Time-Aware Packet Forwarding in Programmable Data Planes

Yuqun Song
School of Electrical Engineering and Computer Science
Washington State University, Pullman, WA, USA
Email: yuqun.song@wsu.edu

Monowar Hasan
School of Electrical Engineering and Computer Science
Washington State University, Pullman, WA, USA
Email: monowar.hasan@wsu.edu

***Abstract*—Networks in many safety-critical systems like avionics, automotive, and industrial plants have strict end-to-end delay requirements to be met for correct system operation. Existing software-defined real-time networks do not support *data plane programmability* provided by recent protocol-independent switch architectures such as P4. Our research enables time-aware flow forwarding in P4-enabled software-defined time-critical networks. In this paper, we introduce time-aware flow scheduling for P4-enabled SDN architectures. We study two scheduling policies: the first one prioritizes flows based on *slack* (i.e., how much time is left to reach the destination), and the second one uses *finish time* as a priority metric, which is determined from its data rate requirements. Both approaches were implemented and tested in the P4 software stack. We find that the slack-based forwarding scheme performs better in retaining real-time requirements. Our publicly released scheduler implementations will assist network engineers in adapting programmable switches to safety-critical applications that demand precise timing guarantees.**

I. Introduction

Real-time networks are integral to various safety-critical cyber-physical applications such as avionics, automobiles, power grids, energy delivery systems, and industrial machine-to-machine communications, to name a few. Such networks have strict Quality of Service (QoS) requirements, mainly in terms of end-to-end delay [1], [2]. The flow delay must not exceed its predefined bounds at any point in system operation. Any violation of temporal constraints can cause serious consequences. For example, modern manufacturing plants are equipped with tens to hundreds of "things" (e.g., sensors and actuators) that communicate with each other with different timing requirements [3]. If such requirements are temporarily broken, failures may occur in controlling the machines, which can disrupt the entire supply chain.

Traditionally, safety-critical real-time systems maintain separate networks (both hardware and software) for different types of traffic, typically for safety and security reasons. However, this approach leads to significant overheads regarding equipment/weight, management, and updates and can introduce potential system errors and faults. Existing implementations, such as Avionics Full-Duplex Switched Ethernet (AFDX) [4] and Controller Area Network (CAN) [5], widely used in these domains, are often proprietary, complex, expensive, and may require custom hardware. To address this problem, researchers propose to leverage the capabilities of Software-Defined Networks (SDNs) [6] for better management and control of real-time networks [7]–[9]. In SDNs, the *control plane* (responsible for routing) and the *data plane* (responsible for packet forwarding) are separated, which allows the designers to control networks more flexibly and efficiently. While there exist efforts for forming time-aware protocols such IEEE 802.1 (TSN) [10], they are often complementary to SDNs (i.e., not directly comparable). Our focus in this research is on the SDN technology, in particular, programmable data planes. The benefit of using SDN for real-time networks is configuring and optimizing network resources flexibly across protocol layers through a logically centralized SDN controller. The centralized, global view of the network in an SDN-enabled architecture helps ensure end-to-end guarantees required for real-time applications.

In traditional SDN-enabled networks (both real-time and non-real-time), we can define the behavior of the control plane but not the data plane. Specifically, we can define routing policies in SDN controllers but not corresponding actions in SDN switches. Programming Protocol-independent Packet Processors (P4) is a new programming language for network devices, such as switches, that overcomes this challenge [11]. P4 enhances the programmability of SDN by enabling network operators to specify the behavior of the data plane precisely (for instance, how the data plane processes packets). P4 provides flexibility in implementing various network applications and designs, such as bandwidth management, duplicate address detection, traffic offloading, denial of service mitigation, routing mechanisms, and traffic metering [12].

While real-time networks have recently adapted for SDNs, such networks are *not* yet configured with P4-enabled SDN architecture. Existing P4 implementations do not reason about end-to-end delay experienced by individual flows. To fully leverage the benefits of P4 in SDN-enabled real-time systems, it is essential to account for delay constraints. The proposed research is the *first to introduce time-aware flow scheduling for programmable data planes* (i.e., P4 switches). This is not trivial, as the current packet forwarding rules supported

This research is partly supported by the U.S. National Science Foundation Award 2345653. Any findings, opinions, recommendations, or conclusions expressed in the paper are those of the authors and do not necessarily reflect the sponsor's views.

by P4 switches do not provide any notion of temporal constraints. Enabling real-time requirements for forwarding flows in P4 requires switch-level modifications and new scheduling policies.

