IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 3 May 2023, accepted 8 June 2023, date of publication 15 June 2023, date of current version 22 June 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3286726

RESEARCH ARTICLE

A Hardware Architecture of a Dynamic Ranking Packet Scheduler for Programmable Network Devices

MOSTAFA ELBEDIWY, BILL PONTIKAKIS, JEAN-PIERRE DAVID, (Member, IEEE), AND YVON SAVARIA, (Fellow, IEEE)
Department of Electrical Engineering, Polytechnique Montréal, Montreal, QC H3T 1J4, Canada

Corresponding author: Mostafa Elbediwy (mostafa.elbediwy@polymtl.ca)

This work was supported in part by Kaloom, Intel, Noviflow, Prompt Quebec; and in part by the Natural Sciences and Engineering Research Council of Canada.

ABSTRACT The introduction of Software-Defined Networking (SDN) separated the control and data forwarding planes, but the data plane still requires a fully programmable packet scheduler that can adapt to different traffic patterns and offer high expressiveness at line rate. The Dynamic Ranking Push-In-First-Out (DR-PIFO) is a novel programmable hardware queue architecture, introduced for the widely-used Portable Switch Architecture (PSA) used in modern network switches and routers. With the aid of a re-ranking mechanism, the DR-PIFO offers a flexible and expressive solution for a wide range of scheduling algorithms while still meeting line rate requirements. Our design, synthesized using TSMC's 65nm technology, achieves the desired timing rate of 1GHz while maintaining a throughput that matches the fastest existing schedulers and incurs a mere 15.5% increase in area compared to the state-of-the-art PIFO design. The proposed DR-PIFO's hardware implementation is shown to closely approach the behavior and performance of its algorithmic model by efficiently executing various scheduling algorithms, leading to precise bandwidth distribution among traffic flows. Additionally, the DR-PIFO offers a significant reduction in the relative flow completion time (FCT) errors, exhibiting a minimum of 30% less error compared to the previously proposed models when implementing various scheduling policies with workloads collected from data centers. Thus, we believe that the DR-PIFO is a significant step toward making hardware packet schedulers more programmable.

INDEX TERMS Software-defined networking, programmable packet schedulers, hardware queues, traffic management.

I. INTRODUCTION

Historically, network equipment such as switches and routers were not programmable, and many still are not today. Telecom operators (Telcos) were at the mercy of hardware equipment vendors and their lengthy ASIC update cycles to implement well-known scheduling algorithms. This left operators without the flexibility to introduce new or modify existing algorithms on their deployed network equipment. But with the advent of network equipment programmable using high-level languages like P4, the paradigm is shifting, giving operators more control in a software-defined networking environment. Currently, programmable network equipment still relies on configurable packet schedulers, which are implemented as fixed-function modules having parameters that can be configured.

Designing and implementing a programmable packet scheduler is a complex endeavor. To be truly effective, the scheduler must have the flexibility to accommodate modifications to existing algorithms and allow for the introduction of new ones, potentially ones that have not been created yet, throughout the life-cycle of the network switches. This level of flexibility requires a programmable, not just configurable, packet scheduler. However, the hardware implementation

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

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

must also be efficient enough to keep up with the demanding pace of network equipment, which must handle a high volume of incoming packets at a line rate, without sacrificing speed during the enqueue and dequeue of packets.

In [1], a programmable packet scheduler known as the Dynamic Ranking Push-In-First-Out (DR-PIFO) was presented as an algorithmic model. Unlike prior proposals, the DR-PIFO exhibits a more expressive, algorithm-agnostic model, allowing it to operate independently of any scheduling algorithm. Its added expressiveness arises from innovative features such as the forced dequeue primitive, error detection mechanism, and dynamic updating of flow priorities while waiting in hardware queues. While these new elements enhance the DR-PIFO's flexibility, they also increase the complexity of the original PIFO architecture outlined in [2]. When deploying the DR-PIFO in hardware switches, careful consideration must be given to the design to ensure it can meet the expected throughput of 1 enqueue and 1 dequeue every 1ns and minimize any increase in the switch chip area.

In this work, we present a hardware architecture of the DR-PIFO packet scheduler's algorithmic model that is both efficient and effective. The proposed architecture accommodates the new features introduced by the DR-PIFO algorithmic model while ensuring the required frequency of 1GHz, which is standard in modern switches operating at line rate [2]. We outline the hardware implementation of the DR-PIFO architecture in this paper to validate its operating frequency and compute its area consumption. The results show that the hardware design of the DR-PIFO introduces a mere 15.5% overhead to the original PIFO packet scheduler's chip area. We believe that the added overhead in the chip area is a reasonable trade-off for the increased versatility in expressing scheduling algorithms achieved through this design.

The efficacy of our proposed architecture was evaluated through a series of experiments, in which the workloads from a previous study [1] were reapplied. Our design was compared against the DR-PIFO algorithmic model, as well as the PIEO [2] and PIFO [3] models. Despite the added latency of one clock cycle due to the new features, the results demonstrate that our DR-PIFO hardware design operates at the necessary throughput while retaining superior scheduling expressiveness compared to the PIEO and PIFO. Moreover, the design maintains the algorithm-agnostic behavior of the DR-PIFO algorithmic model.

The key contributions of this work are the following:

- A presentation of a novel hardware architecture for the DR-PIFO packet scheduler.
- A relatively low area overhead of 15.5% compared to the PIFO packet scheduler hardware design while achieving improved expressiveness with the DR-PIFO.
- A validation of the proposed design's ability to perform 1 dequeue and 1 enqueue operation per clock cycle of 1ns.
- An evaluation of the hardware design's effectiveness in mimicking the behavior of the DR-PIFO packet scheduler algorithmic model when implementing the pFabric and VDS scheduling algorithms used as significant case studies.

II. BACKGROUND

A. SOFTWARE IMPLEMENTATIONS OF PACKET SCHEDULERS

The desired flexibility can be achieved by implementing the scheduler in software. However, this approach is known to result in low performance in terms of the number of CPU cycles required to execute such software implementations [4], leading to decreased performance in cloud services due to added latency in the network [5]. The number of CPU cycles used for enqueueing, dequeueing, and forwarding packets could pose a challenge for more complex applications [2]. In general, software-based packet switching and traffic shaping exhibit poor CPU utilization and result in inaccuracies in rate control [4]. To avoid this performance degradation, a programmable hardware abstraction of a packet scheduler integrated into the data plane of programmable switches can offer the required flexibility while still preserving performance.

B. HARDWARE ARCHITECTURES OF FIXED-FUNCTION PACKET SCHEDULERS

One of the pioneering studies on hardware packet schedulers can be traced back to [6]. The authors conducted a comprehensive comparison of various queuing architectures and introduced novel concepts for hardware schedulers. The scalability of hardware queues was explored, with the authors demonstrating that the use of shift registers in priority queues can eventually impede the scheduler's ability to process a greater volume of packets. The capability of a packet scheduler architecture to adapt as traffic activity rises is crucial, particularly as the advent of 5G technology and its promising wireless-wireline convergence is poised to drive a surge in traffic activity and further emphasize the importance of scalability [7].

The hardware implementation of a packet scheduler within a programmable switch was presented in [8]. The authors focused on expressing the Weighted Fair Queuing (WFQ) scheduling algorithm into an OpenFlow switch [9]. The scheduler was implemented on a NetFPGA executing at a clock frequency of 125 MHz and utilizing dedicated FIFO queues, one for each flow, as buffers. To ensure that packets were dequeued in the proper order, the authors employed a comparator at the output level of the FIFO queues. The challenge of finding the optimal implementation of the WFQ on fixed switch abstraction has previously been raised in [10], while [11] explores the use of reconfigurable switches to execute the approximated fair queueing. Such approximations are crucial for implementing fair queuing and Active Queue Management (AQM) algorithms on modern programmable switches. This is because these switches do not have a built-in abstraction for a programmable packet scheduler. Consequently, any new scheduling algorithm

must be compatible with the existing fixed-function packet schedulers. By introducing a programmable packet scheduler, the cost and time required to implement a new AQM can be minimized while also providing greater opportunities for innovation and customization [12].

C. HARDWARE ARCHITECTURES OF PROGRAMMABLE PACKET SCHEDULERS

In [13], the authors introduced a general hardware architecture for a packet scheduler to meet 5G specifications. This architecture was implemented on an FPGA with a 40 Gb/s throughput and 80 MHz frequency for 64-byte packets. The authors then expanded their proposal to a programmable traffic manager on an FPGA platform [14], which achieved a latency of 2 clock cycles at a frequency of 150 MHz, suitable for handling minimum packet sizes in 100 Gbps Ethernet networks.

A high potential hardware architecture for implementing a programmable packet scheduler was proposed in [3], [15], which is based on the Push-In-First-Out (PIFO) queue primitive [15]. This design inspired other hardware scheduler designs such as the Push-In-Extract-Out (PIEO) [2], calendar queue [16], PIPO [17], AIFO [18], FAIFO [19], and SP-PIFO [20] [21]. Although each of these schedulers offers advantages over the PIFO, they also come with limitations. For instance, the PIEO allows for a wider range of algorithms with its forced dequeue operation and has greater scalability than PIFO as it can support up to 30x more flows, but it also has a 50x lower throughput and cannot be pipelined [2]. The SP-PIFO has been implemented on a real-time programmable hardware switch. However, the main limitation of the SP-PIFO is that it approximates the PIFO's behavior with no guarantee to perfectly emulate it [20]. In addition, the SP-PIFO requires one ingress pipeline stage per queue in order to properly compute the packets' ranks. Furthermore, in the SP-PIFO implementation, all input packets must be processed by the same sequence of pipeline stages, even if there are other available parallel pipelines. Although calendar queues share a packet sorting mechanism similar to the PIFO, they introduce a more complex ranking structure and cannot accommodate packets with scheduling times in the past or beyond the highest available future time in the calendar. Despite its limitations, the PIFO remains the preferred hardware abstraction available.

