Discussion on Push-In-Extract-Out Scheduler in Hardware

Cristina McLaughlin
Department of Electrical Engineering
University of Hawaii at Manoa
Honolulu, United States
cemclaug@hawaii.edu

***Abstract*—Increasing link speeds, network congestion, and an influx of connected devices have created a demand for packet schedulers that can support quantity and speed while maintaining quality of service. Offloading packet scheduling to hardware can meet the demands of speed but has serious drawbacks in scalability and flexibility of design. Current hardware schedulers that are being used, like Push-In-First-Out (PIFO) or First-In-First-Out (FIFO), lack the ability to express a wide range of packet scheduling algorithms and policies. This paper summarizes research conducted in the paper "Fast, Scalable, and Programmable Packet Scheduler in Hardware" by Vishal Shrivastav; it describes a new iteration of the PIFO primitive called Push-In-Extract-Out (PIEO), which maintains an ordered list of elements and allows dequeue from any position by supporting a programmable filtering function. Next, the hardware implementation, results, and evaluation are reviewed. Finally, I present my thoughts on the paper, what I found most interesting, and issues that I have with the experiments conducted. Overall, the research on the PIEO primitive is new and interesting but lacks in areas that would make the case truly compelling.**

Keywords—hardware scheduler, programmable networks, programmable packet scheduler

I. INTRODUCTION

The Internet has become an all-encompassing medium for global communications, information access, streaming services, entertainment, and business. The 2020 Covid-19 pandemic has created a sudden reliance on services that allow us to work and learn from home like Zoom, Google Classroom, and Microsoft Teams. Globally, internet service providers saw a 40-50% network traffic increase, with some companies throttling streaming quality to prevent congestion overload [1-2]. The billions of connected devices coming online every year (a projected 75 billion by 2025 [3]) and increasing link speeds has created a critical demand for packet schedulers that can support the growing packet quantity and speed while maintaining quality of service.

Packet schedulers are functions at the network protocol level responsible for allocating bandwidth to competing connections and implementing scheduling algorithms or policies. The goal of a packet scheduler is to address network congestion, latency, load balancing, and packet loss without simply relying on a "best effort" delivery. Scheduling determinations are made by different queuing algorithms and protocols such as *Deficit Round Robin* (DRR) or *Token Bucket Filter* (TBF). Schedulers can also work based on quality of service (QoS) protocols like *Resource Reservation Protocol* (RSVP) or *Differentiated Services* (DiffServ) to prioritize certain types of packets over others on the network.

Scheduling can occur both in software and on hardware. For instance, hardware implementations occur on network interfaces (NICs) field-programmable gate arrays (FPGAs) while software implementations are included in operating systems like Linux, dummynet, and BSD operating systems. Software schedulers allow for experimentation with policies or algorithms but come at the cost of high CPU utilization and lower time precision. For instance, software scheduling can barely support 1-2 millions of packets per second (Mpps), while a high-end NIC reaches peak rate of 59.5 Mpps [4]. Meeting nanosecond-level precision in software is difficult due to the serialization of data, low resolution software timers, and transmission jitter. On the other hand, hardware implementations are generally fast and precise, but lack programmability and flexibility. Hardware schedulers are also difficult to modify after execution and generally can only support one scheduling algorithm, resulting in a overall preference for software schedulers. The motivation of many new schedulers is to bridge the gap between the flexibility of software and speed of hardware by making the hardware scheduler configurable to support several algorithms.

Research conducted by Vishal Shrivastav [5] proposed a new programmable packet scheduler in hardware, which is fast and scalable to overcome the limitations of schedulers in software. The proposed scheduling primitive *Push-In-Extract-Out* (PIEO) maintains an ordered list of elements and filters to select the eligible element with the smallest rank to dequeue. To demonstrate the feasibility of this new scheduler Shrivastav also prototyped the design on a Stratix V FPGA.