This work explores design alternatives to enable real-time scheduling in programmable data planes. For this, we study two flow scheduling policies for P4 switches. The first one, Slack Monotonic (SM) scheduling, measures the laxity or *slack*, i.e., how much time is left before a flow's timing bounds and forwards the flow with the smallest slack first at each switch. The latter is called Shortest Finish Time First (SFTF), which uses a metric called *finish time* (a function of flow arrival time and its data rate requirement). A flow with the shortest finish time will be forwarded at each switch first. The key difference between these two policies is that the slack-based one considers end-to-end timing requirements while forwarding packets. In contrast, the second scheme intends to be proportionally fair (in terms of data rate requirements) and locally optimize switch share for each flow.

In this work, we made the following contributions.

- We introduce two time-aware packet forwarding policies for P4 switches that track the timing requirements and prioritize flows based on their urgency (§III).
- We implemented the proposed scheduling policies in the P4 software stack (§IV). We release our scheduler implementation for community use.[1]

We tested the performance of our scheduler implementation using an emulated network topology (§V). Our experiments show that forwarding packets considering end-to-end timing requirements allows the flows to reach their destinations in a shorter time than scheduling them at each switch following a proportional-fair manner considering their data rate requirements.

II. Background and Model

We start with a background on real-time networks (§II-A) and programmable switches (§II-B), followed by the network model considered in this paper (§II-C).

A. Real-Time Networks

Safety-critical real-time networks typically have a well-defined network structure (e.g., topologies, hosts, and links) and flow specifications. They are generally under the control of a centralized authority. For example, Airbus has complete control over the A380 topology and its associated subsystems [9]. The closed nature of such networks makes it amenable for enforcement of system-wide policies that take into account safety requirements, such as worst-case end-to-end latency. The key requirement of real-time networks is that the *packets must be delivered between hosts with guaranteed upper bounds on delay*. The timing requirement of such a network is determined by a parameter termed *deadline*. If the packets of all flows in the network reach the destinations before their deadline, the flows meet the real-time requirement, and hence, the cyber-physical network is *safe*. The key challenge in modeling/analyzing real-time networks is to understand the delay(s) caused by individual switches and compose them along the delays caused by the presence of other flows in that switch, as well as the network.

[1]Available on GitHub: https://github.com/CPS2RL/rtp4/.

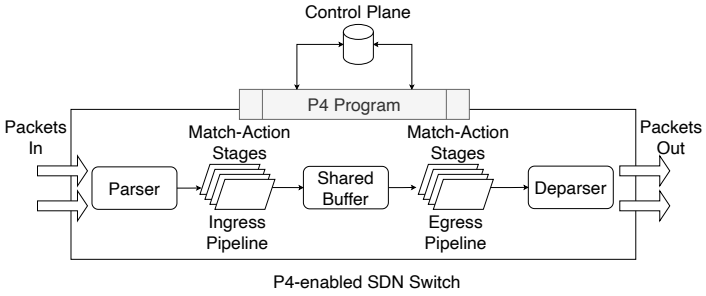


Fig. 1. High-level illustration of the packet forwarding process in P4 switches.

B. Programmable Switch and P4

In traditional SDN, the data plane is relatively fixed, i.e., the SDN protocol such as OpenFlow [13] predefines the supported headers. P4 [11], in contrast, introduces an abstraction model for programming the network data plane. Unlike OpenFlow, P4-enabled devices are protocol-independent, meaning they do not assume inherent support for any particular protocols. Instead, P4 programs define packet headers and specify how packets should be parsed and processed. A P4 program consists of several key elements: *(a)* header definitions, which specify field names and widths for protocol headers; *(b)* metadata, providing packet-specific state; *(c)* registers, meters, and counters for the state that is independent of packets; *(d)* a packet parser specification; *(e)* match-action tables, which define packet and metadata fields to be read and specify actions to be executed; *(f)* actions, which are parameterized functions that invoke one or more primitives; and *(g)* control flow, which dictates the sequence of table execution and supports conditional branching.