Generally speaking, a PIFO is a simple priority queue that inserts elements based on their priority into any position within the queue. The elements, either packets or references to packets, reside in logical PIFOs. However, PIFO can only dequeue elements from the head of the queue [15]. The PIFO, with its 1 GHz frequency, is an ideal solution for switches operating at line rate. Its total area overhead is modest, amounting to just 3.7% of the minimum chip area utilized in switches [3]. The PIFO model is derived from two observations: most scheduling algorithms can rank packets in the ingress pipeline stage, and for many algorithms, the scheduling task is performed across flows with monotonically decreasing priority. This allows for only the head packets of the flows to be sorted, reducing overhead, but can also result in low-priority flows starvation if high-priority flows use more bandwidth than they should. This can arise as a consequence of a strict priority scheduling scheme [22]. The limitation of only being able to dequeue the head packets restricts the functionality of certain scheduling algorithms, including pFabric [23], Virtual Deadline Scheduling (VDS) [24], and several data-center protocols [25], [26], which require the ability to update the priority of packets after they have been enqueued.

D. THE DR-PIFO PACKET SCHEDULER

The algorithmic model of the DR-PIFO packet scheduler has been presented in [1]. The DR-PIFO is a programmable packet scheduler drawing inspiration from the PIFO [3] and the Push-In-Arbitrary-Out (PIAO) [27] models. It comprises four crucial components: a rank computation unit, a re-ranking unit, a packet rank store, and a flow scheduler. The scheduling algorithms can be expressed using the rank computation and re-ranking units, making them suitable for implementation on programmable targets using a domain-specific language, such as Domino [28] or P4 [29].

The rank computation unit assigns a rank to each incoming packet. The re-ranking unit evaluates the necessary actions, such as updating the rank of packets belonging to a flow or forcing the dequeue operation on the current flow, to ensure correct scheduling. The DR-PIFO packet scheduler features a packet rank store and a flow scheduler that operate at line rate to preserve the switch's overall throughput. These components support the essential features of a re-ranking scheme, including updating ranks and executing forced dequeue operations.

The DR-PIFO packet scheduler matches the level of expressiveness of the PIFO scheduler. In addition, it supports an array of unique features, including the ability to dynamically update flow priorities within its queues, perform a forced dequeue from a specific flow, and implement an error detection mechanism for departures. These features make the DR-PIFO packet scheduler a superior option compared to the PIFO and PIEO schedulers since the reported hardware implementation of the DR-PIFO can support more accurately and at line rate a wider variety of scheduling algorithms as we will show in later sections.

The performance of the DR-PIFO algorithm was evaluated using two different case studies in [1]. The first case study involved the pFabric algorithm with starvation prevention and used real-world data center workloads [23]. The second case study employed the VDS algorithm [24], with the same workload selected in [30]. These case studies showed that the DR-PIFO algorithm can express the chosen algorithms in a more effective way compared to PIFO and PIEO. Furthermore, the results demonstrate that, unlike PIFO and PIEO, DR-PIFO is an algorithm-agnostic model and

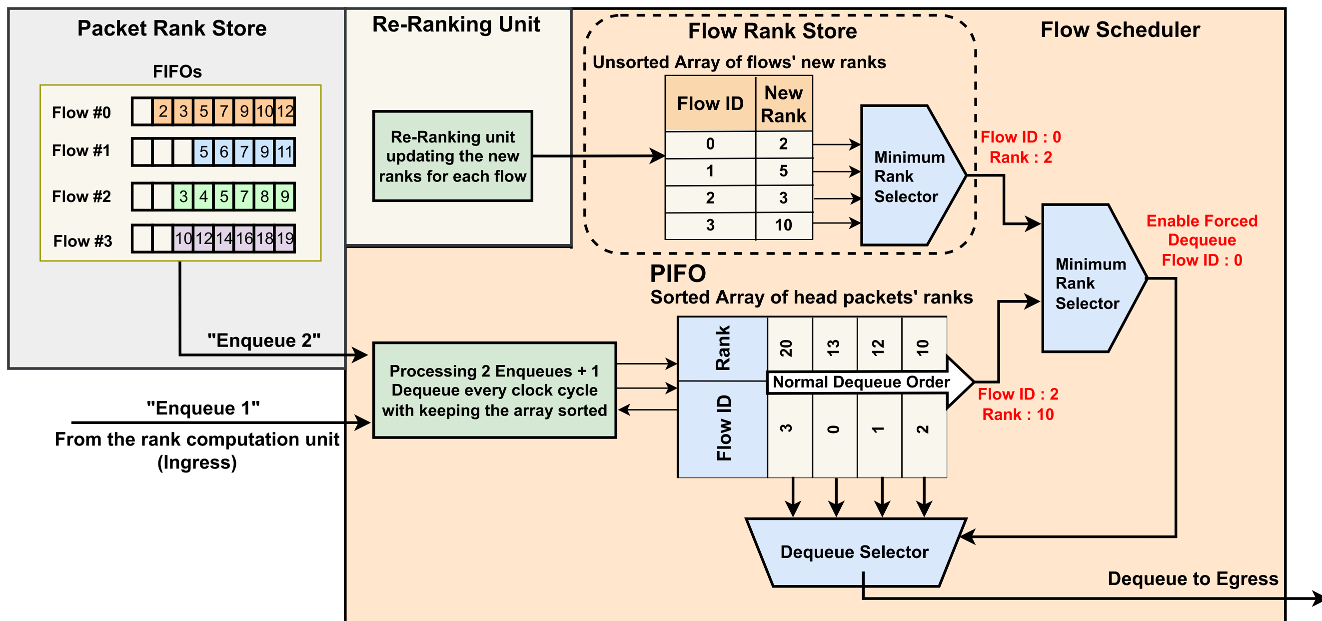

FIGURE 1. A high-level architecture of the DR-PIFO packet scheduler. To illustrate the concept of DR-PIFO, this figure presents an example where 4 flows are scheduled by the pFabric algorithm using the DR-PIFO packet scheduler.

therefore, is not dependent on any specific scheduling algorithm.

III. THE DR-PIFO ARCHITECTURE

In the rest of the present paper, we present a hardware implementation of a DR-PIFO packet scheduler that operates at a line rate. The proposed high-level architecture, shown in Fig. 1, calculates packet rank at the ingress pipeline using a method specified with a domain-specific language such as P4 [20] or Domino [3] that target programmable network devices. The fixed-function unit of the scheduler, including the packet rank store and the flow scheduler, is designed to ensure line-rate performance. On the other hand, the re-ranking unit, being programmable with more relaxed timing constraints [3], can be implemented using any programmable hardware. The focus of the present paper is thus on the design of the fixed-function units and the interfaces between the programmable and fixed-function components within the DR-FIFO.

When a new packet arrives at the switch, it is first evaluated to determine if there are any other packets with the same flow ID in the queue. If there are no other packets, the new packet is considered the head of its flow and inserted directly into the flow scheduler using the "Enqueue 1" operation, as shown in Fig. 1. However, if there are other packets, the new packet is added to the dedicated First-In-First-Out (FIFO) queue for its flow ID, located within the packet rank store, which contains a bank of FIFOs. When a head packet from a particular flow ID is removed from the flow scheduler, its subsequent packet, belonging to the same flow ID and located in the packet rank store, is added to the flow scheduler using the "Enqueue 2" procedure. The flow rank store, as depicted in Fig. 1, holds an unsorted array of the new ranks for each flow, which can be calculated based on events such as the arrival or departure of packets.

The flow rank store serves three main purposes: it stores the updated rank of each flow, shares these ranks with other blocks, and allows the re-ranking unit to update the rank of a single flow at any given time. The flow scheduler, using a PIFO-based queue, sorts packets based on their ranks in ascending order. It performs up to two enqueue operations and dequeues the head packet with the lowest rank, which should be first in the queue. Additionally, it offers a forced dequeue operation that removes the head packet of a specific flow ID, regardless of its place in the PIFO queue.

The flow scheduler may experience errors in the departure order of packets, as it may dequeue a packet from a flow that does not have the lowest rank according to updated ranks. This occurs as the scheduler is unable to constantly re-sort all the head packets with every change in flow ranks. As a result, if a flow's rank is updated, its head packet in the flow scheduler may not reflect the updated rank, leading to errors in the departure order. Additionally, this inability may result in the starvation of flows. To address these issues, the flow scheduler implements an error detection mechanism that recognizes if a recently dequeued packet does not belong to the flow with the minimum rank, according to the new ranks array. The minimum rank selectors in Fig. 1 are then utilized to identify any errors or discrepancies according to some idealized or specified schedule in the packet departure order. The flow rank store calculates the minimum rank among the updated ranks and shares it with the flow scheduler. In the event of an error in the departure order, the flow

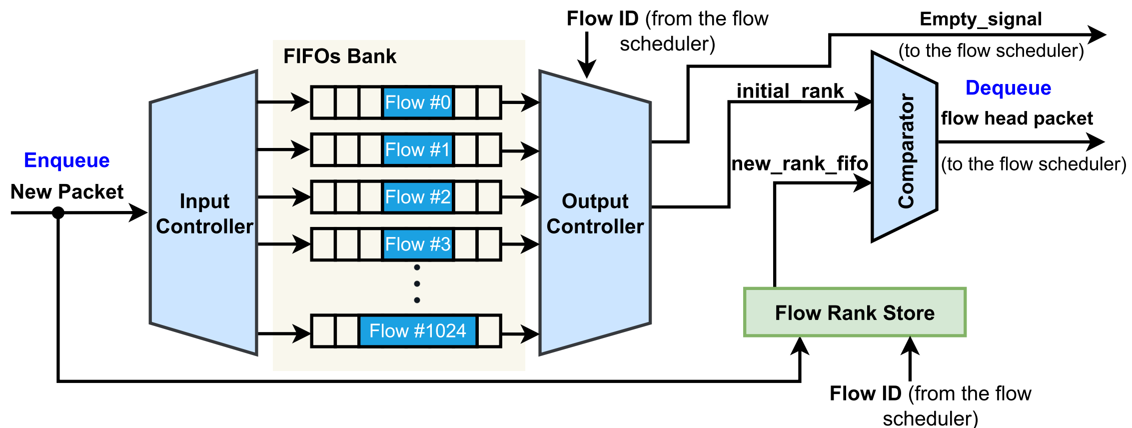

FIGURE 2. The hardware implementation of the packet rank store. All packets from a flow are stored in the FIFO queue dedicated to their flow except the current head packet of this flow.

scheduler employs a forced dequeue primitive to dequeue the appropriate packet from the flow in the next cycle.

