Received 1 May 2023, accepted 23 May 2023, date of publication 31 May 2023, date of current version 7 June 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3281699

RESEARCH ARTICLE

Providing Near Per-Flow Scheduling in Commodity Switches Without Per-Flow Queues

SHIE-YUAN WANG, (Senior Member, IEEE), CHEN-YO SUN, YU-CHEN HSIAO, AND YI-BING LIN, (Fellow, IEEE)
Computer Science, National Yang Ming Chiao Tung University, Hsinchu 300, Taiwan

Corresponding author: Shie-Yuan Wang (shieyuan@cs.nycu.edu.tw)

This work was supported in part by the Center for Open Intelligent Connectivity from the Featured Areas Research Center Program within the framework of the Higher Education Sprout Project by the Ministry of Education (MOE) in Taiwan, in part by the National Science and Technology Council under Project NSTC 111-2622-8-A49-022 and Project NSTC 110-2221-E-A49-032-MY3, and in part by the Academia Sinica under Project 236c-1012030.

ABSTRACT Network quality of service (QoS) is essential for network applications. For many applications, getting a fair share of available bandwidth for their flows can prevent them from being blocked by other flows that do not respond to congestion. Providing per-flow scheduling in each output port of a commodity switch can isolate the flows that compete for the bandwidth of a bottleneck link. Although per-flow scheduling can maintain fair shares among competing flows, due to the high implementation costs of providing per-flow queues in commodity switches, this capability is rarely provided in commodity switches on the market. To address this need, we design and implement a near-per-flow scheduling scheme named Near Per-flow Scheduling (NPFS) in P4 programmable hardware switches and evaluate its performance. NPFS provides near-per-flow scheduling effectiveness in commodity switches that do not have per-flow queues in their output ports. NPFS utilizes the priority queues provided in most commodity switches and dynamically assigns competing flows to these queues based on their protocol types and current sending rates. Experimental results show that, when the number of competing flows is less than three times the number of queues, NPFS guarantees that the achieved bandwidths of these flows only deviate from their ideal fair shares by 5%.

INDEX TERMS Near per-flow scheduling (NPFS), per-flow scheduling, programmable switches, programming protocol-independent packet processors.

I. INTRODUCTION

Congestion control is very important for networks and identified as a top-10 problem in [1]. Allowing a flow to have its fair share of available bandwidth is essential. Otherwise, a flow may be blocked by other flows that exceed their fair shares of available bandwidth and do not respond to congestion. Nowadays, most commodity switches use FIFO queues to forward packets [2]. Both simplicity and low cost can be achieved if only one FIFO queue is needed for each output port [3]. Using such a FIFO queue scheme, however, an unfair situation is likely to occur when a TCP flow competes with a UDP flow for the bandwidth of a bottleneck link [1]. We call this problem the "TCP-vs-UDP" problem in this paper.

The unfair problem also occurs among competing UDP flows, which is called the "UDP-vs-UDP" problem in this paper. When a single queue is used and the total sending rate of all competing UDP flows exceeds the bandwidth of the bottleneck link, the packets of these UDP flows will be dropped by the queue when the queue is full. Suppose that there are N competing UDP flows, the sending rates of these flows are $r_1, r_2, \ldots, r_N$, respectively, and the sum of

The associate editor coordinating the review of this manuscript and approving it for publication was Quansheng Guan.

these sending rates is R. The achieved bandwidth of these competing UDP flows will be about $(r_1/R, r_2/R, \ldots, r_N/R)$. This bandwidth allocation is unfair and the cause of it is that the packets of a flow with a higher sending rate will achieve a higher proportion of the bottleneck bandwidth due to the FIFO scheduling [1].

As for the "TCP-vs-TCP" case in which N large TCP flows are competing for the bandwidth of a bottleneck link, each TCP flow will get its fair share (i.e., 1/N) of the bottleneck bandwidth if the sending rate of each TCP flow is higher than its fair share [1].

An effective way to provide fair bandwidth allocation is to use a per-flow scheduling scheme in each output port of a switch to maintain the fairness. However, to support such a scheme, the switch needs to provide per-flow queues in its output ports so that packets of different flows can be stored in different queues. Such a design is required to avoid the head-of-line blocking problem that may happen when a single queue is used. Once per-flow queues are provided, these per-flow queues can be manipulated with a simple scheduling policy (for example, round-robin) to fairly allocate bottleneck bandwidth among the competing flows.

Although a per-flow scheduling scheme is essential to maintain fairness among competing flows, this scheme is rarely supported by commodity switches on the market due to its high design complexity and implementation cost. In most commodity switches, the number of queues supported in an output port is small. For example, to support the eight different QoS priorities defined in the IEEE 802.1p task group, most commodity switches support only eight priority queues in an output port. As for the support of per-flow queues, when a large number of flows are passing an output port, it is much more difficult to dynamically manage a large and nondeterministic number of per-flow queues at a high speed for a high-bandwidth output port such as 10 Gbps or even 100 Gbps. Therefore, most commodity switches do not support per-flow queues in their switching ASIC chips.

A fair bandwidth allocation scheme can be provided by using an active queue management (AQM) approach (e.g., [4], [5], [6], [7]) or a fair queueing (FQ) approach (e.g., [8], [9], [11], [12], [13]). An AQM scheme determines which packet to be dropped when the queue is full while a FQ scheme determines which packet to be transmitted when the output port is free. In the last three decades, many fair bandwidth allocation schemes have been proposed in the literature. However, most of them were only evaluated by mathematical analyses, simulations, or software implementation in the FreeBSD/Linux kernel.

Recently, data-plane programmable hardware switches have been available on the market. P4 (Programming Protocol-Independent Packet Processor) [14], [15] is an open domain-specific programming language that can be used to program such switches. P4 provides a programmable data plane that can process packets as fast as fixed-function switches. By using P4, researchers can design, implement, and experiment their schemes with realistic network traffics to obtain reliable results. Due to this capability, there is a trend of using programmable switches to support fair bandwidth allocation. In a recent survey paper [16], the authors surveyed nine works that use P4 switches to implement customized or standard AQM algorithms. This shows the strong motivation for fair sharing of available bandwidth.

Although in [16] the authors listed nine works using P4 switches to implement AQM algorithms, these works do not implement FQ algorithms. Furthermore, only two of these nine works actually implemented their proposed algorithms in P4 commodity switches (the other seven works were implemented in the BMv2 P4 software switch). The authors in [17] and [18] proposed P4-based scheduling algorithms for FQ. However, both of these works need special hardware supports and cannot be implemented in P4 commodity switches. The research gaps in the community are designing and implementing a FQ approach in P4 commodity switches that achieves the FQ effect enabled by per-flow scheduling. The research challenge is designing and implementing such an approach in P4 commodity switches by using a very limited number of pipeline operations for each packet.

In an output port whose bandwidth is 100 Gbps (which is the port bandwidth of the hardware switches used in this work), the time interval (12 nanoseconds) between two consecutive 1500-byte packet transmissions is very short. This means that a packet scheduler designed for such an output port needs to finish each scheduling decision within 12 nanoseconds (or even shorter if the packet size is less than 1500 bytes). Due to this high-speed forwarding requirement, a sophisticated fair bandwidth allocation design may not be practically implemented in high-speed switching chips and instead only a very limited number of operations can be applied to a packet while it is passing the packet processing pipeline.

To bridge the gap, in this work we design and implement a scheme named Near Per-flow Scheduling (NPFS) to achieve the effectiveness of per-flow FQ in P4 commodity switches that do not support per-flow queues at their output ports.

The novelty of our scheme is that we use the limited number of priorities queues provided in most commodity switches and dynamically assign competing flows to these queues based on their protocol types (i.e., TCP or UDP) and current sending rates. To address the "TCP-vs-UDP" problem, we separate the flows into the TCP groups and UDP groups, respectively, and then assign these groups to different queues. To address the "UDP-vs-UDP" problem, we put UDP flows with similar sending rates into the same UDP group and assign the UDP groups of different rates to different queues. We dynamically assign weights to these queues based on the number of flows assigned to these queues and use a weighted round-robin packet scheduler to serve these queues. These weights are adjusted so that all TCP flows and UDP flows competing on the same bottleneck link can get their fair shares of the bottleneck bandwidth.

The contributions of our work consist of both novel designs and solid implementation. We have designed novel algorithms that prevent the "UDP-vs-UDP" and "TCP vs. UDP" unfairness problems and the "TCP vs. TCP" unfairness problem when multiple queues are used. Using Inventec D5264 P4 switches [19], we have successfully implemented these algorithms in P4 commodity switches and validated their functionality. Furthermore, experimental results show that, when the number of competing flows is less than three times the number of queues, our scheme guarantees that the flows obtain bandwidths on a bottleneck link that only deviate from their ideal fair shares by 5%.

The significant contributions of our work are summarized as follows:

- We have designed and successfully implemented a novel near-per-flow scheduling scheme NPFS for P4 commodity switches that do not have per-flow queues. This is the first such design and implementa-tion in P4 switches.
- Experimental results show that when the number of competing flows is less than three times the number of queues, NPFS guarantees that their achieved bandwidths on a bottleneck link only deviate from their ideal fair shares by 5%. Such results are near optimal.

This paper is organized as follows. In Section II, we survey related works. Section III briefly introduces the P4 switch architecture. Section IV presents the definition of fair share and our design goals. Section V presents the design and implementation of NPFS in a P4 switch. In Section VI, we evaluate the performance of NPFS using real-world traffic flows. In Section VII, we discuss several issues of NPFS. Finally, we conclude the paper and discuss future work in Section VIII.

II. RELATED WORK

A fair bandwidth allocation to competing flows can be provided by an active queue management (AQM) scheme, in which a switch maintains fairness by dropping the packets of ill-behaved flows [4], [5], [6], [7]. Another approach is a fair queueing (FQ) scheme, which ensures that traffic sources can get their fair bandwidth shares in the network by determining which packet to be transferred when the output port is free. NPFS can be viewed as an FQ scheme.

Nagle [20] proposed the first per-flow FQ scheme that has a set of independent queues for packets of each flow. Many follow-up solutions provided various types of FQ algorithms [8], [9], [10], [11], [12], [13]. Generally, a per-flow queueing algorithm needs two mechanisms — classifying packets by flows and managing the states of flows, both of which can be efficiently performed in P4 commodity switches.

Later on, several proposals improved the time efficiency of per-flow FQ [21], [22]. However, these sophisticated algorithms still could not be implemented in high-speed switches. To overcome this problem, the authors in [23] and [24] leveraged the core-edge switch architecture to reduce the complexity of core switches. Although the implementation complexity of core switches can be reduced, the implementation complexity of edge switches is still high.

These above schemes were evaluated only by mathematical analyses, simulations, or software implementation in the FreeBSD/Linux kernel. They were not implemented in high-speed commodity switches.