This paper focusses on summarizing and discussing the PIEO primitive. Section 2 provides background information and reviews relevant work in new scheduling primitives and implementations in hardware. Section 3 will provide more background information on PIEO scheduling. Section 4 will describe the hardware design and implementation on an FPGA. Next, Section 5 contains a discussion of PIEO; this will include my thoughts on the paper, issues with the experimentation, and

my suggestions for future work based on relevant research in Section 4. Lastly, Section 6 will conclude the paper.

II. Background

All scheduling algorithms are based on two decisions: i) *when* packets should be scheduled and ii) the *order* packets are scheduled based *rank* and all other elements enqueued. The rank property is determined by the chosen scheduling algorithm and programmed as function to assign ranks to individual elements. Generally, both decisions can be made as soon as a packet is enqueued.

A. Packet Scheduling Primitives

Typical hardware schedulers are based on one of the two scheduling primitives—First-In-First-Out (FIFO) or Push-In-First-Out (PIFO). FIFO schedules packets in order of arrival. PIFO is a priority queue that pushes elements into position based on rank and always dequeues from the head of the queue [6]. PIFO supports two operations: *enqueue(f)*, which inserts the element *f* into the queue according to its rank, and *dequeue()*, which extracts the head element of the queue.

However, both primitives lack in scalability and expressiveness. The current PIFO design only scales to 2048 flows, without finer granularity to support one flow per packet [6]. FIFO and PIFO are also not expressive enough to define classes required for certain scheduling algorithms. PIFO cannot support the dynamic filtering at dequeue that is becoming common in cloud network scheduling policies.

B. Packet Scheduling Algorithms

Packet scheduling algorithms or policies determine *when* and *what order* elements are scheduled while the primitive provides abstractions for implementation. Different rank functions and eligibility predicates gives the ability to express a wide range of scheduling algorithms through a single primitive. In addition, scheduling algorithms are classified into two categories: work-conserving and non-work conserving algorithms. The work-conserving category continuously sends packets as long as there are elements queued [7]. The scheduler always attempts to keep available resources busy. On the other hand, non-work conserving schedulers allow the link to be idle despite the presence of packets needing to be scheduled [7]. These schedulers are useful for traffic shaping, enhancing traffic predictability, and reducing jitter.

C. Related Work

Scheduling algorithms have been studied extensively, and there are multiple research papers that expand on scheduling in hardware with new architectures or programmability functions.

LOOM [8]. Loom proposes a new NIC design that moves all per-flow scheduling decisions into the hardware. It is a customizable and flexible packet scheduler that can implement a variety of scheduling policies. Loom uses an OS/NIC interface and enables the OS to drive the link speeds. Offloading scheduling to the NIC ensures low CPU utilization.

Loom is also the only multi-queue NIC design that can efficiently enforce network policy. It achieves this through two key components in the programmable NIC design: scheduling hierarchy and the OS/NIC interface that controls the NICs packet scheduling. The hierarchy is implemented through a common tree of priority queues that rely on the PIFO primitive. Different scheduling algorithms are implemented by changing the rank computation and this model can emulate any scheduling algorithm.

A Configurable Hardware Scheduler for Real-Time Systems [9]. This research presents a configurable hardware scheduler architecture that can implement three different scheduling policies: priority-based, rate monotonic, and earliest deadline first. The scheduler is fully run on an FPGA which allows for high-resolution timing and can be scaled by implementing fixed cycle operations. However, this implementation lacks in programmability and flexibility to implement more complicated scheduling policies.

This research differs from other hardware implementations because it takes advantage of recent FPGA technology that has hardware that is reconfigurable in less than 1.5 ms. They place part of an RTOS in the hardware to reduce scheduling overhead and time-tick processing by thousands of assembly instructions for a system with 50 tasks.