Fig. 1 illustrates the case of expressing the pFabric scheduling algorithm [23] using the DR-PIFO packet scheduler. The illustration depicts four distinct flows, each with one head packet waiting in the PIFO queue within the flow scheduler and multiple packets stored in the FIFO queues within the packet rank store. The rank of a packet in the pFabric algorithm is determined by the remaining size of its flow, with the flow having the lowest packet rank being scheduled first [23]. The ranks of the flows are subject to change with the arrival of new packets. As a result, the ranks of newly arrived packets inside the flow rank store may differ from the ranks of their head packets in the flow scheduler. In the depicted example, the head packet with the minimum rank (10) from flow ID (2) is dequeued, although the actual minimum rank (2) belongs to flow ID (0), as shown in the array inside the flow rank store. This departure error is detected by the minimum rank selectors, which indicates that the head packet from flow ID (0) should have been dequeued instead. The PIFO queue in the flow scheduler cannot keep pace with new ranks and is unable to re-sort all head packets with every new packet arrival, leading to the error. To rectify this, the detected error causes an interruption in the future departure order of the PIFO queue, which forcibly dequeues the head packet from flow ID (0) in the next possible dequeue operation.

IV. HARDWARE DESIGN

In this section, a detailed discussion of the hardware design of the DR-PIFO architecture is provided. Our emphasis is placed on explaining the enqueue and dequeue operations applied to the packet rank store and flow scheduler units. We will also explain the handling of scheduling errors in the flow scheduler unit. Without loss of generality and for simplicity, we will focus on the case of a single logical PIFO queue. However, the hardware components required for implementing multiple logical queues are previously introduced in [3], and we added them to the proposed architecture with no modifications. Accordingly, the concepts described in this section can easily be extended to the case of multiple logical PIFO queues as in [3].

A. THE PACKET RANK STORE UNIT

Modern network switches are usually implemented with a shared buffer implementing a bank of FIFO queues. The shared buffer is used to store the arriving packets, before forwarding them to the output ports. There are typically two types of buffers used in data-center switches. Either on-chip shared RAM buffers, which are small (several MBs), or much larger external DRAM buffers [31]. Our discussion will focus on describing the logic controlling the packet rank store unit during the enqueue and the dequeue operations applied to packets.

1) ENQUEUE OPERATION

The packet rank store unit works as the primary stage for any packet enqueued into the scheduler. As depicted in Fig. 2, the packet rank store employs an input controller in the enqueue stage at the input of the bank of FIFOs. A flow ID is attached to each new packet, and the input controller block checks if this flow ID has, or not, a head packet in the flow scheduler (PIFO queue). If it does not have a detected head packet yet, the new packet will bypass the packet rank store and will be inserted directly into the flow scheduler. If the new packet is not the first packet in the current flow, it will then be inserted at the tail of the FIFO queue belonging to its flow ID. The logic of this controller also determines the correct FIFO queue to which a packet should be inserted. In addition, the input controller checks the occupancy of the FIFO queue, and it drops a packet if a FIFO's occupancy exceeds a certain configurable threshold. In parallel to this mechanism, the information of an enqueued packet is forwarded to the re-ranking unit, to be used in the re-ranking process, if deemed necessary.

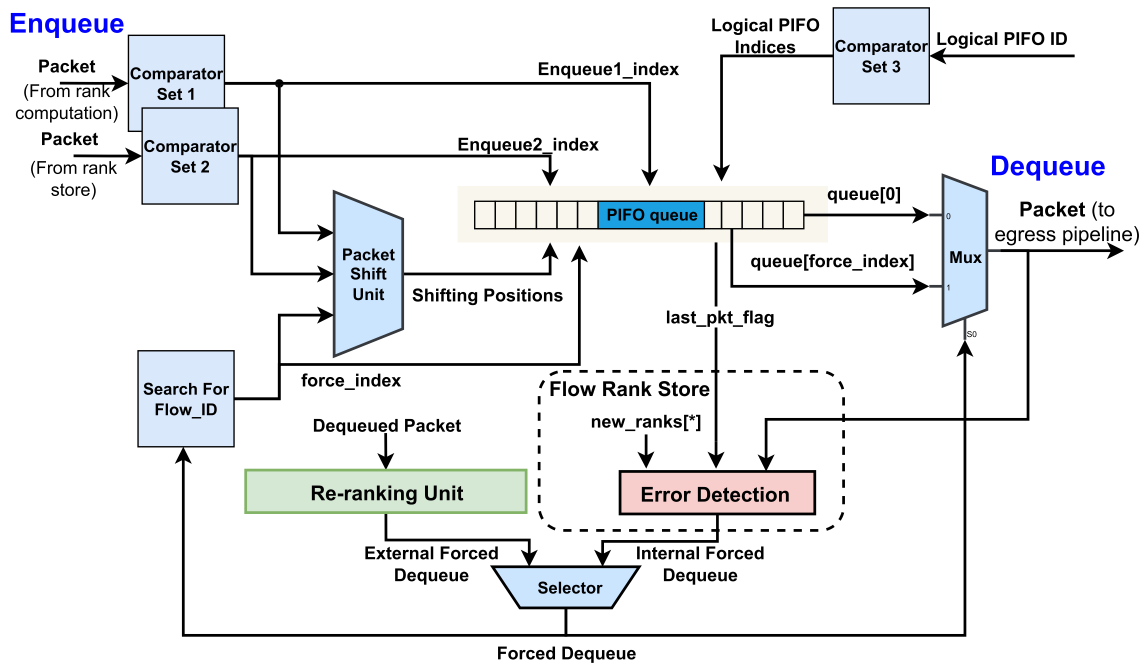

FIGURE 3. Hardware architecture of the flow scheduler. Due to pipelining, each component in this architecture is working all the time in each clock cycle.

2) DEQUEUE OPERATION

The dequeue operation is executed in two phases. During the first phase, as depicted in Fig. 2, the packet rank store employs a controller at the output of the FIFO bank. The output controller receives, from the flow scheduler, the Flow ID of the most recently dequeued packet, and uses it to find the corresponding FIFO queue. In other words, the output controller decodes the flow ID and matches it with its corresponding FIFO queue. Subsequently, the output controller extracts the packet from the head of that queue, if one exists, to propagate it to the second phase. If the FIFO queue is empty, the output controller sends a signal to the flow scheduler indicating that there are no packets waiting in the queue. The second phase is used whenever a rank update is needed, before sending the packet to the flow scheduler. This step works in conjunction with the flow rank store that sends the updated rank, "new rank fifo", of the Flow ID received from the flow scheduler. The comparator at the last output stage decides if the initial or the updated rank should be used as the output of the packet rank store. As in the original algorithm of the packet rank store in [1], this comparator transmits the updated rank from the flow rank store if it is not equal to zero, otherwise, the initial rank is transmitted instead.

B. THE FLOW SCHEDULER UNIT

The flow scheduling unit is responsible for the scheduling of the head-of-line packets belonging to different flows. The architecture of this unit is more complex than that of the rank store unit. At the core of the flow scheduler resides the PIFO queue, as can be observed in Fig. 3. At each clock cycle, the scheduler supports two enqueues and one dequeue operation. A flow's first arriving packet to the scheduler is handled by one of the enqueue operations. The other enqueue operation handles the next-in-line head packet of the recently dequeued flow. This packet is received from the packet rank store unit. The dequeue operation has two modes. In the normal operation mode, it dequeues the head element in the PIFO array whereas, in the forced dequeue mode it dequeues from an arbitrary position of the PIFO array, much like the "extract out" operation in the PIEO [2]. In addition, the scheduler detects whether an error has occurred in the departure order of the packets in which case, it automatically applies a forced dequeue operation for the correct flow in the next possible dequeue cycle. In our hardware implementation of the DR-PIFO, it takes four (4) clock cycles to detect and correct a packet ordering error with a forced dequeue. Finally, the PIFO array has indices from 0 to (the maximum number of flows – 1). Each head packet is sorted with one of these indices and thus, these indices are used to represent the relative order in the queue.