More recently, the authors in [17] used reconfigurable switches to develop the Approximating Fair Queueing (AFQ) scheme, which dynamically assigns the dequeue round number to physical queues and enqueues the packets into the corresponding queue to achieve approximate fair queueing. This scheme needs a new packet scheduler, called the "Rotating Strict Priority" scheduler, which is a custom design and requires the priority of a queue to be adjusted with respect to other queues after it is completely drained by the dequeue module. This mechanism is not currently supported in commodity switches, and existing reconfigurable switches do not expose the programmability of internal queues. Therefore, the authors implemented the AFQ scheme in a hardware prototype based on a Cavium programmable network processor to measure the performance results of the AFQ scheme. It is unclear from the paper whether the AFQ scheme can be exercised in a more general P4 commodity switch instead of the specific hardware prototype.

The authors in [18] presented a scheme called Fair Dynamic Priority Assignment (FDPA), which uses rate estimators to dynamically assign traffic flows to different priority queues. In FDPA the priorities of the queues can remain fixed and need not be varied at a high frequency. FDPA is implemented in the Linux kernel running on a desktop machine instead of a P4 commodity switch. Furthermore, it can only handle the case where all competing flows are TCP flows. In contrast, NPFS can handle other cases such as the "TCP-vs-UDP" case and the "UDP-vs-UDP" case.

Rather than providing a new FQ scheme, several recent studies have proposed innovative packet schedulers that can be used to implement sophisticated algorithms in switches. For example, the authors in [25] introduced the programmable calendar queues with hardware prototype implementation; the authors in [26] and [27] proposed the hardware design of PIFO- based schedulers. Currently, these innovative programmable packet schedulers are only prototypes and are not available yet in commodity switches. The authors in [28] provided an approximate PIFO scheduler (SP-PIFO) by using native strict-priority queues in a P4 switch. Although SP-PIFO is implemented in P4 hardware switches, its purpose is to approximate the PIFO scheduler rather than provide fair bandwidth allocation among competing TCP and UDP flows. Thus, it does not support fine-grained per-flow scheduling.

Recently, the authors in [29] enhanced their Core-Stateless Fair Queueing (CSFQ) scheme presented in [23] to support hierarchical packet scheduling. They implemented their new scheme (named HCSFQ) in a P4 hardware switch and evaluated its performance. Although HCSFQ, like our NPFS scheme, has been implemented in a P4 hardware switch,

TABLE 1. Comparison of NPFS with related works.

Ref	Method	Adopt FIFO Queue	Enable Fine-grained Scheduling	No Need to Modify Packet Header	No Need to Require Multiple Switches	Implemented in Commodity P4 Switches
[17]	Queueing Algorithm	Yes	Yes	Yes	Yes	No
[18]	Queueing Algorithm	Yes	Yes	Yes	Yes	No
[26]	Packet Scheduler	No	No	Yes	Yes	No
[27]	Packet Scheduler	No	No	Yes	Yes	No
[28]	Packet Scheduler	No	No	Yes	Yes	Yes
[29]	Queueing Algorithm	Yes	Yes	No	No	Yes
NPFS	Queueing Algorithm	Yes	Yes	Yes	Yes	Yes

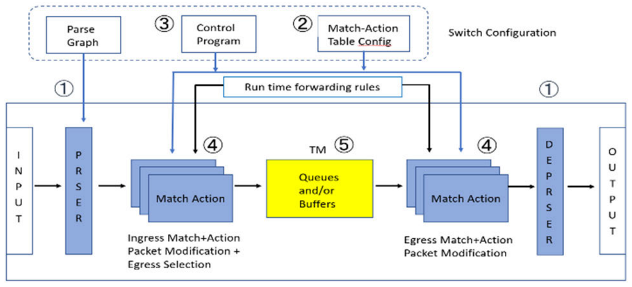

Parse Graph
③
Control Program
②
Match-Action Table Config
Switch Configuration
①
Run time forwarding rules
①
INPUT
PRSER
④
Match Action
Ingress Match+Action Packet Modification + Egress Selection
TM ⑤
Queues and/or Buffers
④
Match Action
Egress Match+Action Packet Modification
DEPRSER
OUTPUT

FIGURE 1. The packet forwarding model of a P4 hardware switch (blue blocks are programmable and the yellow block is reconfigurable).

its designs are totally different from those of NPFS. Firstly, in HCSFQ the header of every packet needs to be modified to carry two pieces of state including: (1) the arrival rate estimation of the flow that the packet belongs to; and (2) a list of node IDs that indicate the flow aggregates that the packet belongs to in the flow hierarchy. In contrast, NPFS need not modify the header of any packet. Secondly, HCSFQ uses probabilistic packet dropping in its scheme while NPFS need not. Thirdly, HCSFQ requires coordination between edge and core switches in a network while NPFS does not require coordination between switches.

There are several specific examples of using programmable switches for packet scheduling [30], [31], [32]. Although their intended purposes are different from ours, they show the attractiveness of this technique. In [33], the authors uses ECN over several packet schedulers to achieve the effects of these schedulers. Since ECN marking must be carried by TCP ACK packets to trigger the congestion control of TCP senders, this work can only solve TCP vs. TCP problems. However, our NPFS scheme can also solve "TCP vs. UDP" and "UDP vs. UDP" problems.

To facilitate a comprehensive understanding of the distinctions between NPFS and the related works, we present a detailed comparative analysis in Table 1, encompassing diverse aspects.

III. P4 SWITCH ARCHITECTURE

A P4 switch is a reconfigurable switch that uses multi-stage pipelines to process packets. With reconfigurability, the programmers can change the way the switch processes packets.

The pipeline logic of a P4 switch is controlled by the P4 language [34], [35]. A P4 program defines how the headers of a packet are parsed into header fields and how the fields are used by the match-action tables. Figure 1 shows the packet forwarding model for a P4 hardware switch. A P4 program can be mainly split into three components: Parser graph (Figure 1 (1)), Match-Action Table Configuration (Figure 1 (2)), and Control Program (Figure 1 (3)).

The parser graph (Figure 1 (1)) defines the process of packet header extraction, including the header formats and the parser behavior. The parser is a finite state machine. In each state, the parser extracts the corresponding bits from the packet and sends the packet to the next state. The parser stores the extracted bits in the header structure objects (e.g., the Ethernet header and IPv4 header) called header fields. In addition, the parser can generate an arbitrary value in the extraction process and send it to other components by storing it in the switch's metadata objects. The header fields and metadata objects can be accessed and modified by the match-action tables according to the needs of the P4 application. The deparser reorganizes the header fields and packet payload to form the packet before forwarding it out of the switch.

Both the ingress and egress pipelines are composed of several match-action tables. In a match-action table, a pair of a key and an action is called a match-action entry. Match-action table configuration (Figure 1 (2)) defines the key fields (i.e., the header fields of a packet used for matching) and the action code of the match-action table. In addition, the processing order of match-action tables is defined by the control program (Figure 1 (3)). The match-action tables are read-only for the data plane, but their entries can be modified by the control plane.

A control plane program can use pre-built APIs to manage the value of key and the input parameter data of the action used in a match-action table. For each packet, the match-action table (Figure 1 (4)) uses the information of the packet to compare with the key value of each match-action entry. If there is a match, the packet will be processed by the action associated with the matched entry. If the packet cannot match any entry in the table, the default action will be executed to handle the packet. An action can modify the header fields of the packet or insert/delete a header into/from the packet. The output port and the queue in the selected output port for a packet can be determined in the ingress pipeline by specifying the values of the metadata. Once a packet leaves the ingress pipeline, the output port and output queue determined for the packet cannot be changed.

The Traffic Manager (TM; Figure 1 (5)) manages physical queues and buffer space to store the packets before they are sent out of the switch. The TM can perform several scheduling algorithms such as FIFO, Weighted Round-Robin, and Strict Priority to determine the forwarding order of the packets.

The P4 switch used in this study has 64 ports and the bandwidth of each port is 100 Gbps. Packets can be forwarded by this switch at line rate.

IV. DESIGN GOALS OF NPFS

In this section, we first define the fair shares of bottleneck bandwidth for the flows competing over a link. Then, we explain the design goals of NPFS. P4 switches use the match-action pipeline architecture to process packets at a high speed in an output port. However, to support a large port bandwidth such as 100 Gbps, the types, functionality, and complexity of actions that can be executed in a table are very limited. For example, floating-point numbers cannot be used in an action and the multiply and divide operations cannot be executed in an action. Besides, a design that can be easily implemented as a software program running on a general CPU must be transformed to a series of match-action tables. At run time, these tables must be configured with proper rules to correctly trigger the execution of user-defined actions. Due to these constraints, most sophisticated FQ algorithms described in Section II cannot be practically implemented in P4 switches. On the other hand, we can implement the designs of NPFS as a series of match-action tables with properly configured rules. NFPS can provide near per-flow scheduling effectiveness at line rate of 100 Gbps, which cannot be achieved by the previous studies.

A. DEFINITIONS OF FAIR SHARE AND NOTATIONS USED IN THIS PAPER

Assume that there are $N_{f,t}$ TCP flows and $N_{f,u}$ UDP flows competing for the bottleneck bandwidth B_B at an output port, where $N_{f,t}$ or $N_{f,u}$ may be 0 but will not be 0 at the same time. Let N_f denote the total number of flows passing the output port and its value is $N_{f,t} + N_{f,u}$. We define a flow to be a small flow if its bandwidth usage is less than B_B/N_f. Otherwise, it is a large flow. The number of small flows and large flows are denoted by N_s and N_l, respectively, and $N_s + N_l = N_f$. In the following equations, we assume that N_l is larger than 0.

For a small flow, since it cannot use all of B_B/N_f, its fair share F_s is its transmission rate. As for a large flow, suppose that the bandwidth usages of these small flows are denoted by $u_s(i)$, where i denotes the index of the i-th small flow, then the fair share F_l of each large flow is defined as below:

$$B_R = \sum_{i=1}^{N_s} (B_B/N_f - u_s(i))$$
$$F_l = B_B/N_f + B_R/N_l$$

In the above equations, B_R is the total amount of the original fair- share bandwidth that is not used by small flows. Because B_R should be fairly shared by all large flows, B_R is divided by N_l and then B_R/N_l is added to B_B/N_f, which is the original fair share of a large flow, to obtain the new fair share F_l of a large flow. If the new fair shares of some large flows are larger than their bandwidth usages, then the above process is repeated to reallocate their left bandwidth among those flows whose bandwidth usages are above the new fair shares. The above process is repeated until the fair shares of all flows are determined. Table 2 summarizes the above notations.

TABLE 2. Notations used for determining the fair shares of competing flows.

Symbol	Definition
$N_{f,t}$	Number of TCP flows
$N_{f,u}$	Number of UDP flows
N_f	Number of TCP and UDP flows $= N_{f,t} + N_{f,u}$
B_B	Bottleneck bandwidth at an output port
$u_s(i)$	Bandwidth usage of the i-th small flow
N_l	Number of large flows
N_s	Number of small flows
B_R	Fair share bandwidth not used by small flows
F_l	Fair share of a large flow
F_s	Fair share of a small flow

B. DESIGN GOALS

There are three design goals for NPFS: (i) Preventing unfairness problems in the "TCP-vs-UDP" case, (ii) Preventing unfairness problems in the "TCP-vs-TCP" case when multiple queues are used, and (iii) Mitigating unfairness problems in the "UDP-vs-UDP" case. In the following section, we structure the presentation of the design and implementation of NPFS based on these goals.

