Fast, Scalable, and Programmable Packet Scheduler in Hardware

Vishal Shrivastav
Cornell University

ABSTRACT

With increasing link speeds and slowdown in the scaling of CPU speeds, packet scheduling in software is resulting in lower precision and higher CPU utilization. By offloading packet scheduling to the hardware such as a NIC, one can potentially overcome these drawbacks. However, to retain the flexibility of software packet schedulers, packet scheduler in hardware must be programmable, while also being fast and scalable. State-of-the-art packet schedulers in hardware either compromise on scalability (Push-In-First-Out (PIFO)) or the ability to express a wide range of packet scheduling algorithms (First-In-First-Out (FIFO)). Further, even a general scheduling primitive like PIFO is not expressive enough to express certain key classes of packet scheduling algorithms. Hence in this paper, we propose a generalization of the PIFO primitive, called *Push-In-Extract-Out (PIEO)*, which like PIFO, maintains an ordered list of elements, but unlike PIFO which only allows dequeue from the head of the list, PIEO allows dequeue from arbitrary positions in the list by supporting a programmable predicate-based filtering at dequeue. Next, we present a fast and scalable hardware design of PIEO scheduler and prototype it on a FPGA. Overall, PIEO scheduler is both more expressive and over 30× more scalable than PIFO.

CCS CONCEPTS

• **Networks → Programmable networks**; • **Hardware → Networking hardware**;

KEYWORDS

Programmable Packet Scheduling; Networking Hardware

1 INTRODUCTION

"In the era of hardware-accelerated computing,
identify and offload common abstractions and primitives,
not individual algorithms and protocols."

A packet scheduler enforces a scheduling algorithm or a scheduling policy which specifies when and in what order to transmit packets on the wire. Implementing a packet scheduler in software gives one the flexibility to quickly experiment with and adopt new scheduling algorithms and policies. However, this flexibility comes at the cost of burning CPU cycles which could have otherwise been used for running applications. In public cloud networks, this translates to loss in revenue [15], as the CPU overhead of packet scheduling [31] takes away from the processing power available to customer VMs. Unfortunately, this issue is only getting worse [3] with the growing mismatch between increase in link speeds and slowdown [11, 14] in the scaling of CPU speeds.

Next, with increasing link speeds, the time budget to make scheduling decisions is also getting smaller, e.g., at 100 Gbps link speeds, to enforce a scheduling policy at MTU timescale precision, a scheduling decision needs to be made every 120 ns. To put this in perspective, a single DRAM access takes about a 100 ns. Further, new transport protocols, such as Fastpass [30], QJump [16], and Ethernet TDMA [41], as well as recently proposed circuit-switched designs [35, 21, 42, 25], and protocols that rely on very accurate packet pacing [2, 19], require packets to be transmitted at precise times on the wire, in some cases at nanosecond-level precision [35]. Meeting these stringent requirements in software is challenging [31, 22, 28, 2, 35], mainly due to non-deterministic software processing jitter, and lack of high resolution software timers.

A packet scheduler in the hardware, such as a NIC, can potentially overcome the aforementioned limitations of software packet schedulers [31]. However, to retain the flexibility of software packet schedulers, packet scheduler in the hardware must be programmable. Further, today's multi-tenant cloud networks rely heavily on virtualization, with hundreds of VMs or light-weight containers running on the same physical machine. This can result in tens of thousands of traffic classes or flows per end-host [32, 31]. Hence, the packet scheduler must also be scalable.

State-of-the-art packet schedulers in hardware are based on one of the two scheduling primitives—(i) First-In-First-Out (FIFO), which simply schedules elements in the order of their arrival, and (ii) Push-In-First-Out (PIFO) [37], which provides a priority queue abstraction, by maintaining an ordered list of elements (based on a programmable *rank* function) and always scheduling from the head of the ordered list ("smallest ranked" element). Unfortunately, packet schedulers based on these primitives either compromise on scalability (PIFO-based scheduler), or the ability to express a wide range of packet scheduling algorithms (FIFO-based scheduler). Further, even a general scheduling primitive like PIFO is not expressive enough to express certain key classes of packet scheduling algorithms (§2.3). Hence in this paper, we propose a new programmable packet scheduler in hardware, which is fast, scalable, and more expressive than state-of-the-art.

To design a programmable packet scheduler, we use the insight that most packet scheduling algorithms have to make two key decisions in the process of scheduling: (i) *when* an element (packet/flow) becomes eligible for scheduling (decided by some programmable *predicate* as a function of time), and (ii) in *what order* to schedule amongst the set of eligible elements (decided by some programmable *rank* function). To that end, at any given time, packet scheduling algorithms first filter the set of elements eligible for scheduling at the time (using the predicate function), and then schedule the smallest ranked element from that set (§2.2, §2.3, §4).

SIGCOMM '19, August 19–23, 2019, Beijing, China

ACM ISBN 978-1-4503-5956-6/19/08…$15.00
https://doi.org/10.1145/3341302.3342090

Hence, most packet scheduling algorithms can be abstracted as the following scheduling policy—At any given time, schedule the "smallest ranked eligible" element.

Next, to realize the policy of scheduling the "smallest ranked eligible" element, one needs a primitive that provides abstractions for: (i) predicate-based filtering, and (ii) selecting the smallest element within an arbitrary subset of elements. Unfortunately, state-of-the-art priority queue based primitives such as PIFO do not provide these abstractions. Hence in this paper, we propose a new scheduling primitive called *Push-In-Extract-Out (PIEO)* (§3.1), which can be seen as a generalization of the PIFO primitive. PIEO associates with each element a *rank* and an eligibility *predicate*, both of which can be programmed based on the choice of the scheduling algorithm. Next, PIEO maintains an *ordered list* of elements in the increasing order of rank, by always enqueuing elements at the appropriate position in the list based on the element's rank ("Push-In" primitive). And finally, for scheduling, PIEO first filters out the subset of elements from the ordered list whose eligibility predicates are true at the time, and then dequeues the element at the smallest index in that subset ("Extract-Out" primitive). Hence, PIEO always schedules the "smallest ranked eligible" element. Further, we use the insight that for most packet scheduling algorithms, the time an element becomes eligible for scheduling ($t_{eligible}$) can be calculated at enqueue into the ordered list, and the eligibility predicate evaluation for filtering at dequeue usually reduces to ($t_{current} \geq t_{eligible}$). Here t could be any monotonic increasing function of time. This insight enables a very efficient hardware implementation of the PIEO scheduler (§5). Finally, we present a programming framework for the PIEO scheduler (§3.2), using which we show that one can express a wide range of packet scheduling algorithms (§4).

PIEO primitive maintains an ordered list of elements ("Push-In"), atop which filtering and smallest rank selection happens at dequeue ("Extract-Out"). However, implementing both a fast and scalable ordered list in the hardware is challenging, as it presents a fundamental trade-off between time complexity and hardware resource consumption. E.g., using $O(1)$ comparators and flip-flops, one would need $O(N)$ time to enqueue an element in an ordered list of size N, assuming the list is stored as an array[1] in memory. On the flip side, to enqueue in $O(1)$ time, designs such as PIFO [37] exploit the massive parallelism in hardware by storing the entire list in flip-flops and associating a comparator with each element in the list following the classic *parallel compare-and-shift* architecture [29], thus requiring $O(N)$ flip-flops and comparators, which limits the scalability of such a design [37]. In this paper, we present a hardware design (§5) of an ordered list for the PIEO scheduler which is both fast and scalable. In particular, our design executes both "Push-In" and "Extract-Out" primitive operations in $O(1)$ time (four clock cycles), while requiring only $O(\sqrt{N})$ flip-flops and comparators, while the ordered list sits entirely in SRAM.