1) ENQUEUE OPERATION

As already mentioned, the flow scheduler executes two enqueue operations at a time. This is accomplished with the aid of a set of two comparator units, and the packet shift unit located at its input stage, which comprises a number of parallel comparators used in each comparator set unit that is equal to the number of supported flows. Thus, one comparator set is sufficient to compare the rank of the inserted packet against all the other head packets already stored in the

PIFO queue. The comparison produces a binary (0/1) result. This result is used in the first clock cycle of the enqueue operation to determine the first element with a rank higher than that of the newly arrived packet as depicted in Fig. 4(a). Subsequently, the newly arrived packet is inserted ahead of the first element with a higher rank that was found.

If the packet that arrives at the scheduler is the first one of a new flow, it is enqueued in the PIFO queue as the head packet of that flow. Since it is the first packet of a flow, it is arriving directly from the ingress pipeline and thus, it bypasses the packet rank store unit. After going through the Comparator Set 1, as depicted in Fig. 3, the appropriate index for the new packet is generated in the PIFO queue "Enqueue1 index" based on its rank, and the rank of the remaining packets waiting in the queue. Subsequently, the packet is placed in that index position of the queue.

The second enqueue operation takes place whenever a head packet from a certain Flow ID has just departed from the scheduler. As already discussed, the remaining packets of the flow for which the head packet was recently dequeued from the PIFO queue are kept in the packet rank store. Thus, the packet rank store unit provides the next packet of the flow to which the just departed packet belongs. This next packet is considered as the next lowest rank packet of its flow according to the observation stating that priorities of packets are monotonically decreasing through flows [3]. After going through the Comparator Set 2, as depicted in Fig. 3, the "Enqueue2 index" is generated, which places the packet into its appropriate position in the PIFO queue, based on its rank.

As depicted in Fig. 3, the packet shift unit has three indices as inputs, two of which belong to the two distinct enqueue operations, whereas the third input is coming from the dequeue operation. Although the index of the dequeued packet is usually 0, it could be any arbitrary index (up to the number of flows – 1), whenever a forced dequeue operation is executed. Thus, the packet shift unit does not assume a fixed forced dequeue index and it is responsible for determining the direction of the shift for each element in the PIFO queue based on the input indices. This is completed in the first clock cycle of the enqueue as depicted in Fig. 4(a). After the appropriate position of the current packet has been determined within the PIFO queue, the elements in the queue are shifted simultaneously in the following clock cycle. Following the completion of the shifting of the residing elements in the PIFO queue, in the appropriate direction, an unoccupied space will be left. This space will be used to enqueue new packets in their appropriate indices namely, "Enqueue1 index" and "Enqueue2 index".

2) DEQUEUE OPERATION

The dequeueing of the head element from the PIFO queue is the default dequeueing operation of the flow scheduler. We will refer to this action as the normal dequeue operation. A single physical PIFO can support multiple logical PIFOs. Thus, the desired logical PIFO ID must be compared against the logical PIFO IDs of all the head packets currently residing

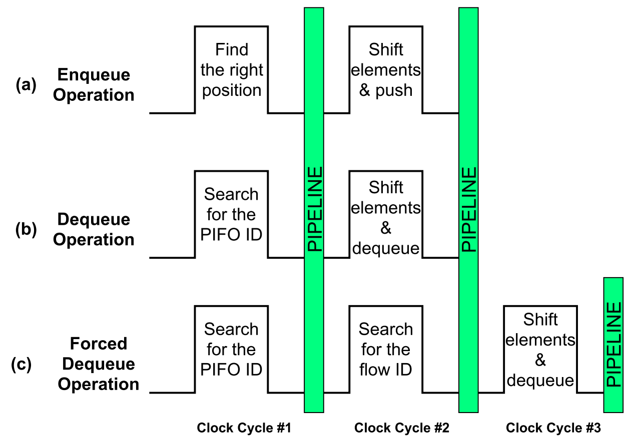

FIGURE 4. The flow scheduler operations in 2 & 3 pipeline stages. (a) Shows the Enqueue pipeline operation (same for "Enqueue1" and "Enqueue2"), afterwards (b) the normal dequeue is executed or (c) the forced dequeue is executed.

in the physical queue. The comparator set 3 in Fig. 3, provides the indices of the packets belonging to the desired logical PIFO queue from which they are dequeued. An index of zero is then sent to the packet shift unit, which also receives the indices of the input comparator units 1 and 2, for the two enqueue operations.

The value of the first element, indicated as queue[0] in Fig. 3, is one of the inputs of the multiplexer at the output stage of the scheduler. The second input to the multiplexer is the dequeued packet, used in the case of a forced dequeue and indicated as queue[force index] in Fig. 3. The select signal of this multiplexer is the "Forced Dequeue" flag signal which stays on until the head packet, corresponding to the forced Flow ID, is finally dequeued. The forced dequeue operation can be triggered either externally by the re-ranking unit or internally by the error detection unit, which sends a forced dequeue flag signal that corresponds to the Flow ID. Then, the position in the queue from which a packet should be dequeued, is determined using this received flow ID. The selector that generates the forced dequeue signal, checks if the forced dequeue is triggered internally and externally at the same time, then the external forced dequeue triggered by the re-ranking unit is applied, and the internal forced dequeue is discarded. Since the original PIFO can only dequeue elements from the head of the queue, the DR-PIFO employs some additional logic that adds complexity to the design to support the needed forced dequeue functionality.

The normal dequeue operation generates an output in two (2) clock cycles, without pipelining, as depicted in Fig. 4(b). It can be observed in Fig. 4(c) that the forced dequeue operation takes three (3) clock cycles. In the first clock cycle, the identification of the appropriate logical PIFO ID takes place using the Comparator Set 3 similar to the first step of the normal dequeue operation. In the second clock cycle, the scheduler finds the head packet of the desired flow. For this to be accomplished, the desired Flow ID is compared against all Flow IDs currently stored in the PIFO queue.

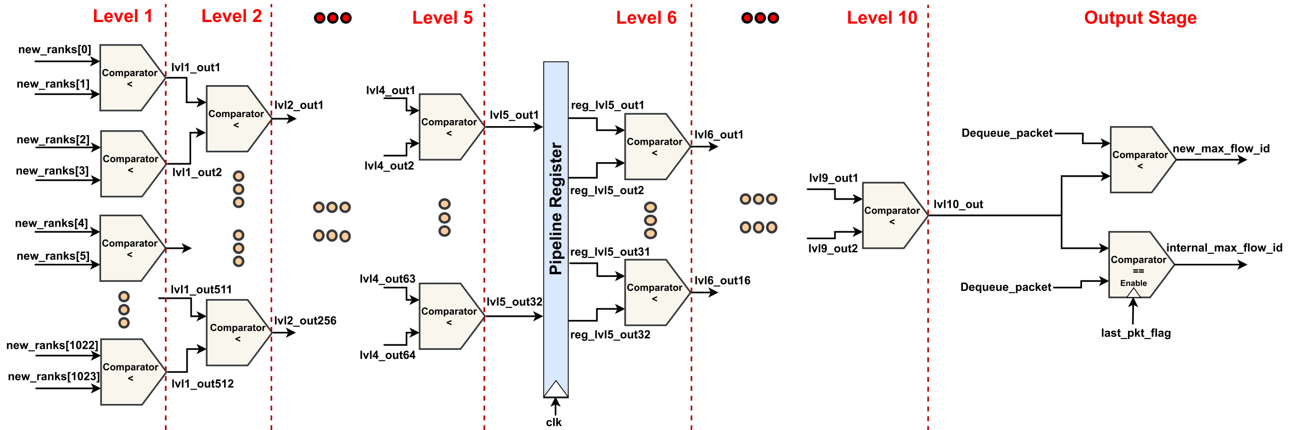

FIGURE 5. The hardware architecture for the error detection unit, showing an example of supporting 1024 flows. It also illustrates the number of cascaded comparators required to detect an error.

Parallel comparators are used to find the desired dequeue index. The number of parallel comparators used to find the desired dequeue index is equal to the number of supported flows. In the third clock cycle, the selected head packet is dequeued to the egress pipeline, and the forced dequeue index is sent back to the packet shift unit at the enqueue side of the scheduler.

Compared to the enqueue and normal dequeue operations, the forced dequeue operation takes an additional clock cycle to complete. This forces the enqueue operation to stall for one clock cycle while waiting for the forced dequeue operation to provide the forced index. The output from the multiplexer is also sent to the error detection block, to be used by the re-ranking unit to produce the "Force Dequeue" signal, whenever it is needed. This "Force Dequeue" signal is used by both, the "Search for Flow ID" block at the input and the multiplexer at the output of the scheduling unit. The error detection unit evaluates whether there is an error in the departure order of the output packets or not. The additional clock cycle needed by the forced dequeue operation introduces a latency of one clock cycle to the packet scheduler. However, we strongly believe that this added latency does not affect the total throughput of the DR-PIFO packet scheduler for two reasons. The first one is that the forced dequeue operation is not the common dequeue operation, but it is rarely executed as illustrated in section IV. The second reason is that, even in a non-pipelined manner, the non-pipelined DR-PIFO with its forced dequeue operation can enqueue and dequeue a packet in 5 clock cycles. This number of cycles is sufficient for the highest speed links today [3].

3) THE ERROR DETECTION UNIT

The flow scheduler is unable to rearrange the head packets of flows whose ranking has been altered within the PIFO queue. Neither the re-ranking unit nor any other control block is capable of monitoring all the packets stored within the queue. A simple approach to overcome this limitation would be to duplicate the flow scheduler components, but this would result in potential degradation of performance due to the added area overhead. Instead, our design features an error detection unit that enhances the PIFO queue's awareness of any changes to the rank of its hosted packets.

