High-Throughput Stateless-but-Complex Packet Processing within a Tbps Programmable Switch

Yutaro Yoshinaka*, Yuki Koizumi*, Junji Takemasa* and Toru Hasegawa†
*Graduate School of Information Science and Technology, Osaka University
†Faculty of Materials for Energy, Shimane University

***Abstract*—Programmable switches are promising platforms for fast and flexible in-network computation; however, a standard mechanism, packet recirculation, degrades throughput due to bandwidth consumption caused by the loopback of not only packet headers but also cumbersome payloads. This paper proposes P^{4}QRS, a mechanism for retaining payloads within the switch, reducing payload recirculations. Specifically, P^{4}QRS bifurcates packets into headers and payloads, which undergo the computation process through pipelines and the buffering process leveraging the switch's queue behavior, respectively; they then rendezvous for reassembly into complete packets to be sent out. To validate its effectiveness, we evaluated P^{4}QRS using an analytical model and implementation on state-of-the-art hardware programmable switches. Our evaluation shows that P^{4}QRS operates stably and intrinsically boosts complex in-switch computations.**

***Index Terms*—Programmable switches, In-network computing**

I. Introduction

Programmable switches with pipeline-based architecture [1] mark a significant milestone in the networking field by offering a high degree of flexibility in packet forwarding operations.

Beyond the traditional role of packet forwarding, programmable switches are increasingly being utilized for network and data processing functions, emulating the role traditionally played by middleboxes. Recirculation-based computation, which circulates packets through the switch multiple times until the necessary computation is completed, is a vital approach to tasks of complexity that exceed the capacity executable within a packet's single passage through the switch's pipelines. Recirculation-based computation enables various applications, including advanced routing and forwarding [2], [3], virtualizing and chaining of network services [4]–[10], and security and privacy [11], [12]. Also, recirculation enables the in-switch cryptography [13]–[15], machine learning [16], [17], and other unique tasks [18]–[20], potentially integrated into network functions and benefitting new applications.

Despite its potential, recirculation-based computation has a significant drawback: it consumes a substantial amount of port bandwidth for recirculation, introducing a tradeoff between performance and complexity of the computation. Many studies emphasize this drawback and prioritize avoiding packet recirculation at the cost of richer designs [21]–[23]. For instance, Fu et al. [23], [24] incorporate a lightweight but weak cipher to realize their security function without recirculations. In contrast, others, such as network function chaining [7]–[9], advocate the use of recirculation for seeking the variety and complexity of computations, albeit at the performance cost. Overcoming this drawback could pave the way for implementing richer functionalities without degrading performance.

The recent innovative ideas [25], [26] address the issue of port bandwidth wastage. They propose splitting a packet at the switch into distinct parts: one integral to computation and the other extraneous to the computation, which still consumes a significant portion of the port bandwidth. For notation simplicity, the former and latter segments of a packet are referred to as a *header* and *payload*, respectively. The header is then forwarded to an external device for computation, whereas the payload is kept waiting in the switch's registers [25] or external RDMA servers [26]. Once the computation is completed, the processed header retrieves the corresponding payload, and they are reassembled. These approaches reduce traffic injected into the computing device, thereby alleviating the load on its ports and improving the utilization of computational power.

Although they focus on performing the computation on external devices, in contrast to our aim of computations within the switch itself for further acceleration, the ideas are applicable to resolve the drawback of in-switch recirculation-based computation. However, adapting these approaches to in-switch header computation, rather than external devices, does not resolve the challenges inherent in storing payloads in the switch's registers or RDMA servers. Specifically, the former approach restricts the payload size that registers can accommodate, limiting its effectiveness. Meanwhile, the latter depends on external RDMA servers, which must scale with input rates to handle bandwidth-consuming payloads.

In this study, we introduce *P^{4}QRS*, an acceleration scheme for in-switch computation, addressing these challenges. In P^{4}QRS, while headers undergo recirculation-based computation, payloads are buffered in the switch's queues, sometimes being evicted and enqueued again by recirculation. The essence is to reduce payload recirculations performed until the computation on headers is completed, with the aid of the queue scheduling mechanism. Precisely, payloads are pushed into a low-weighted queue, whereas headers are pushed into a high-weighted queue. Thus, the switch picks up headers more frequently than payloads from these queues, thereby recirculating payloads slowly while processing headers rapidly.

The paper's contributions are summarized as follows: P^{4}QRS enables buffering large payloads within the switch and significantly accelerates in-switch stateless processing on

979-8-3503-5171-2/24/$31.00 ©2024 IEEE

packet headers, which has important applications [27], particularly focusing on complex computations. As the payload buffering in P^{4}QRS relies on the dynamics of queues, which seems sensitive and peaky at first glance, it raises concerns about the system's stability, packet loss, and latency. To address this concern, we construct an analytical model, on which the feasibility of the proposed scheme is confirmed by rigorous analysis. Moreover, we bridge the gap between theory and practice by implementing P^{4}QRS on the state-of-the-art programmable switches. With the implementation, we demonstrate that our approach significantly improves the throughput of recirculation-based complex computation. The evaluation results indicate that P^{4}QRS can accelerate computation 2+ times on the Tofino switch and achieves throughput of several Tbps on the Tofino 2 switch, with tolerable latency overhead.

The rest of this paper is organized as follows: Section II contrasts our proposal with competing approaches. Section III formulates the problem addressed. Section IV details the design of P^{4}QRS, while Section V describes its implementation. Section VI presents an analytical model of P^{4}QRS, validating its feasibility. We then evaluate the implementation on actual switches in Section VII. The discussion is presented in Section VIII, and the paper concludes with Section IX.

II. Related Work

This section compares the existing and proposed computation techniques that utilize programmable switches. The table in Fig. 1 assesses them based on resource demand, speed, acceleration of short and long packets, and use cases.

In-switch computation (Fig. 1a) has an advantage in its overwhelming speed for stateless and simple tasks, reaching up to tens of Tbps, and holds promise as an approach for in-network computing. However, due to intrinsic architectural limitations, it necessitates recirculation for more complex processing, and the naïve approach recirculates both headers and payloads. Consequently, the recirculated payloads, which are irrelevant to the computation, waste the capacity of switch ports allocated for recirculation, thereby reducing throughput.

Such an issue, where computations are bottlenecked by payload transfer, is not unique to in-switch computation; it also affects stateful computation performed on network function (NF) servers by placing pressure on the bandwidth of their network interfaces. In response, some studies propose insightful strategies to reduce this wasteful payload transfer.

PayloadPark [25] (Fig. 1b) saves port bandwidth by storing payloads within the switch. The switch splits an incoming packet into a header and payload, and the payload is stored in a memory space accessible from the switch's pipeline, known as registers (1). The header is forwarded to an NF server for computation. Once the computation is completed, the header is returned to the switch (2). Finally, the switch retrieves the corresponding payload from its registers, merges the header and payload into a complete packet, and sends it out (3). However, programmable switches typically have limitations on the amount of data that can be stored in the register during a single pipeline pass. This restricts PayloadPark to storing a limited fragment of a payload, specifically 160 bytes, in a single pass, and additional recirculations are needed for storing more fragments. This limitation restricts performance gain for long packets of around 1500 bytes, which dominate the majority of the bandwidth in Internet and data center networks with bimodal packet size distributions [28], [29], to a marginal level.

Ribosome [26] (Fig. 1c) addresses this issue. Its packet processing flow is similar to that of PayloadPark, but a novelty involves the storage of the entirety of payloads in RDMA servers rather than the switch's registers. However, despite its efficacy and applicability, it adds resource demands by relying on external RDMA servers proportional to the input bandwidth. It is of concern that four external RDMA servers, each connected to the switch through a 100-Gbps link, are necessary to handle 300-Gbps input traffic in their testbed [26].

Our proposal, P^{4}QRS (Fig. 1d), is distinguished from these predecessors by three points: First, it aims to accelerate stateless yet complex computations to speeds of up to several Tbps by in-switch computation. Second, it buffers the entirety of large payloads, which are particularly bandwidth-consuming and resource-demanding. Third, it avoids sending these large payloads outside the switch, thereby preventing the imposition of these burdens on external entities. In essence, computations are executed and accelerated within a single switch.