V. DESIGN AND IMPLEMENTATION

We have designed and implemented NPFS in Inventec D5264 P4 switches (D5264-P4 in short) [19], which use Barefoot/Intel's Tofino chip as their switching ASIC [35] chips.

A. DESIGNS FOR PREVENTING UNFAIRENESS PROBLEMS IN THE "TCP-vs-UDP" CASE

D5264-P4 has up to 32 physical queues in each 100 Gbps port and this number can be changed by a pre-built API function [19]. Assume that there are N_q queues used in an output port, we split the queues into three different queue sets: (i) the default queue set, (ii) the TCP queue set, and (iii) the UDP queue set. The default queue set has one queue, the TCP queue set has $\lceil N_q/2 \rceil - 1$ queues, and the UDP queue set has $\lfloor N_q/2 \rfloor$ queues.

The default queue set has only one queue (called the default queue) to temporarily store the packets of new flows whose match-action entries have not been inserted into the match-action table yet. The bandwidth usage of a new flow will be monitored by the control program of NPFS for one second. After one second, it will be assigned to a queue in the TCP (or the UDP) queue set based on its protocol type. Because D5264-P4 supports the strict-priority scheduling method, which is commonly supported in commodity switches, NPFS gives the default queue a higher priority than the queues in the TCP and the UDP queue sets. This design allows NPFS to observe the real bandwidth usage of a flow in the first second of its duration before starting to control its bandwidth usage.

The queues in the TCP queue set (called TCP queues) and the queues in the UDP queue set (called UDP queues) are used to store the packets of TCP flows and UDP flows, respectively, and their priorities are set the same. Algorithm 1 shows the mechanism used to dispatch TCP and UDP flows into the respective TCP and UDP queue sets. (Note that all Algorithms shown in this paper are in the pseudocode format.) Further details about the operations between the control plane and data plane will be explained in Sections V-D and V-E. Figure 2 shows the system architecture of NPFS, including a default queue, several TCP queues, and several UDP queues. In NPFS, because all flows sharing a TCP queue are TCP flows and all flows sharing a UDP queue are all UDP flows, the "TCP-vs-UDP" problem is avoided.

Algorithm 1 The algorithm used to dispatch TCP and UDP flows into the respective TCP and UDP queue sets

```
Require:
    Array of new flows, $F$;
    Array of TCP flows, $T$;
    Array of UDP flows, $U$;
1: for $i \leftarrow 0$ to $len(F) - 1$ do
2:     if $F[i]$ is a UDP flow then
3:         $U$ append $F[i]$
4:     else
5:         $T$ append $F[i]$
```

For the TCP queues and UDP queues, NPFS uses the deficit-weighted round-robin scheduling method [21] to determine the transferring amount of each queue in each round. Due to its simplicity and effectiveness, this method is supported in most commodity switches. The control program of NPFS uses pre-built APIs to dynamically set the weight

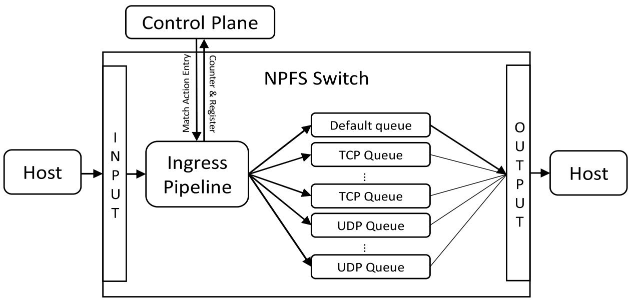


FIGURE 2. The system architecture of the NPFS scheme.

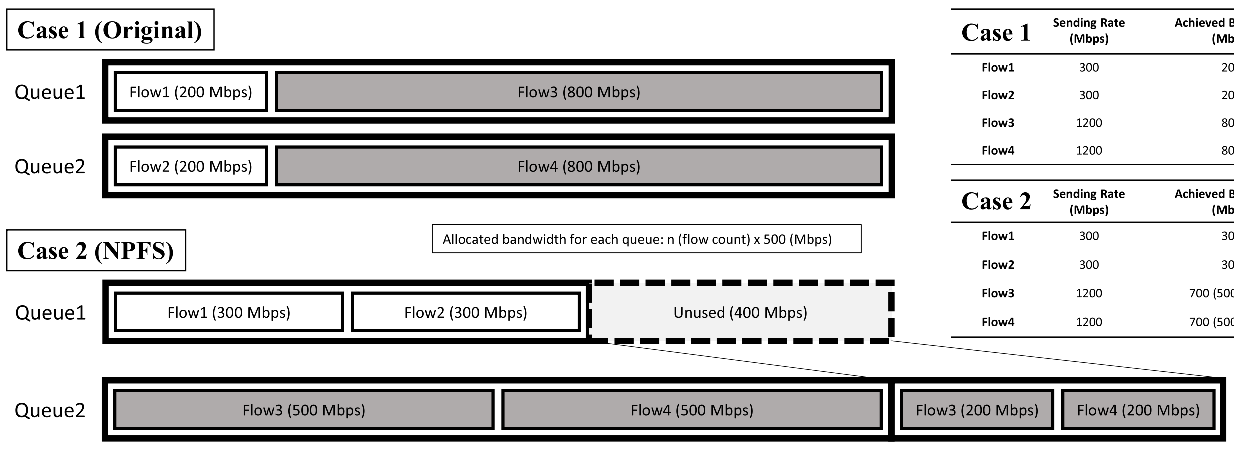
TABLE 3. Notations used for describing the NPFS system architecture.

Symbol	Definition
N_q	Total number of queues in an output port
$N_{q,t}$	Number of queues reserved for TCP flows in an output port = $\lceil N_q/2 \rceil - 1$
$N_{q,u}$	Number of queues reserved for UDP flows in an output port = $\lfloor N_q/2 \rfloor$
N_q^*	Number of queues reserved for TCP and UDP flows in an output port = $N_{q,t} + N_{q,u} = N_q - 1$
$N_{f,q}$	Number of flows sharing a specific queue, which is either a TCP or a UDP queue
w_i	The weight assigned to the i-th queue, which is either a TCP or a UDP queue
$B_{q,d}$	Bandwidth consumed by the packets dispatched to the default queue, which has the highest priority
$B_{q,t}$	Aggregate bandwidth of the queues in the TCP queue set, achieved from the deficit weighted round-robin packet scheduler
$B_{q,u}$	Aggregate bandwidth of the queues in the UDP queue set, achieved from the deficit weighted round-robin packet scheduler

of each queue. The maximum weight of a queue in the used P4 switch can be set to 1,024. NPFS sets the weight of each queue to $N_{f,q} \times 20$, where $N_{f,q}$ is the number of flows sharing the queue. Let w_i denote the weight assigned to the queue with the index of i. Let $B_{q,d}$ denote the bandwidth consumed by the packets dispatched to the default queue. Then, the available bandwidth of the i-th queue is equal to $(B_B - B_{q,d}) \times (w_i / \sum_{k=1}^{N_q^*} w_k)$, where N_q^* is the total number of queues in the TCP queue set and UDP queue set. With these settings, the available bandwidth of each TCP queue or UDP queue will be proportional to the number of flows using the queue. This bandwidth allocation among the TCP and the UDP queues is the first step towards achieving the goal of fair sharing. Table 3 summaries the above notations.

B. DESIGNS FOR PREVENTING UNFAIRNESS PROBLEMS IN THE "TCP-vs-TCP" CASE WITH MULTIPLE QUEUES

The "TCP-vs-TCP" case with a single queue has been discussed in Section I, in which all TCP flows sharing a single queue will get their fair shares without problems. However, if $N_{f,t}$ TCP flows share a set of queues and some of their sending rates are less than the aggregate bandwidth of these queues $B_{q,t}$ divided by $N_{f,t}$, a TCP flow whose sending



Case 1	Sending Rate (Mbps)	Achieved Bandwidth (Mbps)
Flow1	300	200
Flow2	300	200
Flow3	1200	800
Flow4	1200	800

Case 2	Sending Rate (Mbps)	Achieved Bandwidth (Mbps)
Flow1	300	300
Flow2	300	300
Flow3	1200	700 (500 + 200)
Flow4	1200	700 (500 + 200)

FIGURE 3. An example showing how the NPFS scheme assigns UDP flows to the UDP queues.

rate is higher than $B_{q,t}/N_{f,t}$ may unfairly get an amount of bandwidth that is higher than its fair share. NPFS resolves this issue as follows.

After allocating a set of queues for TCP flows, NPFS determines the matching between TCP flows and TCP queues. We use one of the TCP queues (called the small flow queue) to store the packets of the TCP flows with sending rates less than $B_{q,t}/N_{f,t}$. If we assign a TCP flow whose bandwidth usage is less than $B_{q,t}/N_{f,t}$ and another TCP flow whose usage is larger than $B_{q,t}/N_{f,t}$ to the same queue, the remaining bandwidth left by the small TCP flow will be solely used by the large one. However, the remaining bandwidth should be reallocated to all other flows in the system, rather than just the flow sharing the same queue.

For example, if a small and a large TCP flows share a queue and their sending rates are 300 Mbps and 1,000 Mbps, respectively. Assuming that $B_{q,t}/N_{f,t}$ is 500 Mbps. Then, the bandwidth allocated to the queue to serve the two flows is 1,000 (500 $\times$ 2) Mbps. In this scenario, the achieved bandwidth of the two TCP flows will be 300 Mbps and 700 Mbps, respectively. Because the small flow does not consume all of the 500 Mbps bandwidth allocated to it, its left bandwidth 200 Mbps is used by the large flow, which explains why the large flow receives 700 Mbps bandwidth. However, it is fairer if the left bandwidth 200 Mbps is shared by all other flows in the system, rather than just the large flow using the same queue.

To solve this unfairness problem, we assign the TCP flows whose sending rates are less than $B_{q,t}/N_{f,t}$ to the small flow queue. As explained in Section V-A, NPFS uses the deficit weighted round-robin scheduler and assigns a weight to a queue that is proportional to the number of flows assigned to the queue. Since the aggregate sending rate of all small flows assigned to the small flow queue will be less than the bandwidth of the small flow queue achieved from the deficit weighted round-robin scheduler, the left bandwidth of the small flow queue will be automatically and fairly shared by all other queues (and thus the flows served by them) managed by the deficit weighted round-robin scheduler. As for the TCP

Algorithm 2 The algorithm used to move TCP flows between the small flow queue and non small flow queues

```
Require:
    Fair share of a TCP flow, $fairShare$;
    Array of TCP flows, $T$; Flow's sending rate, $T[i].rate$;
    Array of TCP queues containing arrays of mapped TCP flows, $Q_T$;
    Array containing mapped TCP flows for the special TCP small queue, $Q_{Tsmall}$;
    Index used to assign TCP flows to TCP queues in a round robin manner, $next$;
1:  for $i \leftarrow 0$ to $len(Q_T) - 1$ do
2:      for $j \leftarrow 0$ to $len(Q_T[i]) - 1$ do
3:          if $Q_T\ [i][j].rate < fairShare$ then
4:              $Q_{Tsmall}$ append $Q_T[i]\ [j]$
5:              $Q_T[i]$ remove element at index $j$
6:
7:  for $i \leftarrow 0$ to $len(Q_{Tsmall}) - 1$ do
8:      if $Q_{Tsmall}[i].rate >= fairShare$ than
9:          $Q_T[next]$ append $Q_{Tsmall}[i]$
10:         $next \leftarrow next + 1 \bmod len(Q_T)$
11:         $Q_{Tsmall}$ remove element at index $i$
```

flows whose sending rates are larger than $B_{q,t}/N_{f,t}$, NPFS assigns them to non-small flow queues and sets the weights of these queues to their corresponding values as explained in Section V-A. Due to the property of TCP congestion control algorithm, these large TCP flows will share the available bandwidth of a queue equally.

