamma_C. \tag{6}$$

In contrast to Equation 4, where the rest token rate (i.e., θ_{rest}) is assigned to C's less prior sibling classes, the lendable token rate (i.e., $\theta_{lendable}$) in Equation 6 can be shared by any eligible starvation class. The scheduling function identifies eligible lenders from the QoS borrowing label (Section IV-B). Then it sequentially queries the shadow bucket of each lender class. The borrowing procedure is simply another practice of rate-limiting process as shown in Figure 8. Packets are forwarded only if they get enough tokens either from their original class

bucket or one of shadow buckets of lender classes. Otherwise, they would be dropped due to inadequate bandwidth.

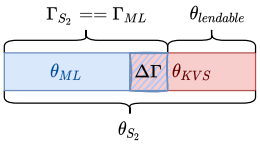


Fig. 9: Interior classes bandwidth sharing example.

Our borrowing procedure supports fine-grained traffic control. Such requirements frequently occur in network isolation scenarios between multiple tenants. Bandwidth sharing is preferent among interior classes by default. Continue on the example in Figure 6, at the time when KVS is idle while WS and ML are hungry. Since ML and KVS are deployed by the same tenant (i.e., interior classes), bandwidth sharing between them should be prior. This requires that the administrator sets S_2 to the *borrowing class label* of WS. She then sets S_2 and KVS to the label of ML. Figure 9 illustrates the relation of token rates between ML and KVS. Note that ML has class S_2 on its scheduling tree path. So that its flow rate is fully reflected on S_2's token consumption rate (i.e., $\Gamma_{S_2} == \Gamma_{ML}$). Then the lendable token rate of S_2 (i.e., $\theta_{lendable} = \theta_{S_2} - \Gamma_{S_2}$) already subtracts the ML-occupied lendable share of KVS (i.e., $\Delta\Gamma$). The more ML occupies, the less WS can borrow. If the administrator changes WS's label from S_2 to KVS and ML. Then WS and ML become equal in sharing KVS's reservation bandwidth because WS can directly query the shadow bucket of KVS as ML does.

Subprocedure 3: Expired status removal. Traffic class update is driven by packet arrival events. However, this may lead to some expired flow status left on the scheduling tree, which easily misleads the behavior of subsequent traffic classes. For example, when calculating token rates, results become expired after the processing of the last several packets within a flow. We solve this problem by adding timestamps in all kinds of shared status. Upon usage, the recorded timestamp is compared to the current one. If the interval is larger than a predefined threshold, we identify the status as expired. We restore expired status to its initial value. These operations are encapsulated into the *update* procedure.

Algorithm. Algorithm 1 details operations in the scheduling function. It first sequentially refreshes token buckets along the packet's scheduling tree path (Lines 1-4). This step also evaluates per class's token rate and lendable token rate. Meanwhile, the consumed token counter of each class records per packet passing (Line 5). At leaf class, the meter function throttles traffic rate. The green color result indicates sufficient bandwidth and passing approval (Lines 6-8). Otherwise, the packet undergoes an additional borrowing procedure. The scheduling function again sequentially queries the shadow bucket of lender classes specified in the borrowing label (Lines 9-13). The packet passes if there is residual bandwidth (Line 14-15). Otherwise, the final decision is drop (Line 16).

Algorithm 1: Scheduling Function

Input: Packet QoS Label *M*, Packet *P*, Scheduling Tree *Tree*
Output: Forwarding decision *DROP/FORWARD*

```
1  for each class c in M.hierarchy_class_label do
2      if grap_lock then
3          update (Tree[c].θ, Tree[c].Bucket);
4          release_lock;
5      count (Tree[c].Consume_Counter, P);
6  color ← meter (Tree[c].Bucket, P);
7  if color == GREEN then
8      return FORWARD;
9  for each class c in M.borrowing_class_label do
10     if grap_lock then
11         update (Tree[c].θ_lendable, Tree[c].Shadow_Bucket);
12         release_lock;
13     color ← meter (Tree[c].Shadow_Bucket, P);
14     if color == GREEN then
15         return FORWARD;
16 return DROP;
```

D. Scheduling Accuracy

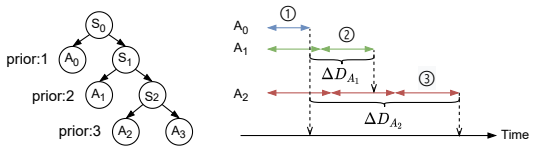


Fig. 10: Propagation delay of asynchronous update procedures.