P4 operates using *match-action pipelines* for packet forwarding in SDN switches, which is performed through *table lookups* (match) and corresponding *actions*. As shown in Fig. 1, when a packet arrives, the parser first processes its header while assuming the packet data is buffered. The *parser* extracts header fields based on a programmed parse graph and passes them to *ingress match-action tables*. These tables contain lookup *keys* (e.g., IP and MAC addresses) and corresponding *actions* (e.g., forward or drop). The packet header is processed based on the lookup results, and an egress port and queue are selected. The packet is then passed to the *egress match-action tables*, where additional processing can occur before finally being forwarded to the output port.

We note that while P4 enables flexibility in switch programming, it was *not* designed for hard real-time applications that require precise end-to-end timing guarantees. The key contribution of our work is to enable real-time capabilities in P4 switches.

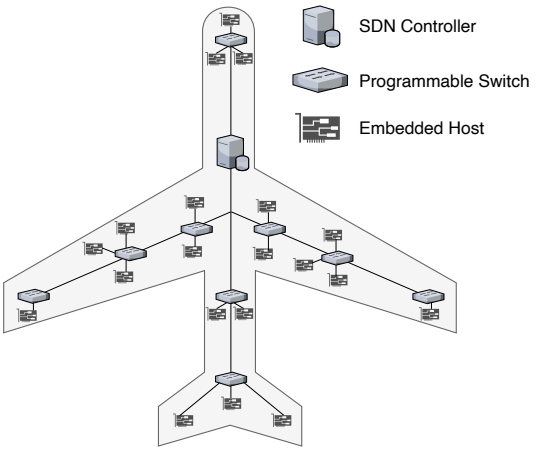


Fig. 2. Schematic of a software-defined real-time cyber-physical network.

C. Network Model

We consider a software-defined real-time network [8] represented by a graph $N(\mathcal{V}, \mathcal{E})$ managed by an SDN controller C, where $\mathcal{V}$ denotes the set of network switches and $\mathcal{E}$ denotes the links between switches, as shown in Fig. 2. Each switch $\pi_s \in \mathcal{V}$ is programmable (i.e., P4-enabled) by assumption. The host systems (generally embedded devices) are attached to the switches. The real-time applications running on the hosts generate traffic flows. We assume a set of $\mathcal{F}$ flows present in the network. Each flow $F_i \in \mathcal{F}$ traversed from a source host h^i_{src} to destination host h^i_{dst} through a set of switches by using a source routing algorithm [8]. The length of the packets for the flow F_i is L_i, and the bandwidth requirement of F_i is denoted by B_i. While routed through the switches, a flow will experience two kinds of delay: *(a)* processing delay from ingress to egress ports, including delay due to the presence of other flows in a given switch, and *(b)* transmission and propagation delay for packet communications over the medium. When the real-time application generates a flow, it must reach the destination by its deadline D_i. The bandwidth and deadline constraints typically originate from real-time applications and are often tightly coupled with underlying cyber-physical systems.

Note: In this work, the focus is on self-contained, localized, real-time networks typically connected over wired LANs. In such networks, the number of flows and their specifications are known apriori and typically do not change frequently. The schemes proposed and evaluated in this work are not meant for open wide-area networks such as the Internet.

III. Time-Aware Flow Scheduling

Recall that programmable switches do not support real-time flows, as there is no accountability for temporal constraints in current P4 implementations. We want to conduct *design-space explorations* and devise time-aware time-aware schedulers that consider timing constraints and schedule flows based on timing budgets to enable deadline guarantees in programmable switches. For instance, until now, it is unknown that ***is it better to consider flow urgency*** (i.e., time left to reach the destination) or ***schedule them in a proportional-share manner based on their completion time*** (i.e., the difference between arrival at ingress and departure at the egress)?

To answer this question, we study two schedulers. Our first algorithm calculates the *slack* (i.e., the available time to reach the destination before the deadline) and schedules the flows based on slack values. We refer to this scheme as *Slack Monotonic (SM)* policy.[2] For each switch $\pi_s \in \mathcal{V}$, SM policy prioritizes flows with shorter deadlines or traversed longer in the path (i.e., takes more time to reach π_s from source). We also explore a "proportional share"-like policy based on *finish time*. Specifically, we calculate each flow's tentative (viz., virtual) finish time and prioritize the flow with the shortest finish time. We refer to this latter scheme as the *Shortest Finish Time First (SFTF)* policy.

We now present both scheduling techniques in detail (§III-A and §III-B) and then discuss how to incorporate them in P4 switches (§IV).