In NPFS, the control program will periodically monitor, update, and store the sending rates of all flows in the counters of a P4 switch every second. After these operations are performed in each second, the control program will compare the sending rate of each TCP flow with the current value of $B_{q,t}/N_{f,t}$, where $N_{f,t}$ may vary over time. If a TCP flow is currently served by the small flow queue and its sending rate has become larger than $B_{q,t}/N_{f,t}$, the control program will

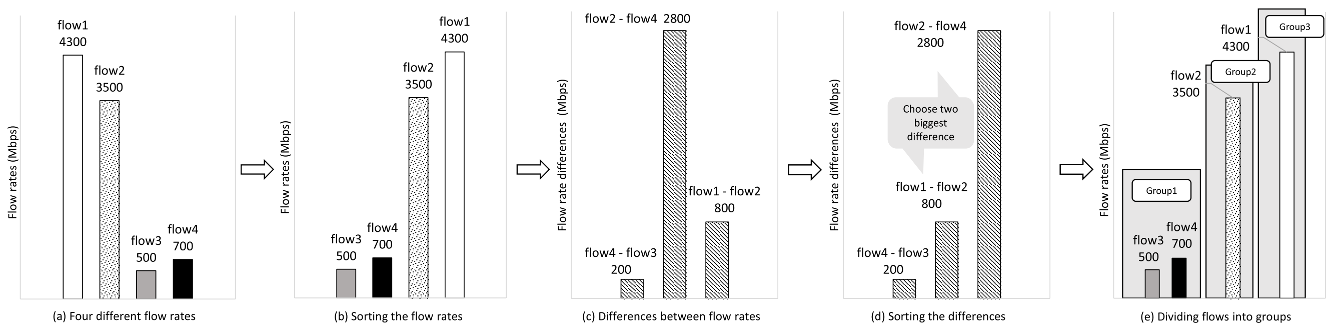


FIGURE 4. An example illustrating how Algorithm 3 partitions four flows into three groups.

move it to a non-small flow queue. In contrast, if a TCP flow is currently served by a non-small flow queue and its sending rate has become less than $B_{q,t}/N_{f,t}$, the control program will move it to the small flow queue. NPFS avoids frequently moving a TCP flow among different queues in order not to generate excessive TCP out-of-order packets. Algorithm 2 describes the above designs.

C. DESIGNS FOR MITIGATING UNFAIRNESS PROBLEMS IN THE "UDP-vs-UDP" CASE

The "UDP-vs-UDP" problem may occur when there are multiple UDP flows sharing a queue. In this scenario, the achieved bandwidths of these flows will be proportional to their sending rates, which may exceed their fair shares. When assigning multiple UDP flows to multiple queues, the "UDP-vs-UDP" problem may occur if the assignment does not consider this problem. In the following, we use Figure 3 as an example to illustrate this problem and show how it can be solved in NPFS. In this example, Case 1 represents the case in which the problem occurs and thus competing flows do not receive their fair shares. In contrast, Case 2 represents the case in which competing flows receive their fair shares in NPFS.

In this example, there are four UDP flows and two UDP queues. Among the four flows, two flows are small flows each sending at the rate of 300 Mbps while the other two flows are large flows each sending at the rate of 1,200 Mbps. We assume that $B_{q,u}/N_{f,u}$ is 500 Mbps and thus the bandwidth allocated for each queue is 500 Mbps multiplied by the number of flows sharing the queue. Since each queue serves two flows, its allocated bandwidth is 1,000 Mbps. In the "UDP-vs-UDP" situation, the Case 1 table of Figure 3 shows the achieved bandwidth of each flow, where the achieved bandwidth (200 Mbps) of each of these small flows is lower than its fair share (300 Mbps). This is because each of them is served by a queue that also serves a large flow.

Ideally, the UDP flows served by a queue should receive their fair shares from the bandwidth of the queue. If we assign the UDP flows that have similar sending rates to the same queue, the proportions of the bandwidths that these flows obtain from the queue will be close. With this design, NPFS can mitigate the "UDP-vs-UDP" problem. We have designed and implemented an algorithm to achieve this goal. It is

Algorithm 3 The algorithm used to mitigate unfairness problems in the "UDP-vs-UDP" case

Require:
Array of UDP flows, U; Flow's sending rate, $U[i].rate$;
Array of sending rate differences between pairs of adjacent flows, D; Index of the right flow of the pair, $D[i].idx$; Value of the difference, $D[i].diff$;
Array of UDP queues containing mapped UDP flows, Q_U;

```
1: SortAscending(U, key: U.rate)            ▷ Sort, flows by sending rates
2: for $i \leftarrow 0$ to $len(U)-1$ do     ▷ Get sending rate differences of adjacent flows
3:     $D[i].idx \leftarrow i+1$
4:     $D[i].diff \leftarrow U[i+1].rate - U[i].rate$
5:
6: SortDescending(D, key: D.diff)
7:   $D \leftarrow D[: len(Q_U)-1]$          ▷ Get largest $len(Q_U)-1$ sending rate differences
8:
9:   SortAscending(D, key: D.idx)
10: $l \leftarrow 0$
11: for $i \leftarrow 0$ to $len(Q_U)-1$ do   ▷ Assign flows into queues
12:     $Q_U[i] \leftarrow U[l : D[i].idx]$
13:     $l \leftarrow D[i].idx$
```

periodically executed by the control program of NPFS every second. It will assign the UDP flows to UDP queues based on their sending rates to minimize the maximum sending rate difference in a UDP queue. Case 2 of Figure 3 shows how this algorithm assigns the four UDP flows to the two UDP queues. The achieved bandwidth of each flow is shown in the Case 2 table of Figure 3. One can see that the two small flows now can achieve their fair shares and the left $400 = (1000 - 600)$ Mbps bandwidth of queue 1 can be used by queue 2 due to using the deficit weighted round-robin packet scheduler. As a result, each of the two large flows can achieve $700 = (500 + 400/2)$ Mbps bandwidth, which is its fair share in this situation.

Algorithm 3 below describes the designs of the above algorithm. In the pseudocode, Array[i:j] represents a subarray containing Array[i], Array[i+1], ..., Array[j−1]. For $N_{f,u}$ UDP flows and $N_{q,u}$ UDP queues in the system, this algorithm will partition these flows into $N_{q,u}$ groups by finding the top $N_{q,u} - 1$ largest gaps between every two adjacent flows. Algorithm 3 first sorts the $N_{f,u}$ flows by their sending rates in ascending order. Next, it calculates the sending rate difference between every two adjacent flows in the sorted array. Then, it sorts these differences and obtains the top $N_{q,u} - 1$ largest sending rate differences to separate flows into $N_{q,u}$ groups, where each group corresponds to a queue. For the flows assigned to a group, NPFS will direct their packets to the queue corresponding to this group. Directing a packet to a queue can be dynamically and efficiently performed in a P4 switch by specifying the value of the output queue metadata associated with the packet.

Figure 4 shows an example to illustrate this design. In this example, Algorithm 3 partitions four UDP flows into three groups, where each group corresponds to a UDP queue. After calculating the differences and selecting the places of the top two largest differences, Algorithm 3 uses these places to partition the four flows into three groups.

Algorithm 3 is executed by the control program every second. There are three sorting parts in it, each of which has a time complexity of $O(nlogn)$, where n is the number of flows. Therefore, the time complexity of Algorithm 3 is $O(nlogn)$. We have measured and will report its execution time with different numbers of flows in Section VI.

D. IMPLEMENTATION OF DATA PLANE OPERATIONS

NPFS identifies a flow by its FID (flow ID), which is the hash value of the 5-tuple information (source IP address, source port number, destination IP address, destination port number, and the value of the protocol type) in the packet header. NPFS uses the CRC-16 hash algorithm to compute the FID. NPFS uses several match-action tables in the data plane. The first table is *match_queue_table*. The key of an entry in this table is FID and the returned value of the entry is the identity QID of the queue assigned to the flow. When matched by a packet, the action of the entry will write the QID of the entry into the output queue metadata of the packet, thus directing the packet to the assigned queue. The second table is *match_counter_table*. The key of an entry in this table is FID and the returned value of the entry is counterID. Each counterID represents the index to a counter used in the data plane. When matched by a packet, the action of the entry will add the packet size to the corresponding counter. NPFS uses these counters to calculate the current sending rates of flows every second. Figure 5 shows the flowchart of the data plane operations with the following steps:

Step 1 (Parsing): When a packet arrives, the packet is parsed into multiple headers (Figure 5 (1)). The parser extracts the Ethernet header, IP header and the protocol type (TCP or UDP) from the packet step by step. Because the match-action tables in the ingress pipeline need the information of the protocol type and source/destination port numbers, NPFS uses several metadata to store and carry them with the packet. After the values of these metadata are set, the packet will leave the parsing process.

Step 2 (Computing FID): To compute the FID for each packet (Figure 5 (2)), NPFS defines a match-action table whose default action executes a 16-bit hash function to output a value between 0 and 65,535. This hash function uses the 5-tuple information of the packet to compute the FID. This table matches every packet and the default action is applied to each packet.

Step 3 (Determining the QID for a Packet): When a packet arrives, NPFS uses its FID to check whether it is the first packet of a new flow or it is a packet of an existing flow whose QID has been assigned. For the former case, the entry for this new flow has not been inserted into the table and the packet will not match any entry. For the latter case, the entry for this existing flow has been inserted into the table and the packet will match it.

NPFS uses *match_queue_table* to perform the above checks (Fig. 5 (3)). The key of this table is FID. If a packet is matched with an entry in the table, it will be processed by an action that copies the QID stored in the entry to the packet's QID metadata, which specifies the output queue for the packet (Fig. 5 (4)). On the other hand, if a packet matches no entry in the table, then the default action, which processes new flows, will be invoked. Since the QID for the new flow has not been assigned, this default action copies 0 to the packet's QID metadata to indicate this situation (Fig. 5 (6)).

NPFS then uses the metadata obtained in Step 1 to determine whether this new flow is a TCP flow or a UDP flow (Fig. 5 (7)). If it is a TCP flow, then NPFS writes the FID of the new flow to a register (Fig. 5 (9)), which will be read by the control program of NPFS every second to detect the arrival of a new flow and get its FID. On the other hand, if it is a UDP flow, then NPFS writes the FID of the new flow plus 100,000 to the register (Fig. 5 (8)). Since the range of FID is from 0 to 65,535 due to the use of 16 bits and the register has 32 bits, NPFS purposely adds 100,000 to FID for a new UDP flow. Thus, when the control program reads a value from the register and the value is above 65,535, it knows that the new flow is a UDP flow and 100,000 should be deducted from the read value to get the FID of the new flow. On the other hand, if the register value is less than 65,536, the control program knows that the new flow is a TCP flow and the read value is its FID.