PSPAT [4]. PSPAT is an efficient and robust software implemented packet scheduler than can reach near hardware speed. This architecture decouples clients, scheduler, and device driver through the use of lock-free, memory-friendly mailboxes. This allows for distribution of work within in the system and maximum parallelism leading to a peak scheduling rate of almost 40 Mpps. PSPAT maintains scalability and flexibility to implement several different scheduling policies like DRR, WF^2Q, and QFQ; however, it has high CPU utilization and memory stalls in the system are heavily dependent on the CPU architecture used.

Programmable Packet Scheduling at Line Rate [6]. This was the original paper describing the Push-In-First-Out (PIFO) queue. The research details the implementation of a programmable PIFO scheduler in hardware that supports 5-level hierarchical scheduling and runs at a clock frequency of 1 GHz. It also has the same buffer size as a typical shared memory switch in a data center which support ~60K packets and ~1K flows, while only used 4% additional chip area.

III. Push-In-Extract-Out (PIEO) Primitive and Hardware Implementation

This section describes the PIEO primitive and summarizes the programming framework that was used in the original research.

A. PIEO Definition

Push-In-Extract-Out is a generalization of the Push-In-First-Out primitive. Each PIEO element has a rank and eligibility predicate that are programmed based on the user's choice of scheduling policy. PIEO also maintains an ordered list of elements—increasing by rank—through the "Push-In" enqueue function. This ensures that each element is placed in the proper position within the ordered list. Finally, during scheduling, the "Extract-Out" function filters through elements

whose eligibility predicates are true, and dequeues the element with the smallest rank. As a result, PIEO always schedules the "smallest ranked eligible" element.

PIEO works through the use of three functions: *enqueue(f), dequeue(),* and *dequeue(f)*. These operations are described below:

enqueue(f): This function inserts the element *f* into the ordered list according to *f*'s rank.

dequeue(): This function filters elements from the ordered list whose eligibility predicates are true. It selects the smallest ranked element and dequeues it. When a tie occurs for eligible elements, the oldest element is dequeued. If there are no eligible elements, the function returns NULL.

dequeue(f): If a specific element *f* needs to be dequeued, this function dequeues it from the list without having to rely on rank or eligibility predicates. If element *f* does not exist within the list, the function returns NULL. This operation provides the flexibility to update the rank of an element if needed. For example, one would *dequeue(f)* the element, update the rank or predicate, then *enqueue(f)* again.

B. PIEO in Scheduling Algorithms

This section summarizes the expressiveness of the PIEO primitive among different scheduling algorithms including work-conserving, non-work conserving, hierarchical, asynchronous, and priority scheduling.

Under work conserving algorithms, PIEO can express Deficit Round Robin (DRR), Weighted Fair Queuing (WFQ), and Worst-case Fair Queueing (WF^2Q+). For example, WF^2Q+ calculates a virtual start and finish time for each packet within a flow. It then schedules the flow whose head packet has the smallest finish time among all other flow heads. Within the PIEO primitive, element ranks would be equal to the finish time and the scheduling predicate would filter for elements that are eligible to be scheduled.

In non-work conserving algorithms, PIEO can express Token Bucket (TB) and Rate-controlled Static-Priority Queuing (RCSP). RCSP is an algorithm that is used to shape traffic, by assigning eligibility times to each packet in a flow [10]. At any given time, it schedules the highest priority flow amongst all other flows. Within the PIEO structure, element rank would be assigned by the priority variable and the predicate would filter for elements whose time eligibility is true.

Hierarchical scheduling is more difficult than flat scheduling due to grouping flows into different classes. The classes can contain custom scheduling policies that are unique to the class, so using a single PIEO is not possible. To express hierarchical scheduling a tree structure is used, where non-leaf nodes are higher classes, and leaf nodes represent flows. A PIEO is associated with each non-leaf node and used to schedule the node's children according to an eligibility predicates and rank. The packets propagate upward until they dequeue at the root of the tree. If we have an n-level tree, n PIEOs are required to complete the scheduling.