As illustrated in Figure 5, the error detection unit operates in two pipeline stages and produces an output at every clock cycle. It becomes activated whenever the flow scheduler performs its normal dequeue operation. The error detection unit leverages the unsorted "new ranks" array provided by the updated flow rank store and uses it to search for the minimum updated rank of all flows stored in the PIFO queue. If the minimum rank found is lower than the recently dequeued packet, the error detection unit detects an error in the departure order and initiates a forced dequeue operation. The "last pkt flag" produced at the output stage indicates whether the dequeued packet is the last one from its flow. This flag changes in response to the "Empty signal" from the packet rank store, as depicted in Figure 2. If the "last pkt flag" is set, the error detection unit searches for another flow ID with the same rank, and when found, schedules its head packet for the next dequeue cycle via an internal forced dequeue operation.

Assuming that N flows are supported by the scheduler, if one was to implement the classical compare and shift architecture to find the minimum rank of all flows, that design would incur a critical path delay of $N\times$ delay_of_comparator_block. Thus, as N becomes large, it would be difficult to achieve the desired frequency of 1GHz with this naive approach. To overcome this issue as N may scale to a large number, we propose the use of multistage comparators to determine the minimum rank of all flows. Our proposed architecture is depicted in Fig. 5. As an example, to find the minimum value out of N values, the first stage consists of $N/2$ comparators operating in parallel and generating $N/2$ output values. Subsequently, these $N/2$ values are forwarded to the second stage where $N/4$ comparators operate in parallel. The results

from the second stage propagate to the third stage and so on and so forth. Using this technique to compare the ranks can theoretically reduce the critical path delay to $\log_2 N \times$ delay_of_comparator_block plus the delay of some additional internal logic blocks. Our results show that the delay of this critical path can be split over two clock cycles in a pipelined architecture, as illustrated in Fig. 5.

The implementation reported in the present paper supports up to 1024 flows. Accordingly, the error detection unit must find the minimum rank value among the 1024 rank values in the "new ranks" array. In the first level, labeled "level1," 512 comparators compare the ranks of the odd indices of the "new ranks" array against those of even indices, with the index at this level corresponding to the flow ID. The results, including a rank and a flow ID, are then propagated from one level to the next one until they reach the pipeline register. The outputs from "level5" are registered before being transmitted to the next level, "level6." The final result is collected from level 10 ($2^{10} = 1024$). This final result is then compared with the values from the recently dequeued packet. The pipeline register reduces the number of cascaded comparators triggered at each clock cycle from 10 to 5, which reduces the critical delay path and enables our design to meet the timing constraints of the 1GHz clock cycle, even for as many as 1024 flows.

V. RESULTS

In this section, the hardware implementation of the DR-PIFO is analyzed and compared to the PIFO-based scheduler from [3]. In addition, the expressiveness of the DR-PIFO architecture is evaluated and compared to the algorithmic models of the DR-PIFO, PIFO-based scheduler, and PIEO scheduler from [1]. The experiments conducted in this work are similar to those in [1] for a fair comparison.

A. HARDWARE PERFORMANCE

The hardware implementation of the DR-PIFO scheduler can service up to 1024 flows, each having 16-bit rank values. It can store up to 64k packets in its buffer. These specifications match the ones found in [3], and they target switches with shallow buffer sizes. An example of such a switch is Broadcom's Trident II, which has a buffer size of 12Mbytes.

The DR-PIFO scheduler was developed using SystemVerilog, a widely used hardware description and verification language, and synthesized using the 65nm TSMC CMOS technology and Synopsys Design Compiler. The validity and performance of the resulting scheduler, along with the flow and packet rank stores, were verified through post-synthesis simulations. The re-ranking unit was not synthesized, as it was built as a fixed function for all algorithms in our experiments (as described in [1], but it is recommended to implement this unit in a programmable platform to allow for the implementation of new algorithms or modifications without requiring a new ASIC chip. The performance of the DR-PIFO was compared against the original PIFO-based scheduler, which has the capability to process one enqueue and one dequeue operation per packet every 1ns, which is the optimum throughput as noted in [3]. Since the results of the PIFO were obtained with a 16nm technology, it was re-synthesized using the same 65nm technology to ensure a fair comparison between the two schedulers.

TABLE 1. DR-PIFO vs PIFO hardware performance (65nm).

Performance	DR-PIFO	PIFO
Frequency (GHz)	1.00	1.00
Area (mm^2)	15.07	13.05
Supported Flows	1024	1024

As shown in Table 1, the DR-PIFO operates at the same clock frequency of 1GHz as the PIFO scheduler. This throughput of 1 packet enqueue and dequeue every 1ns is considered an optimal value, as a lower throughput may affect the total performance of modern switches [3]. The hardware logic added by the DR-PIFO, such as the flow rank store and forced dequeue support, did not hinder its maximum operating frequency. For up to 1024 flows, both the DR-PIFO and PIFO designs are guaranteed to meet the 1ns timing constraint at the 65nm technology node. To accommodate more flows, either a more advanced technology node or more efficient design optimization is necessary. The DR-PIFO's post-synthesis area includes the packet rank store, flow rank store, and flow scheduler and is only 15.5% larger than the PIFO design. This area comparison does not take into account the programmable components such as the rank computation and re-ranking units.

The impact of the programmable re-ranking unit's area overhead is not taken into account. However, the fixed scheduling components such as the flow scheduler and the packet rank store are found to be consuming three times more area than the programmable components in the PIFO scheduler as reported in [3]. The rank computation unit's area has already been calculated in [3]. However, it cannot be assumed that the re-ranking unit would consume the same amount of area. To date, no scheduling algorithm has been found to require more processing in the re-ranking unit than in the rank computation unit, however, it may be necessary for the re-ranking unit to operate with lower latency. As a result, the hardware requirements and area consumption of the re-ranking unit are topics for future research.

B. EVALUATION

To evaluate the expressiveness of our proposed hardware design, we utilized two scheduling algorithms, the pFabric and the VDS. These algorithms were chosen for their distinct differences and intrinsic importance. The pFabric assigns priority to flows based on the arrival of new packets, while the VDS changes priority based on both the arrival and departure of packets [1]. We evaluated the DR-PIFO, PIFO, and PIEO using the Bandwidth Utilization (BU) and Flow

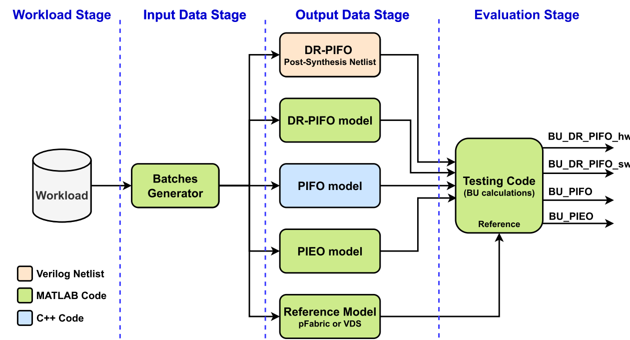

FIGURE 6. Adopted experimental setup showing the implemented testing stages in our experiments.

Completion Time (FCT) metrics. The results for the proposed hardware design of the DR-PIFO were obtained from Verilog netlist simulations targeting the 65nm TSMC technology node, while the results for the algorithmic DR-PIFO, PIFO, and PIEO schedulers were obtained from functional software models from the work in [1]. However, bug-free and portable hardware codes for PIFO and PIEO designs were not accessible, which forced us to re-implementing them as functional software models from the original papers' functional description.

1) EXPERIMENTAL SETUP

In our experimental setup, as depicted in Fig.6, it is crucial to specify for each experiment the workload and the input data, from which the output data is obtained and processed through an evaluation stage. Our experiments start with generating a unique workload for each of the two adopted case studies. The workloads used in our experiments are based on previous research in the literature, with detailed information provided further in the section. Each workload consists of a number of flows, with a specific number of packets per flow, packet length, order of packets at the sending point, and parameters for calculating priorities. The objective is to transmit all packets from the sending point to the receiving point. As we are testing packet schedulers, the sending point serves as the input side of the scheduler, and the receiving point is the output side. The specific information describing each workload is clearly stated in each individual case study.

The organization of the data generated by the workload is an essential aspect of the testing process. The input data stage is responsible for organizing this data to be passed to the scheduler under test. The process begins with the batches generator, which splits the packets in the workload into batches. Each batch consists of a random number of packets from different flows, which are stored and sorted together in the scheduler. To avoid overflow, the number of packets from each flow must not exceed the total available capacity for that flow. At least one packet from each non-empty flow is included in each batch.

Once the packets have been organized into batches, they are passed to the scheduler, one by one, at each positive edge of the clock cycle. The packets are sorted in ascending order based on their flow ID. The scheduler waits until it has received and stored all the packets in the current batch before dequeueing them, one by one, based on their priorities and the scheduling algorithm. There is no overlap between batches, and the next batch is not passed until all the packets from the current batch have been dequeued.

In this test setup, the scheduler can be in one of two states: enqueue or dequeue, with no overlap between them. The test bench uses communicating flags to synchronize between the input stage (batches generator) and the scheduler under test. The number of batches generated for each case study is stated in the case studies.

The output data stage is the third step in the testing process. It monitors and reports the packets transmitted from the single output port of the scheduler under test. This stage generates a report that showcases the sequence of packets dequeued by the packet scheduler. This data is collected during the dequeue state of the scheduler. During the enqueue state, the output port is idle as there is no dequeueing of packets. This output data is not modified until it is delivered to the next stage which is the evaluation stage. The assumption is made that the total available bandwidth at the output port of the scheduler is always higher than the combined bandwidth required to send each flow in the current batch. This assumption is made for two reasons, the first one is that the two adopted case studies, which sufficiently assess and challenge the new features of the DR-PIFO design, are work-conserving algorithms. Therefore, they provide neither a rate-limiting scheme to avoid severe congestion nor a specific dropping scheme to manage packet losses. The second reason is to avoid congestion collapse at the output port and the dropping of packets since the packet loss could affect the evaluation of the schedulers-under-test in expressing the implemented scheduling algorithms. Accordingly, the elimination of the packet loss provides a fair comparison for all the tested packet schedulers. Thus, the present work mainly evaluates the accuracy of expressing different scheduling policies and does not cover packet-dropping policies. Nevertheless, all the schedulers under-test apply the same Tail-Dropping technique, if needed.