To demonstrate the feasibility of our hardware design of the PIEO scheduler, we prototype[2] it on a FPGA (§6). Our prototype executes each primitive operation in few tens of nanoseconds (which is sufficient to schedule MTU-sized packets at 100 Gbps line-rate), while also scales to tens of thousands of flows (over 30× more scalable than PIFO). To further evaluate the performance of our prototype, we program it with a two-level hierarchical scheduling algorithm implementing rate-limit and fair queue policies. We show that our prototype could very accurately enforce these policies atop FPGAs with 40 Gbps interface bandwidth (§6.3).

This work does not raise any ethical issues.

[1]Linked list implementation also has overall time complexity of $O(N)$. Even probabilistic datastructures such as skip lists have average time complexity of $O(log(N))$, and worst-case time complexity of $O(N)$.

[2]Code for the FPGA implementation of PIEO scheduler is available at https://github.com/vishal1303/PIEO-Scheduler

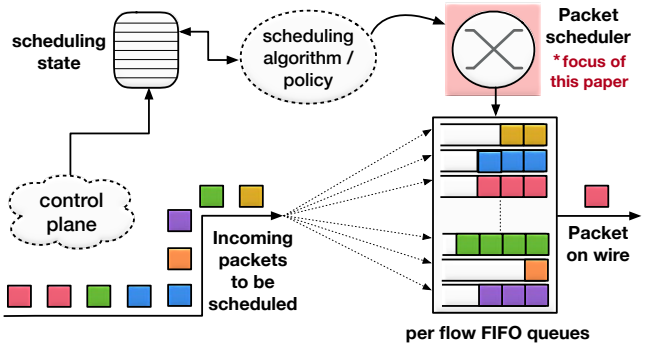

Figure 1: A generic packet scheduling model.

2 BACKGROUND

2.1 Packet scheduling model

Fig. 1 shows the packet scheduling model assumed in this paper. Packets ready to be scheduled for transmission are stored in one of the *flow queues* or *traffic classes*[3]. Packets within each flow queue are scheduled in FIFO order, whereas packets across queues are scheduled according to some custom *packet scheduling algorithm or policy*, which specifies when and in what order packets from each queue should be transmitted on the wire. To facilitate scheduling, a custom *scheduling state* is maintained in memory. Typically, this state could also be accessed and configured by the control plane. And finally, a *packet scheduler* is used to express and enforce the chosen scheduling algorithm/policy. The focus of this paper is to design an efficient packet scheduler in hardware, which could be programmed to express a wide range of packet scheduling algorithms/policies, in a fast and scalable manner.

2.2 Packet scheduling algorithms

Most packet scheduling algorithms can be abstracted as the following scheduling policy:

Assign each element (packet/flow) an eligibility predicate and a rank. Whenever the link is idle, among all elements whose predicates are true, schedule the one with the smallest rank.

The predicate determines *when* an element becomes eligible for scheduling, while rank decides in *what order* to schedule amongst the eligible elements. By appropriately choosing the predicate and rank functions, one can express a wide range of packet scheduling algorithms (§4). Further, packet scheduling algorithms can be broadly classified into two key classes:

[3]We use *flow queues* and *traffic classes* interchangeably in the paper.

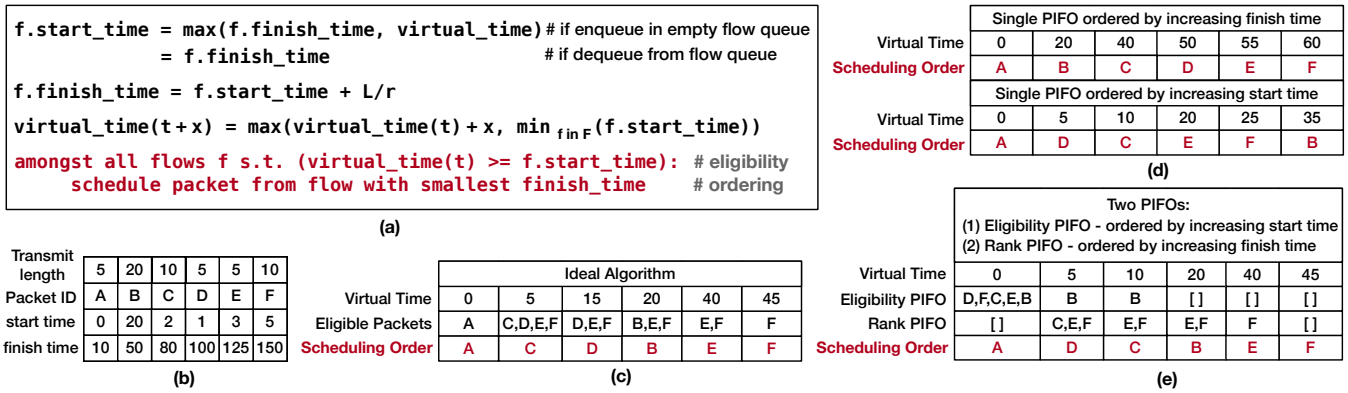

Figure 2: (a) WF^{2}Q+ algorithm [5]. L is the length of packet at the head of flow queue, r is the rate for flow f, x is the transmission length of current packet being transmitted, and F is the set of back-logged flows. (b) Packets at the head of six different flows in the example system, where packets can be of different sizes (transmission length). We also show start and finish times for each packet. (c) Scheduling order in an ideal run of WF^{2}Q+. (d) and (e) scheduling orders when running WF^{2}Q+ using PIFO.

Work-conserving algorithms. This class of packet scheduling algorithms ensure that the network link is never idle as long as there are outstanding packets to send. Hence, in these algorithms, the eligibility predicate of at least one active element is always true, and whenever the link is idle, the next eligible element in the order of rank is scheduled. Examples of work-conserving packet scheduling algorithms include Deficit Round Robin (DDR) [34], Weighted Fair Queuing (WFQ) [13], Worst-case Fair Weighted Fair Queuing (WF^{2}Q) [5], and Stochastic Fairness Queuing (SFQ) [23].

Non-work conserving algorithms. Under this class of packet scheduling algorithms, the network link can be idle even when there are outstanding packets to send, i.e., the eligibility predicate associated with each active element could all be false at the same time. Non-work conserving packet scheduling algorithms generally specify the *time* to schedule an element, and the eligibility predicate for an element p generally takes the form ($t_{current} \geq p.t_{eligible}$). Non-work conserving algorithms are naturally suited to express *traffic shaping* primitives such as rate-limiting and packet pacing [31, 32]. A classic example of a non-work conserving packet scheduling algorithm is Token Bucket [50].

2.3 Packet scheduling primitives

First-In-First-Out (FIFO). FIFO is the most basic scheduling primitive, which simply schedules elements in the order of their arrival. As a result, FIFO primitive is not able to express a wide range of packet scheduling algorithms. And yet, FIFO based schedulers are the most common packet schedulers in hardware, as their simplicity enables both fast and scalable scheduling.

Push-In-First-Out (PIFO) [37]. PIFO primitive assigns each element a programmable *rank* value, and at any given time, schedules the "smallest ranked" element. To realize this abstraction, PIFO maintains an ordered list of elements in the increasing order of rank, and supports two primitive operations atop the ordered list—(i) *enqueue(f)*, which inserts an element f into the ordered list, at the position dictated by f's rank, and (ii) *dequeue()*, which extracts the element at the head of the ordered list.