Asynchronous scheduling is a way to break out of the current scheduling algorithm to respond to packet scheduling issues. For instance, asynchronous tasks can be used to avoid starvation in strict priority scheduling or can be used to schedule back on network feedback. As described before, the *dequeue(f)* function can be used for asynchronous scheduling to dequeue, re-rank, then re-enqueue the necessary element. In the case of starvation avoidance, a user can define an alarm function and handler to observe time spent in queue and if the flow is starving, update the PIEO rank to increase its priority.

Priority scheduling like Shortest Job First (SJF), Shortest Remaining Time First (SRTF), and Earliest Deadline First (EDF) are easily expressed in a priority queue data structure. A priority queue is easily expressed in PIEO by setting the rank to the correct priority value, and just setting the eligibility predicate for each element to true. With this method, elements are only scheduled according to the rank.

C. PIEO Hardware Design and Implementation

This section summarizes the PIEO hardware scheduler design and implementation. Shrivastav conducted the prototype on the Stratix V FPGA which is comprised of ~2500 20Kbit dual port SRAM blocks with block access latency of one clock cycle. The main goal of implementing a scheduler in hardware is to keep up with link speeds, therefore the implementation of PIEO needs to execute each enqueue or dequeue function in O(1) time. The hardware design stores the ordered list of elements within SRAM as an array ($2\sqrt{N}$) of sublists of size $\sqrt{N}$ elements. This hardware design requires $O(\sqrt{N})$ flip flops and comparators. The sublists are ordered in increasing rank, and increasing order of eligibility time; therefore, both lists can be accessed and compared in one clock cycle.

The enqueuing and dequeuing functions are conducted in the hardware as follows. First, parallel comparisons are made between to the two sublists to find the correct list to enqueue or dequeue from. The correct sublist is extracted from SRAM. Next, using parallel comparisons again and the priority policy, an element within the chosen sublist is enqueued or dequeued, and the new sublist is updated in SRAM.

IV. Results and Evaluation

This section summarizes the results and evaluation of the PIEO hardware prototype. The performance was evaluated across three different metrics: scalability, scheduling rate, and programmability. Then, each field was compared against a PIFO implementation synthesized on the same FPGA.

A. Scalability

The scalability metric in the research evaluated the percentage of Adaptive Logic Modules (ALMs) and the percentage of SRAM consumed. The ALMs were used to implement the combinational and flip-flop-based logic, while the SRAM was used to store the ordered list.

Figure 1 shows the results of the percentage of consumed ALMs using PIFO versus PIFO. The control PIFO implementation consumed 64% of the available resources when handling a scheduler size of 1 K elements. The figure also shows that PIFO reaches it maximum size before 2 K elements

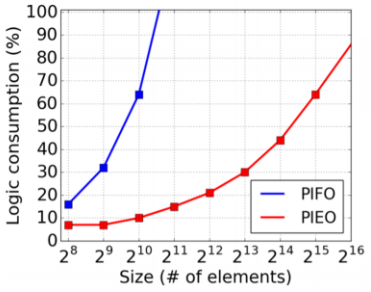

Fig. 1. Percentage of logic modules (ALMs) consumed.

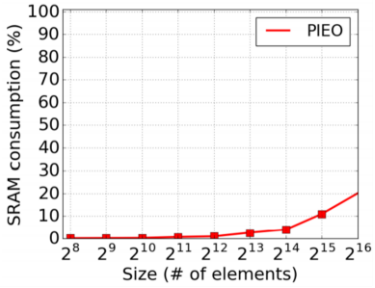

Fig. 2. SRAM consumption out of 6.4 MB.