III. Problem Formulation

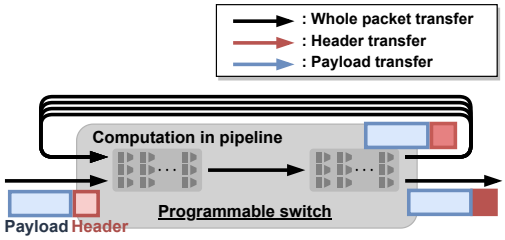
This section introduces the assumed switch architecture and in-switch computation and discusses the scope of P^{4}QRS.

A. Programmable Switches

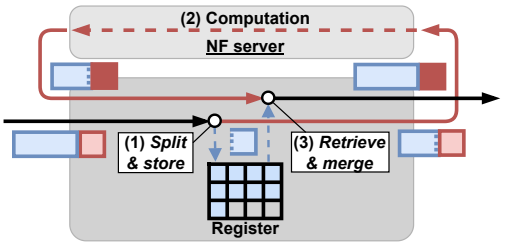
We assume a switch based on the *Reconfigurable Match Tables* (*RMT*) [1], [30] as a platform. It comprises the *ingress* and *egress pipelines* with the *traffic manager* between the pipelines. The ingress and egress pipelines consist of cascading stages, each is equipped with *match-action units* (*MAU*). An MAU has SRAM/TCAM blocks and ALUs, thereby enabling single-step exact/ternary table matching operations followed by single-step arithmetic and logical operations. Furthermore, MAUs can access a variety of externs offering advanced functionality, such as a *register*, which allows per-packet storage, loading, and manipulation of state variables. The traffic manager has a set of queues for each egress port for tasks such as QoS control. Each queue set can be configured to operate under one of the embedded scheduling policies, including *Deficit Weighted Round Robin* (*DWRR*) [31].

B. In-switch Computation and Recirculation

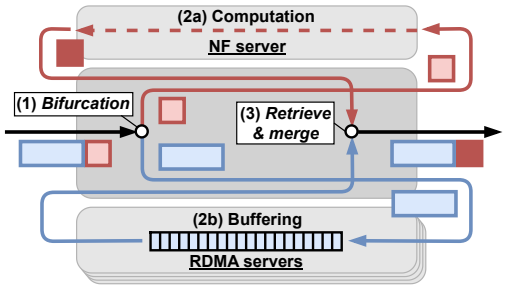
The computational effort exerted by one pass, which means packets' passage through ingress and egress pipelines once, is bounded by the number of stages in these pipelines. Hence, performing complex computations that exceed this limitation necessitates *packet recirculation*. This approach redirects the packets transmitted from the egress port back to the ingress port for multiple iterative processing. However, recirculation spoils computational performance by wasting the port bandwidth. If the port bandwidth is the bottleneck, n-time recirculations degrade computation throughput by a factor of $1/(n+1)$.



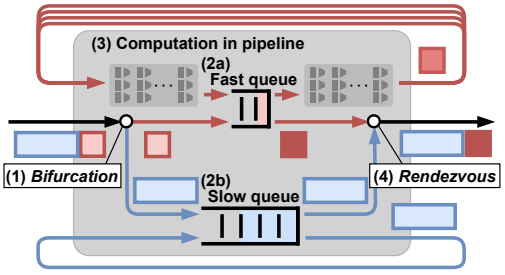
(a) Naïve in-switch computation



(b) PayloadPark



(c) Ribosome



(d) P^{4}QRS

	Split & merge	Header computation	Payload buffering	Resource demand	Order of speed	Accelerate short packets	Accelerate long packets	Class of computation	Use case
Naïve in-switch	—	Switch	—	Low	10s of Tbps	No	No	Stateless, simple	Forwarding, telemetry, aggregation
PayloadPark	Switch	NF server	Switch registers	Medium	100s of Gbps	Yes	Marginal	Stateful, complex	Load balancer, NAT, firewall
Ribosome	Switch	NF server	RDMA servers	High	100s of Gbps	Yes	Yes	Stateful, complex	Load balancer, NAT, scheduler, advanced telemetry
P^{4}QRS	**Switch**	**Switch**	**Switch queues**	**Low**	**Several Tbps**	**No**	**Yes**	**Stateless, complex**	**NF chain [7]–[9], security policies [11], cryptography [12], [14], [15]**

Figure 1: Comparison of existing and proposed approaches

By presuming that only a small portion of each packet is involved in computation, we can avoid this wastage by selectively recirculating only the relevant portion. This assumption is supported by Chen et al. [27], which shows that a substantial portion of common NFs are stateless and localized to packet headers. Indeed, fundamental functionalities such as routing and forwarding [2], [32] typically operate without accessing payloads. This principle also applies to security applications like policy enforcing [11] and privacy protocol [12]. Moreover, the complexity of header computations increases with the virtualizing and chaining of NFs [4], [7]–[9].

Based on this, we define the scope of computational acceleration provided by P^{4}QRS as network functions with *stateless-but-complex computation on headers*: The targets of computational tasks are localized to only a small portion of a packet, while the remaining parts are subject to simple or no computations. Hereafter, we refer to the former part of packets as the *header* and the latter as the *payload*. Additionally, to ensure that the benefits surpass the overhead of P^{4}QRS while naturally meeting the hardware requirements of the switches, we presume that the computation applied to headers is stateless and complex (i.e., requiring more than two recirculations).

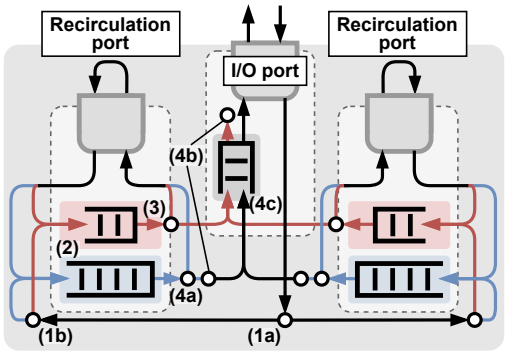


Figure 2: Rendezvous unit

IV. Design

A. Overview

P^{4}QRS is designed with the following intents: i) It accelerates recirculation-based computation on the switch by reducing the utilization of recirculation bandwidth. ii) It accommodates large payloads and stores them inside the switch, addressing the challenges posed by PayloadPark and Ribosome.

Fig. 1d illustrates our approach, and Fig. 2 depicts a *rendezvous unit*, the specific unit in the switch that realizes the key functionality for the challenges. It is equipped with one I/O port and two recirculation ports. Given a port speed of 100 Gbps, this unit handles traffic up to 100 Gbps while preparing a recirculation bandwidth of 200 Gbps for computation. Queues are associated with these ports, specifically designating one queue for the I/O port and two distinct queues—a header queue and a payload queue—for each recirculation port.

In our approach, the following key processes occur within the unit: (1) *bifurcation*, (2) *payload buffering*, (3) *header computation*, and (4) *rendezvous*. Initially, an incoming packet is assigned an identifier, allocated one of two recirculation ports of the unit (1a), and bifurcated into its header and payload (1b). The header and payload are then separately pushed to and recirculated through distinct queues, with fast and slow service rates, respectively, enabling buffering of the entire payload within the switch (2). Concurrently, the header undergoes computation in pipelines while being recirculated rapidly through the corresponding queue (3). Upon completion of the computation, the corresponding payload exits the queue and reunites with the header—a process we refer to as rendezvous. In this process, the header first stores the processed data in registers (4b). The payload is recirculated through the slow queue until the rendezvous is ready (4a) and retrieves the processed data from the registers (4b). Finally, the payload, now combined with the header, undergoes one more recirculation, enters the queue associated with the I/O port (4c), and departs from the switch.

The rendezvous unit is scalable in that it can be duplicated within the switch depending on the incoming packet rate through run-time control plane settings. Packets can be switched between these units after the final recirculation, enabling packet forwarding to arbitrary egress ports.