Step 4 (Counting the Bytes of Flows): NPFS needs to measure the current sending rates of every flows to dynamically assign them to appropriate queues. This can be performed by continuously counting the number of bytes of every flows and using the number of bytes transmitted in the last second to compute the current sending rates of flows. Since the number of flows passing an output port at any time may be a few hundreds or less, the number of counters should be flexibly set depending on the real-world traffic patterns to save the counter resources in a P4 switch. In our experiments, because the maximum number of flows is a few hundreds, NPFS uses 1,024 counters and each counter

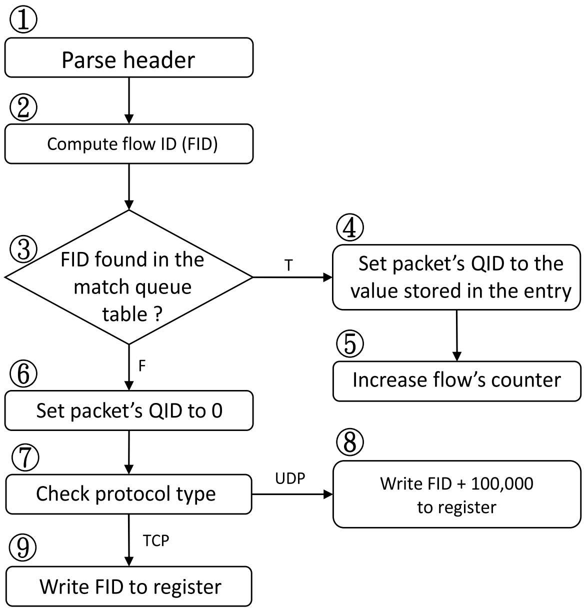

① Parse header
② Compute flow ID (FID)
③ FID found in the match queue table ?
T
④ Set packet's QID to the value stored in the entry
⑤ Increase flow's counter
F
⑥ Set packet's QID to 0
⑦ Check protocol type
UDP
⑧ Write FID + 100,000 to register
TCP
⑨ Write FID to register

FIGURE 5. The flowchart of data plane operations.

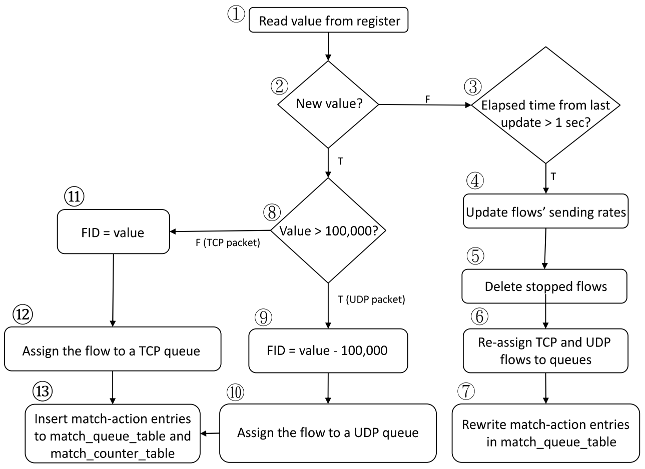

① Read value from register
② New value?
F
③ Elapsed time from last update > 1 sec?
T
④ Update flows' sending rates
⑤ Delete stopped flows
⑥ Re-assign TCP and UDP flows to queues
⑦ Rewrite match-action entries in match_queue_table
T
⑧ Value > 100,000?
F (TCP packet)
⑪ FID = value
T (UDP packet)
⑨ FID = value - 100,000
⑫ Assign the flow to a TCP queue
⑩ Assign the flow to a UDP queue
⑬ Insert match-action entries to match_queue_table and match_counter_table

FIGURE 6. The flowchart of control plane operations.

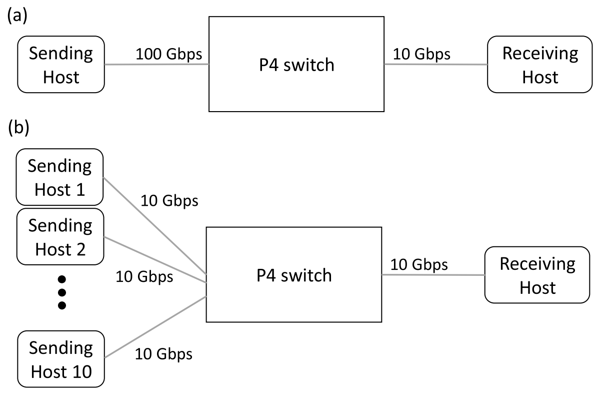


FIGURE 7. The network topology used in the experiments.

is accessed by its counter ID ranging from 0 to 1,023. NPFS uses *match_counter_table* and the FID of a packet as the key of the table to get the ID of the counter used for the flow of the packet. With the counter ID, NPFS increases the indexed counter by the packet length in bytes (Figure 5 (5)). The control program of NPFS will fetch the values of these counters every second to measure the current sending rates of flows.

E. IMPLEMENTATION OF CONTROL PLANE OPERATIONS

Figure 6 shows the workflow of the control plane operations. The control program uses pre-built APIs to read the register value every second (Figure 6 (1)). It maintains a list that stores the IDs of the flows being processed. For each read register value, the control program checks whether it is a new value not in the list (Figure 6 (2)). If so, then it starts the process of adding a new flow. Otherwise, it performs the process of assigning flows to queues. In the process of adding a new flow, the control program first determines the protocol type of the new flow by checking whether the register value is greater than 100,000 (Figure 6 (8)). If the value is greater than 100,000, then the new flow is a UDP flow and its FID is the value minus 100,000 (Figure 6 (9)). Then, the new flow is assigned to a UDP queue (Figure 6 (10)). Otherwise, the new flow is a TCP flow (Figure 6 (11)) and the new flow is assigned to a TCP queue (Figure 6 (12)).

For a new flow, NPFS assigns it to a TCP queue or a UDP queue depending on its protocol type. In the next second the new flow will be moved to an appropriate queue determined by the methods presented in Section V-B and Section V-C. Let QID be the ID of this initial queue selected for the new flow. In addition to finding a queue for the new flow, the control program finds an unused counter (with the identity counterID) for the new flow. Then, the control program inserts an entry with (FID, QID) into *match_queue_table* and inserts an entry with (FID, counterID) into *match_counter_table* (Figure 6 (13)) so that the flow can start using the specified queue and counter.

To assign flows to queues, the control program continuously checks the elapsed time from the last update of the sending rates of flows. When the elapsed time exceeds one second (Figure 6 (3)), the control program performs the following operations to assign flows to queues. The control program first updates the current sending rates of flows by reading their counter values (Figure 6 (4)). It also checks whether some flows have stopped sending data in the last second (Figure 6 (5)). If so, the control program deletes their entries from *match_queue_table* and *match_counter_table*. In addition, the method in Section V-A is used to update the weights of the queues that served these flows to release the bandwidth allocated to them. Also, NPFS uses the mechanisms presented in Section V-B and Section V-C to assign flows to queues (Figure 6 (6)). Then, it rewrites the entries in *match_queue_table* for the flows whose queues need to be changed to start using the new mapping (Figure 6 (7)).

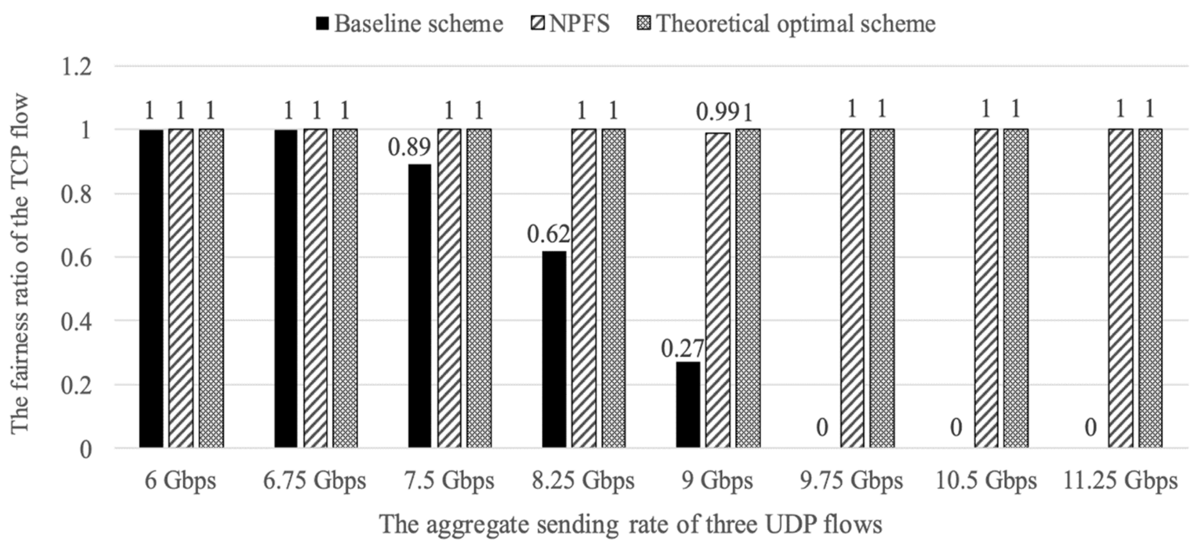


FIGURE 8. The fairness ratio of the TCP flow under different aggregate sending rates of three competing UDP flows.

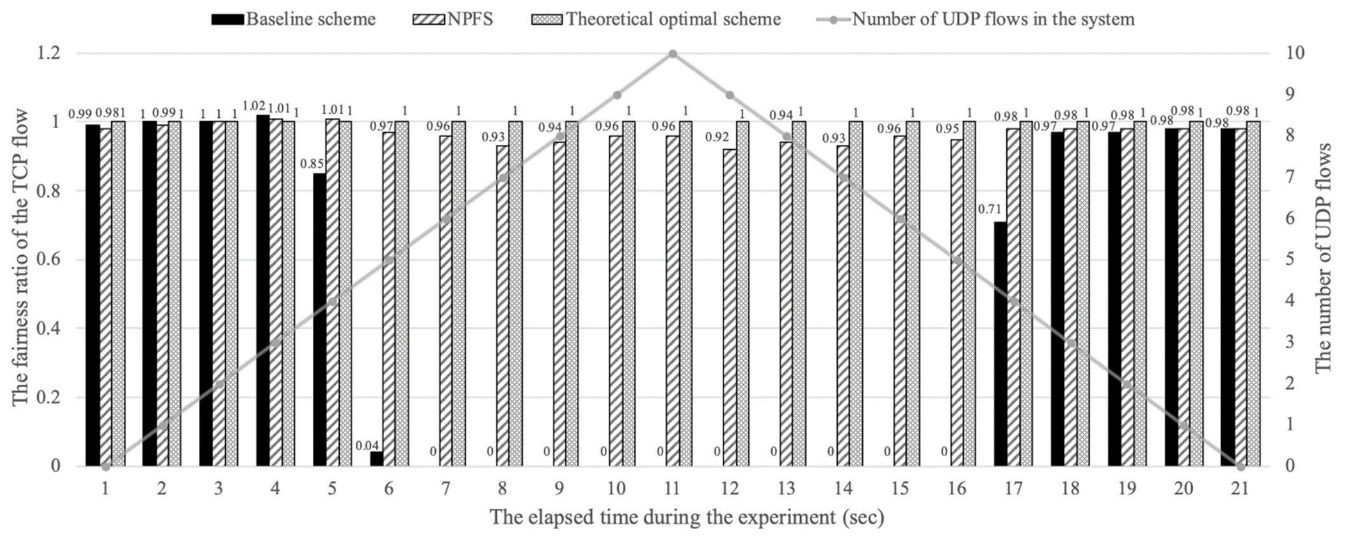


FIGURE 9. The fairness ratio of the TCP flow with different numbers of competing UDP flows.