The final stage of our experimental setup is the evaluation phase, which involves the use of a testing code to determine the Bandwidth Utilization (BU) for each scheduler being tested. This code calculates the bandwidth for each flow during each batch period using the output data from the schedulers, and then compares these bandwidths to the ideal bandwidths assigned by the reference model. The reference models, implemented in MATLAB, are designed to simulate the respective ideal behavior of the scheduling algorithms (pFabric or VDS). To ensure fairness in the comparison, the testing code assumes that all schedulers have equal processing capacity (1 enqueue and 1 dequeue per clock cycle).

TABLE 2. Bandwidth utilization (BU) compared to a nominal (ideal) pFabric.

BU	DR-PIFO HW*		DR-PIFO SW**		PIFO***		PIEO**	
	Web search	Data mining	Web search	Data mining	Web search	Data mining	Web search	Data mining
Average	1.0127	1.0012	1.0012	1.0001	1.1076	1.009	0.9993	1.0006
Standard Deviation ±	0.02	0.02	0.0045	0.002	0.42	0.23	0.2	0.13
Minimum (BU < 1)	0.84	0.43	0.95	0.93	0.05	0.03	0.13	0.04
Maximum (BU > 1)	4.77	7	1.037	1	11.2	2.05	62	21

*Results from simulation after synthesis, 65nm TSMC.
**Results from MATLAB simulations.
***Results from PIFO C++ model.

2) pFabric CASE STUDY

In this study, we utilized two workloads collected from real-world data centers [23]. The first workload, the web search, is characterized by its heavy-tailed distribution, where 30% of flows account for over 95% of all sent bytes. The flow sizes range from 1 MB to 20 MB. The second workload, collected from data centers performing data mining jobs, is more skewed, with only 3.6% of flows accounting for 95% of all bytes. Most of the flows, around 80%, are smaller than 10KB. The web search workload, being more challenging due to its characteristics, can result in starvation of certain flows [23]. Thus, our primary focus in this case study will be on the results obtained when using the challenging web search workload.

In our experiment, we simulated 100 unique flows, generating a total of over 109k packets for the web search workload and over 631k packets for the data mining workload. The number of flows is carefully selected to strike a balance between simulation time and testing the DR-PIFO Verilog netlist. Previous research [1] has revealed that increasing the number of flows degrades the performance of PIFO and PIEO, but has no impact on the performance of the DR-PIFO model. The size of each packet is the standard Ethernet MTU of 1, 500 bytes. The number of batches is randomly generated and includes 844 batches for the web search workload and approximately 37k batches for the data mining workload throughout the simulation time. We use the bandwidth utilization (BU) factor, defined in equation (1), to evaluate the proposed hardware architecture of the DR-PIFO scheduler, as previously mentioned in [1]. An ideal value of 1 for the BU represents 100% bandwidth utilization, and any reduction from this ideal value is considered bandwidth (BW) loss, while any increase is considered an erroneous BW gain for the given flow.

$$BU = \frac{BW_{nominal_pFabric}}{BW_{actual_scheduler}} \tag{1}$$

The BU metric measures the precision of the packet schedulers in realizing the desired scheduling algorithm during each batch of the workload, yet it fails to illustrate its impact on end-to-end performance. Hence, we also resort to the Flow Completion Time (FCT), a widely used metric for evaluating scheduling policies' impact on end-to-end delays. To assess the accuracy of scheduling policy realization, we use the relative error in FCT values as our evaluation metric. The relative errors are computed by comparing the FCT values obtained by the scheduler under evaluation with those of the ideal scheduling algorithm, either pFabric or VDS. The relative error in the FCT values is defined in equation (2).

$$FCT_error = \frac{|FCT_{scheduler} - FCT_{nominal_pFabric}|}{FCT_{nominal_pFabric}} \tag{2}$$

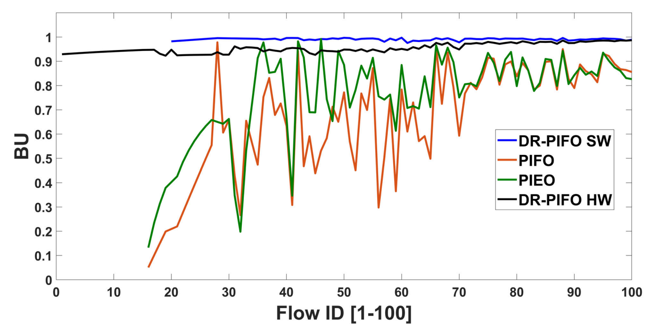

FIGURE 7. Bandwidth Utilization (BU) for the pFabric, for 100 flows and from four schedulers; the DR-PIFO software version, the PIFO, the PIEO [1] and the proposed DR-PIFO hardware version.

As can be observed in Table 2, the DR-PIFO hardware design achieved an average BU close to 1, with the lowest standard deviation for the two tested workloads. The standard deviation is a crucial metric for evaluating models, as it measures the dispersion of BU values from the average. Additionally, for the web search workload, DR-PIFO obtained the optimum values for minimum and maximum BU for a flow. Thus, its average BU is comparable to its algorithmic model and is closer to the ideal pFabric than the results from the PIFO and PIEO software models.

As seen in Fig. 7, both the algorithmic model and hardware implementation of the DR-PIFO scheduler produced a constant BU for all starved flows (BU < 1), unlike the results from the PIFO and PIEO. The impact of these BU

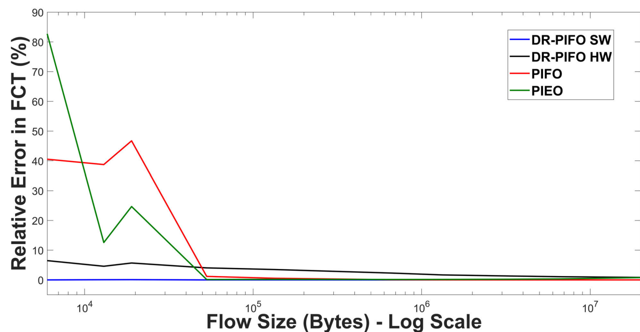

FIGURE 8. The relative errors in the Flow Completion Time (FCT) for different flow sizes while implementing the pFabric scheduling algorithm with the web search workload.

values on end-to-end performance is reflected in the FCT errors displayed in Fig. 8, where DR-PIFO still achieved much lower FCT errors compared to PIFO and PIEO. Only the algorithmic model and hardware implementation of DR-PIFO were able to consistently produce low FCT errors for all flows.

The forced dequeue primitive is executed 908 times during the simulation of over 109k dequeued packets, constituting roughly 0.83% of all dequeues. This figure is lower in the hardware design due to the implementation of stricter triggering criteria for the forced dequeue operation. The operation requires an extra clock cycle to complete in the hardware design, and therefore, to mitigate its impact on the BU results, it should be triggered judiciously.

The disparities in the BU values between the algorithmic model and the hardware implementation of the DR-PIFO stem from the hardware constraints incorporated into our design. Our hardware implementation is limited by a 1 GHz operating frequency, and the forced dequeue primitive contributes an additional clock cycle latency that impacts the BU factor of the relevant flows. Furthermore, the error detection unit operates sequentially, requiring an extra clock cycle to detect departure order errors, which were omitted from the algorithmic model of the DR-PIFO.

3) VDS CASE STUDY

The DR-PIFO hardware implementation was evaluated using the VDS scheduling algorithm. Our experiment, conducted in the same manner as described in [1], simulated six flows with a total of 95.3k packets over a 100-second period. The window constraints parameters of (m, k) were $(3, 5)$, $(5, 7)$, $(7, 9)$, $(9, 11)$, and $(13, 15)$, one pair for each of the six flows, and were used to determine the priority of each flow during simulation. The priority computation steps of VDS are outlined in [1], [24], and [30]. The six flows had various arrival rates as follows; 333kbps, 200kbps, 143kbps, 110kbps, 90kbps, and 77kbps, respectively. The simulation is divided into 804 batches, and the number of batches is randomly chosen with different randomized sizes. The comparison was performed between DR-PIFO and PIEO

TABLE 3. Bandwidth utilization for different flows in the VDS Case study.

Bandwidth Utilization	DR-PIFO HW*	DR-PIFO SW** [1]	PIEO** [1]
Average	1.008	1.0025	0.9951
Standard Deviation $\pm$	0.02	0.0086	0.04
Minimum (BU $<$ 1)	0.87	0.91	0.63
Maximum (BU $>$ 1)	1.3529	1.1455	1.3235

*Results from simulation after synthesis, 65nm TSMC.
**Results from MATLAB simulations.

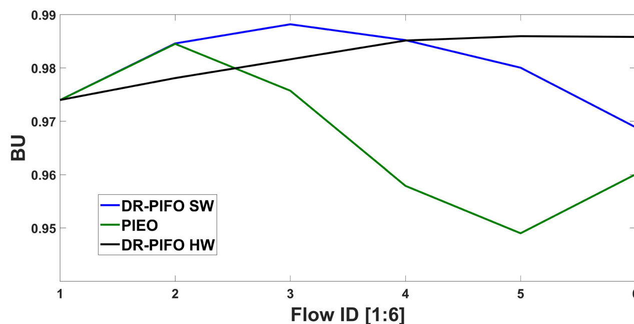

FIGURE 9. Bandwidth Utilization (BU) for the VDS algorithm for 6 flows and from three schedulers; the software version of the DR-PIFO, the PIEO [1] and the proposed hardware version of the DR-PIFO.

schedulers, as the PIFO scheduler does not support the VDS algorithm. The calculation of BU was performed using Equation (1).