Limitations of PIFO. PIFO fundamentally provides the abstraction of a priority queue, which at any given time, schedules the "smallest ranked" element in the entire list. [37] shows that this simple abstraction can express a wide range of scheduling algorithms which either specify when or specify in what order to schedule each element. However, PIFO's abstraction is not sufficient to express a more general class of scheduling algorithms/policies which specify *both* when and in what order to schedule each element — This class of algorithms/policies schedule the smallest ranked element, but only from the set of elements that are eligible for scheduling at the time, which, in principle, could be any arbitrary subset of active elements[4]. Hence, they invariably require a primitive that supports dynamically filtering a subset of elements at dequeue and then return the smallest ranked element from that subset, something that PIFO primitive does not support. Such complex packet scheduling policies are becoming more common in today's multi-tenant cloud networks [38], and a classic example of one such algorithm is the Worst-case Fair Weighted Fair Queuing (WF^{2}Q) [6]. WF^{2}Q is the most accurate packet fair queuing algorithm known, making it an ideal choice for implementing fair queue scheduling policies. Further, non-work conserving version of WF^{2}Q can very accurately implement shaping primitives such as rate-limiting [31].

WF^{2}Q+[5] (Fig. 2(a)) tries to schedule a packet whenever the link is idle, which could be at any arbitrary discrete time t. However, challenge with WF^{2}Q+ is that the eligibility predicate of any arbitrary subset of elements can become true at t, as shown in Fig. 2(c), and hence scheduling the smallest ranked eligible element at time t becomes challenging. In Fig. 2(d) and (e), we try to express WF^{2}Q+ using PIFO. First, we try using a single PIFO (Fig. 2(d)). It is easy to see that this is not sufficient—if we order the PIFO by increasing finish times, it results in wrong scheduling order if some arbitrary element becomes eligible before the element at the head, and if we order the PIFO by increasing start times, the right scheduling order could still be violated if multiple elements become eligible at the

[4]PIFO could express a special case of such algorithms, where the smallest ranked element in the entire list is always eligible at dequeue.
[5]WF^{2}Q+ [5] is the implementation-friendly version of WF^{2}Q [13].

same time, and the element with the smallest finish time is not at the head of the PIFO. A more promising approach is to use two PIFOs, ordered by increasing start and finish times respectively, and move elements between the two PIFOs as and when the elements become eligible, as demonstrated in Fig. 2(e). However, this approach is also not sufficient, precisely because an arbitrary number of elements can become eligible at any given time, e.g., in Fig. 2, C,D,E, and F all become eligible at $t = 5$, and ideally C should have been scheduled as C has the smallest finish time amongst the eligible elements. But since the eligibility PIFO is ordered by increasing start time, D is released to rank PIFO before C, resulting in the wrong scheduling order. In general, $O(N)$ elements (N is PIFO size) could become eligible at any given time, which in the worst-case could result in $O(N)$ deviation from the ideal scheduling order for an element.

Further, PIFO primitive does not allow dynamically updating the attributes (such as rank) of an arbitrary element in the ordered list, as required by certain scheduling algorithms, e.g., updating the priority of an element based on it's age to avoid starvation in a strict-priority based scheduling algorithm.

Finally, the hardware design of the ordered list used to implement the PIFO primitive achieves very limited scalability ([37], Fig. 8). Hence, PIFO scheduler is also not scalable. In principle, one could use *approximate* datastructures, such as a multi-priority fifo queue [1], a calendar queue [10], a timing wheel [40], or a multi-level feedback queue [4], to implement an approximate version of the PIFO primitive. These datastructures could approximate the behavior of a priority queue or an ordered list in a fast and scalable manner by using multiple FIFO queues. However, by design, they could only express approximate versions of key packet scheduling algorithms [33, 4], invariably resulting in weaker performance guarantees [52]. Further, these datastructures also tend to have several performance-critical configuration parameters, e.g., number of priority levels in a multi-priority fifo queue, or size and number of buckets in a calendar queue, which are not trivial to fine-tune.

Universal Packet Scheduling (UPS) [27]. In the same vein as PIFO, which tries to propose a general packet scheduling primitive, UPS tries to propose a single scheduling algorithm that can emulate all other packet scheduling algorithms. While [27] proves that no such algorithm exists, it also shows that the classical Least Slack Time First (LSTF) [45] algorithm comes close to being universal. However, just like PIFO, LSTF also uses a priority queue abstraction at it's core, as it always schedules the "smallest slack first", just as PIFO would schedule the "smallest rank first". As a result, LSTF has the same limitations as PIFO discussed above.

3 PUSH-IN-EXTRACT-OUT (PIEO)

In this section, we describe the PIEO primitive and present a programming framework to program the PIEO scheduler.

3.1 PIEO primitive

PIEO primitive assigns each element an eligibility *predicate* and a *rank*, both of which can be programmed based on the choice of the scheduling algorithm, and at any given time, it schedules the "smallest ranked eligible" element. To realize this abstraction, PIEO maintains an ordered list of elements in the increasing order of rank, and supports three primitive operations atop the ordered list:

(1) **enqueue(f):** This operation inserts an element f into the ordered list, at the position dictated by f's rank. This operation realizes the "Push-In" primitive.
(2) **dequeue():** This operation first filters out a subset of elements from the ordered list whose eligibility predicates are true at the time, and then dequeues the element at the smallest index in that subset. Hence, this operation always dequeues the "smallest ranked eligible" element. If there are multiple eligible elements with the same smallest rank value, then the element which was enqueued first is dequeued. If no eligible element exists, NULL is returned. This operation realizes the "Extract-Out" primitive.
(3) **dequeue(f):** This operation dequeues a specific element f from the ordered list. If f does not exist, NULL is returned.

While the *enqueue(f)* and *dequeue()* operations are sufficient to schedule elements according to the PIEO primitive, the additional *dequeue(f)* operation provides an added flexibility to asynchronously extract a specific element from the list, to say, dynamically update the scheduling attributes (such as rank) of the element (and then re-enqueue using *enqueue(f)*), as illustrated in §4.4.

Limitations on complexity of predicate function. PIEO primitive associates a custom predicate with each element, which is evaluated at dequeue to filter a subset of elements. However, the complexity of predicate function is limited by the practical constraints of a fast and scalable packet scheduler. In particular, we want each primitive operation to execute in as few clock cycles as possible to keep up with increasing link speeds, while encode the predicate in as few bits as possible for scalability. Fortunately, for most packet scheduling algorithms, the predicate usually takes the form ($t_{current} \geq t_{eligible}$), where t could be any monotonic increasing function of time, and $t_{eligible}$ is when the element becomes eligible for scheduling and can be calculated at enqueue into the ordered list (§4). This allows for a fast and parallel evaluation of predicates at dequeue. Further, one only needs to encode $t_{eligible}$ for each element as the predicate, thus also ensuring a small storage footprint, important for scalability. One can potentially use PIEO primitive with more complex predicate functions for problems where constraints on time and memory are more relaxed.

3.2 PIEO programming framework

In this section, we describe a framework to program the PIEO scheduler. PIEO assumes that each packet is stored in one of the *flow queues*, and that the packets within each flow are scheduled in a FIFO order, as discussed in §2.1. Hence, each element in the ordered list refers to a flow, and scheduling a flow f results in the transmission of the packet at the head of flow queue f.

The programming framework for PIEO scheduler is shown in Fig. 3. PIEO scheduler comprises an ordered list datastructure that implements the PIEO primitive (§3.1), and can be programmed through the *rank* and *predicate* attributes assigned to each element.