VI. PERFORMANCE EVALUATION

The experiments include one sending host, one receiving host, and one D5264-P4 switch, and the used topology is shown in Figure 7(a). Each of the hosts has a 32-core Intel 1.8 GHz E5-2675 CPU and a Mellanox MT27800 ConnectX-5 [37] network interface card (NIC). We use iperf version 2.0.9 [38] to generate traffic in the experiments. Because this server has 32 CPU cores, many iperf programs are executed on different CPU cores, and the total sending rate of all iperf programs in any experiment is less than 30 Gbps, the sending host is powerful enough to generate the desired traffic load.

The NICs of hosts and the ports of the P4 switch can support 100 Gbps bandwidth. To create bottleneck conditions, we purposely set the bandwidth of the port of the P4 switch that connects to the receiving host to 10 Gbps in the experiments. This setting can create 10:1 congestion pressure on the bottleneck port. Effectively, the used topology is equivalent to the topology shown in Figure 7(b), where each of ten sending hosts connects to the P4 switch via a 10 Gbps link.

We set the number of queues used in this bottleneck output port to 8, which is the default number of queues provided in most commodity switches for an output port [36]. Note that in this datacenter-grade P4 switch the number of queues provided in an output port can go up to 32 when the port bandwidth is 100 Gbps [36].

In each experiment, the sending host and receiving host run many iperf programs to set up multiple TCP/UDP flows. The iperf program can specify the application-layer sending rate of a flow, either TCP or UDP. After receiving traffic from the sending host, the P4 switch forwards the traffic to the receiving host. After each experiment is finished, the throughput of each flow reported by the iperf program running on the receiving host is recorded. Using the fair share definition presented in Section IV-A, we define the fairness ratio of a flow as its achieved throughput divided by its fair share. A flow is fairly treated if its fairness ratio is 1. In a scenario with multiple competing flows, the ideal outcome of a FQ scheme is that the fairness ratio of each flow is 1. If a flow receives a fairness ratio larger than 1, some flow(s) must receive a fairness ratio smaller than 1, meaning that the bandwidth allocation among them is not fair. We repeat each experiment 10 times. In the following experiments, we compare the performance of NPFS with that of the baseline scheme, in which only one queue is used to serve all packets at an output port, and that of the theoretic optimal scheme, which shows the theoretic optimal results.

Figure 8 shows the fairness ratio of a TCP flow under different aggregate sending rates of three UDP flows in the baseline scheme, NPFS, theoretical optimal scheme, respectively. In this set of experiments, a TCP flow and three UDP flows compete for the 10 Gbps bandwidth of the bottleneck port. Note that the throughput reported by the iperf program is the application-layer throughput. Since the preambles, Ethernet header, IP header, and TCP/UDP headers consume network bandwidth, the maximum achievable throughput at the application layer on a 10 Gbps link is about 9.5 Gbps. Thus, when computing the fairness ratio of a flow, we use 9.5 Gbps rather than 10 Gbps as the bottleneck bandwidth.

The sending duration of each flow in the experiments is set to 20 seconds. Initially, the sending rates of the TCP flow and three UDP flows are set to 2 Gbps. Then, we gradually increase the sending rate of each UDP flow from 2 Gbps to 3.75 Gbps for different experiments. Figure 8 shows the fairness ratio of the TCP flow under different aggregate sending rates of the three UDP flows, which ranges from $6 = 2 \times 3$ Gbps to $11.25 = 3.75 \times 3$ Gbps. The figure shows that NPFS avoids the "TCP-vs-UDP" problem and the TCP flow achieves its fair share in each experiment. In contrast, in the baseline scheme, the TCP flow can only use the bandwidth left by the UDP flows and thus its fairness ratio continuously decreases when the aggregate sending rate of the UDP flows increases. When the aggregate sending rate is larger than 9.75 Gbps (which is higher than the 9.5 Gbps bandwidth available at the application layer), the TCP flow cannot send out any data.

Figure 9 shows the fairness ratio of a TCP flow with different numbers of competing UDP flows in the system. At the beginning of the experiment, the sending host launches a TCP flow whose sending duration is 21 seconds. It then adds a UDP flow into the system at a 1-second interval until the number of UDP flows reaches 10. In this experiment, we do not limit the sending rate of the TCP flow. The sending duration of each UDP flow is 10 seconds and each UDP flow uses 2 Gbps as its sending rate. Because the experiment duration is 21 seconds and the duration of each UDP flow is 10 seconds, the number of UDP flows in the system is increased from 0 to 10 and then decreased down to 0 during the experiment. The fairness ratios of the TCP flow in NPFS are very close to the results of the theoretic optimal scheme, even when the total number of flows varies. In contrast, in the baseline scheme, the fairness ratios of the TCP flow drop to almost 0 when the number of UDP flows is larger than 4.

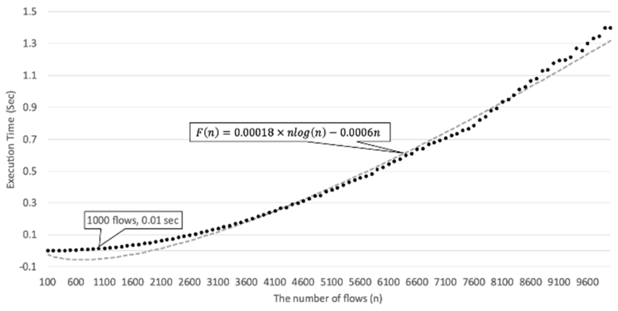


FIGURE 10. The execution time of Algorithm 3 with different numbers of flows.

In the above experiments, we evaluate whether NPFS can protect TCP flows from competing UDP flows. In the following, we evaluate the fairness among competing UDP flows. Because Algorithm 3 is executed every second to (re)assign UDP flows to appropriate queues, its execution time should be less than one second. Figure 10 shows the execution time of Algorithm 3 running on the operating system of the P4 switch with different numbers of flows. For these flows, we assign random sending rates to them. To obtain reliable results, each dot in Figure 10 is the average execution time of 1,000 runs under a given number of flows.

Figure 10 shows that the execution time of Algorithm 3 is only 0.01 seconds when the number of UDP flows is 1,000. This number is already higher than the maximum number of flows that can be effectively supported by NPFS. (We will discuss this issue later.) The execution time can be approximated by $0.00018nlog(n) - 0.0006n$ in Figure 10, which confirms the time complexity analysis of Algorithm 3 performed at the end of Section V-C.

Figure 11 shows the fairness ratios of ten UDP flows that are competing for the bottleneck bandwidth, where the sending rate of each flow ranges from 100 Mbps to 5 Gbps. The duration of each UDP flow is set to 20 seconds. Ideally, the fairness ratio of each flow should be 1. However, the fairness ratios of the 4 Gbps flow and 5 Gbps flow are 1.34 and 1.66 in the baseline scheme, which means that they unfairly take some bandwidth from the fair shares of other flows. For example, the flows whose sending rates are set to less than or equal to 2 Gbps unfairly receive a fairness ratio much less than 1. This is the phenomenon of the "UDP-vs-UDP" problem. In contrast, comparing NPFS with the theoretical optimal scheme, one can see that in NPFS the "UDP-vs-UDP" problem is greatly mitigated, evidenced by the fact that most UDP flows receive a fairness ratio very close to 1.

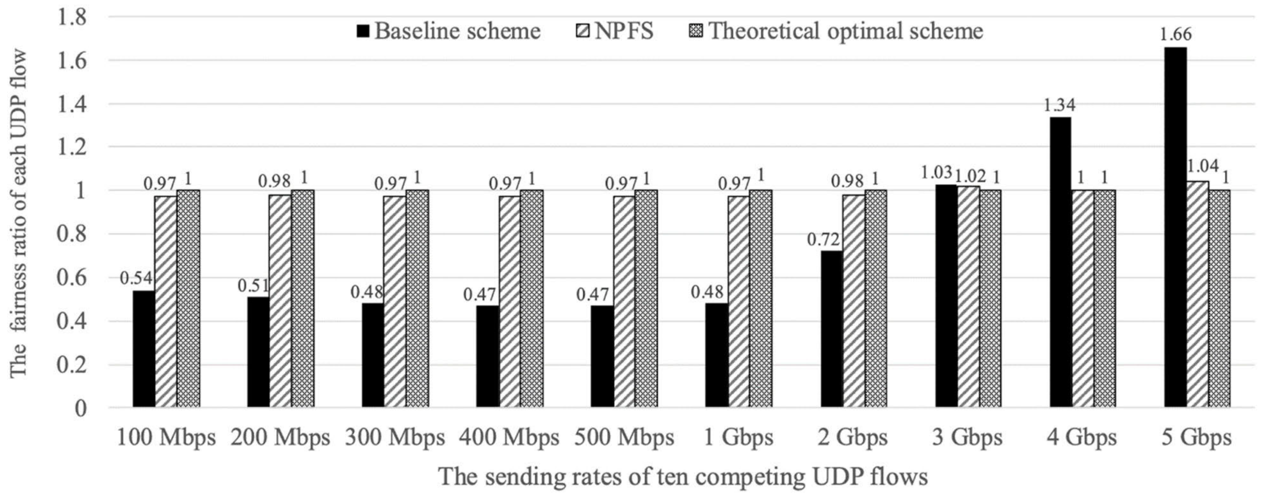


FIGURE 11. The fairness ratio of each UDP flow when ten UDP flows are simultaneously competing.