Generally, the scheduling accuracy depends on the precision of single class rate-limiting and the impact of multi-class hierarchy. Single class rate-limiting on NPs is accurate, which mainly benefits from their specialized hardware design. Numeric instructions can execute on the transactional memory, which provides guaranteed consistency and high efficiency. For example, our *meter* function is essentially a wrapper around the atomic meter instruction [28]. Similar examples include various counters used in our scheduling functions. Moreover, we guard complex multiplication and division with software locks to avoid inaccuracy caused by data races. So that the main error of scheduling function would exist in multi-class interoperation, especially the propagation delay of token rate fluctuation.

Figure 10 illustrates a priority scheduling case. The prior sequence is A_0, A_1 and A_2. The flow rate change of class A_0 is evaluated and written back to the shared memory at the end of *update* stage ①. So that the latest info of A_0 can be only acquired by A_1 at the beginning of stage ②. Then, A_1 adjusts its token rate accordingly and replenishes its bucket with this new rate. Changes of A_1 finally take effect at the end of stage ② when the new rate and token number are recorded in the shared memory. This produces a propagation delay of ΔD_{A_1}. Similarly, the adjustment of A_2 takes effect at the end of stage ③ with the delay of ΔD_{A_2}. Note that

the *meter* operation is simultaneously executing during each *update* stage (Figure 8). This results in class A_1 and A_2 throttle at original flow rates during their delay intervals. The delay time is affected by the *update* code execution speed and tree depth of the class. Fortunately, modern NPs operate at high frequency (e.g., 1.2GHz), making each update stage finish within tens of milliseconds. In the next section, we demonstrate that FlowValve offers good rate conformance.

V. Evaluation

We implement the front end of FlowValve prototype in Python and build its back end on a Netronome Agilio CX 40GbE SmartNIC [6]. The backend processing pipeline is developed in P4 [29] and the scheduling function is written in Micro-C [30]. The P4 and Micro-C programs are linked together to run on the SmartNIC. The Netronome SmartNIC inserts in a PCIe Gen 3x8 slot. There is also an Intel X710 Quad Port 10GbE SFP+ Ethernet NIC. These two cards are wire connected. We use the Netronome NIC to send and schedule packets, while the Intel NIC receives and responses. Other settings of the end-host include 32GB memory and an 8-Core 2.3GHz CPU. The OS is CentOS 7. We choose DPDK driver because it provides high throughput, which is suitable for stress testing. There is also kernel compatible driver which FlowValve can run on.

Methodologies. Our evaluation aims to demonstrate the performance gain brought by offloading scheduling functions. Specifically, we compare FlowValve with two well-known and widely-adopted software schedulers, i.e., Linux HTB and DPDK QoS Scheduler. For tests on HTB, we use iperf3 to saturate network link and netperf to measure one-way latency. As for tests on FlowValve and DPDK QoS Scheduler, we build a TCP performance analyzing tool coupled with user space mTCP [31] stack. We first run network-bound testing applications with different QoS settings to observe the behavior of HTB, FlowValve, and DPDK QoS Scheduler. Then, we compare the metrics of maximum packet throughput, one-way latency, and CPU utilization to profile their scheduling capabilities.

Results. FlowValve can accurately enforce QoS policies while driving TCP traffic at 40Gbps, which contributes to freeing two CPU cores. It can further save more CPU resources as the packet rate increases. Although FlowValve has a little higher one-way delay at 40Gbps, it significantly lowers delay variation, making the egress traffic pattern more smooth and predictable.