B. Bifurcation

This process has two roles: Firstly, the switch assigns identifiers to packets to ensure successful rendezvous between headers and payloads. This needs to be accomplished despite practical challenges such as the finite size of registers, varying input rates, and accidental packet reordering and loss. Secondly, it bifurcates packets into headers and payloads, which are subsequently treated as independent packets.

In the first step, each packet is assigned two identifiers: a packet key (pkt_key) and a packet identifier (pkt_id). A packet key, a 16-bit sequential value, is utilized to look up entries in registers. This key is generated by a counter that produces sequential numbers in modulo N, where N represents the size of the registers. A packet identifier, a 32-bit value, identifies a packet and ensures appropriate rendezvous. This identifier is issued by another counter, which is incremented each time the former counter rolls over.

Next, packets are bifurcated through packet mirroring and truncation. Initially, incoming packets are mirrored, resulting in two copies: the original and the mirrored packet. The mirrored packet is subsequently truncated to remove its payload. In this context, we assume the Ethernet and IP headers, a 16-byte metadata segment, and 64 bytes of data from the IP payload are involved in the computation, although this assumption is adjustable. The combined 114-byte section is hereafter referred to as the *header*, and the entire original packet is broadly referred to as the *payload*.

C. Payload Buffering

This process aims to save the recirculation bandwidth by retaining payloads inside the switch queue for as long as possible, by fulfilling the following requirements: Firstly, the waiting time for payloads in the queue must be significantly longer than that of headers, which must be kept short. Secondly, the lengths of the header and payload queues must not diverge. Thirdly, the process must withstand accidental packet loss.

To this end, we exploit the switch's queue scheduling mechanism. Two separate queues managed with DWRR scheduling are configured: the *header queue* and the *payload queue*. Assigning a lower *weight* to the latter than the former ensures that payloads endure longer waiting times within the queue than headers. The discrepancy in waiting times between headers and payloads simulates payload buffering. We will elaborate on a strategy for determining these weights in Section VI. As an aside, it is worth mentioning that the switch dynamically manages these queue sizes to minimize packet loss.

This process, however, cannot completely prevent unintended eviction of payloads. When this happens, the evicted packets are recirculated and re-entered into the payload queue. Essentially, the purpose of this process is to minimize these inevitable payload recirculations, conserving recirculation bandwidth. Headers and payloads continue to be recirculated and enqueued until further recirculation is no longer necessary. In each recirculation, payloads inspect the state variable maintained by the rendezvous process and stop their recirculation once their corresponding header is ready for rendezvous.

Furthermore, we introduce a mechanism to prevent payloads from being indefinitely recirculated when their corresponding headers have been accidentally lost prior to rendezvous. Each packet has a recirculation counter (recir_ctr) in its metadata, which is incremented each time the packet is recirculated, and any over-recirculated payloads are discarded.

D. Header Computation

This process performs computations similarly to naïve recirculation-based computations but only on headers. Though the headers are recirculated, the utilization of recirculation port bandwidth remains inexpensive as it does not handle payloads.

E. Rendezvous

This process aims at the following objectives: Firstly, payloads should be evicted from the queue promptly upon the completion of the computation process—timing synchronization. Ideally, eviction should occur concurrently with the completion; however, this is challenging since the switch's FIFO queues lack the ability to evict a packet selectively

or at accurate timing. Furthermore, this synchronization must be executed reliably, despite fluctuations in input rates and potential issues such as packet reordering and loss. Secondly, the processed data, a total of 64 bytes, must be correctly transferred from headers to payloads, adhering to the switch's hardware requirements—data transfer. Specifically, access to each state variable must be confined to a single register within a single stage. The width and the number of register entries must meet the switch's specifications.

Timing synchronization. The processed headers notify the corresponding payloads of the completion of computation, by leaving a state variable in a register in a pipeline stage. The register has N entries, which are accessed using packet keys pkt_key, and each entry contains a packet identifier pkt_id. Upon arrival of a packet (a processed header or waiting payload) at this stage, it searches for its entry in the register with its packet key. In the case of the processed header, it places its packet identifier in the respective entry, indicating its readiness. When the corresponding payload is subsequently evicted from the queue, it consults this entry. If the packet identifier contained matches that in the header's metadata, the payload is released from the buffering process and advances to data transfer; if not, it is recirculated and returned to the queue.

In this algorithm, the probability of an incorrect rendezvous is the collision probability of two packet identifiers possessed by packets with the same packet keys. Due to the identifier assigning method, their collision can occur between packets $N \times 2^{32}$ times apart. In practice, this probability is negligible because it is impossible that a packet remains within the system for such a long duration (e.g., over 70k seconds, when $N = 2^{15}$ and 2.0 Gpps rate), owing to the aforementioned mechanism to discard over-recirculated payloads based on recir_ctr.

Data transfer. The processed data is transferred from a header to a payload through 16 registers distributed across multiple stages. The header deposits the data into the pkt_key-th entry of these registers. Subsequently, the corresponding payload accesses these entries and replaces its conveyed data with the concatenation of the retrieved data pieces.

V. Implementation

We implement the P[4]QRS in the P4 language [30] on the Intel Tofino [33] and Tofino 2 [34] ASICs. This implementation seeks to maximize performance while satisfying the switch's specifications and the following requirements: i) The bifurcation process requires the mirroring and truncation functions, both performed at the traffic manager. ii) The computation process must be distributed across all stages, in multiples of complete passes, of the pipeline to ensure that P[4]QRS performs the same computations as the naïve recirculation-based approach. iii) The timing synchronization and the data transfer steps of the rendezvous process must be located within the same pipeline of the same pass to ensure process consistency. iv) Packets must enter the output queue associated with the output port after the rendezvous process.

Fig. 3 illustrates the layout satisfying the requirements. The figure depicts an instance of the computation necessitating n recirculations, resulting in $n + 1$ pipeline passes. In this configuration, headers are recirculated n times and undergo computation spanning n passes out of the total computation. In contrast, the number of recirculations for payloads, denoted as m, is determined by the dynamics of the queues, which are subject to analytical and experimental analyses in the following sections. The rendezvous occurs during the penultimate pass of the payloads through pipelines. Following this, in the final pass, the merged packets are directed to the output queue corresponding to their designated output port, where, simultaneously, the last pass of computation is executed.

VI. Analytical Evaluation

This section investigates P[4]QRS using the analytical model to address the following research questions:

- **RQ1.** *In what ways does P[4]QRS accelerate computations?*
- **RQ2.** *How do parameters, such as the queue weights, a packet arrival rate, and a header recirculation count, influence the system's behavior in a steady state?*
- **RQ3.** *How can the parameters of P[4]QRS be configured to enhance the system's performance and stability?*

A. Mathematical Modeling

Model Structure. The model shown in Fig. 4 incorporates the header queue Q_H, payload queue Q_P, and output queue Q_O. Q_H and Q_P are governed by the DWRR arbiter with service rate μ and weights (ω_H, ω_P), whereas Q_O operates alone at service rate μ_O. For ease of explanation, these weights are normalized by the header and payload size, respectively, and we define $\omega = \omega_H/\omega_P$. They denote the queue weights as a ratio *in packets* rather than *in bytes*, differing from the original weights used in DWRR. Furthermore, we treat these queues as M/M/1 queues, treating cascades and loopbacks of queues as individual simple queues [35], [36]. Then, we derive the individual service rates of Q_H and Q_P by simulating the behavior of the DWRR arbiter using *aggregation of variables* [37]–[39].

Solving the Model. The model is calculated as follows: First, assume a hypothetical average payload recirculation count m. This value is used in conjunction with given parameters, such as header recirculation count n, input rate r, and weights (ω_H, ω_P), to calculate the mean waiting times of Q_H, Q_P, and Q_O, denoted by W_H, W_P, and W_O, respectively. Then, based on these values, m', the resulting payload recirculation count, is obtained by solving the following equation:

$$n(W_H+W_{const})+W_O+W_{const} = m'(W_P+W_{const}) \quad \text{(VI.1)}$$

This equation asserts that the time elapsed between the packet arrival and rendezvous is equal for headers (LHS) and payloads (RHS). W_{const} is the constant portion of latency for a single pass of the switch, empirically set to 750 ns. Iterating this process, we find the expected payload recirculation count m by minimizing $|m' - m| \to 0$ under the constraint that queues do not diverge and Eq. (VI.1), using SLSQP [40]. This minimization leads to m coinciding with m', which implies the successful rendezvous with m recirculations of payloads.