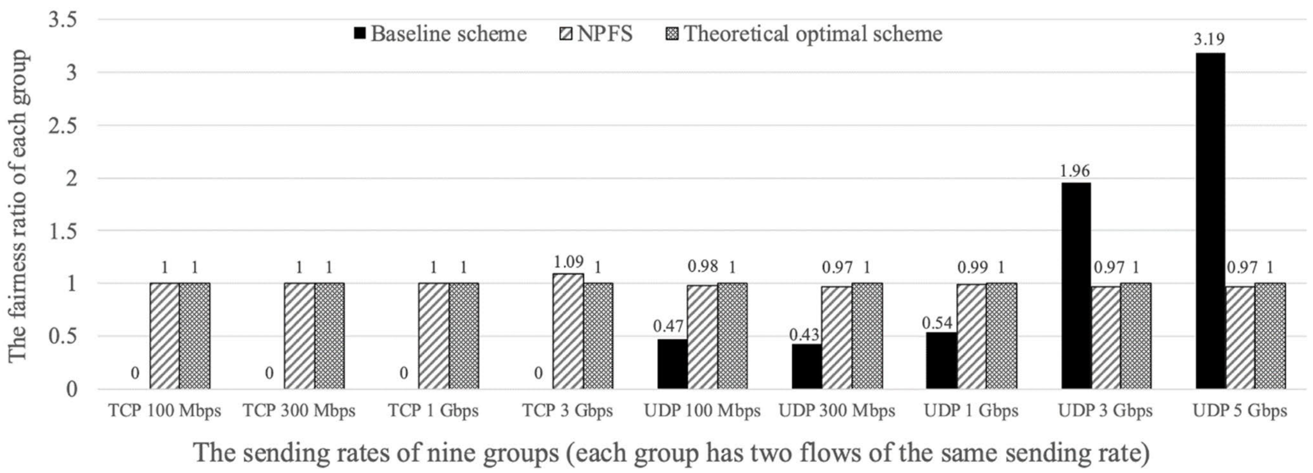


FIGURE 12. The fairness ratio of each group when the flows of the nine groups are simultaneously competing.

Figure 12 shows the fairness ratios of multiple groups of flows that are simultaneously competing for the bottleneck bandwidth. There are nine groups of flows in this experiment and each of them has either two TCP flows or two UDP flows with the same sending rate. Different TCP groups use different sending rates and different UDP groups use different sending rates. Since each group has two flows, the fairness ratio of a group is defined as the average of the fairness ratios of its two flows. In this experiment, the sending duration of each flow is set to 20 seconds.

From this figure, one can see that in the baseline scheme, none of the flows in these TCP groups can send out any data, and thus the fairness ratios of these TCP groups are 0. Besides, the "UDP-vs-UDP" problem occurs among the UDP groups. In contrast, in NPFS the fairness ratios of both the TCP groups and UDP groups are all close to 1, which shows that the performance of NPFS can approach the results of the theoretic optimal scheme. These results show that in NPFS, when multiple TCP flows and UDP flows are simultaneously competing for the bottleneck bandwidth, they can still achieve their fair shares. These results also show that the "TCP-vs-TCP with multiple queues" problem presented in Section V-B is solved in NPFS. This is because if this problem is not solved in NPFS, the "TCP 100 Mbps" flows or "TCP 300 Mbps" flows may be assigned to the same queue with "TCP 1Gbps" flows or "TCP 3Gbps" flows. If this is the case, the "TCP 1Gbps" flows or "TCP 3Gbps" flows will get a fairness ratio much larger than 1. However, as shown in the figure, their fairness ratios are very close to 1.

Figure 13 shows the performance of NPFS using different numbers of queues in an output port. In this experiment, we design three different traffic sets with 30 flows, 60 flows, and 90 flows, respectively. In each traffic set, the number of TCP flows is equal to the UDP flows. The sending rates of TCP flows are taken from the numbers in an arithmetic sequence, and the sending rates of UDP flows are also taken

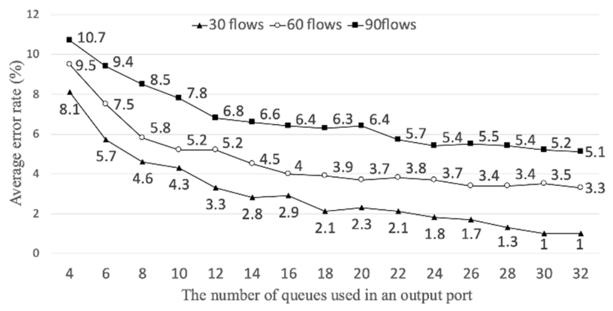


FIGURE 13. **The average absolute error rate of all competing flows with different numbers of queues in an output port.**

from these numbers. Note that an arithmetic sequence is an ordered sequence and the difference between two neighboring numbers is the same. If a is its first number, d is the difference between two neighboring numbers, and there are N numbers in the sequence, then the n-th number in the sequence is $a + (n - 1) \times d$, where n = 1, 2, . . . , N. The sum S of all of the numbers in the sequence is $n \times a + (n - 1) \times n \times d \div 2$. For all of the three traffic sets, the starting number a of their arithmetic sequences is set to 100 Mbps.

To be more specific, the 30-flow traffic set has 15 TCP flows and 15 UDP flows, and the sending rates of these TCP flows form an arithmetic sequence starting from 100 Mbps and ending at 1500 Mbps, with an increment of 100 Mbps. That is, the sending rates of these TCP flows are 100, 200, 300, 400, . . . , 1, 400, and 1,500 Mbps, respectively. The sending rate settings for the 15 UDP flows are the same as those for the TCP flows. With these settings, the sum of the sending rates of all flows in the 30-flow traffic set is 24 Gbps.

To maintain the same load on the bottleneck bandwidth, for the 60-flow and 90-flow traffic sets, we also set the sum of the sending rates of their flows to 24 Gbps. Because the values of a and N for the 60-flow and 90-flow traffic sets are known, the d values used for them are set to 48.28 and 19.70, respectively.

In addition to the above adjustment, to reflect that the total volume of traffic is increased but the bottleneck bandwidth remains the same, we also enlarge the sending duration of each flow in the 60-flow traffic set and 90-flow traffic set to 2 and 3 times of the sending duration of the 30-flow traffic set, respectively.

For each traffic set, after the experiment is finished, we compute the fairness ratio of each flow in it and then subtract 1 from the fairness ratio to get its error rate. Since the error rate can be positive or negative, we take its absolute value to reflect how much the fairness ratio deviates from the ideal value, which is 1. Then, we average such values of the flows in a traffic set to get the average absolute error rate of all flows in a traffic set.

The results in Figure 13 show that NPFS can achieve a lower average error rate when the number of queues used in an output port increases. When the number of queues increases from 4 to 12, the error rate decreases significantly. Over 12 queues, the error rate decreases at a lower speed with more queues. In addition, when 32 queues are used in an output port, even though the number of flows grows to 90, the error rate can still be kept at only 5%. These results show that NPFS can be a solution for fair queueing in commodity switches when the number of flows is less than three times the number of queues in an output port.

VII. DISCUSSIONS

Experiments show that NPFS can avoid "TCP-vs-UDP" problem, mitigate the "UDP-vs-UDP" problem, and allow competing TCP flows to achieve their fair shares based on the AIMD property of the TCP congestion control algorithm. However, these results are achieved with TCP flows using the same congestion control algorithm where the UDP flows do not implement congestion control. In a network where these requirements are not met, the performance of NPFS may degrade. These issues are discussed below.

Currently, TCP CUBIC is the default TCP congestion control algorithm used for all Linux hosts. (Note that this TCP congestion control algorithm is also used in our experiments.) However, other congestion control algorithms such as "new reno" and reno can be enabled to replace CUBIC. Since TCP flows using these different TCP congestion control algorithms may not share the bottleneck bandwidth fairly when they compete, NPFS should be deployed in a network where most TCP hosts use the same TCP congestion control algorithm.

Another issue is the assumption made by NPFS that the UDP protocol does not respond to congestion like TCP and thus NFPS separates UDP flows from TCP flows. As a transport-layer protocol implemented in the operating system (OS), indeed the UDP protocol does not implement any congestion control inside the OS. However, some network applications that use the UDP protocol to transmit their data may implement their own application-layer congestion control to respond to congestion. For example, the QUIC [40] protocol is an application-layer library that is based on the UDP protocol but provides TCP-like congestion control.

For QUIC UDP flows, if NPFS puts them into a queue based on their initial bandwidth usages and this queue is shared with other UDP flows that do not respond to congestion, they may not compete fairly with these UDP flows. However, since QUIC uses UDP port number 443 as its port number, in the future version of NPFS, we can treat a UDP flow using 443 as its port number as a TCP flow and process it as a TCP flow in NPFS. We will consider this issue in our future work.

Currently, the control-plane program of NPFS measures the bandwidth usage of every flow, uses Algorithm 3 to re-assign UDP flows to appropriate queues, and adjusts the weight of every queue at the 1-second interval. Since NPFS is aimed to providing fair sharing to elephant flows, most of which are long-lived, currently we use 1 second as the monitoring and re-acting interval. This interval can be shortened to a smaller value such as 0.1 seconds to react to flow changes more quickly. The reaction speed can be improved at the cost of increasing CPU usage of the switch. However, as Figure 10 shows, when the number of flows regulated by NPFS is less than 1000, the execution time of Algorithm 3 (which performs the updates) is less than 0.01 second. This means that even when a very short interval such as 0.1 second is used, the CPU of the switch can still finish the updates in time in each monitoring and updating interval.

Due to its simplicity, the baseline with only one queue is the most widely used scheme in commodity switches and thus we compare NPFS with it in Section VI "Performance Evaluation". Several schemes that use multiple queues such as the priority scheduling scheme and the traffic class-based round-robin scheduling scheme exist. However, their purposes are different from ours. For example, the priority scheduling scheme is used for serving traffic of different priorities while the traffic class-based round-robin scheduling scheme is used for providing fair sharing at the per-traffic-class level rather than at the per-flow level. Thus, we did not compare our scheme with them.

The FID (flow ID) currently used by NPFS is computed from the CRC-16 hash function. To reduce the hash collision probability, the future version of NPFS can use the CRC-32 or even CRC-64 function. If collisions still occur, we can use several common hash collision resolution methods such as liner probing to solve it.

The target application of NPFS is to regulate the bandwidth usages of elephant flows so that small flows and elephant flows can both get their fair shares. The traffic of elephant flows in our intended applications can be commodity traffic or other types of traffic such as high-definition video streaming.

Priority-based scheduling does not conflict with fair bandwidth allocation. Traffic flows of different priorities can be served first by the priority-based scheduling, followed by traffic flows of the same priorities served using a round-robin or weighted round-robin approach.

VIII. CONCLUSION AND FUTURE WORK

Network quality of service (QoS) is essential to network applications. Enabling a flow to get its fair share of available bandwidth can prevent its data transfer from being blocked by those flows that do not respond to congestion.

In this paper, we have designed and implemented a near per-flow queueing scheme named NPFS in P4 hardware switches and evaluated its performance under many different conditions. NPFS outperforms the single-queue-based baseline scheme in maintaining fair shares among competing flows. Specifically, the "TCP-vs-UDP" problem can be avoided and the "UDP-vs-UDP" problem can be greatly mitigated in our scheme. Experimental results show that, for the fair share of a flow, NPFS can achieve an error rate of 5% or less when the number of flows is less than three times the number of queues in an output port.

In most networks, elephant (i.e., large) flows account for a very small portion of the total flows of a network but carry most of the traffic. For example, the authors in [39] found that on the WIDE network, elephant flows were only 4.7% of all flows but occupied 41.3% of all data transmitted. NPFS is a near-optimal per-flow queueing scheme operating on a limited number of queues. In a practical deployment of P4 commodity switches with high-bandwidth ports, NFPS is useful for the networks where around one hundred elephant flows or more are simultaneously competing for the bandwidth of an output port.

In the future, we will perform the mathematical analysis of NPFS to shed more insight of the performance results.