During 95.3k dequeueing cycles, the forced dequeue operation was executed 564 times, constituting about 0.6% of the total dequeueing time. The values in Table 3 demonstrate that the hardware implementation of DR-PIFO has an average buffer utilization (BU) of 1.008 with a standard deviation of 0.02. This result is consistent with the algorithmic model. However, there is a 13% maximum loss in BU in the hardware implementation compared to the 9% loss in the algorithmic model, in the worst scenario. Despite this, the hardware design still surpasses the BU values observed with PIEO, except for the maximum BU, even with the hardware limitations. Although the maximum BU of DR-PIFO is slightly higher than PIEO, they both remain close to 1, especially when compared to pFabric. Fig. 9 presents the BU values for the suffered flows with BU $<$ 1 for all 3 schedulers, while Fig. 10 demonstrates the relative errors of FCT values for the end-to-end performance assessment. As it can be observed, the differences in the BU values caused some relative errors in the FCT values.

It should be noted that the maximum throughput achieved by the PIEO is significantly lower, 4 times less than the throughput of the DR-PIFO and PIFO, as per [2]. However, in the two case studies conducted, it was assumed that the PIEO could operate at the same throughput as the DR-PIFO

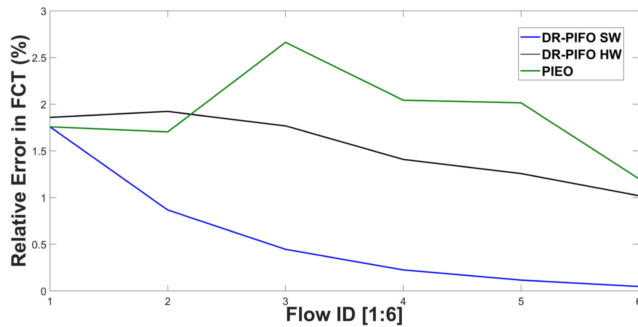

FIGURE 10. The relative errors in the Flow Completion Time (FCT) for 6 flows while implementing the VDS scheduling algorithm in each model.

and PIFO. Without this assumption, the BU values of the PIEO might decline due to its lower throughput capacity. The results of the case studies provide robust support for the theory presented in [1], which asserts that the DR-PIFO is an algorithm-agnostic model.

VI. DISCUSSION

Despite advances in programmable network switches, such as Intel's Tofino series [32], the traffic manager unit still operates as fixed-function logic. The scheduling algorithms integrated into the hardware during the fabrication of network equipment are widely used but limited. The scheduler in the traffic manager is configurable rather than programmable, limiting network operators to modifying specific parameters and being unable to alter or create new scheduling algorithms. To meet Service Level Agreements (SLAs), the operators must configure the priority levels and flow transmission rate. However, there has been a recent surge in research aimed at increasing the programmability of traffic managers. New hardware abstractions and data structures, such as the PIFO and PIEO schedulers, have been proposed to expand the range of scheduling algorithms. Although these abstractions offer greater expressiveness, they are still not fully programmable. The DR-PIFO improves upon the PIFO design with dynamic re-ranking and forced dequeue on packets, as discussed in detail in [1].

1) PERFORMANCE OF DR-PIFO'S HARDWARE IMPLEMENTATION

The DR-PIFO hardware implementation achieves a 1GHz operating frequency and a 1 packet per 1ns throughput, thanks to its pipelined design. To meet these timing constraints, various hardware micro-architecture designs were introduced in section IV, providing line-rate performance while allowing dynamic re-ranking and forced dequeueing of packets from any position in the queue. While the DR-PIFO introduces more programmability, it also incurs a 15.5% area overhead compared to state-of-the-art PIFO packet scheduler hardware designs [3]. In today's switches, a fixed packet scheduler is implemented in the chip area without providing any programmability to the network operators. Thus, for today's switches, the DR-PIFO packet scheduler, as a replacement for the fixed scheduler, offers outstanding expressiveness and programmability for a wide class of scheduling algorithms at the cost of its area overhead while maintaining the line rate processing. Since no optimization efforts were made to reduce this overhead, it remains a potential area for future work.

2) DESIGN SCALABILITY

The architecture of the DR-PIFO design inherits scalability limitation from the original PIFO. The DR-PIFO design scales proportionally to the number of supported flows since the proposed design requires $O(N)$ flip-flops and comparators to support N number of flows. Thus, the proposed DR-PIFO supports around 1k flows without violating the desired timing constraints. However, there are some practical techniques to increase the number of supported flows. For example, the network traffic from multiple sources can be grouped into aggregate flows [33]. Accordingly, the maximum number of grouped flows is 64k which is the maximum number of packets that the DR-PIFO can store. Moreover, parallel DR-PIFO packet schedulers can be separately implemented, one dedicated DR-PIFO for each output port, whereas today's switch provides up to 64 output ports. In addition, technologies more advanced and much faster than 65nm CMOS could be used.

With the advent of 5G and the increasing demand for wireless-wireline convergence, the need for servicing hundreds of thousands, or even millions of flows at once is expected to grow. Previous best-known designs that increased the number of supported flows compared to the PIFO can be found in [2] and [20]. Due to the possible starvation of flows in the original PIFO design, parallel processing cannot be adopted for multiple PIFO queues. On the other hand, this starvation is not possible in the DR-PIFO design as illustrated in section IV. Thus, the DR-PIFO is poised to be a key solution for addressing this challenge, making traffic managers more programmable and scalable. The focus of our current research efforts is on improving the scalability of a hierarchical version of the DR-PIFO and introducing a software abstraction compatible with the P4 language.

3) DR-PIFO LIMITATIONS

The DR-PIFO scheduler incorporates an operation to determine the lowest rank and detect scheduling errors in packet ordering. However, as the number of flows grows beyond a certain point, implementing this operation while meeting timing constraints is very challenging. While finding the minimum value of an unsorted array is a well-known issue, there have been various techniques proposed to enhance its performance. Future work will investigate micro-architectural improvements to address this scalability limitation and align with current and future traffic demands. Moreover, it is crucial to keep the rank updating function as straightforward as possible, as a complex mechanism

will negatively impact the performance of the DR-PIFO scheduler.

VII. CONCLUSION

In a recent work, we introduced the algorithmic model of the Dynamic Ranking Push-In-First-Out (DR-PIFO), an expressive programmable packet scheduler. In this paper, we presented its hardware design and implementation developed using TSMC's 65nm CMOS technology. The DR-PIFO sorts packets based on their priority, which can be updated while they are stored in the queues. The architecture also allows packet dequeueing from any position and can detect errors in their departure order. Our proposed hardware implementation can enqueue and dequeue packets each 1 ns, while handling up to 1024 flows. This performance makes it suitable for deployment in modern programmable network switches. The DR-PIFO adds a mere 15.5% overhead in chip area compared to a previous packet scheduler, which is acceptable given its added flexibility and programmability. Finally, we assess the proposed hardware implementation by implementing two vital scheduling algorithms, the pFabric and the VDS. It was found that the hardware implementation of the DR-PIFO is as flexible as its algorithmic model and that in terms of bandwidth utilization, it outperforms well-known schedulers such as the PIFO and the PIEO. In addition, it achieves much lower relative flow completion time (FCT) errors for different flow sizes when implementing the pFabric scheduling algorithm with the web search workload. Following this work, we are currently investigating the implementation of a hierarchical DR-PIFO architecture. In a future work, the expressiveness and the performance characteristics of the DR-PIFO scheduler will be validated and evaluated in FPGA-based network devices. Ultimately, language constructs that will support the implementation of the DR-PIFO scheduler could be introduced to the P4 language and deployed in programmable hardware devices. This will pave the way forward for P4 programmers to implement a diverse set of scheduling algorithms.

REFERENCES

[1] M. Elbediwy, B. Pontikakis, A. Ghaffari, J.-P. David, and Y. Savaria, "DR-PIFO: A dynamic ranking packet scheduler using push-in-first-out queue," *IEEE Trans. Netw. Service Manag.*, Aug. 2022.

[2] V. Shrivastav, "Fast, scalable, and programmable packet scheduler in hardware," in *Proc. ACM Special Interest Group Data Commun.*, Aug. 2019, pp. 367–379.

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

[4] M. T. Arashloo, M. Ghobadi, J. Rexford, and D. Walker, "HotCocoa: Hardware congestion control abstractions," in *Proc. 16th ACM Workshop Hot Topics Netw.*, Nov. 2017, pp. 108–114.

[5] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, and E. Chung, "Azure accelerated networking: Smartnics in the public cloud," in *Proc. 15th USENIX Symp. Networked Syst. Design Implement.*, 2018, pp. 51–66.

[6] K. G. Shin, J. Rexford, and S.-W. Moon, "Scalable hardware priority queue architectures for high-speed packet switches," *IEEE Trans. Comput.*, vol. 49, no. 11, pp. 1215–1227, Nov. 2000, doi: 10.1109/12.895938.

[7] A. Gupta and R. K. Jha, "A survey of 5G network: Architecture and emerging technologies," *IEEE Access*, vol. 3, pp. 1206–1232, 2015.