A. Slack Monotonic (SM) Scheduling

We define the *slack* at a given decision instance as the time remaining for a flow to reach the destination. Let T^s_i denote the time a flow F_i takes to reach an intermediate switch π_s from its source. Hence, at τ_s, the slack of flow F_i is calculated by subtracting its deadline D_i from the traversed time. Specifically, we calculate the slack as follows:

$$slack^s_i = D_i - \big(T^s_i + \frac{L_i}{B_i}\big). \tag{1}$$

For each switch π_s, the scheduler will select the flow with the lowest slack, i.e., if both F_i and F_j are present at the ready queue of π_s, F_i will be scheduled first if $slack^s_i < slack^s_j$. For equal slack, we break ties arbitrarily. As we shall see in the paper (§IV), switch programmability allows us to keep track of the slacks in packet headers, which can be used to make scheduling decisions.

B. Shortest Finish Time First (SFTF) Scheduling

The slack-based scheduling scheme introduced above favors the flow of those with shorter deadlines or those that consume more time earlier in their path. We now introduce an alternate scheme that aims to be "fair" at each switch and retains real-time guarantees. In this case, the scheduling decision is based on a *virtual finish time*, which approximates the completion time of a flow F_i considering the behavior of other flows routed through the same set of switches in the path of F_i. Let a packet of a flow F_i arrive at the ingress port of the switch π_s at time A^s_i. The arrival time depends on the average transmission rate and departure time of the previous packet of the flow. We calculate the departure time (at egress) for F_i as follows:

$$finish^s_i = A^s_i + \frac{L_i}{B_i}. \tag{2}$$

[2]This policy is also known as Least Laxity First (LLF) in literature.

In this case, the switch scheduler will process the flow with the shortest finish time. For instance, for a pair of flows F_i and F_j, the switch will schedule F_i first if $finish_i^s < finish_j^s$. As before, for equal finish times, the scheduler will pick one arbitrarily.

Note that, in this policy, the amount of switch share each flow gets depends on their packet length and bandwidth requirements. However, SFTF does not consider end-to-end deadline constraints while making forwarding decisions.

IV. Scheduler Implementation in P4

We now present our implementation details of the flow scheduling algorithms introduced in §III. In P4-enabled switches, we can program the data plane according to specific requirements. In programmable switches, *flow tables* store information about flows. Following prior work [8], we assume there exists one flow per priority level. While flow multiplexing [9] is doable, we left this as future work. In our implementation, we create a new flow table called `priority_table`, which matches based on the flow identifier and executes corresponding *actions* (in this case, setting flow priorities). P4 supports up to 8 priority levels per port. We use P4's `set_priority()` API as a flow action to set the priority. For instance, the flow will be scheduled with the highest priority if we assign the priority value to 7 using `set_priority()`.

As mentioned earlier, the current P4 implementation does not support real-time flow scheduling, as there is no deadline or time-specific data structure in P4 switches. However, data plane programmability allows us to modify packet headers based on application requirements. Hence, we modify the packet headers for UDP traffic.[3] Specifically, we include `deadline`, `slack`, and `finish_time` fields in the UDP packet header to carry on-demand timing information as the flow is routed through its path. For a flow packet `p`, we use `p.extract()` function to get the UDP header from the Ethernet/IP packet and obtain its timing information to make scheduling decisions.

In our current implementation, we use *flow tables* and *registers* to perform flow scheduling. The registers in P4 are array-like data structures that store information. P4 provides P4Runtime [14] and ThriftAPI [15] to modify flow tables and control the behavior of the data plane. We define a set of registers to store the ingress and egress timestamp, deadline, slack, and packet length extracted from the packet. The registers are accessible through the P4 `get_register()` API. We use `bm_register_write()` and `bm_register_read()` APIs to perform write/read operations on the registers. We partition the registers and isolate the data of different flows by writing them down in different positions in the registers.

We select the candidate flow for scheduling based on the calculated slack values (for SM) or finish time (for SFTF). For this, we implemented a new API in the controller named `table_add()` that interacts with the switch with scheduling information. The function takes the *flow table*, corresponding *actions*, and *match keys* as inputs. We use P4 `get_res()` API to retrieve the table resources associated with the flow table and then use `get_action()` function to find the corresponding action (e.g., setting candidate flow's priority). Next, the `parse_match_key()` API is called to ensure the target action is pushed for the desired flow (i.e., high-priority flow to be scheduled). Once the rules are set, the switch will process and dispatch the flows in a high to low-priority order.