ACKNOWLEDGMENT

The authors thank Barefoot/Intel for their supports of this work.

REFERENCES

[1] J. Kurose and K. Ross, *Computer Networking: A Top-Down Approach*, 7th ed. London, U.K.: Pearson, 2016.

[2] F. Gebali, "Scheduling algorithms," in *Analysis of Computer and Communication Networks*. Boston, MA, USA: Springer, 2008, doi: 10.1007/978-0-387-74437-7_12.

[3] D. A. Devi and S. Jaga, "Analysis of scheduled routing algorithms on 5-port router for network on chip application," *Int. J. Sci. Technol. Res.*, vol. 8, no. 9, pp. 2148–2153, Sep. 2019.

[4] R. Pan, B. Prabhakar, and K. Psounis, "CHOKe—A stateless active queue management scheme for approximating fair bandwidth allocation," in *Proc. IEEE Conf. Comput. Commun., 19th Annu. Joint Conf. IEEE Comput. Commun. Soc. (INFOCOM)*, vol. 2, Mar. 2000, pp. 942–951.

[5] W.-c. Feng, K. G. Shin, D. D. Kandlur, and D. Saha, "The BLUE active queue management algorithms," *IEEE/ACM Trans. Netw.*, vol. 10, no. 4, pp. 513–528, Aug. 2002.

[6] W. Feng, D. Kandlur, D. Saha, and K. Shin, "Stochastic fair blue: A queue management algorithm for enforcing fairness," in *Proc. IEEE Conf. Comput. Commun., 20th Annu. Joint Conf. IEEE Comput. Commun. Soc. (INFOCOM)*, vol. 3, Apr. 2001, pp. 1520–1529.

[7] D. Lin and R. Morris, "Dynamics of random early detection," in *Proc. ACM SIGCOMM Conf. Appl., Technol., Archit., Protocols Comput. Commun. (SIGCOMM)*. New York, NY, USA: Association for Computing Machinery, Oct. 1997, pp. 127–137, doi: 10.1145/263105.263154.

[8] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 19, no. 4, pp. 1–12, Aug. 1989, doi: 10.1145/75247.75248.

[9] P. Goyal, H. M. Vin, and H. Cheng, "Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks," *IEEE/ACM Trans. Netw.*, vol. 5, no. 5, pp. 690–704, Oct. 1997.

[10] A. K. Parekh and R. G. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: The single-node case," *IEEE/ACM Trans. Netw.*, vol. 1, no. 3, pp. 344–357, Jun. 1993.

[11] P. McKenney, "Stochastic fairness queueing," in *Proc. 9th Annu. Joint Conf. IEEE Comput. Commun. Soc. The Multiple Facets Integr. (INFOCOM)*, vol. 2, Jan. 1990, pp. 733–740.

[12] J. C. R. Bennett and H. Zhang, "WF^{2}Q: Worst-case fair weighted fair queueing," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, vol. 1, Mar. 1996, pp. 120–128.

[13] S. J. Golestani, "A self-clocked fair queueing scheme for broadband applications," in *Proc. Conf. Comput. Commun. (INFOCOM)*, vol. 2, 1994, pp. 636–646.

[14] 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, doi: 10.1145/2656877.2656890.

[15] *P4 Website*. Accessed: Jun. 5, 2021. [Online]. Available: https://p4.org

[16] E. F. Kfoury, J. Crichigno, and E. Bou-Harb, "An exhaustive survey on P4 programmable data plane switches: Taxonomy, applications, challenges, and future trends," *IEEE Access*, vol. 9, pp. 87094–87155, 2021.

[17] N. K. Sharma, M. Liu, K. Atreya, and A. Krishnamurthy, "Approximating fair queueing on reconfigurable switches," in *Proc. 15th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*. Renton, WA, USA: USENIX Association, Apr. 2018, pp. 1–16. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/sharma

[18] C. Cascone, N. Bonelli, L. Bianchi, A. Capone, and B. Sansò, "Towards approximate fair bandwidth sharing via dynamic priority queuing," in *Proc. IEEE Int. Symp. Local Metrop. Area Netw. (LANMAN)*, Jun. 2017, pp. 1–6.

[19] *Inventec D5264 Datacenter Programmable 100GbE Switch*. Accessed: Jul. 17, 2021. [Online]. Available: http://productline.inventec.com/switch/Download/D5264.pdf

[20] J. Nagle, "On packet switches with infinite storage," *IEEE Trans. Commun.*, vol. COM-35, no. 4, pp. 435–438, Apr. 1987.

[21] M. Shreedhar and G. Varghese, "Efficient fair queuing using deficit round-robin," *IEEE/ACM Trans. Netw.*, vol. 4, no. 3, pp. 375–385, Jun. 1996.

[22] S. S. Kanhere, H. Sethu, and A. B. Parekh, "Fair and efficient packet scheduling using elastic round Robin," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, pp. 324–336, Mar. 2002.

[23] I. Stoica, S. Shenker, and H. Zhang, "Core-stateless fair queueing: A scalable architecture to approximate fair bandwidth allocations in high-speed networks," *IEEE/ACM Trans. Netw.*, vol. 11, no. 1, pp. 33–46, Feb. 2003.

[24] Z. Cao, Z. Wang, and E. Zegura, "Rainbow fair queueing: Fair bandwidth sharing without per-flow state," in *Proc. IEEE Conf. Comput. Commun., 19th Annu. Joint Conf. IEEE Comput. Commun. Soc. (INFOCOM)*, vol. 2, Mar. 2000, pp. 922–931.

[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 *Proc. 17th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*. Santa Clara, CA, USA: USENIX Association, Feb. 2020, pp. 685–699. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/sharma

[26] 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 *Proc. ACM SIGCOMM Conf.* New York, NY, USA: Association for Computing Machinery, Aug. 2016, pp. 44–57, doi: 10.1145/2934872.2934899.

[27] V. Shrivastav, "Fast, scalable, and programmable packet scheduler in hardware," in *Proc. ACM Special Interest Group Data Commun. (SIGCOMM)*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 367–379, doi: 10.1145/3341302.3342090.

[28] A. G. Alcoz, A. Dietmuller, and L. Vanbever, "SP-PIFO: Approximating push-in first-out behaviors using strict-priority queues," in *Proc. 17th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*. Santa Clara, CA, USA: USENIX Association, Feb. 2020, pp. 59–76. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/alcoz

[29] Z. Yu, J. Wu, V. Braverman, I. Stoica, and X. Jin, "Twenty years after: Hierarchical core-stateless fair queueing," in *Proc. 18th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*. USENIX Association, Apr. 2021, pp. 1–18. [Online]. Available: https://www.usenix.org/conference/nsdi21/presentation/yu

[30] M. Kang, G. Yang, Y. Yoo, and C. Yoo, "TensorExpress: In-network communication scheduling for distributed deep learning," in *Proc. IEEE 13th Int. Conf. Cloud Comput. (CLOUD)*, Beijing, China, Oct. 2020, pp. 25–27.

[31] C. Zhang, Z. Chen, H. Song, R. Yao, Y. Xu, Y. Wang, J. Miao, and B. Liu, "PIPO: Efficient programmable scheduling for time sensitive networking," in *Proc. IEEE 29th Int. Conf. Netw. Protocols (ICNP)*, Dallas, TX, USA, Nov. 2021, pp. 1–11.

[32] H. Huang, P. Li, and S. Guo, "Traffic scheduling for deep packet inspection in software-defined networks," *Concurrency Comput., Pract. Exper.*, vol. 29, no. 16, p. e3967, 2017.

[33] W. Bai, K. Chen, L. Chen, C. Kim, and H. Wu, "Enabling ECN over generic packet scheduling," in *Proc. 12th Int. Conf. Emerg. Netw. Exp. Technol.*, Dec. 2016, pp. 191–204.

[34] *P4 16 Language Specification Version 1.2.0*. Accessed: Jul. 17, 2021. [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.2.0.html

[35] *Intel Tofino ASIC Chip*. Accessed: Jul. 17, 2021. [Online]. Available: https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html

[36] "10k-series device family overview," Early Access Product Specifications, Barefoot Netw. Confidential Proprietary, Oct. 2018.

[37] *Mellanox MT27800 ConnectX-5*. Accessed: Jul. 17, 2021. [Online]. Available: https://www.mellanox.com/files/doc-2020/pb-connectx-5-en-card.pdf

[38] *Iperf Version 2.0.9*. Accessed: Jul. 17, 2021. [Online]. Available: https://iperf.fr/iperf-doc.php#doc

[39] T. Mori, R. Kawahara, S. Naito, and S. Goto, "On the characteristics of Internet traffic variability: Spikes and elephants," in *Proc. Int. Symp. Appl. Internet.*, 2004, pp. 99–106.

[40] J. Iyengar and M. Thomson, *QUIC: A UDP-Based Multiplexed and Secure Transport*, document RFC 9000, IETF, 2021.

SHIE-YUAN WANG (Senior Member, IEEE) received the master's and Ph.D. degrees in computer science from Harvard University, in 1997 and 1999, respectively. He is a Full Professor with the Department of Computer Science, National Yang Ming Chiao Tung University (NYCU), Taiwan. He has published many high-quality journals and conference papers in the fields of computer networks, such as IEEE/ACM Transactions on Networking and *Journal of Network and Computer Applications* (Elsevier). His current research interests include the Internet of Things and P4 programmable networks. He received the Outstanding Information Technology Elite Award of Taiwan Government, in 2012; the President Award of Tokyo University of Science, for his contributions to computer network researches, in 2014; and the 19th Y. Z. Hsu Scientific Paper Award from Y. Z. Hsu Foundation, Taiwan, in 2021. Since 2021, he has been selected as a world's top 2% scientist for his career impact. He has served as the General Chair, the Technical Program Co-Chair, and a member for many prestigious IEEE conferences, such as ICC, GLOBECOM, NOMS, PIMRC, VTC, and ISCC. He is serving as an Associate Editor for *ACM Computing Surveys*.

CHEN-YO SUN received the bachelor's degree in computer science from the National Yang Ming Chiao Tung University (NYCU), Taiwan, in 2021. He is currently pursuing the master's degree with Carnegie Mellon University (CMU).

YU-CHEN HSIAO received the bachelor's degree in computer science from the National Yang Ming Chiao Tung University (NYCU), Taiwan, in 2021, where he is currently pursuing the master's degree with the Institute of Computer Science and Engineering.

YI-BING LIN (Fellow, IEEE) received the bachelor's degree from the National Cheng Kung University, Taiwan, in 1983, and the Ph.D. degree from the University of Washington, USA, in 1990. He is a Winbond Chair Professor with the National Yang Ming Chiao Tung University (NYCU). From 1990 to 1995, he was a Research Scientist with Bellcore (Telcordia). Then, he joined NCTU, Taiwan, where he became a lifetime Chair Professor, in 2010, and the Vice President, in 2011. From 2014 to 2016, he was the Deputy Minister of the Ministry of Science and Technology, Taiwan. Since 2016, he has been the Vice Chancellor of the University System of Taiwan (for NCTU, NTHU, NCU, and NYM). He is an AAAS Fellow, an ACM Fellow, and an IET Fellow.