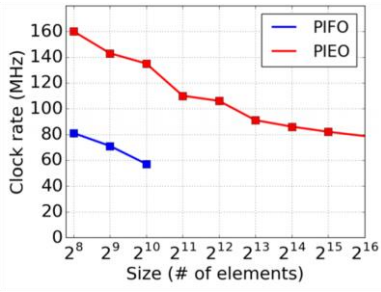

Fig. 3. Clock rates achieved by PIEO and PIFO.

meaning the scalability is limited. In contrast, PIEO compares well, reaching only about 10% consumption at 1 K elements and ~15% at 2 K elements. The graph does not even show it reaching maximum resource capacity, so the FPGA can easily fit a 30 K element scheduler.

PIFO consumes exponentially more ALMs that PIEO because it does not rely on SRAM and flip-flops to distribute storage and processing. This is also why Figure 2 does not show the SRAM consumption of PIFO. However, it does shows that despite PIEO's SRAM overhead, the total consumption remained well below 10% for the first 16 K elements and remained under 30% usage up to 65.5 K elements.

B. *Scheduling Rate*

The next metric evaluated was scheduling rate or the rate at which the scheduler could make decisions. The scheduling rate is a function of the clock rate of the scheduler circuit and the number of cycles needed to execute each primitive operation. Figure 3 shows the clock rates achieved by both PIFO and PIEO. Due to the scalability limits of PIFO the clock rate could only be tested up to 2^{10} elements. Through the three points collected PIFO performed twice as well as PIEO did. It is also important to note that PIEO was not pipelined in this test while PIFO was. PIEO's design is limited by the number of SRAM access ports and memory stages in different operations cannot be executed in parallel meaning it can never be fully pipelined.

PIEO takes 4 clock cycles or about 50 ns per every primitive operation. However, the clock rates achieved are both a function of design and the hardware device used. The author suggests that the design would run at much higher clock rates on more powerful FPGAs or ASIC hardware. For example, the PIFO design had a rate of 57MHz on the FPGA used while performing at 1 GHz on an ASIC. The report extrapolates that PIEO would take 4 ns at a 1 GHz clock rate.

C. *Programmability*

The last metric evaluated was programmability that the PIEO primitive has and its ability to express a wide range of scheduling algorithms. The author programmed two algorithms using PIEO—Token Bucket and WF^{2}Q—that were discussed earlier. Token Bucket and WF^{2}Q were chosen because they implement rate-limiting and fair queuing which are widely used policies in real-life scheduling implementations. Rate-limiting is used to control the rate of requests that are sent or received, and it is used to prevent denial of service attacks. Fair queuing is designed to achieve fairness over all flows to prevent starvation or greedy usage. The algorithms were then prototyped on the FPGA using System Verilog. A two-level hierarchical scheduler was prototyped, with ten nodes at level-2 and 10 flows within each node for a total of 100 flows. Figure 4 shows the rate limit enforcement of the PIEO and that it was successful across all Gbps. Figure 5 shows the results from the fair queueing experiments. PIEO was able to accurately enforce fair queueing across all flows with all rates.

V. Discussion

PIEO represents a novel data structure that a hardware packet scheduler can rely on. The research presented both a fast and scalable implementation of an ordered list in hardware. This section will be discussing my overall thoughts of the paper; it will be split into a) my general opinions of the paper, b) what I found most interesting and enjoyed, c) my criticisms of the research and issues I had with experimentation, and lastly d) suggestions for future work based on related work discussed in Section 2.

A. *General Opinions*

Starting with my general opinions about the work, I thought the premise was fascinating. After reading through the abstract and introduction, I was immediately interested because the paper promised to bridge the gap between software and hardware schedulers. They proposed a programmable hardware scheduler that is "fast, scalable, and more expressive than state-of-the-art" and I wanted to read further about how a new, simple primitive could achieve these statements. I also appreciated that it was instantly clear what the author was evaluating and how he continuously reiterated the three criteria: speed, scalability, and programmability so the reader was never lost on the main points. The paper also fit into my general interests because it combined low-level algorithms with FPGAs.