3.2.1 Programming functions. In this section, we describe three generic types of functions that the programmers can implement to program the PIEO scheduler. We also describe the default behavior of each function, which the programmers can then extend based on the choice of the scheduling algorithm they intend to program. All the state needed for scheduling can be stored as either per flow or

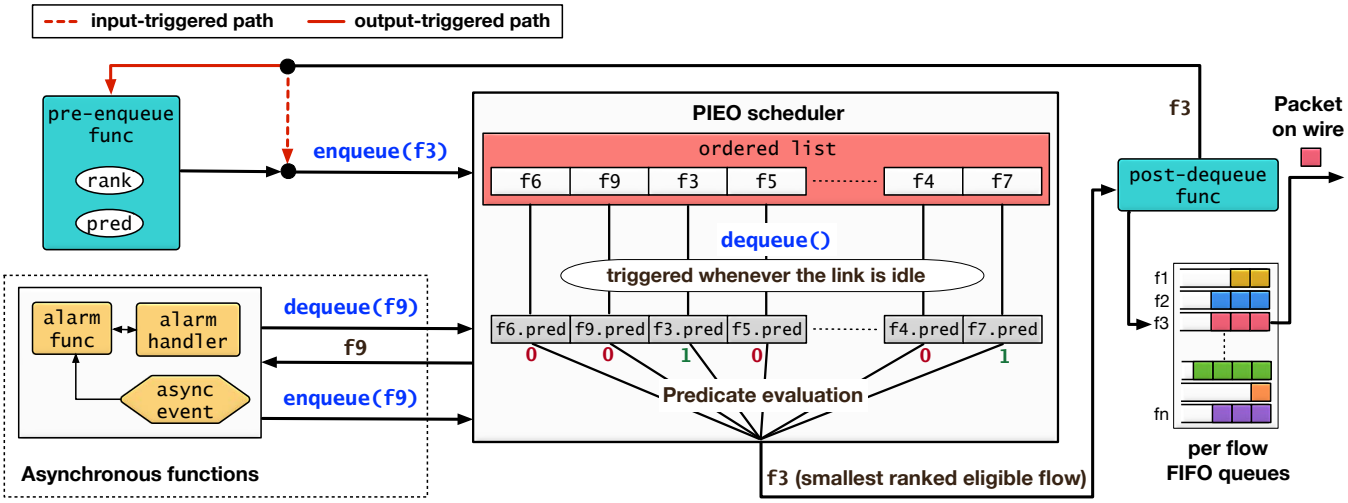

Figure 3: PIEO programming framework.

global state, and should be accessible by both the control plane and the programming functions. The control plane can use the memory to store control states, e.g., per-flow rate-limit value or QoS priority, while the programming functions can use the memory to store algorithm-specific state, e.g., virtual finish time in WFQ [13].

Pre-Enqueue and Post-Dequeue functions. Pre-Enqueue function takes as argument the flow f to be enqueued into the ordered list, and at the very minimum, assigns f a *rank* and a *predicate* as dictated by the choice of the scheduling algorithm being programmed. Note that while the predicate is assigned during enqueue into the list, it is only evaluated during the process of dequeue.

Post-Dequeue function takes as argument a flow f dequeued from the ordered list, and at the very minimum, transmits the packet at the head of flow queue f and re-enqueues f back into the ordered list if f's queue is not empty after current packet transmission.

While the Post-Dequeue function is always triggered after each *dequeue()* operation on the ordered list, the Pre-Enqueue function can be triggered in two different ways:

1. Input-triggered model: In this model, the Pre-Enqueue function is triggered whenever a packet is enqueued into a flow queue. The behavior of the default implementation of Pre-Enqueue and Post-Dequeue functions under this model is shown below.

```
pre-enqueue-func(flow f): { # default
    f.curr_pkt.rank = 1
    f.curr_pkt.predicate = True
    if (pkt enqueue into an empty f.queue):
        ordered_list.enqueue(f)
}
post-dequeue-func(flow f): { # default
    send(f.queue.head)
    if (f.queue not empty):
        f.rank = f.queue.head.rank
        f.predicate = f.queue.head.predicate
        ordered_list.enqueue(f)
}
```

2. Output-triggered model: In this model, the Pre-Enqueue function is triggered whenever a packet is dequeued from a flow queue, or whenever a packet is enqueued into an *empty* flow queue. The behavior of the default implementation of Pre-Enqueue and Post-Dequeue functions under this model is shown below.

```
pre-enqueue-func(flow f): { # default
    f.rank = 1
    f.predicate = True
    ordered_list.enqueue(f)
}
post-dequeue-func(flow f): { # default
    send(f.queue.head)
    if (f.queue not empty):
        pre-enqueue-func(f)
}
```

Programmers have the flexibility to choose between the two models. The trade-off is that while the output-triggered model can provide more precise guarantees for certain shaping policies [37], it also puts the Pre-Enqueue function on the critical path of scheduling, which means the complexity of rank and predicate calculations would affect the overall scheduling rate.

Alarm function and handler. The ability to asynchronously enqueue and dequeue specific elements to/from the ordered list using the *enqueue(f)* and *dequeue(f)* operations gives programmers the ability to define custom *events* which could trigger a custom *alarm function* that can asynchronously enqueue or dequeue a particular flow in or out of the ordered list. Programmers can also define a custom *alarm handler* function to operate upon the dequeued flow. By default, these functions are disabled in PIEO.

```
async_event e = NULL # default
alarm-func(async_event e): {} # default
alarm-handler(flow f): {} # default
```

3.2.2 Implementing programming functions. While we present a programming framework for the PIEO scheduler, the paper does not focus on a specific programming language to implement the programming functions used to program the PIEO scheduler. This would depend upon the underlying architecture of the hardware device. E.g., for FPGA devices, one could use languages such as System Verilog [49], Bluespec [8], or OpenCL [46] to implement the programming functions. In our FPGA prototype, we use System Verilog to implement the programming functions (§6.3). For ASIC hardware devices with RMT [9] architecture, one could potentially adapt one of the domain-specific languages for programmable switches such as Domino [36] (used to program the PIFO scheduler). We leave the exploration of new programmable hardware architectures and domain-specific languages (and compliers) for network hardware devices as an avenue for future work.

4 THE EXPRESSIVENESS OF PIEO

In this section, we use the PIEO primitive and the programming framework described in §3 to express a wide range of packet scheduling algorithms. Without loss of generality, the pseudo code for the programming functions presented in this section assumes the output-triggered model described in §3.2.1.

4.1 Work-conserving algorithms

Deficit Round Robin (DRR) [34]. DRR schedules flows in a round-robin fashion. When a flow is scheduled, DRR transmits packets from the flow until the flow runs out of credit (`deficit_counter`).

```
post-dequeue-func(flow f): {
    f.deficit_counter += f.quanta
    while (f.queue not empty
           & f.deficit_counter ≥ size(f.queue.head)):
        f.deficit_counter -= size(f.queue.head)
        send(f.queue.head)
    if (f.queue empty): f.deficit_counter = 0
    else: pre-enqueue-func(f)
}
other functions: default as described in §3.2.1
```

Weighted Fair Queuing (WFQ) [13]. WFQ calculates a virtual finish time for each packet in a flow, and at any given time, schedules the flow whose head packet has the smallest finish time value.

```
pre-enqueue-func(flow f): {
    r = Link_Rate / f.weight        # rate for flow f
    f.finish_time = max(f.finish_time, virtual_time)
                    + (size(f.queue.head) / r)
    f.rank = f.finish_time
    f.predicate = True
    ordered_list.enqueue(f)
}
post-dequeue-func(flow f): {
    virtual_time += (size(f.queue.head) / Link_Rate)
    rest is default as described in §3.2.1
}
other functions: default as described in §3.2.1
```