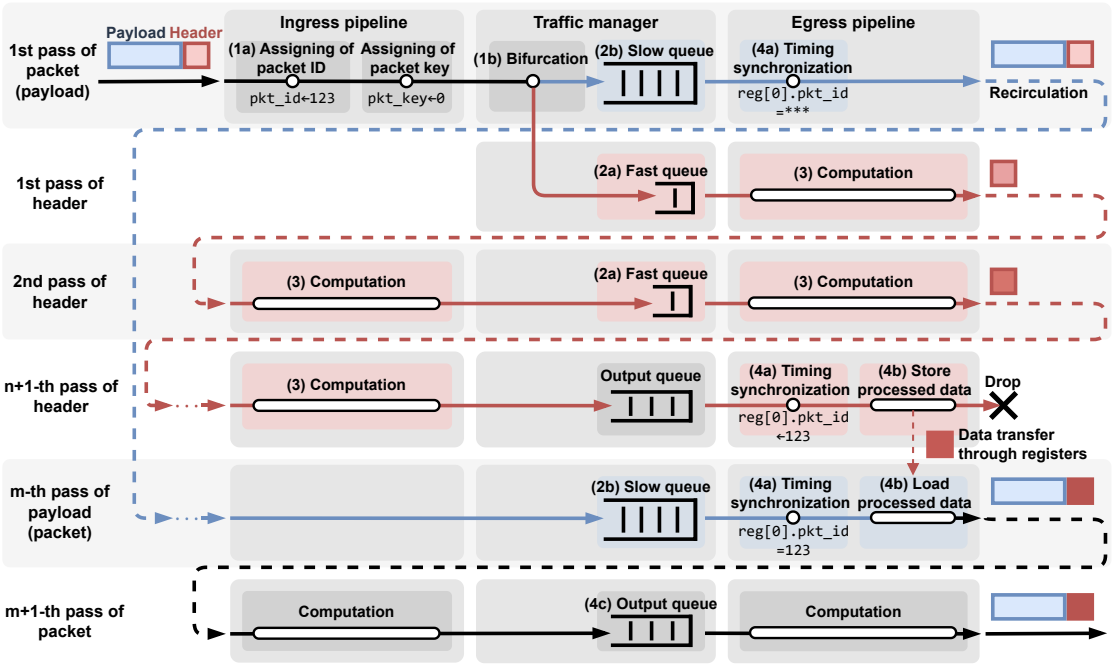


Figure 3: Pipeline layout

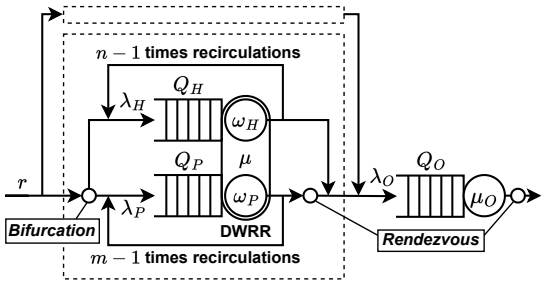


Figure 4: Queueing model of a rendezvous unit

B. Results

Response to RQ1. To accelerate computation, P[4]QRS needs to satisfy the following two conditions:

- There must be a steady state where the queue lengths do not diverge, and the payload recirculation count is significantly less than the header recirculation count ($m < n$).
- Even if the system state is perturbed from this steady state, such as in terms of the payload recirculation count, due to external influences, it should return to the steady state. This requirement means that the system must operate as a closed-loop system with respect to m.

As long as these conditions are met, P[4]QRS can self-maintain its system state. That is, it will eventually converge to the steady state where computation acceleration is accomplished, thereby ensuring robust operation despite fluctuations.

The first condition can be reformulated using W_H and W_P. Given the condition $m < n$ and Eq. (VI.1), the queues must be managed to meet the following condition:

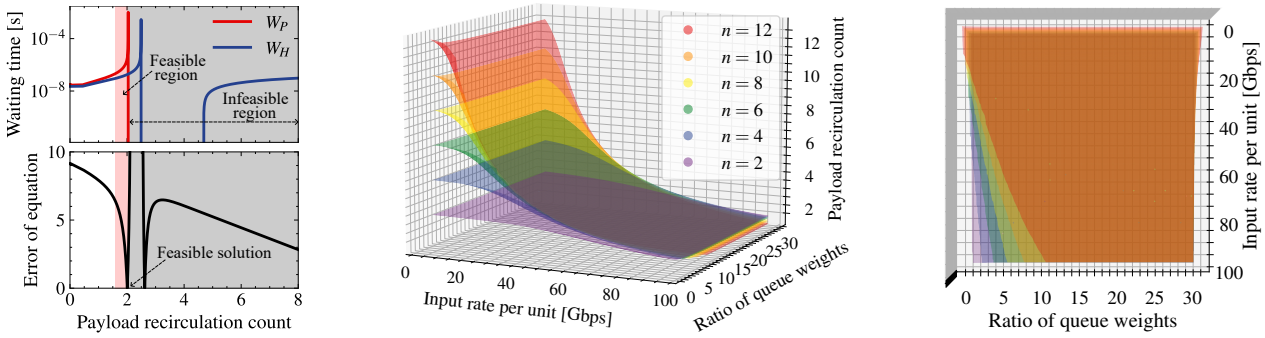
$$W_P - W_H > \frac{W_O + W_{const}}{n} \tag{VI.2}$$

This condition implies that $W_P - W_H$, which depends on m, must exceed a constant determined by the input rate and the header recirculation count, which are given parameters.

To visualize the regions where this condition is satisfied, we calculate how W_P and W_H change with respect to variations in the value of m, as shown in the upper part of Fig. 5a. As an example, we consider the system with $n = 8$ (header recirculation count), $r = 70$ Gbps per port (input rate), $\omega = 20$ (queue weight ratio), and header and payload sizes of 114 and 1500 bytes, respectively. The grey shaded areas indicate infeasible regions where any of the queues diverge, and the red shaded areas represent feasible regions where condition (VI.2) is satisfied. The result indicates that the average payload recirculation count m must fall within the range of 1.58 and 2.04.

Next, we find the payload recirculation where the one derived by the simulation (m') and the hypothetical one (m) are identical. The lower part of Fig. 5a plots the model's objective function, $f(m) = |m' - m|$. It concludes that $m = 2.00$, which is represented by the point on the left in the figure, is a unique solution that results in $f(m) = 0$ and no queue divergence. This solution indeed falls within the feasible region, enabling computation acceleration.

To show P[4]QRS satisfies the second condition, we consider the system's response to a fluctuation in the length of the payload queue. For example, suppose the payload queue temporarily expands due to variations in the input rate or perturbations in m. This expansion extends the waiting time of payloads in the queue during a single pipeline pass.



(a) Queue wait time, equation error (b) Payload recirculation count on analytical model (c) Top view of Fig. (b)

Figure 5: Results of model analysis

As the waiting time of headers in the header queue—and thus the overall time for which payloads must wait—is only marginally affected by the fluctuation, the required number of recirculations for the rendezvous, namely m, decreases. This subsequently leads to a contraction of the payload queue by reducing its arrival rate. In other words, the system exerts a negative feedback on payload queue size, which contributes to the system's stability and adaptability to various input rates.