A. QoS Policy Enforcement

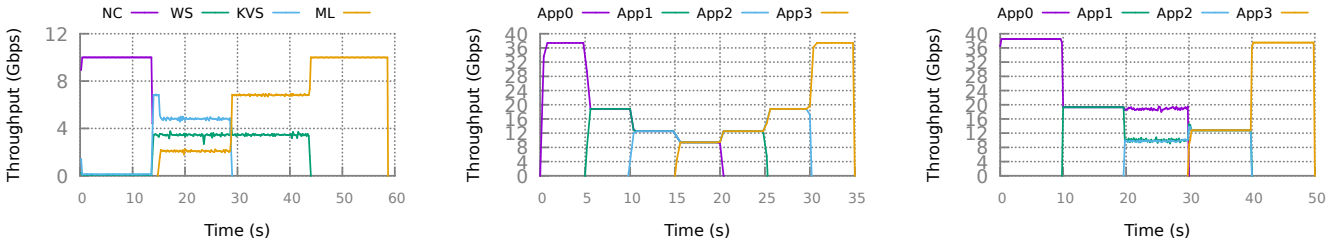
We demonstrate that our FlowValve prototype can enforce QoS policies accurately and efficiently. We perform experiments with the following settings. There are four active processes (App0-App3). Each process runs on a separated CPU core and sends traffic to the SmartNIC from an isolated virtual function. The virtual function associated interface has one transmit queue and one receive queue. This setting prevents interventions between the traffic of different processes.

Figure 11 illustrates the throughput achieved by different apps over time. In the first experiment, we replace HTB (and PRIO) in Figure 3 with FlowValve and enforce the same QoS policies to the traffic of four applications. Each application maintains a single TCP connection. Figure 11(a) shows that FlowValve outperforms HTB in the following aspects. First, FlowValve better prioritizes traffic of NC before time 15s by giving it all the available bandwidth. Second, FlowValve accurately distributes bandwidth among active traffic classes according to their weight and priority settings from time 15s to 30s. However, HTB ignores the priority between the KVS and ML classes to let them equally share the bandwidth. Third, FlowValve limits the total traffic rate to nearly 10Gbps all the way, while HTB breaks this limitation when the traffic of the KVS class stops at time 30s. This indicates high accuracy of flow rate estimation and token distribution algorithms.

Next, we scale the traffic volume to Netronome's line rate and increase the TCP connection number of each process from one to four. This can make the traffic rate of one app add up to 40Gbps because the receiver side has four 10GbE ports. We demonstrate that FlowValve can still enforce network policies with two kinds of QoS settings, fair queueing and weighted fair queueing. Figure 11(b) shows that FlowValve precisely distributes bandwidth among active flows and drives line rate. Further, we dynamically adjust TCP connection numbers in the range of 4 to 256 per process and ensure that different processes maintain different numbers of connections. The results remain the same as Figure 11(b). In the experiment of weighted fair queueing, as shown in Figure 11(c), we set policies as shown in Figure 12. The weight value of each traffic class is recorded in the third column. For example, the aggregated bandwidth of S_1's children should amount to App0 because of their equal weight. So we can observe that the appearance of App2's traffic at time 20s does not affect the traffic of App0. However, when App0 stops sending at time 30s, the other three classes equally share link bandwidth because we do not enforce weighted borrowing. Similarly, we randomize the TCP connection number of each traffic class. The overall results change little compared with Figure 11(c). In contrast to HTB, DPDK QoS Scheduler also succeeds in scheduling traffic at 40Gbps in our tests. So we further compare more metrics to profile their capabilities.

B. Offloading Effectiveness

We evaluate the benefit of offloading scheduling functions by comparing the maximum throughput and one-way delay of different packet schedulers. For both experiments, we omit tests on HTB when the total bandwidth exceeds 10Gbps because HTB cannot enforce network policies correctly on these high speed links. In the first experiment, we set network policies the same as the fair queueing experiment in Section V-A and inject fixed-length packets at full speed. Figure 13 compares the maximum throughput of FlowValve and DPDK QoS Scheduler over varying packet sizes. FlowValve can achieve higher throughput in all cases meanwhile saving CPU resources. In contrast, DPDK QoS Scheduler



(a) (b) (c)

Fig. 11: TCP throughput on 10Gbps and 40Gbps network links. (a) Motivation example. (b) 40Gbps fair queueing. (c) 40Gbps weighted fair queueing with scheduling policies illustrated in Figure 12.

Class	Parent class	Sibling class	Weight
App0	S0	S1	App0 : S1 = 1 : 1
App1	S1	S2	App1 : S2 = 1 : 1
App2	S2	App3	App2 : App3 = 1 : 1

Fig. 12: Weighted scheduling policies of experiment apps.