Worst-case Fair Weighted Fair Queuing (WF^{2}Q+) [5]. WF^{2}Q+ calculates a virtual start and finish time for each packet in a flow, and at any given time, schedules the flow whose head packet has the smallest finish time value amongst all the flows whose head packet has the start time less than or equal to current virtual time.

```
pre-enqueue-func(flow f): {
    calculate f.start_time as in Fig. 2(a)
    calculate f.finish_time as in Fig. 2(a)
    f.rank = f.finish_time
    f.predicate = (virtual_time(at deq) ≥ f.start_time)
    ordered_list.enqueue(f)
}
post-dequeue-func(flow f): {
    calculate virtual_time as in Fig. 2(a)
    rest is default as described in §3.2.1
}
other functions: default as described in §3.2.1
```

4.2 Non-work conserving algorithms

Token Bucket (TB) [50]. TB schedules packets from eligible flows, i.e., flows with enough tokens, or else defers the scheduling of the flow to some future time by when the flow has gathered enough tokens.

```
pre-enqueue-func(flow f): {
    f.tokens += f.rate * (now - f.last_time)
    if (f.tokens > f.burst_threshold):
        f.tokens = f.burst_threshold
    if (size(f.queue.head) ≤ f.tokens):
        send_time = now
    else:
        send_time = now +
            (size(f.queue.head) - f.tokens) / f.rate
    f.tokens -= size(f.queue.head)
    f.last_time = now
    f.rank = send_time
    f.predicate = (wall_clock_time(at deq) ≥ send_time)
    ordered_list.enqueue(f)
}
other functions: default as described in §3.2.1
```

Rate-controlled Static-Priority Queuing (RCSP) [53]. This class of algorithms shape the traffic in each flow by assigning an eligibility time to each packet within the flow, and at any given time, schedules the highest priority flow amongst all the flows with an eligible packet at the head of the queue.

```
pre-enqueue-func(flow f): {
    send_time = f.queue.head.time
    f.rank = f.priority
    f.predicate = (wall_clock_time(at deq) ≥ send_time)
    ordered_list.enqueue(f)
}
other functions: default as described in §3.2.1
```

4.3 Hierarchical scheduling

So far we have only discussed flat scheduling. However, in practice, it is often desirable to group flows into a hierarchy of classes, e.g., a two-level hierarchy comprising a group of VMs, with a group of flows within each VM. In this example, the link bandwidth can be shared amongst the VMs using some scheduling policy, e.g., a rate-limit for each VM, while one can use a different scheduling

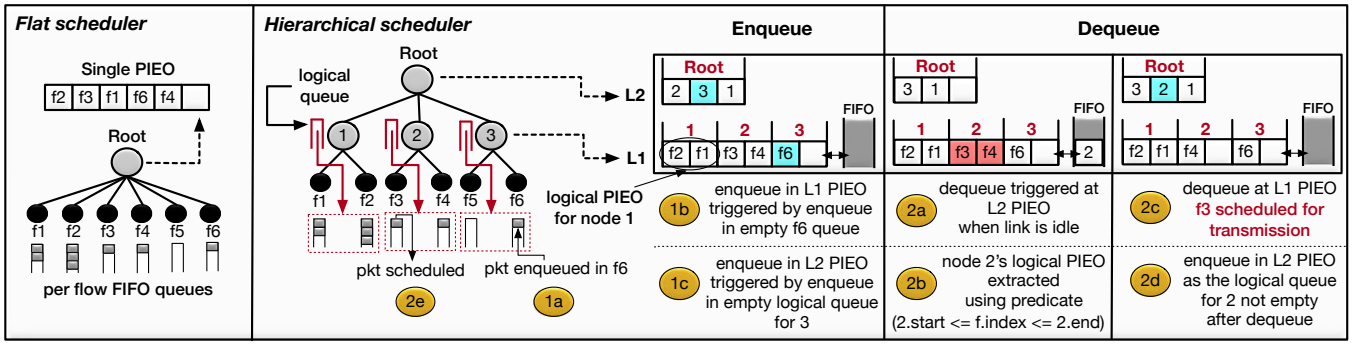

Figure 4: Hierarchical packet scheduling in PIEO.

policy to schedule the flows within each VM, e.g., fair queuing. In general, we can represent such hierarchies using a tree structure, as shown in Fig. 4, where the leaf nodes represent the flows, while the non-leaf nodes represent higher-level classes, such as VMs. Unfortunately, since each non-leaf node can implement it's own custom scheduling policy to schedule it's children, a single PIEO is not sufficient to express hierarchical scheduling policies. However, we can support hierarchical scheduling using multiple PIEOs.

We associate each node in the tree (except the root node) with a queue—for leaf nodes, these are per flow FIFO queues storing the packets, while for non-leaf nodes, these are *logical queues*, which are references to the set of child queues for that node. Next, we associate each non-leaf node with a *logical PIEO*, which schedules the node's children. Since PIEO allows us to filter a subset of elements from an ordered list using a predicate, all the nodes at the same level of hierarchy can share the same physical PIEO, which can then be logically partitioned into a set of logical PIEOs, one for each node at the same level in the hierarchy, with the size of each logical PIEO equal to number of corresponding node's children (Fig. 4). Next, each non-leaf node p maintains the start and end indices of it's logical PIEO and the eligibility predicate of each of it's child element f is extended with ($p.start \leq f.index \leq p.end$), thus allowing one to extract the ordered list of elements in p's logical PIEO (p's children) from the physical PIEO.

Enqueue in each level happens independently and is triggered by the same conditions as for flat scheduling, e.g., packet enqueue into an empty queue (§3.2.1). Dequeue always starts at the root PIEO, and propagates down to the lower levels in the tree hierarchy. Each lower level PIEO is associated with a FIFO to store the dequeued ids from the parent level. Dequeue at a level i is triggered whenever the corresponding FIFO in not empty. The logical PIEO corresponding to node *fifo.head* is then extracted, and the smallest ranked eligible element in the logical PIEO is dequeued and put into the FIFO at level $i-1$, until we reach the lowest non-leaf level, at which point the dequeued element (a leaf node representing a flow) is scheduled for transmission. All this is demonstrated in Fig. 4.

Finally, to support n–level hierarchical scheduling with arbitrary tree topologies, we need n physical PIEOs. We map this hierarchy to the hardware as an array of n independent physical PIEOs with a FIFO as the interface between any two consecutive PIEOs in the arraylist (Fig. 4).

4.4 Asynchronous scheduling

Starvation avoidance in strict priority scheduling. A common way to avoid starvation of lower priority flows in a strict priority based scheduling algorithm is to periodically increase the priority of the flow being starved. This is generally triggered whenever a flow has spent time larger than some threshold without being scheduled. Assuming flows are ranked by their priority in PIEO, one can define an alarm function and handler that can asynchronously update the starving flow's priority to avoid starvation.

```
async_event e = (curr_time - f.age ≥ threshold)
alarm-func(async_event e): ordered_list.dequeue(f)
alarm-handler(flow f): {
    f.age = curr_time
    f.priority = f.priority - 1
    pre-enqueue-func(f)
}
```

Scheduling based on asynchronous network feedback. Certain datacenter protocols such as [51, 12] can result in change of a flow's rank or eligibility based on some asynchronous feedback from the network. E.g., in D^3 [51], already scheduled flows can be quenched asynchronously based on network feedback.

```
async_event e = receipt of pause or resume feedback
alarm-func(async_event e): {
    if (recvd pause feedback for flow f):
        f.block = True
        ordered_list.dequeue(f)
    if (recvd resume feedback for flow f):
        f.block = False
        pre-enqueue-func(f)
}
alarm-handler(flow f): default as described in §3.2.1
```

4.5 Priority scheduling

Several scheduling algorithms assign a *priority* to each element, and schedule elements in the order of their priority. Examples include Shortest Job First (SJF) [47], Shortest Remaining Time First (SRTF) [48], Least Slack Time First (LSTF) [45], and Earliest Deadline First (EDF) [44]. Such algorithms can be expressed using a priority queue datastructure. One can easily emulate a priority queue using PIEO, by setting the rank of each element as equal to it's priority value, and setting the eligibility predicate of each element as true.