Response to RQ2. Fig. 5b illustrates the solutions of the model. The horizontal axes represent the parameter space, input rate per rendezvous unit r and queue weight ratio ω. The vertical axis represents solution m. Each surface in the figure corresponds to each header recirculation count n. Areas not covered by a surface indicate no feasible solutions for that parameter. The results lead to the following observations:

- P^{4}QRS more effectively accelerates computation for scenarios with higher input rates because the payload recirculation count (m) decreases as the input rate increases. Additionally, even for scenarios with low input rates, it can still complete the computation at the given rate.
- P^{4}QRS more effectively accelerates computation with higher header recirculation counts (n). With $n = 2$ serving as the baseline, where the computation requires two recirculations and uses two recirculation ports, the computational efficiency is nearly equal to that of the naïve approach. When $n > 2$, P^{4}QRS conserves recirculation bandwidth, enabling the system to perform computations at a higher rate compared to the naïve approach.
- The queue weight configuration of P^{4}QRS is robust to changes in the input rate. Precisely, the queue weight suitable for higher input rates remains valid for lower input rates. This feature allows users to configure the queue weights only assuming the peak input rate.

Response to RQ3. Through our analyses, we derive a guideline for configuring the queue weights. As observed in Fig. 5c, smaller ω make it difficult to reach a steady state, whereas larger ω do not significantly harm the system. Hence, it is recommended to configure the queue weights such that they are significantly different from each other, e.g., $\omega = 20$.

Limitation. Though our model can provide a rough explanation of the system's behavior, the weights derived from the model may not necessarily be optimal for the P^{4}QRS system on a real switch. Additionally, the performance predicted by the model may not be achievable on the switch. This limitation arises from the approximations used when reducing the switch's characteristics into the model using M/M/1 queues and Eq. (VI.1). Hence, we experimentally evaluate P^{4}QRS implemented on a real switch in the following section.

VII. Experimental Evaluation with Real Switches

This section evaluates the performance of P^{4}QRS and demonstrates its practicality in a real-world deployment on real switches by addressing the following research questions:

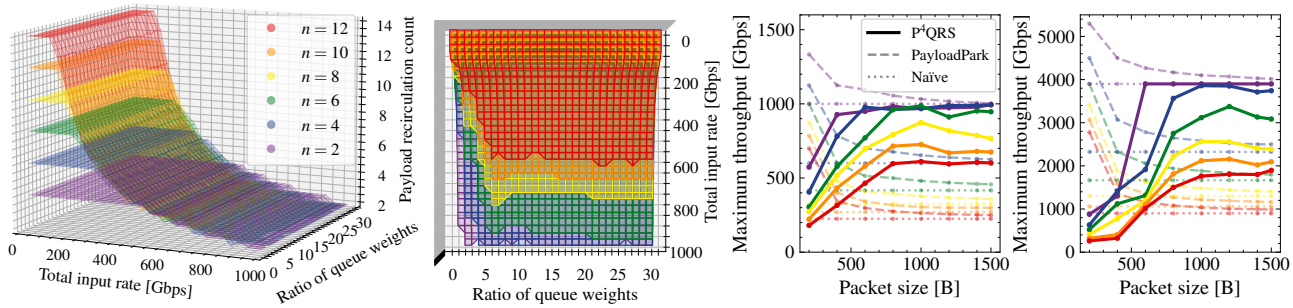
RQ4. *Can P^{4}QRS operate steadily, and do the observations in our analytical model hold true on a real switch?*
RQ5. *Does P^{4}QRS outperform naïve recirculation and the existing in-switch payload storage method?*
RQ6. *Does P^{4}QRS cause excessive latency overhead?*
RQ7. *Do queues in P^{4}QRS exhaust the switch's packet buffer for packet rendezvous?*
RQ8. *How does P^{4}QRS behave in transient states, i.e., in the case of input traffic whose rate changes dynamically?*
RQ9. *How much does P^{4}QRS cause packet reordering?*
RQ10. *How much pipeline resources does P^{4}QRS consume, and can it accommodate various network functions?*
RQ11. *Can P^{4}QRS accommodate complex network functions while maintaining its performance advantages?*

A. Setup

We deploy the P^{4}QRS implementation on the 100BF-32X switches [41] and AS9516-32D [42] switches, equipped with Tofino and Tofino2 ASICs, respectively. The former handles traffic of up to 1.0 Tbps with ten rendezvous units of P^{4}QRS and $N = 2^{15}$. In contrast, the latter manages traffic up to 3.9 Tbps by employing 39 rendezvous units, and N is increased to 47k to accommodate the increased input rate.

B. Results

Response to RQ4. Fig. 6a and 6b show the result on the Tofino switch with the varying header recirculation count, queue weights, and input rate. The input rate per port was



(a) Payload recirculation count (Tofino) (b) Top view of Fig. (a) (c) Max speed (Tofino) (d) Max speed (Tofino 2)

Figure 6: Evaluation results of the P4QRS implementation

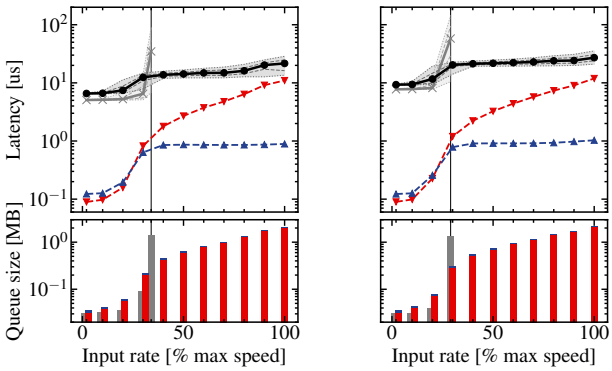
adjusted between 2 Gbps and 98 Gbps. The size of packets is fixed to 1500 bytes. For the queue weights, we set the weight of queue Q_P to 10 *in byte* and set the Q_H weight so that ω varies between 1 and 30 *in packet*. Each lattice point on the surfaces in these figures represents successful rendezvous for that parameter set, whereas areas not covered by the surfaces indicate failed rendezvous, i.e., packet loss.

The following observations support the analytical results:

- P4QRS significantly boosts recirculation-based computation. For instance, the maximum feasible throughput of P4QRS with $n = 8$ in Fig. 6a is 800.1 Gbps, while the naïve method with 8-times recirculation can achieve up to 333.3 Gbps using the same configuration of 30 ports. This translates into a computational acceleration factor of 2.40.
- The experimental results in Figs. 6a and 6b exhibit a trend consistent with the analytical results in Figs. 5b and 5c, although that P4QRS with large n does not completely achieve the performance predicted by the model.
- The rendezvous operation consistently succeeds as long as the queue weights are appropriately configured; that is, there is no internal missing point on all the surfaces.
- The queue weights that enable successful rendezvous at high input rates are also valid at lower input rates.
- The strategy of weight assignment, which is derived in the model analysis, remains valid on the real switch.

Response to RQ5. Fig. 6c and 6d compare the maximum throughput achieved by P4QRS on the Tofino and Tofino 2 switch, respectively, with the naïve approach and PayloadPark (with storage of 160 bytes for each payload), using 30 ports of the switch and varying input packet sizes from 200 to 1500 bytes. The line colors correspond to n as in Fig. 6a. For P4QRS, the weight of Q_P is set to 10, while the weight of Q_H is set to ensure the rate ratio ω *in packets* approximates 20.

The figure indicates that PayloadPark shows strength in handling smaller packet sizes. However, its efficiency diminishes as the packet size increases due to the decreasing ratio of the 160 byte payload segments stored in registers to the total packet size. In contrast, P4QRS outperforms PayloadPark with large packet sizes, except for the case of $n = 2$. Precisely, when dealing with packet sizes of 1000–1500 bytes and when $n \geq 6$, P4QRS accelerates computation to a speed exceeding



(a) $n = 6$ (b) $n = 10$

(Tops): Gray and black plots show the median latency of the naïve approach and P4QRS, respectively. Dotted lines indicate Q1 and Q3, shaded areas represent the 5th to 95th percentiles. Red and blue plots represent W_P and W_H in P4QRS. (Bottoms): Gray bar shows the average total size of recirculation queues. Red and blue bars represent Q_P and Q_H in P4QRS.