[8] H. T. Hong, Q. B. Xuan, D. D. Van, N. P. Ngoc, and T. N. Huu, "Hardware-efficient implementation of WFQ algorithm on NetFPGA-based OpenFlow switch," in *Proc. Int. Conf. Adv. Technol. Commun. (ATC)*, Oct. 2016, pp. 431–436.

[9] J. Naous, D. Erickson, G. A. Covington, G. Appenzeller, and N. McKeown, "Implementing an OpenFlow switch on the NetFPGA platform," in *Proc. 4th ACM/IEEE Symp. Architectures Netw. Commun. Syst.*, Nov. 2008, pp. 1–9.

[10] K. McLaughlin, S. Sezer, H. Blume, X. Yang, F. Kupzog, and T. Noll, "A scalable packet sorting circuit for high-speed WFQ packet scheduling," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 16, no. 7, pp. 781–791, Jul. 2008.

[11] N. K. Sharma, M. Liu, K. Atreya, and A. Krishnamurthy, "Approximating fair queueing on reconfigurable switches," in *Proc. 15th USENIX Symp. Networked Syst. Design Implement.*, 2018, pp. 1–16.

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

[13] I. Benacer, F. Boyer, and Y. Savaria, "Design of a low latency 40 Gb/s flow-based traffic manager using high-level synthesis," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2018, pp. 1–5.

[14] I. Benacer, F. Boyer, and Y. Savaria, "A high-speed, scalable, and programmable traffic manager architecture for flow-based networking," *IEEE Access*, vol. 7, pp. 2231–2243, 2019.

[15] A. Sivaraman, S. Subramanian, A. Agrawal, S. Chole, S.-T. Chuang, T. Edsall, M. Alizadeh, S. Katti, N. McKeown, and H. Balakrishnan, "Towards programmable packet scheduling," in *Proc. 14th ACM Workshop Hot Topics Netw.*, Nov. 2015, pp. 1–7.

[16] N. K. Sharma, C. Zhao, M. Liu, P. G. Kannan, C. Kim, A. Krishnamurthy, and A. Sivaraman, "Programmable calendar queues for high-speed packet scheduling," in *Proc. 17th USENIX Symp. Networked Syst. Design Implement.*, 2020, pp. 685–699.

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

[18] Z. Yu, C. Hu, J. Wu, X. Sun, V. Braverman, M. Chowdhury, Z. Liu, and X. Jin, "Programmable packet scheduling with a single queue," in *Proc. ACM SIGCOMM Conf.*, Aug. 2021, pp. 179–193.

[19] M.-Y. Zhu, K.-F. Chen, Z. Chen, and N. Lv, "FAIFO: UAV-assisted IoT programmable packet scheduling considering freshness," *Ad Hoc Netw.*, vol. 134, Sep. 2022, Art. no. 102912.

[20] A. G. Alcoz, A. Dietmuller, and L. Vanbever, "SP-PIFO: Approximating push-in first-out behaviors using strict-priority queues," in *Proc. 17th USENIX Symp. Networked Syst. Design Implement.*, 2020, pp. 59–76.

[21] B. Vass, C. Sarkadi, and G. Rétvári, "Programmable packet scheduling with SP-PIFO: Theory, algorithms and evaluation," in *Proc. IEEE INFOCOM Conf. Comput. Commun. Workshops (INFOCOM WKSHPS)*, May 2022, pp. 1–6.

[22] Y. Jiang, C.-K. Tham, and C.-C. Ko, "A probabilistic priority scheduling discipline for high speed networks," in *Proc. IEEE Workshop High Perform. Switching Routing*, May 2001, pp. 1–5.

[23] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pFabric: Minimal near-optimal datacenter transport," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 435–446, Aug. 2013.

[24] Y. Zhang, R. West, and X. Qi, "A virtual deadline scheduler for window-constrained service guarantees," in *Proc. 25th IEEE Int. Real-Time Syst. Symp.*, Dec. 2004, pp. 151–160.

[25] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, "Better never than late: Meeting deadlines in datacenter networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 50–61, Aug. 2011.

[26] C. DeSanti. (2011). *802.1QBB—Priority-Based Flow Control*. [Online]. Available: http://www.ieee802.org/1/pages/802.1bb.html

[27] S.-T. Chuang, A. Goel, N. McKeown, and B. Prabhakar, "Matching output queueing with a combined input/output-queued switch," *IEEE J. Sel. Areas Commun.*, vol. 17, no. 6, pp. 1030–1039, Jun. 1999.

[28] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking, "Packet transactions: High-level programming for line-rate switches," in *Proc. ACM SIGCOMM Conf.*, Aug. 2016, pp. 15–28.

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

[30] Y. Zhang and R. West, "End-to-end window-constrained scheduling for real-time communication," in *Proc. 10th Int. Conf. Real-Time Embedded Comput. Syst. Appl.*, 2004.

[31] A. Bechtolsheim, L. Dale, H. Holbrook, and A. Li, "Why big data needs big buffer switches," Arista, Santa Clara, CA, USA, White Paper, 2016, vol. 10. [Online]. Available: https://www.arista.com/assets/data/pdf/Whitepapers/BigDataBigBuffers-WP.pdf

[32] Intel. (2021). *Intel Programmable Ethernet Switch Products*. [Online]. Available: https://www.intel.ca/content/www/ca/en/products/network-io/programmable-ethernet-switch.html

[33] P. Valente, "Providing near-optimal fair-queueing guarantees at round-robin amortized cost," in *Proc. 22nd Int. Conf. Comput. Commun. Netw. (ICCCN)*, Jul. 2013, pp. 1–7.

MOSTAFA ELBEDIWY received the B.Sc. degree in nanotechnology and nanoelectronics engineering from the Zewail City of Science and Technology, Egypt, in 2018. He is currently pursuing the Ph.D. degree with the Electrical Engineering Department, Polytechnique Montréal. His Ph.D. research focuses on the design of a hierarchical, scalable and programmable packet scheduler that is able to operates at line rate, and efficient enough to be implemented in today's programmable software-defined networks. His research interests include network traffic management, software-defined networking, computer architecture, digital systems design, high-level synthesis, and reconfigurable computing.

BILL PONTIKAKIS received the B.Eng. degree in computer engineering and the M.A.Sc. degree in electrical and computer engineering from Concordia University, Montreal, QC, Canada, in 2002 and 2003, respectively, and the Ph.D. degree in electrical engineering from Polytechnique Montréal, QC, Canada, which he was about to complete when he terminated it just before submitting his final thesis. Currently, he is a Research Associate with the Canada NSERC Industrial Research Chair for High-Speed and Programmable Packet Processors, Department of Electrical Engineering, Polytechnique Montréal. His contributions to the research chair are vital to its success, as he actively participates in the scientific, operational, budgeting, public relations, and research vision aspects of the chair. His current research interests include programmable network data planes and P4 language. He presented his first paper in the IEEE ISCAS Conference during his undergraduate studies, which earned him the ReSMIQ Undergraduate Research Award in Microelectronics in Montreal.

JEAN-PIERRE DAVID (Member, IEEE) received the B.Ing. degree in electrical engineering from Université de Liège, Liège, Belgium, in 1995, and the Ph.D. degree from Université Catholique de Louvain, Louvain-la-Neuve, Belgium, in 2002. He has been an Assistant Professor with Université de Montréal, QC, Canada, in 2002, and moved to Polytechnique Montréal, QC, Canada, in 2006, where he is currently a Professor with the Department of Electrical Engineering. His research interests include digital system design, reconfigurable computing, high-level synthesis, high speed communications, and neural networks and their applications.

YVON SAVARIA (Fellow, IEEE) received the B.Ing. and M.Sc.A. degrees in electrical engineering from École Polytechnique Montreal, in 1980 and 1982, respectively, and the Ph.D. degree in electrical engineering from McGill University, in 1985. Since 1985, he has been with Polytechnique Montréal, where he is currently a Professor with the Department of Electrical Engineering. He has carried out work in several areas related to microelectronic circuits and microsystems, such as testing, verification, validation, clocking methods, defect and fault tolerance, effects of radiation on electronics, high-speed interconnects and circuit design techniques, CAD methods, reconfigurable computing and applications of microelectronics to telecommunications, aerospace, image processing, video processing, radar signal processing, and the acceleration of digital signal processing. He is currently involved in several projects related to embedded systems in aircrafts, wireless sensor networks, virtual networks, software-defined networks, machine learning (ML), embedded ML, computational efficiency, and application-specific architecture design. He holds 16 patents, has published 190 journal articles and 485 conference papers, and was the thesis advisor of 190 graduate students who completed their studies. He was the Program Co-Chairman of NEWCAS'2018 and the General Chairman of NEWCAS'2020. He has been working as a consultant or was sponsored for carrying out research with Bombardier, Buspass, CNRC, Design Workshop, Dolphin, DREO, Ericsson, Genesis, Gennum, Huawei, Hyperchip, Intel, ISR, Kaloom, LTRIM, Miranda, MiroTech, Nortel, Octasic, PMC-Sierra, Space Co-Design, Technocap, Thales, Tundra, and Wavelite. Since June 2019, he has been the NSERC-Kaloom-Intel-Noviflow (KIN) Chair Professor. He is a fellow of the Canadian Academy of Engineering. He is a member of the Executive Committee of Regroupement Stratégique en Microélectronique du Québec (RESMIQ) and a member of Ordre des Ingénieurs du Québec (OIQ). In 2001, he was awarded the Tier 1 Canada Research Chair (www.chairs.gc.ca) on the design and architecture of advanced microelectronic systems, until June 2015. He received the Synergy Award of the Natural Sciences and Engineering Research Council of Canada, in 2006.