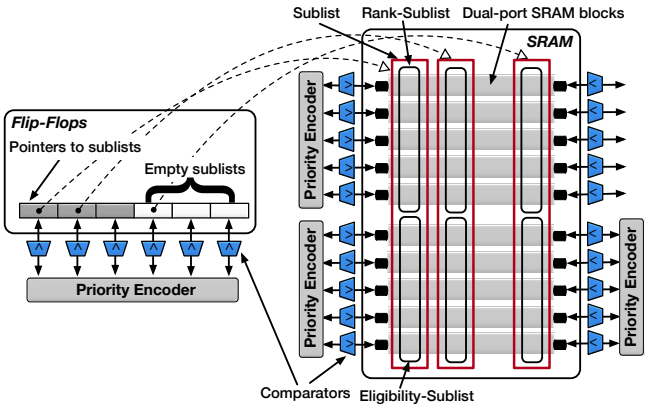

Figure 5: PIEO scheduler hardware architecture. A priority encoder takes as input a bit vector and returns the smallest index containing 1.

5 HARDWARE DESIGN

"All problems in computer science can be solved by another level of indirection." — *David Wheeler*

In this section, we describe the hardware design of PIEO scheduler.

We assume that the target hardware device is equipped with SRAM. Further, we assume that SRAM is divided into multiple blocks, and each SRAM block comprises independent access ports. Such a memory layout is very common in hardware devices, e.g., Stratix V FPGA [17] used in our prototype (§6) comprise ~2500 dual-port SRAM blocks of size 20 KBits each, where each SRAM block has access latency of one clock cycle.

5.1 Architecture

PIEO scheduler comprises an ordered list, that supports three primitive operations (§3.1)—*enqueue(f)*, *dequeue()*, and *dequeue(f)*. However, implementing an ordered list in hardware presents a fundamental trade-off between time complexity and hardware resource consumption. To keep up with increasing link speeds, we want to execute each primitive operation atop the ordered list in $O(1)$ time. However, to achieve this, state-of-the-art designs such as PIFO [37] require parallel access to each element in the list using the classic *parallel compare-and-shift* architecture [29], and hence have to store the entire list in flip-flops (as opposed to a more scalable memory technology such as SRAM), and associate a comparator with each element. Thus, such a design requires $O(N)$ flip-flops and comparators for a list of size N, and with the slowdown in transistor scaling [11, 14], this limits the scalability of such a design.

In this paper, we present a design of the ordered list that still executes primitive operations in $O(1)$ time, but only needs to access and compare $O(\sqrt{N})$ elements in parallel, while the ordered list sits entirely in SRAM. The key insight we use is to store and access the ordered list using one level of indirection (Fig. 5). More specifically, the ordered list is stored as an array (of size $2\sqrt{N}$) of *sublists* in SRAM, where each sublist is of size $\sqrt{N}$ elements. Elements within each sublist are ordered by both increasing rank (*Rank-Sublist*), and increasing order of eligibility time (*Eligibility-Sublist*). We stripe the elements of each sublist across $O(\sqrt{N})$ dual-port SRAM blocks, which allows us to access two entire sublists in one clock cycle. Next, we maintain an array (of size $2\sqrt{N}$) in flip-flops, which stores the pointers to the sublists, with sublists in the array ordered by increasing value of the smallest rank within each sublist. Thus, by sweeping across the sublists in the order they appear in the pointer array, one can get the entire list of elements in the order of increasing rank.

Enqueue and dequeue operations proceed in two steps—First, we figure out the right sublist to enqueue into or dequeue from, using parallel comparisons and priority encoding on the pointer array, and then extract the corresponding sublist from SRAM. Second, we use parallel comparisons and priority encoding on the extracted sublist to figure out the position within the sublist to enqueue/dequeue the element, and then write back the updated sublist to SRAM. Thus, with this design, we only require $O(\sqrt{N})$ flip-flops and comparators, unlike $O(N)$ in PIFO, at the cost of few extra clock cycles to execute each primitive operation (§5.2) and 2× SRAM overhead (Invariant 1). We evaluate these trade-offs in §6.

5.2 Implementation

In SRAM, PIEO maintains an array (of size $2\sqrt{N}$) of sublists, called *Sublist-Array*. Each sublist in the array is of size $\sqrt{N}$. Further, each sublist comprises two ordered sublists—*Rank-Sublist* and *Eligibility-Sublist*. Each element in *Rank-Sublist* comprises three attributes:

(1) **flow_id:** This is the flow id of the element.
(2) **rank:** This is the *rank* value assigned to the element by the enqueue function.
(3) **send_time:** This encodes the eligibility predicate assigned to the element by the enqueue function. PIEO exploits the fact that most scheduling algorithms use eligibility predicate of the form (*curr_time* ≥ *send_time*) (§4), where *send_time* is the time the element becomes eligible for scheduling. Thus, the eligibility predicate in PIEO is encoded using a single *send_time* value for each element. Predicate that is always true is encoded by assigning *send_time* to 0, and predicate that is always false is encoded by assigning *send_time* to ∞.

The *Rank-Sublist* is ordered by increasing *rank* values. Further, corresponding to each *Rank-Sublist*, there is an *Eligibility-Sublist* of the same size, which maintains a copy of the *send_time* attribute from it's corresponding *Rank-Sublist*. *Eligibility-Sublist* is ordered by increasing *send_time* values.

In flip-flops, PIEO maintains an array of size $2\sqrt{N}$, called *Ordered-Sublist-Array*, where each entry in the array points to a sublist in the *Sublist-Array*. More specifically, each entry in the *Ordered-Sublist-Array* comprises three attributes:

(1) **sublist_id:** This is the index (pointer) into the *Sublist-Array*, pointing to sublist *Sublist-Array[sublist_id]*.
(2) **smallest_rank:** This is the smallest *rank* value in the sublist *Sublist-Array[sublist_id]*, i.e., *Sublist-Array[sublist_id].Rank-Sublist[0].rank*.
(3) **smallest_send_time:** This is the smallest *send_time* value in the sublist *Sublist-Array[sublist_id]*, i.e., *Sublist-Array[sublist_id].Eligibility-Sublist[0]*.
(4) **num:** This stores the current number of elements in *Sublist-Array[sublist_id]*.