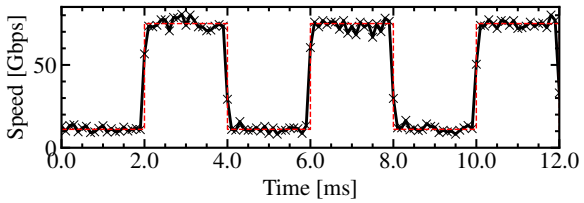
Figure 7: Latency and queue size of P4QRS (Tofino)

twice of the naïve approach on Tofino and achieves speeds of several Tbps on Tofino 2. In comparison, PayloadPark only offers a performance gain of 10–20%.

Response to RQ6. The upper row of Fig. 7 compares P4QRS (black plot) and the naïve approach (gray plot) on the Tofino switch, in terms of their latency with 1500-byte packets, using the same numbers of I/O and recirculation ports for both methods. Firstly, P4QRS introduces slight additional latency at low input rates, but the increase is limited to a few microseconds. Secondly, unlike the naive method, where latency increases rapidly at a certain rate, the latency of P4QRS grows moderately. Ultimately, P4QRS handles much higher input rates than the naive method while maintaining lower maximum latency.

The figures also delineate the intrinsic mechanism of P4QRS: the discrepancy of mean waiting time of Q_P (red plot) and Q_H (blue plot), as explained in RQ1. This discrepancy starts to appear around the maximum rate of the naïve approach and expands for handling higher rates.

Response to RQ7. The lower row of Fig. 7 compares P4QRS (red and blue bars) and the naïve approach (gray bars) on the Tofino switch, in terms of the total consumption of the switch's



Red and black are input (configured) and output (measured) speeds per port.

Figure 8: Behavior in transient states (Tofino)

packet buffer memory by queues used for recirculation. The trend in the results is similar to that of latency, as the primary cause of packet latency is the expansion of queues. The maximum queue occupancy of P^{4}QRS is around 2 MB, which is almost the same as the naïve approach. Given the switch's packet buffer size of 20 MB [43], it leaves plenty of room for the original use of queues, such as burst absorption.

Response to RQ8. The experiment involves a computer serving as the packet transmitter and receiver and two switches, one functioning as an amplifier and the other as the P^{4}QRS. The computer generates traffic of a dynamic pattern at speeds of up to 100 Gbps, which is amplified tenfold by the amplifier before being fed into the P^{4}QRS, and one-tenth of its output is redirected to the computer, where the speed is measured every 100 us. We used 1500-byte packets, $n = 8$, and the traffic pattern set to oscillate between high and low rates.

Fig. 8 indicates that the system functions properly in transient states: the measured output rate immediately tracks the input traffic's transition between low and high rates. First, this result substantiates P^{4}QRS's capability to absorb sharp rate increases, which also suggests that P^{4}QRS leaves sufficient room in the packet buffer, as shown in the previous assessment. Moreover, it demonstrates that P^{4}QRS can quickly transition from one steady state to another without control plane intervention, such as queue weight adjustment, highlighting its capability to manage general wild traffic patterns.

Response to RQ9. As P^{4}QRS process packets individually rather than on a per-flow basis, where the decision of whether to recirculate is made per packet instead of per flow, it may cause packet reordering, which can be particularly problematic in TCP use cases. We measured 10^6 packets processed by P^{4}QRS on the Tofino switch and estimated the reordered packet ratio [44] as follows. The instantaneous peak rate of a TCP flow is observed when TCP segments of the window size are sent back-to-back, and it is generally limited by the bottleneck link speed. It is reasonable to assume that flows are fairly aggregated and interleaved as they enter the switch. Under this assumption, for instance, packets of a flow traversing a 500 Mbps bottleneck link are interleaved over a distance exceeding 100 packets at a switch port. In the evaluation results with $n = 6$ and P^{4}QRS operating at maximum speed, the reordered packet ratio for this flow remains under 10^{-5}. Although the ratio increases to 5% at a distance of 50 packets, such high-rate flows are rare on the Internet [45]. Similarly, for $n = 10$, the ratio is below 10^{-5} when the distance exceeds 53 packets.

Also, more generally, we inspected the reordering extent [44], the maximum distance between two packets directed to the same P^{4}QRS queue and reordered by the time of output. Fig. 10 indicates that the reordering extent measured is up to around a hundred packets, which translates to the length of the payload queue. Consequently, the jitter is at most on the order of a microsecond, contrasting with the millisecond-scale jitters typically observed on the Internet.

These results argue that significant packet ordering is generally limited to remarkably fast flows on the Internet. However, concerns remain for bandwidth-intensive and time-sensitive applications, e.g., in data center environments. In such cases, P^{4}QRS should be extended to preserve packet order within each flow, similar to the approach taken by Switcharoo [46]. Specifically, packets are assigned a sequential identifier based on the 5-tuple during bifurcation and released from buffering in this correct order. This is left for future work.

Response to RQ10. As in Fig. 3, P^{4}QRS reserves all stages of n pipeline passes in the computation process, along with the final pipeline pass, exclusively for application-tailored computation. This design leaves programmers of network functions on P^{4}QRS with a design space equivalent to that available with the naïve method using n-times recirculation.

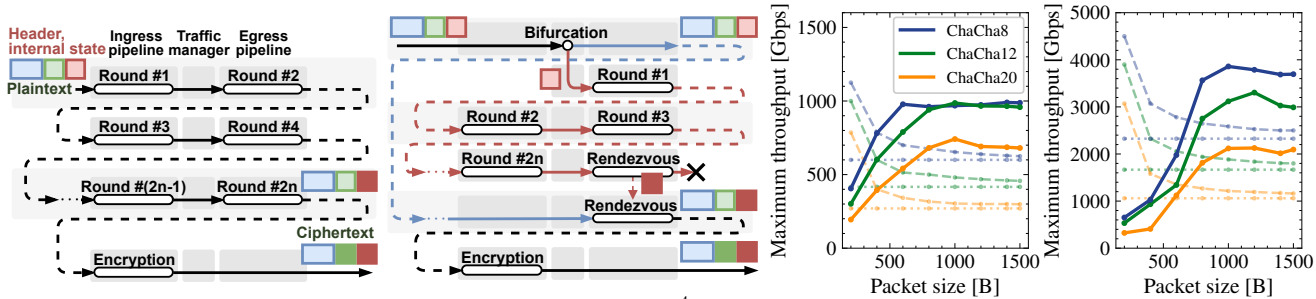
In the configuration of experiments, our P^{4}QRS implementation on Tofino (resp. Tofino 2) utilizes 12.95% (resp. 10.00%), 36.07% (33.38%), 15.48% (9.58%), 11.94% (9.61%), and 8.01% (7.27%) of the switch's VLIW instruction, SRAM, TCAM, exact-match crossbar, and ternary-match crossbar resources, respectively. The SRAM consumption is comparable to that of PayloadPark [25], while it accounts for a total of 10 (resp. 39) rendezvous units, which serve up to 1 Tbps (resp. 3.9 Tbps) traffic. These utilization ratios indicate that the majority of resources are left for computation.

Response to RQ11. We validate the capability of P^{4}QRS to accommodate diverse tasks by examining the implementation of particularly complex headers processing, taking cryptography as an example. We employ ChaCha [47], a stream cipher, and its implementation on the switches [15] for encrypting 64-byte data trailed by headers. Its naïve and accelerated implementation is outlined in Fig. 9a and 9b, respectively.

Fig. 9c and 9d compare performance of the P^{4}QRS-assisted and naïve implementations. ChaCha20, the original ChaCha of 20 rounds, requires 10 header recirculations ($n = 10$), whereas reduced-round variants, ChaCha12 and ChaCha8, involve $n = 6$ and 4, respectively. The figure demonstrates performance gain consistent with previous evaluations of standalone P^{4}QRS. For packets of 1000–1500 bytes, throughput is doubled with ChaCha20 and ChaCha12 and gains 1.5+ times enhancement with ChaCha8 by P^{4}QRS on Tofino. Significant acceleration for these packets is also achieved on Tofino 2, enabling operations with ChaCha20, ChaCha12, ChaCha8 at 2.0 Tbps, 3.0 Tbps, 3.6 Tbps, respectively.