Packet Size (Byte)	FlowValve	DPDK QoS Scheduler	
	Maximum Throughput (Mpps)	Maximum Throughput (Mpps)	Scheduling Core Number
1518	3.23	2.25	1
		3.24	2
1024	4.75	4.49	2
64	19.69	9.06	4

Fig. 13: The maximum throughput of evaluated schedulers when enforcing fair queueing.

reaches performance expectations at the expense of consuming CPU cores. This becomes more obvious as the packet rate increases. For example, DPDK QoS Scheduler takes one core to schedule packets of 1518B at 2.25 Mpps but four cores to schedule packets of 64B at 9.06Mpps. When all connections only exchange 64B small packets, FlowValve can schedule packets at 19.69Mpps, which comes up to using eight CPU cores by DPDK QoS Scheduler. We further dig into the implementation of DPDK hierarchical scheduler block to find the reason for its performance degradation. The main problem is the high complexity for DPDK scheduler to make the queue operations thread safe, especially in terms of multi-core

Bandwidth (Gbps)	Scheduler	One-way Delay (us)	
		Mean	Standard Deviation
10	HTB	36.74	348.25
	FlowValve	30.05	0.30
	DPDK QoS	50.51	41.06
40	FlowValve	162.93 (161.01)	0.30 (0.11)
	DPDK QoS	70.38	83.29

Fig. 14: One-way delay of evaluated schedulers when enforcing fair queueing.

scaling requirements. The impact of using locking primitives (e.g., spinlocks) is significant. Also data structures sharing to external cores easily cause high miss rate of cache lines [24]. These all make the multi-core cooperation of DPDK QoS Scheduler less effective.

Figure 14 compares the one-way delay of different packet schedulers. When the bandwidth is limited to 10Gbps, FlowValve causes the lowest delay in packet transmission. However, as the bandwidth increases to 40Gbps, the delay increases 4x. We consider this is mainly caused by some other necessary processings on the SmartNIC. Because when we disable FlowValve to simply forward packets at 40Gbps, the delay is still as high as 161.01 microseconds. This also indicates that FlowValve is not a processing bottleneck. Surprisingly, FlowValve almost causes no variations in delay. We attribute this to the efficient algorithms performed by our scheduling function. Since the whole pipeline suffers from a bottleneck elsewhere that we could not change, the delay becomes stable and predictable. This makes FlowValve suitable for scheduling jitter-sensitive workloads, e.g., the video traffic.

VI. Discussion

Higher Line rate. The current FlowValve prototype is implemented on a 40GbE Netronome SmartNIC. However, we demonstrate that FlowValve is capable of processing packets as fast as at near 20Mpps (Figure 13). Saturating 100Gbps link with 1500B packets only needs a rate of 8.33Mpps, which indicates the potential of porting FlowValve to 100GbE SmartNICs to accommodate bandwidth growth in data center networks. Besides, a higher rate platform usually has more processing power. For example, as the number of micro-engines or the frequency of single micro-engine grows on the 100GbE cards [32], the peak packet processing rate is expected to improve as well.

Other SmartNICs. Netronome SmartNICs are typical and popular multi-core SoC SmartNICs. Offloading scheduling on similar platforms may borrow some insights from FlowValve's design. However, the programmer still needs considerable efforts to tailor the implementation for specific devices. Because different devices adapt different programming languages and models. As for another distinct product category, the FPGA-based platforms, we do not expect FlowValve to be worthy of reference due to the huge gap in architectural differences.

VII. Related Work

Programmable scheduling primitives. Part of the existing work focuses on developing new programmable scheduling primitives. Once these primitives are integrated into hardware, a range of algorithms can be expressed accordingly. This brings more flexibility into customizing scheduling algorithms. For example, a PIFO-based scheduler can program hierarchical work-conserving (e.g., Weighted Fair Queueing) and non-work-conserving (e.g., Token Bucket Filtering) algorithms at line rate [33]. The PIEO-based scheduler further overcomes PIFO's limitation on dequeueing by supplementing filters to express a wider range of algorithms [34]. Programmable Calendar Queues remove the limitation of finite priority levels in today's switch by performing dynamic escalation of packet priorities. The basic calendar queue abstraction is implemented by managing priority queues in the hardware traffic manager with primitive instructions (e.g., pause/resume queues) [25]. In contrast to these works, FlowValve does not aim to provide another new primitive. Instead, it explores the feasibility of offloading existing software algorithms onto SmartNICs with inherent queueing systems.