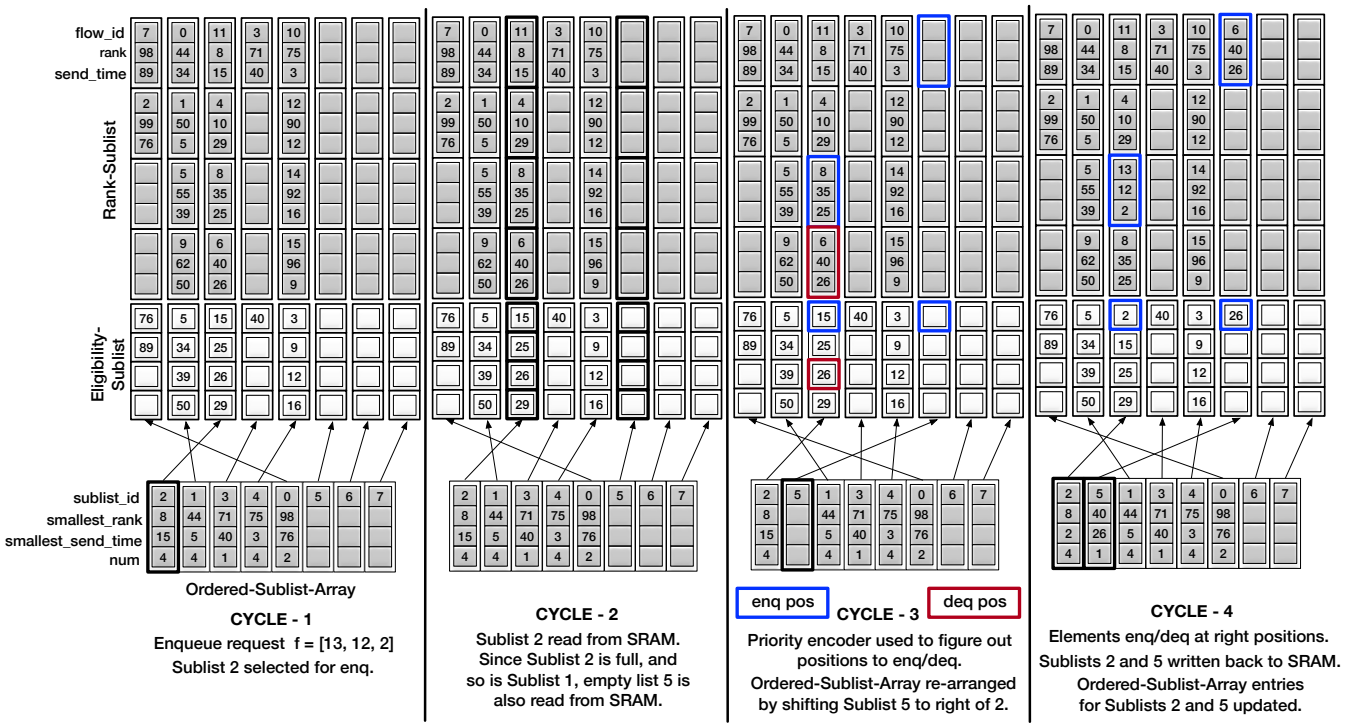

Figure 6: An example enqueue into the PIEO ordered list of size 16 elements (8 sublists each of size 4).

Ordered-Sublist-Array is ordered by increasing *smallest_rank* value. Further, *Ordered-Sublist-Array* is dynamically partitioned into two sections, as shown in Fig. 5—the section on the left points to sublists which are not empty, while the section on the right points to all the currently empty sublists.

By stitching together sublists in the order they appear in the *Ordered-Sublist-Array*, one can get the entire list of elements in PIEO. We call this list *Global-Ordered-List*. *Global-Ordered-List* is ordered by increasing *rank* value. Logically, enqueue and dequeue operations happen on top of the *Global-Ordered-List*.

Enqueue(f). The enqueue operation inserts element f into the *Global-Ordered-List*. It ensures that after every enqueue operation, the resulting *Global-Ordered-List* is ordered by increasing *rank* value. This is implemented in hardware as follows:

- **Cycle 1:** In this cycle, we select the sublist to enqueue f into, using the parallel compare operation (*Ordered-Sublist-Array[i].smallest_rank* > *f.rank*). We feed the resulting bit-vector into a priority encoder, which outputs index *j*. Sublist S pointed by *Ordered-Sublist-Array[j-1]* is selected for enqueue.
- **Cycle 2:** In this cycle, we read the sublist S from SRAM. In case S was full, the enqueue operation would push out an existing element in S. Hence, we also read an additional sublist S' to store the pushed out element. S' is either the sublist to the immediate right of S in the *Ordered-Sublist-Array*, provided it is not full, or else a new empty sublist.
- **Cycle 3:** In this cycle, we figure out the position to enqueue within sublist S, by running priority encoders on top of bit vectors returned by parallel compare operations (*S.Rank-Sublist[i].rank* > *f.rank*), and (*S.Eligibility-Sublist[i]* > *f.send_time*) resp. In case S was full, the tail element in *S.Rank-Sublist* will be moved to the head of *S'.Rank-Sublist*, while we use parallel compare operation (*S'.Eligibility-Sublist[i]* > *S[tail].send_time*), followed by priority encoding, to figure out the position to enqueue the *send_time* value of the element moving from S into the eligibility sublist within S'. In case S' was initially empty, we also re-arrange the *Ordered-Sublist-Array* by shifting S' to the immediate right of S.
- **Cycle 4:** In this cycle, we enqueue/dequeue respective elements at the positions output from the last cycle, and write back S (and S') to the SRAM. We also update the *Ordered-Sublist-Array* entries for S and S' with the new values of (i) *num*, (ii) *smallest_rank* (read from the corresponding *Rank-Sublist*), and (iii) *smallest_send_time* (read from the corresponding *Eligibility-Sublist*).

Invariant 1 [Bounding the number of sublists]. The key to ensuring $O(1)$ enqueue time is choosing a new empty sublist for enqueue whenever both the sublist to which the new element is to be enqueued and the sublist to it's immediate right in the *Ordered-Sublist-Array* are full. This avoids the chain-reaction of shifting the tail element of one sublist to the head of next (which would result in worst-case $O(\sqrt{N})$ SRAM accesses), at the cost of memory fragmentation (Fig. 6). However, an upshot of this design is the invariant that there cannot be two consecutive partially full sublists in the *Ordered-Sublist-Array*. As a consequence, to store N elements using $\sqrt{N}$-sized sublists, one would require at most $2\sqrt{N}$ sublists (2× SRAM overhead). We evaluate this overhead in §6.1.