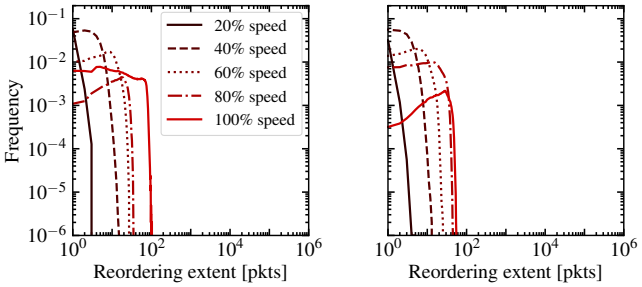
VIII. Discussion

This section discusses the following research questions:



(a) Naïve implementation (b) Implementation using P4QRS (c) Max speed (Tofino) (d) Max speed (Tofino 2)

Figure 9: ChaCha on P4QRS



(a) $n = 6$ (b) $n = 10$

Figure 10: Packet reordering analysis (Tofino)

RQ12. Is recirculating packets within the switches justifiable?
RQ13. What are P4QRS's expected standard applications?
RQ14. Is it possible to apply P4QRS to other types of programmable switches besides the Tofino family?
RQ15. What does P4QRS suggest about the future processor architecture for in-network computing?

Response to RQ12. The switches are a promising option for flexible packet processing in Tbps-scale forwarding, and limited header recirculations do not negate this merit.

Regarding processing performance, in the ideal case where chip speed rather than port bandwidth is the bottleneck, the 160-fold parallelism provided by Tofino 2 (four pipelines, 40 stages per pass), besides further acceleration by incorporating VLIW, MAUs, and other externs, is advantageous. P4QRS focuses on eliminating bottlenecks from port bandwidth and approaching this ideal case, and demonstrates, for instance, that computations requiring 360 stages (8 recirculations) can be performed at 300 Mpps (2.5 Tbps, 1024-byte packets) with about 1 GHz frequency, offering 100-fold parallelism.

Regarding forwarding speed, processing the whole packet within the switch is an essential approach. Retaining payloads externally reduces loads on the NF server but does not resolve the intrinsic problem of transferring a large volume of traffic to computers, leading to high resource demands.

Response to RQ13. Although buffering, payload access, and statefulness occurring in NFs pose challenges to the application of P4QRS, a substantial portion of common NFs can still be implemented with P4QRS, according to the discussion presented by Chen et al. [27]. Examples of such NFs include load balancer, congestion control, coordination, ACL, and monitoring. While they may be simple individually, they are often combined [48], needing multiple recirculations.

The state-of-the-art NF chains adopt pipeline folding, which optimally allocates switch pipelines for NFs [49]. P4QRS currently assumes workload equally distributed across pipelines, leaving implementation of this method as future work.

Response to RQ14. As P4QRS process resorts to only basic features of today's programmable switches, i.e., packet processing pipelines, packet mirroring (multicasting), truncation, stateful memory, and QoS control on queues, it is applicable to RMT-based switches, potentially including Broadcom's Jericho [50], Marvell's Teralynx [51], AMD's Pensando [52], NVIDIA's Spectrum [53], and Centec's GoldenGate [54].

Response to RQ15. The issue of recirculation overhead emphasizes the necessity of retaining large payloads within the switch and retrieving them upon the headers' request, a feature not intrinsically supported by the RMT architecture. dRMT architecture [55] and its variants [56], [57] address this challenge by adopting a run-to-completion architecture. However, such architectures can have fewer ALUs in each computation unit [55], thus reducing parallelism compared to RMT. In contrast, P4QRS suggests a way to solve this problem by leveraging in-switch memory while maintaining the advantages of RMT architecture. Our evaluations confirm that the costs for in-switch payload retention are reasonable.

IX. Conclusion

This paper proposes P4QRS, an acceleration scheme for in-switch computation. By exploiting the switch's queues, P4QRS enables the storage of the entire payload, along with packet bifurcation, computation, and rendezvous, within a single switch. An analytical model and experimental evaluation confirm P4QRS's stable operation and significant acceleration in complex computations scenarios.

Acknowledgement

We thank our shepherd, Giuseppe Di Battista, and the anonymous reviewers from CoNEXT '23, NSDI '24, and ICNP '24 for their constructive feedback. This work was supported by JSPS KAKENHI Grant Numbers 24K02930, 24KJ1629.

REFERENCES

[1] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 99–110, 2013.

[2] K. Subramanian, A. Abhashkumar, L. D'Antoni, and A. Akella, "D2R: Policy-compliant fast reroute," in *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR '21)*, 2021, pp. 148–161.

[3] 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.

[4] D. Hancock and J. Van der Merwe, "HyPer4: Using P4 to virtualize the programmable data plane," in *Proceedings of the 12th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '16)*, 2016, pp. 35–49.

[5] C. Zhang, J. Bi, Y. Zhou, and J. Wu, "HyperVDP: High-performance virtualization of the programmable data plane," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 3, pp. 556–569, 2019.

[6] J. Lee, H. Ko, H. Lee, and S. Pack, "Flow-aware service function embedding algorithm in programmable data plane," *IEEE Access*, vol. 9, pp. 6113–6121, 2020.

[7] D. Wu, A. Chen, T. E. Ng, G. Wang, and H. Wang, "Accelerated service chaining on a single switch ASIC," in *Proceedings of the 18th ACM Workshop on Hot Topics in Networks (HotNets '19)*, 2019, pp. 141–149.

[8] J. Ma, S. Xie, and J. Zhao, "P4SFC: Service function chain offloading with programmable switches," in *2020 IEEE 39th International Performance Computing and Communications Conference (IPCCC)*, 2020, pp. 1–6.

[9] Y. Zhou, J. Bi, C. Zhang, M. Xu, and J. Wu, "FlexMesh: Flexibly chaining network functions on programmable data planes at runtime," in *2020 IFIP Networking Conference (Networking)*, 2020, pp. 73–81.

[10] H. Huang, W. Wu, Y. He, and Z. Guo, "SFP: Service function chain provision on programmable switches for cloud tenants," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2022, pp. 1239–1249.

[11] Q. Kang, L. Xue, A. Morrison, Y. Tang, A. Chen, and X. Luo, "Programmable in-network security for context-aware BYOD policies," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 595–612.

[12] Y. Yoshinaka, M. Kochiyama, Y. Koizumi, J. Takemasa, and T. Hasegawa, "A lightweight anonymity protocol at terabit speeds on programmable switches," *Computer Networks*, vol. 253, no. 110721, pp. 1–16, 2024.

[13] S. Yoo and X. Chen, "Secure keyed hashing on programmable switches," in *Proceedings of the ACM SIGCOMM 2021 Workshop on Secure Programmable network INfrastructure (SPIN '21)*, 2021, pp. 16–22.

[14] X. Chen, "Implementing AES encryption on programmable switches via scrambled lookup tables," in *Proceedings of the Workshop on Secure Programmable Network Infrastructure (SPIN '20)*, 2020, pp. 8–14.

[15] Y. Yoshinaka, J. Takemasa, Y. Koizumi, and T. Hasegawa, "On implementing ChaCha on a programmable switch," in *Proceedings of the 5th International Workshop on P4 in Europe (EuroP4 '22)*, 2022, pp. 15–18.

[16] M. C. Luizelli, R. Canofre, A. F. Lorenzon, F. D. Rossi, W. Cordeiro, and O. M. Caicedo, "In-network neural networks: Challenges and opportunities for innovation," *IEEE Network*, vol. 35, no. 6, pp. 68–74, 2021.

[17] K. Razavi, G. Karlos, V. Nigade, M. Mühlhäuser, and L. Wang, "Distributed DNN serving in the network data plane," in *Proceedings of the 5th International Workshop on P4 in Europe (EuroP4 '22)*, 2022, pp. 67–70.

[18] R. Das and A. C. Snoeren, "Enabling active networking on RMT hardware," in *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets '20)*, 2020, pp. 175–181.