Tying in with the interest factor, is how easy, understandable, and straightforward the writing was. Unlike other research papers I have read, this captured my attention, then maintained it with clean, easy to understand writing. I was surprised at the lack of proofs, equations, and math; the introduction is the only section where time complexity is mentioned, and it is brief. It was also clearly split into two sections: describing PIEO and the hardware implementation of it. The paper is very short, so it is not boring to read through. It is 12 pages without the references, and only 7 pages of text

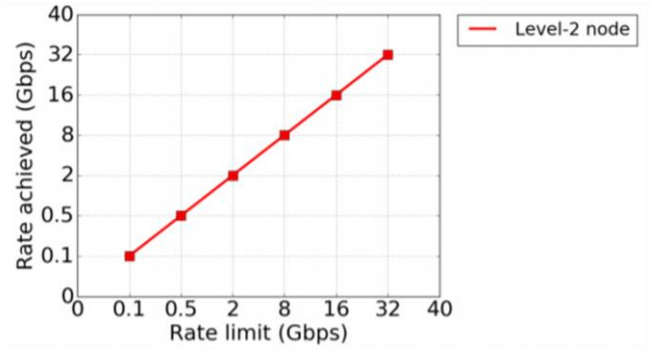

Fig. 4. Rate limit enforcement in PIEO.

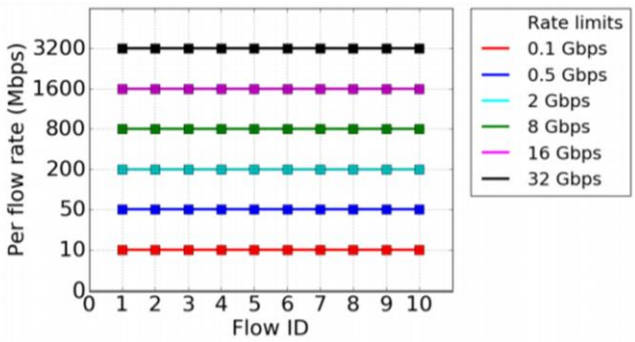

Fig. 5. Fair queue enforcement in PIEO.

when accounting for the number of figures—12 total—and the amount of pseudocode. However, I also felt that the paper was too brief on certain explanations; this problem will be discussed further below.

As mentioned, this paper relies heavily on figures and pseudocode to explain the PIEO model, programming framework, and different types of packet scheduling. Some of the figures were helpful, such as the generic packet scheduling model versus the PIEO scheduling model. These figures make the differences and improvements clear. However, I felt that other figures were poorly designed, and the written explanations were more understandable. For instance, there are two figures that the diagram the enqueue and dequeue operations from PIEO when conducted on an ordered list of size 16 elements (8 sublists each of size 4). Both figures show 32 columns of gray boxes filled with extremely small text; it was so unreadable I ignored them in favor of the step-by-step text descriptions. I feel like figures should be used to provide the reader with more insight into a topic; in this case, I gained a better understanding from the written explanations.

To conclude my general opinions, I think this paper did well in style and readability. However, it could use improvement in sections that were too brief in explanation and figures that were too small to be readable.

B. *Positives in Technical Areas*

This section discusses the positive aspects of the technical areas. First, I really appreciated the simplicity of design. The author proposes PIEO which is a generalized version of PIFO. By making the original PIFO design more generic, the author was able to make the primitive more flexible than its predecessor. The original design uses the first out mechanism, which only dequeues from the head of the queue. The extract mechanisms allow the user to program any dequeue criteria. This small change allows PIEO to express key classes of packet scheduling algorithms that PIFO cannot.

The straightforward design and minimal changes made to PIFO in order to create PIEO also make the results more impactful on the reader. I was surprised by the large differences in scalability and speed between the two primitives. It speaks clearly on how merely changing the dequeue method drastically improves the performance of the scheduler. Looking back on Figures 1 and 3, the differences in ALMs consumed and clock rates achieved are even more impressive when understanding how minimal the design changes were.