Efficient packet scheduling. Another chain of work aims to develop efficient and practical scheduling systems with software optimizations or specialized hardware. For example, Eiffel implements efficient packet ranking based on the Find First Set CPU instruction [35]. Carousel scales end-host traffic shaping significantly by using efficient timestamps and lock-free coordination as well as timely freeing resources in upper network layers [4]. However, finding efficient implementations is challenging. Recent proposals demonstrate that the upcoming programmable switches can also be utilized to offload scheduling functions. For example, SP-PIFO closely approximates the behavior of PIFO using strict-priority queues on Barefoot Tofino [36]. AFQ emulates the bit-by-bit round-robin algorithm to approximate fairness queueing at line rate [37]. HCSFQ further emulates two-layer hierarchy fair queueing with a single FIFO queue [38]. Compared with programmable switches, SmartNICs are more complicated offloading targets. An inappropriate design or implementation may even fail to achieve line rate performance. FlowValve carefully reduces locking overhead to conduct parallel scheduling on multi-core NPs. In contrast, works on programmable switches do not face this challenge because programmable switches deploy sequential pipelines. Similarly, Loom [13] presents a multi-queue NIC design coupled with a PIFO-based scheduler. We present detailed comparison with Loom in Figure 15.

	FlowValve	**Loom**
Programming Target	Multi-core Network Processor	Sequential Match-Action Table Pipeline
Scheduling Primitives	Hierarchical Token Buckets	Push-In-First-Out queues
Ease of Deployment	Off-the-shell products	Prototype

Fig. 15: Comparison with Loom.

VIII. Conclusion

We present FlowValve, the first parallel packet scheduler for NP-based SmartNICs that offloads critical functions of Linux traffic control. FlowValve takes advantage of multi-core parallelism on NPs to run classifying and scheduling functions on many processing cores simultaneously. Evaluations on a real SmartNIC show that FlowVlave offers higher throughput than non-offloaded schedulers and substantially reduces CPU and memory burdens on end-hosts. Our work takes the first step to offload stateful scheduling functions in existing SmartNIC products, and we hope it can bring more interest to this promising direction.

IX. Acknowledgements

We thank the anonymous reviewers for their insightful comments and suggestions. This work was supported by the National Key Research and Development Program of China under Grant No. 2018YFB1800201; the National Natural Science Foundation of China under Grant Nos. 62072091, 62032013 and 61872073; the LiaoNing Revitalization Talents Program under Grant No. XLYC1902010.

References

[1] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, A. Greenberg, and C. Kim, "Eyeq: Practical network performance isolation at the edge," in *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, 2013, pp. 297–311.

[2] R. Mittal, V. T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats, "Timely: Rtt-based congestion control for the datacenter," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 537–550, 2015.

[3] G. Kumar, N. Dukkipati, K. Jang, H. M. Wassel, X. Wu, B. Montazeri, Y. Wang, K. Springborn, C. Alfeld, M. Ryan *et al.*, "Swift: Delay is simple and effective for congestion control in the datacenter," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 514–528.

[4] A. Saeed, N. Dukkipati, V. Valancius, V. The Lam, C. Contavalli, and A. Vahdat, "Carousel: Scalable traffic shaping at end hosts," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017, pp. 404–417.

[5] H. Eran, L. Zeno, M. Tork, G. Malka, and M. Silberstein, "{NICA}: An infrastructure for inline acceleration of network applications," in *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, 2019, pp. 345–362.

[6] Netronome, *Agilio CX SmartNIC (1x40GbE)*, 2022. [Online]. Available: https://open-nfp.org/resources/

[7] Fungible, *The Fungible Data Processing Unit.*, 2022. [Online]. Available: https://www.fungible.com/product/dpu-platform/

[8] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung *et al.*, "Azure accelerated networking: Smartnics in the public cloud," in *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, 2018, pp. 51–66.

[9] Y. Moon, S. Lee, M. A. Jamshed, and K. Park, "Acceltcp: Accelerating network applications with stateful {TCP} offloading," in *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, 2020, pp. 77–92.