[3]As most real-time communications use UDP [8], [9]. However, our implementation can be extended for other packet types without loss of generality.

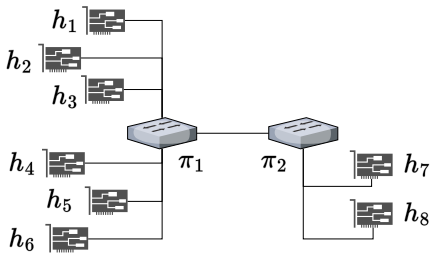


Fig. 3. Mininet topology used for evaluation.

Our P4 scheduler implementation is publicly available: https://github.com/CPS2RL/rtp4/.

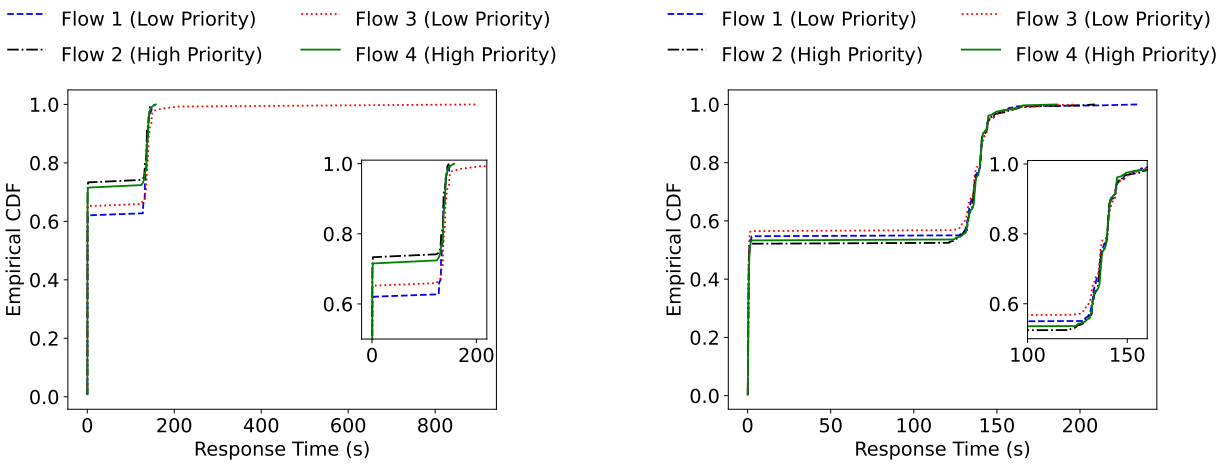
V. Evaluation

A. Setup

We evaluate the efficacy of our scheme using an emulated real-time network created by Mininet [16]. We built a topology with 8 hosts connected to 2 switches in a line as shown in Fig 3. The switches in our network are P4 BMV2 (Behavioral Model Version 2) types [17], where we configured P4 programs (SM and SFTF scheduling logics) to process packets. The link bandwidth was set to 10 Mbps. **Note:** Our focus in this work is to test the flow forwarding policies at the switch level. Hence, simulating larger networks will not provide additional insights. We resort to a simple line network to demonstrate the performance of time-aware schedulers in the P4 ecosystem. We further stress that the setup is created as a proof-of-concept, and our scheduler can be scaled to support large topology and any number of flows.

B. Results

Recall from our earlier discussion (§II) that flows in real-time networks are known ahead of time, and unlike traditional systems, they do not change behavior over time. As indicated above, our evaluation focuses on analyzing whether the real-time schedulers work correctly in the switches. Hence, to demonstrate this behavior, we use fewer flows (2-7 of our evaluations) and thin links (e.g., 10 Mbps). Using a large number of flows or larger capacity links will not change the insights we observed in the experiments. We further assume the flows used here are "control" flows, which are mostly used for signaling and do not require high bandwidth.



(a) Slack Monotonic Scheduling

(b) Shortest Finish Time First Scheduling

Fig. 4. Empirical CDF of flow response times for *(a)* SM (left) and *(b)* SFTF (right) schedulers processing a set of high and low priority flows. The low-priority flows result in longer response times (longer tails in CDF distribution), which suggests that the priority scheduling is correctly applied on the P4 program running on the switches.