Another detail of the technical writing that I thought was positive was the in-depth descriptions showing the expressiveness of PIEO. In the original paper Section 4 goes through multiple classes of algorithms such as work conserving, non-work-conserving, hierarchical, asynchronous, and priority scheduling. I appreciated the comprehensive overview of how PIEO was implemented in each case. The pseudocode also makes it clear on how the primitive is used within the context of each scheduling policy; I could easily see what the eligibility predicate was set to and what determined rank for the enqueueing function.

C. *Drawbacks in Technical Areas*

There were several areas in the paper that did not perform as a well; this section will be discussing issues that I had with the experimentation and evaluation. My first criticism is the lack of thorough comparisons between PIEO and other queueing structures. Throughout the paper all the results show PIEO performs far superior to the PIFO predecessor, but this is the only comparison that is made. As a reader, I feel like the results and evaluation would be more compelling if the there were direct comparisons between the performance of PIEO and other hardware schedulers. Even if the author could not conduct the tests, he could report on the speed values or consumption of resources provided in other research papers. I think even implementing FIFO in hardware to have third data set would greatly improve the reporting. With a third set, the reader could really begin to understand how much an improvement PIEO would be on current scheduling primitives. The current state of the paper makes comparisons between two relatively unknown queuing data structures so some of the significance is lost.

Next, there is also a lack of discussion on limitations and improvements. Much like the point about making comparisons, I feel that discussing limitations is important in making a compelling case to the reader. As I was reading the glowing performance evaluation, I began to question why this design was not being immediately used in all hardware schedulers. Is it because the solution is expensive (requires a $6,000 FPGA), is it because PIEO cannot be pipelined, or is it due to difficulty in implementation? These are questions that I had after reading the results evaluation. In addition, the only suggestion for improvement was to implement PIEO on a better, more

expensive FPGA or on ASIC hardware, which is also very expensive.

Lastly, I had several issues with the specific section: 6.3 Programmability. This section of the paper described the testing and evaluation of the programmability criteria. My main criticism is that programmability is not defined, and it is not clear why it is being tested. The previous section in the paper that discusses the expressiveness of PIEO and implements various algorithms in pseudocode does a great job in showing the programmability. The actual testing only implements two—Token Bucket and WF^2Q—of the many algorithms that were described. The algorithms were chosen because they implement widely used scheduling policies, rate-limiting and fair-queuing. However, the case for flexibility and programmability would be much stronger if more scheduling algorithms were prototyped. In addition, the graphs, shown in Figure 4 and 5 do not show or provide further insights for the reader and the accompanying text was also lacking. Overall, this section was the weakest in the paper, and I feel like it could have been fully left out.

D. Future Work

In this portion I will be discussing ideas for future work for PIEO and the related work in Section 2. My first suggestion would be to improve upon this research by discussing the limitations and drawbacks of PIEO. As described earlier in Part C, the only suggestion for improvement in the original paper was using a stronger, more expensive hardware. I would like to see more work done, using more expensive hardware to understand what the limiting factor of PIEO is.

In conjunction with my previous point, it would also be interesting to implement pipelining on the PIEO design. The experiments conducted during the original research used a non-pipelined design for testing. The author states that some degree of pipelining is possible, but the access to SRAM ports during each memory stage of the primitive operation prevents a fully pipelined design. Future work should be done to implement as much pipelining as possible. The two designs could be compared on various FPGAs and the pipelined version could be tested for scalability and speed. It would also be interesting to see what the hardware requirement would be for the non-pipelined version to reach the same performance level a pipelined version can reach on the original hardware.

Loom, the flexible packet scheduling framework in the NIC, was discussed in Section 2. Loom uses a PIFO scheduler to enforce its scheduling policies. Future work could be done to implement Loom with PIEO rather than PIFO to see the effects of speed and scalability. I would be interested in seeing how difficult it would be to reimplement the design with PIEO as well as any drawbacks in using PIEO over PIFO.