[19] J. Jiang, Z. Huang, Q. Xiang, L. Tang, and J. Shu, "P4-DPLL: Accelerating SAT solving using switching ASICs," in *Proceedings of the ACM SIGCOMM Workshop on Formal Foundations and Security of Programmable Network Infrastructures (FFSPIN '22)*, 2022, pp. 24–30.

[20] R. Glebke, J. Krude, I. Kunze, J. Rüth, F. Senger, and K. Wehrle, "Towards executing computer vision functionality on programmable network devices," in *Proceedings of the 1st ACM CoNEXT Workshop on Emerging in-Network Computing Paradigms (ENCP '19)*, 2019, pp. 15–20.

[21] S. Vaucher, N. Yazdani, P. Felber, D. E. Lucani, and V. Schiavoni, "ZipLine: In-network compression at line speed," in *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '20)*, 2020, pp. 399–405.

[22] M. Chiesa, R. Sedar, G. Antichi, M. Borokhovich, A. Kamisiński, G. Nikolaidis, and S. Schmid, "Fast ReRoute on programmable switches," *IEEE/ACM Transactions on Networking*, vol. 29, no. 2, pp. 637–650, 2021.

[23] S. Fu, K. Xu, Q. Li, X. Wang, S. Yao, Y. Guo, and X. Du, "MASK: Practical source and path verification based on multi-AS-key," in *2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)*, 2021, pp. 1–10.

[24] S. Fu, Q. Li, X. Wang, S. Yao, X. Feng, Z. Wang, X. Du, K. Wan, and K. Xu, "D3: Lightweight secure fault localization in edge cloud," in *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*, 2022, pp. 515–525.

[25] S. Goswami, N. Kodirov, C. Mustard, I. Beschastnikh, and M. Seltzer, "Parking packet payload with P4," in *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '20)*, 2020, pp. 274–281.

[26] M. Scazzariello, T. Caiazzi, H. Ghasemirahni, T. Barbette, D. Kostic, and M. Chiesa, "A high-speed stateful packet processing approach for Tbps programmable switches," in *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023, pp. 1237–1255.

[27] X. Chen, Q. Huang, P. Wang, Z. Meng, H. Liu, Y. Chen, D. Zhang, H. Zhou, B. Zhou, and C. Wu, "LightNF: Simplifying network function offloading in programmable networks," in *2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)*, 2021, pp. 1–10.

[28] W. John and S. Tafvelin, "Analysis of Internet backbone traffic and header anomalies observed," in *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement (IMC '07)*, 2007, pp. 111–116.

[29] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement (IMC '10)*, 2010, pp. 267–280.

[30] 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.

[31] M. Shreedhar and G. Varghese, "Efficient fair queuing using deficit round-robin," *IEEE/ACM Transactions on Networking*, vol. 4, no. 3, pp. 375–385, 1996.

[32] D. Merling, S. Lindner, and M. Menth, "Hardware-based evaluation of scalable and resilient multicast with BIER in P4," *IEEE Access*, vol. 9, pp. 34 500–34 514, 2021.

[33] (2023) Intel® Tofino™. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html. Accessed on May 14, 2024.

[34] (2023) Intel® Tofino™ 2. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html. Accessed on May 14, 2024.

[35] P. J. Burke, "The output of a queuing system," *Operations research*, vol. 4, no. 6, pp. 699–704, 1956.

[36] J. R. Jackson, "Jobshop-like queueing systems," *Management science*, vol. 10, no. 1, pp. 131–142, 1963.

[37] H. A. Simon and A. Ando, "Aggregation of variables in dynamic systems," *Econometrica: journal of the Econometric Society*, pp. 111–138, 1961.

[38] A. J. Fehske and G. P. Fettweis, "Aggregation of variables in load models for interference-coupled cellular data networks," in *2012 IEEE International Conference on Communications (ICC)*, 2012, pp. 5102–5107.

[39] E. Fischer, A. Fehske, G. Fettweis *et al.*, "A flexible analytic model for the design space exploration of many-core network-on-chips based on queueing theory," in *Proceedings of The Fourth International Conference on Advances in System Simulation (SIMUL)*, 2012, pp. 119–124.

[40] D. Kraft, "A software package for sequential quadratic programming," *Technical Report DFVLR-FB 88-28, DLR German Aerospace Center - Institute for Flight Mechanics*, 1988.

[41] (2023) Wedge 100BF-32X. https://www.edge-core.com/wp-content/uploads/2023/08/DCS800-Wedge100BF-32X-R11.pdf. Accessed on May 21, 2024.

[42] "Data center switch AS9516-32D," https://www.edge-core.com/_upload/images/2023-061-DCS810_AS9516-32D_DS_R07_20230503.pdf, 2023, accessed on May 21, 2024.
[43] (2021) $P4_{16}$ Intel® Tofino™ native architecture – public version. https://github.com/barefootnetworks/Open-Tofino/blob/master/PUBLIC_Tofino-Native-Arch.pdf. Accessed on May 15, 2024.
[44] A. Morton, L. Ciavattone, G. Ramachandran, S. Shalunov, and J. Perser, "Packet reordering metrics," Internet Requests for Comments, RFC 4737, November 2006. [Online]. Available: https://www.rfc-editor.org/rfc/rfc4737.txt
[45] S. Bauer, B. Jaeger, F. Helfert, P. Barias, and G. Carle, "On the evolution of internet flow characteristics," in *Proceedings of the 2021 Applied Networking Research Workshop (ANRW '21)*, 2021, pp. 29–35.
[46] T. Caiazzi, M. Scazzariello, and M. Chiesa, "Millions of low-latency state insertions on ASIC switches," *Proceedings of the ACM on Networking*, vol. 1, no. CoNEXT3, pp. 1–23, 2023.
[47] D. J. Bernstein, "ChaCha, a variant of Salsa20," in *Workshop Record of the State of the Art of Stream Ciphers (SASC)*, vol. 8, no. 1, 2008, pp. 3–5.
[48] J. Halpern and C. Pignataro, "Service function chaining (SFC) architecture," RFC 7665, 2015.
[49] T. Pan, K. Liu, X. Wei, Y. Qiao, J. Hu, Z. Li, J. Liang, T. Cheng, W. Su, J. Lu *et al.*, "LuoShen: A hyper-converged programmable gateway for multi-tenant multi-service edge clouds," in *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 2024, pp. 877–892.
[50] B. Wheeler. (2020) Jericho2c+ brings 7nm to routers. https://docs.broadcom.com/doc/88850-Report1-PUB. Accessed on May 16, 2024.
[51] (2023) Marvell® teralynx® 10 data center Ethernet switch. https://www.marvell.com/content/dam/marvell/en/public-collateral/switching/marvell-teralynx-10-data-center-ethernet-switch-product-brief.pdf. Accessed on May 16, 2024.
[52] (2024) AMD Pensando™ DPUs offer high performance while helping lower TCO for cloud service providers. https://www.amd.com/content/dam/amd/en/documents/pensando-business-docs/solution-brief/csp-case-study.pdf. Accessed on May 16, 2024.
[53] (2020) NVIDIA Mellanox Spectrum-3. https://network.nvidia.com/files/doc-2020/pb-spectrum-3.pdf. Accessed on May 16, 2024.
[54] (2016) Centec V350 product introduction. https://people.ucsc.edu/~warner/Bufs/V350_Introduction_R1.5.pdf. Accessed on May 16, 2024.
[55] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy *et al.*, "dRMT: Disaggregated programmable switching," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*, 2017, pp. 1–14.
[56] M. Yang, A. Baban, V. Kugel, J. Libby, S. Mackie, S. S. R. Kananda, C.-H. Wu, and M. Ghobadi, "Using Trio: Juniper Networks' programmable chipset-for emerging in-network applications," in *Proceedings of the ACM SIGCOMM 2022 Conference (SIGCOMM '22)*, 2022, pp. 633–648.
[57] K. Deierling, "NVIDIA's resource transmutable network processing ASIC," in *2023 IEEE Hot Chips 35 Symposium (HCS)*, 2023, pp. 1–14.