In our first set of experiments, we want to test whether our priority scheduling is correctly implemented to the P4 switches. For this, we created flows with two priority classes (High and Low), with two flows for each priority class. Let us denote F_1 and F_3 as Low priority (large slack/finish time values for SM and SFTF, respectively) and flow F_2 and F_4 as High priority (smaller slack/finish time). The flows were dispatched in a round-robin, time-division manner, and only a pair of High and Low priority classes of flows are present at a given scheduling epoch. For instance, at a given scheduling instance t, the set of flows present in the switches were either $\{F_1, F_2\}$ or $\{F_3, F_4\}$. The flows were sent from h_1 and h_2 to h_7. As the focus of this experiment was to test whether high-priority flows are prioritized at egress, we sent the flows in tandem to prevent the effect of burst traffic from other flows.

We want to observe the network behavior in a *congested* scenario. So, we stressed the network with 12 Mbps traffic (in 10 Mbps links) from these two priority classes. The Low priority flows generated smaller traffic with a larger interarrival time (0.01 s). The High priority ones arrived at a faster rate. We determined the interarrival time of the High priority class as a function of packet size and target bandwidth as follows: $\frac{\text{packet size}}{\text{target bandwidth}}$. As a typical UPD packet length is 1472 bytes, we set the interarrival time for the High class to be 0.2352 ms.

In Fig. 4, we show the Empirical Cumulative Distribution Function (ECDF) of the flow response times for SM (Fig. 4(a)) and SFTF (Fig. 4(b)) schedulers. The response time is defined by the time between flow origination at the source host and reception by the destination host, which includes the total processing time at the switches and propagation time over the communication medium. We extracted the response times from switch egress timestamps. The x-axis of Fig. 4 shows the response times, and the y-axis shows the distribution (ECDF).

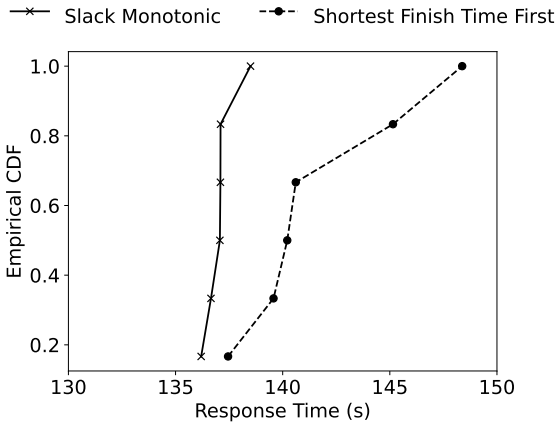


Fig. 5. Resposne times for SM and SFTF schedulers for a varying number of flows (2-7). SFTF schedules the flows based on their bandwidth requirements without considering end-to-end timing constraints, which results in a longer delay than the SM scheme.

We calculate ECDF as follows:

$$G_\alpha(\jmath) = \frac{1}{\alpha} \sum_{i=1}^{\alpha} \mathbb{I}_{[\zeta_i \leq \jmath]}, \tag{3}$$

where α is the total number of experimental observations, ζ_i response times for i-th experimental observation, and $\jmath$ represents the x-axis values (i.e., response times) in Fig. 4. The indicator function $\mathbb{I}_{[\cdot]}$ outputs 1 if the condition $[\cdot]$ is satisfied and 0 otherwise. As Fig. 4 shows, the flows with Low priority class (F_1 and F_3) have a longer response time than the High priority class (F_2 and F_4) in our trials (i.e., the tails in the ECDF distributions are longer). Note that, by definition, if the response times of the flows are less than the deadline, the flows will meet their real-time requirements. These experiments confirm that the priority

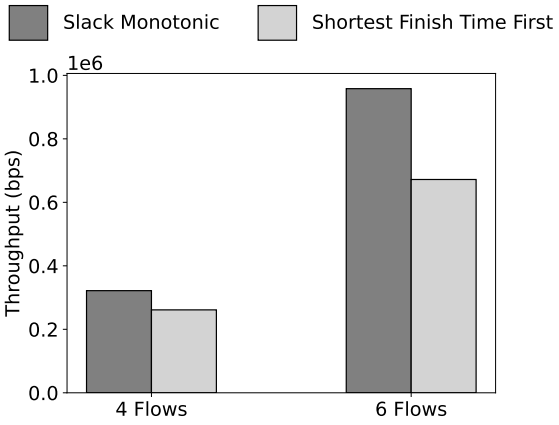


Fig. 6. Average network throughput for SM and SFTF schedulers for 4 and 6 flows. SFTF requires more time to route packets from source to destination, which degrades overall throughput.

scheduling implementation is correctly configured in the P4 switches.