The last suggestion I have for future work would be a survey of direct comparisons between schedulers in hardware and schedulers in software that are built to reach hardware speeds. As discussed earlier, PSPAT is an efficient software packet scheduler that was designed to reach speeds near hardware level. It can implement several scheduling policies just like how PIEO can and remains flexible and scalable due to the design in software. I would be interested in seeing how these designs directly compare since PIEO was measured by clock rate but PSPAT is measured in packets per second.

VI. Conclusion

This paper summarized and discussed a new packet scheduling primitive called Push-In-Extract-Out (PIEO). It assigns each element a rank and an eligibility predicate, pushes elements into place within a queue, and schedules the "smallest ranked eligible" element by extracting it out. PIEO has been shown to be expressive, scalable, and fast by being prototyped and tested on an FPGA against its predecessor PIFO. The hardware implementation was able to schedule 2^{16} elements at an 80 MHz clock rate, which was the maximum clock rate of PIFO when scheduling 2^8 elements. The ALMs consumption was 64% less than PIFO and the SRAM consumption was negligible even when scheduling 2^{16} elements.

My opinion on the research is that it is compelling but lacking in some areas that leave the reader confused or wanting more. It is focused solely on the abilities of PIEO. There are minimal performance comparisons made to similar technology but going into further detail on these topics would make the research feel more well-rounded. An in-depth discussion on the limitations and drawbacks of PIEO would also make the research more convincing. Overall, the paper was engaging and interesting to read and serves as a good baseline for future work to be built off it.

References

[1] E. Koeze and N. Popper, "The Virus Changed the Way We Internet," *The New York Times*, 07-Apr-2020. [Online]. Available: https://www.nytimes.com/interactive/2020/04/07/technology/coronavirus-internet-use.html. [Accessed: 12-May-2020].

[2] R. Browne, "The internet is under huge strain because of the coronavirus. Experts say it can cope - for now," *CNBC*, 27-Mar-2020. [Online]. Available: https://www.cnbc.com/2020/03/27/coronavirus-can-the-internet-handle-unprecedented-surge-in-traffic.html. [Accessed: 12-May-2020].

[3] K. Gyarmathy, "Comprehensive Guide to IoT Statistics You Need to Know in 2020," *Data centers and Colocation Services*. [Online]. Available: https://www.vxchnge.com/blog/iot-statistics. [Accessed: 12-May-2020].

[4] L. Rizzo, P. Valente, G. Lettieri, and V. Maffione, "PSPAT: Software packet scheduling at hardware speed," *Computer Communications*, vol. 120, pp. 32–45, 2018.

[5] V. Shrivastav, "Fast, scalable, and programmable packet scheduler in hardware," *Proceedings of the ACM Special Interest Group on Data Communication*, 2019.

[6] A. Sivaraman, N. Mckeown, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, and S. Katti, "Programmable Packet Scheduling at Line Rate," *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference - SIGCOMM 16*, 2016.

[7] J. Liebeherr and E. Yilmaz, "Workconserving vs. non-workconserving packet scheduling: an issue revisited," *1999 Seventh International Workshop on Quality of Service. IWQoS99. (Cat. No.98EX354)*.

[8] B. Stephens, A. Akella, and M. Swift, "Loom: Flexible and Efficient {NIC} Packet Scheduling," *16th USENIX Symposium on Networked Systems Design and Implementation 19*, 2019.

[9] P. Kuacharoen, M. A. Shalan, and V. J. Mooney, "A Configurable Hardware Scheduler for Real-Time Systems ," *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, pp. 95–101, 2003.

[10] H. Zhang and D. Ferrari, “Rate-controlled static-priority queueing,” *IEEE INFOCOM 93 The Conference on Computer Communications, Proceedings*.

[11] H. Zhang and D. Ferrari, “Rate-controlled static-priority queueing,” *IEEE INFOCOM 93 The Conference on Computer Communications, Proceedings*.