[10] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang, "Kv-direct: High-performance in-memory key-value store with programmable nic," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 137–152.

[11] J. Hypolite, J. Sonchack, S. Hershkop, N. Dautenhahn, A. DeHon, and J. M. Smith, "Deepmatch: practical deep packet inspection in the data plane using network processors," in *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, 2020, pp. 336–350.

[12] M. T. Arashloo, A. Lavrov, M. Ghobadi, J. Rexford, D. Walker, and D. Wentzlaff, "Enabling programmable transport protocols in high-speed nics," in *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, 2020, pp. 93–109.

[13] B. Stephens, A. Akella, and M. Swift, "Loom: Flexible and efficient {NIC} packet scheduling," in *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, 2019, pp. 33–46.

[14] S. Pontarelli, R. Bifulco, M. Bonola, C. Cascone, M. Spaziani, V. Bruschi, D. Sanvito, G. Siracusano, A. Capone, M. Honda *et al.*, "Flowblaze: Stateful packet processing in hardware," in *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, 2019, pp. 531–548.

[15] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar *et al.*, "The design and implementation of open vswitch," in *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, 2015, pp. 117–130.

[16] M. Hedayati, K. Shen, M. L. Scott, and M. Marty, "Multi-queue fair queuing," in *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, 2019, pp. 301–314.

[17] B. Stephens, A. Singhvi, A. Akella, and M. Swift, "Titan: Fair packet scheduling for commodity multiqueue nics," in *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, 2017, pp. 431–444.

[18] Netronome, *Virtual Switch Acceleration with OVS-TC and Agilio 40GbE SmartNICs*, 2018. [Online]. Available: http://www.netronome.com

[19] B. Hubert *et al.*, "Linux advanced routing & traffic control howto," *setembro de*, 2002.

[20] *Tc-prio*, 2001. [Online]. Available: https://man7.org/linux/man-pages/man8/tc-prio.8.html

[21] M. Devera and D. Cohen, *HTB Linux queuing discipline manual - user guide*, 2002. [Online]. Available: http://luxik.cdi.cz/ devik/qos/htb/manual/userg.htm

[22] "FlowValve code repository," https://github.com/Haers/FlowValve.

[23] J. D. Brouer, "Network stack challenges at increasing speeds," in *Proceedings of the Linux Conference, Auckland, New Zealand*, 2015, pp. 12–16.

[24] I. Corporation, *Quality of Service (QoS) Framework*, 2021. [Online]. Available: https://doc.dpdk.org/guides/prog_guide/qos_framework.html

[25] 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.

[35] A. Saeed, Y. Zhao, N. Dukkipati, E. Zegura, M. Ammar, K. Harras, and A. Vahdat, "Eiffel: Efficient and flexible software packet scheduling," in *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, 2019, pp. 17–32.

[26] A. Kumar, S. Jain, U. Naik, A. Raghuraman, N. Kasinadhuni, E. C. Zermeno, C. S. Gunn, J. Ai, B. Carlin, M. Amarandei-Stavila *et al.*, "Bwe: Flexible, hierarchical bandwidth allocation for wan distributed computing," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015, pp. 1–14.

[27] J. Heinanen and R. Guerin, "Rfc2697: a single rate three color marker," 1999.

[28] S. G. J, "Transactional memory that performs an atomic metering command," U.S. Patent 8 775 686, Jun. 7, 2014.

[29] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[30] Netronome, *Packet wire app in Micro-C*, 2021. [Online]. Available: https://open-nfp.org/the-classroom/

[31] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mtcp: a highly scalable user-level {TCP} stack for multicore systems," in *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, 2014, pp. 489–502.

[32] Netronome, *Agilio LX SmartNIC (1x100GbE)*, 2022. [Online]. Available: https://www.netronome.com/products/agilio-lx/

[33] 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*, 2016, pp. 44–57.

[34] 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.

[36] 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.

[37] N. K. Sharma, M. Liu, K. Atreya, and A. Krishnamurthy, "Approximating fair queueing on reconfigurable switches," in *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, 2018, pp. 1–16.

[38] Z. Yu, J. Wu, V. Braverman, I. Stoica, and X. Jin, "Twenty years after: Hierarchical core-stateless fair queueing." in *NSDI*, 2021, pp. 29–45.