In the next set of experiments (Fig. 5), we compare the performance of SM and SFTF schedulers. For this, we varied the number of flows from 2 to 7 and measured response times for both schedulers. We measured the 90^{th} percentile response time value of the most affected (lowest priority) flows. As the figure shows, the SM scheme results in shorter response times than the SFTF policy. This is because SFTF does not account for end-to-end constraints and locally optimizes switch share among the flows based on their bandwidth requirements. As a result, the end-to-end performance suffers. By definition, when response times exceed deadlines, the flows will not retain real-time constraints. Hence, the SM scheduler can ensure timing constraints more likely than the SFTF schemes for a fixed deadline.

In our final experiments (see Fig. 6), we compare SM and SFTF schedulers regarding network throughput. We used iPerf [18] to obtain the network throughput. As illustrated in the figure, SM outperforms SFTF in throughput. As shown in earlier experiments (Fig. 5), SFTF requires more time to process packets of a flow. This reduces the amount of flow that can be served per unit of time. As a result, overall network throughput degrades.

VI. Related work

Several papers consider QoS constraints in traditional and SDN-enabled networks (see the survey [19]). However, they do not consider real-time aspects. Besides, data plane programmability is not supported on those SDN-based architectures. Recent studies that aim to adapt SDNs for real-time applications [7]–[9], [20]–[22] and provide fault tolerance with timing guarantees [23] do not consider programmable switch architectures.

SP-PIFO [24] introduces static scheduling using P4 but does not consider deadline constraints. There has been research on programmable switches for various contexts, such as introspection [25], [26], failure recovery [12], [27], [28], rerouting [29], virtualization [30], queue management [31], and prototyping [32]. However, those techniques are for general-purpose networks and do not consider real-time requirements. To the best of our knowledge, the proposed work is one of the first efforts that enables time-aware packet forwarding for P4 switches.

VII. Conclusion

This paper introduces techniques to enable real-time flow scheduling in P4-enabled SDN switches. We implemented and tested proposed schemes on the P4 software stack. Our experiments show that scheduling policies considering end-to-end requirements (SM) retain better real-time guarantees than the bandwidth-based proportional fair scheme (SFTF). This research is one of the early investigations that enables real-time scheduling for P4-based switches. Our open-source scheduler implementation will assist real-time network engineers in performing design-space exploration of flow configurations in their target systems.

References

[1] "IEEE standard communication delivery time performance requirements for electric power substation automation," *IEEE Std 1646-2004*, pp. 1–36, 2005.

[2] S.-N. Yeung and J. Lehoczky, "End-to-end delay analysis for real-time networks," in *Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001)*, pp. 299–309, IEEE, 2001.

[3] B. Chen, J. Wan, L. Shu, P. Li, M. Mukherjee, and B. Yin, "Smart factory of industry 4.0: Key technologies, application case, and challenges," *Ieee Access*, vol. 6, pp. 6505–6519, 2017.

[4] Z. Ayhan, E. G. Schmidt, and K. W. Schmidt, "Computation of tight bounds for the worst-case end-to-end delay on Avionics Full-Duplex Switched Ethernet," *Journal of Systems Architecture*, p. 103278, 2024.

[5] R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien, "Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised," *Real-Time Systems*, vol. 35, pp. 239–272, 2007.

[6] K. Kirkpatrick, "Software-defined networking," *Communications of the ACM*, vol. 56, no. 9, pp. 16–19, 2013.

[7] S. Oh, I. Shin, and K. Lee, "RT-SDN: adaptive routing and priority ordering for software-defined real-time networking," *IEEE Systems Journal*, vol. 16, no. 2, pp. 2379–2390, 2021.

[8] R. Kumar, M. Hasan, S. Padhy, K. Evchenko, L. Piramanayagam, S. Mohan, and R. B. Bobba, "End-to-end network delay guarantees for real-time systems using SDN," in *2017 IEEE Real-Time Systems Symposium (RTSS)*, pp. 231–242, IEEE, 2017.

[9] A. Kashinath, M. Hasan, R. Kumar, S. Mohan, R. B. Bobba, and S. Padhy, "Safety critical networks using commodity SDNs," in *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*, pp. 1–10, IEEE, 2021.

[10] J. Farkas, L. L. Bello, and C. Gunther, "Time-sensitive networking standards," *IEEE Communications Standards Magazine*, vol. 2, no. 2, pp. 20–21, 2018.

[11] 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.

[12] H. Miura, K. Hirata, and T. Tachibana, "P4-based design of fast failure recovery for software-defined networks," *Computer Networks*, vol. 216, p. 109274, 2022.

[13] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM computer communication review*, vol. 38, no. 2, pp. 69–74, 2008.

[14] T. P. A. W. Group, "P4Runtime specification." https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html. [Accessed 12-10-2024].

[15] "P4-Utils 1.0 documentation." https://nsg-ethz.github.io/p4-utils/index.html. [Accessed 12-10-2024].

[16] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, pp. 1–6, 2010.
[17] "The reference P4 software switch." https://github.com/p4lang/behavioral-model. [Accessed 13-10-2024].
[18] "iPerf - The TCP, UDP and SCTP network bandwidth measurement tool." https://iperf.fr. [Accessed 13-10-2024].
[19] J. W. Guck, A. Van Bemten, M. Reisslein, and W. Kellerer, "Unicast QoS routing algorithms for SDN: A comprehensive survey and performance evaluation," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 1, pp. 388–415, 2017.
[20] A. Kashinath, M. Hasan, S. Mohan, R. B. Bobba, and R. Mittal, "Improving dependability via deadline guarantees in commodity real-time networks," in *2020 IEEE Globecom Workshops (GC Wkshps*, pp. 1–6, IEEE, 2020.
[21] J. W. Guck, A. Van Bemten, and W. Kellerer, "DetServ: Network models for real-time qos provisioning in sdn-based industrial environments," *IEEE Transactions on Network and Service Management*, vol. 14, no. 4, pp. 1003–1017, 2017.
[22] K. Lee, M. Kim, T. Park, H. S. Chwa, J. Lee, S. Shin, and I. Shin, "MC-SDN: Supporting mixed-criticality real-time communication using software-defined networking," *IEEE Internet of Things Journal*, vol. 6, no. 4, pp. 6325–6344, 2019.
[23] K. Lee, M. Kim, H. Kim, H. S. Chwa, J. Lee, and I. Shin, "Fault-resilient real-time communication using software-defined networking," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 204–215, IEEE, 2019.
[24] 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)*, pp. 59–76, 2020.
[25] S. Aalibagi, M. Dolati, S. Sadrhaghighi, and M. Ghaderi, "Low-overhead packet loss diagnosis for virtual private clouds using P4-programmable NICs," in *NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium*, pp. 1–9, IEEE, 2023.
[26] P. Wintermeyer, M. Apostolaki, A. Dietmüller, and L. Vanbever, "P^2GO: P4 profile-guided optimizations," in *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, pp. 146–152, 2020.
[27] Z. Li, Y. Hu, J. Wu, and J. Lu, "P4Resilience: Scalable resilience for multi-failure recovery in SDN with programmable data plane," *Computer Networks*, vol. 208, p. 108896, 2022.
[28] J. Xu, S. Xie, and J. Zhao, "P4Neighbor: Efficient link failure recovery with programmable switches," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 388–401, 2021.
[29] A. Mazloum, E. Kfoury, J. Gomez, and J. Crichigno, "A survey on rerouting techniques with p4 programmable data plane switches," *Computer Networks*, vol. 230, p. 109795, 2023.
[30] D. Hancock and J. Van der Merwe, "HyPer4: Using P4 to virtualize the programmable data plane," in *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pp. 35–49, 2016.
[31] Z. Li, Y. Hu, L. Tian, and Z. Lv, "Packet rank-aware active queue management for programmable flow scheduling," *Computer Networks*, vol. 225, p. 109632, 2023.
[32] F. Geyer and M. Winkel, "Towards embedded packet processing devices for rapid prototyping of avionic applications," in *9th European Congress on Embedded Real Time Software and Systems (ERTS 2018)*, 2018.