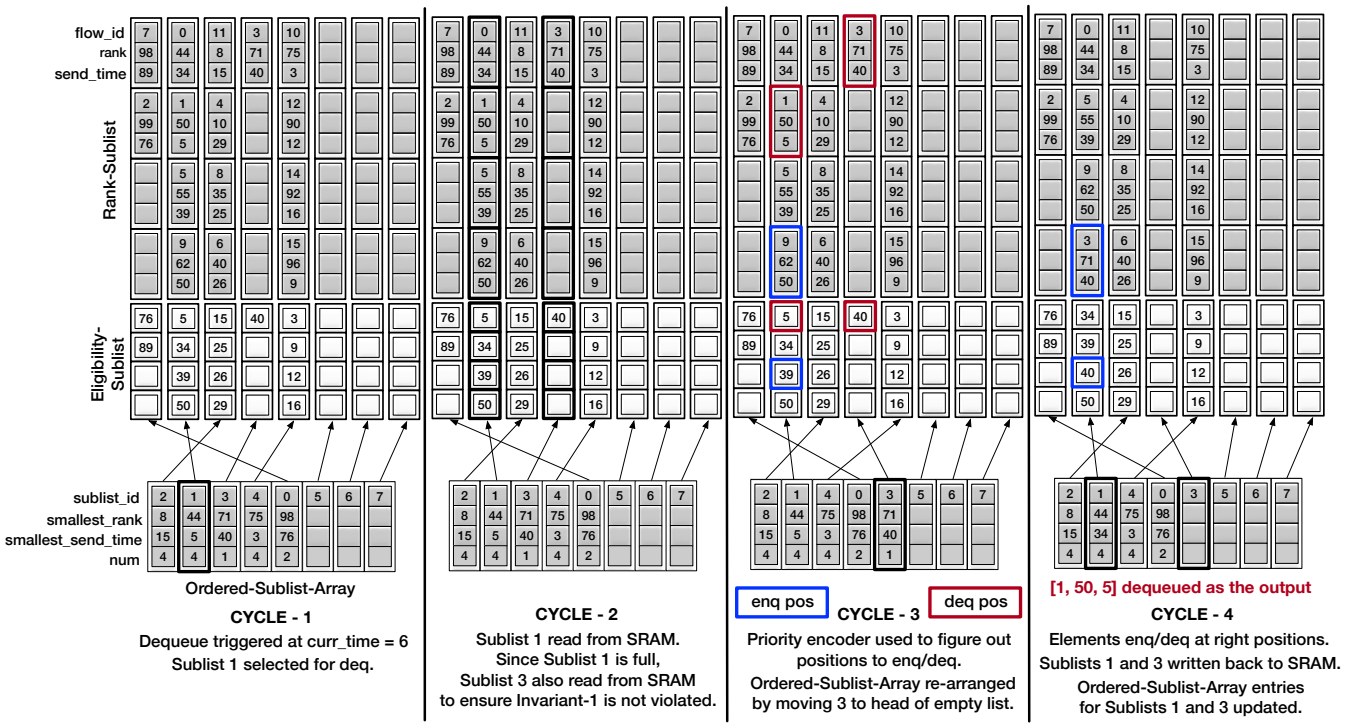

Figure 7: An example dequeue from the PIEO ordered list of size 16 elements (8 sublists each of size 4).

Dequeue(). This operation returns the "smallest ranked eligible" element in *Global-Ordered-List*. It is implemented as follows:

- **Cycle 1:** In this cycle, we select the sublist that contains the "smallest ranked eligible" element. For this, we use the priority encoder to extract the sublist at the smallest index in the *Ordered-Sublist-Array* that satisfies the predicate (*curr_time* ≥ *Ordered-Sublist-Array[i].smallest_send_time*). We call it *S*. The predicate ensures that *S* will have at least one eligible element, and since the *Ordered-Sublist-Array* is ordered by increasing *smallest_rank* value, the "smallest ranked eligible" element in the entire list is guaranteed to be in *S*.
- **Cycle 2:** In this cycle, we read the sublist *S* from SRAM. In case *S* was full, after a dequeue it would be partially full. So, to ensure Invariant 1 is not violated, we read another sublist, either to the immediate left of *S* in the *Ordered-Sublist-Array*, or to it's immediate right, whichever is not full, and choose either in case both of them are not full. We call it *S'*. If both left and right sublists are full, we only read *S*, as in that case even a partially full *S* would not violate Invariant 1.
- **Cycle 3:** In this cycle, we figure out the position to dequeue the "smallest ranked eligible" element from *S*. For that, we use the priority encoder that outputs the smallest index *idx* in *S.Rank-Sublist* satisfying the predicate (*curr_time* ≥ *S.Rank-Sublist[i].send_time*). The "smallest ranked eligible" element to be dequeued and returned as the final output of *dequeue()* operation is *S.Rank-Sublist[idx]*. Further, in the case *S* was full, we move an element from *S'* to *S*, to ensure that *S* remains full even after dequeue, hence ensuring that Invariant 1 will not be violated. The element to be moved, *e*, is deterministically added to either the head (if *S'* is to the left of *S*) or to the tail (if *S'* is to the right of *S*) of *S.Rank-Sublist* in the next cycle. However, we have to rely on priority encoding to figure out the position to dequeue *e.send_time* from *S'.Eligibility-Sublist*, and the corresponding position in *S.Eligibility-Sublist* where it would be enqueued. For that, we use parallel compare operations (*S'.Eligibility-Sublist[i]* == *e.send_time*) and (*S.Eligibility-Sublist[i]* > *e.send_time*) resp. Finally, in case either *S* or *S'* becomes empty after dequeue, we re-arrange the *Ordered-Sublist-Array* by shifting *S* or *S'* to the head of the logical partition comprising empty sublists.
- **Cycle 4:** In this cycle, we enqueue/dequeue respective elements at the positions output from the last cycle, and write back *S* (and *S'*) to the SRAM. We also update the *Ordered-Sublist-Array* entries for *S* and *S'* with the new values of (i) *num*, (ii) *smallest_rank* (read from the corresponding *Rank-Sublist*), and (iii) *smallest_send_time* (read from the corresponding *Eligibility-Sublist*).

Dequeue(f). This operation dequeues a specific element *f* from the *Global-Ordered-List*. PIEO keeps track of the sublist id that each element (flow) within the *Global-Ordered-List* is stored at, as part of the flow state, and updates this information after each primitive operation. In cycle 1, we use that information to select the sublist storing *f*, and then repeat cycles 2-4 from the *dequeue()* operation, with a modification in cycle 3 where we use the predicate (*f* == *S.Rank-Sublist[i].flow_id*) to figure out the index *idx* of the element in *S.Rank-Sublist* to be dequeued and returned as the final output.

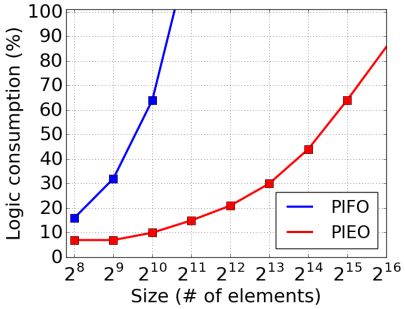

Figure 8: Percentage of logic modules (ALMs) consumed (out of 234 K).

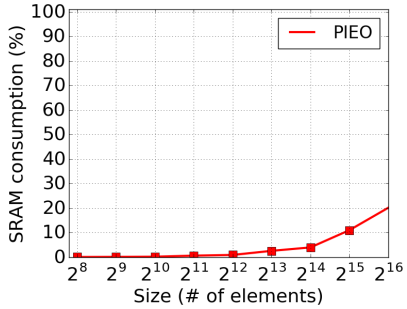

Figure 9: Percentage of SRAM consumed (out of 6.5 MB).

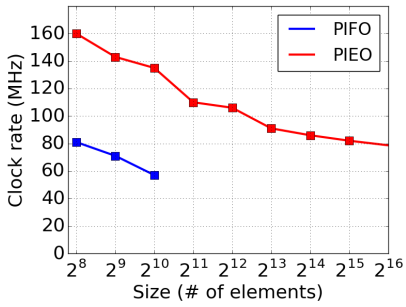

Figure 10: Clock rates achieved by the scheduler circuit.

6 PROTOTYPE AND EVALUATION

We prototyped the PIEO scheduler on an Altera Stratix V [17] FPGA comprising 234 K Adaptive Logic Modules (ALMs), 52 Mbits (6.5 MB) SRAM, and 40 Gbps interface bandwidth. Our prototype was written in System Verilog [49] comprising ~1300 LOCs.

We evaluate the performance of our prototype across three metrics—(i) Scalability, (ii) Scheduling rate, and (iii) Programmability. To serve as the baseline, we synthesized the open-source PIFO implementation [26] atop our FPGA. We use 16-bit rank and predicate fields, same as in PIFO implementation.

6.1 Scalability

In this section, we evaluate how the logic and memory resources consumed by PIEO's design scale with the size of the PIEO scheduler. We report the percentage of available Adaptive Logic Modules (ALMs) consumed to implement the combinational and flip-flop based logic (Fig. 8), and the percentage of available SRAM consumed to store the ordered list (Fig. 9). The baseline PIFO implementation consumes 64% of the available logic modules to implement a PIFO scheduler of size 1 K elements, and this scales linearly with the size of PIFO, meaning we can't fit a PIFO with 2 K elements or more on our FPGA. In contrast, the logic consumption for PIEO increases sub-linearly (as the square root function), and we can easily fit a PIEO scheduler with 30 K elements on our FPGA. This is a direct consequence of PIEO's design, which unlike PIFO, exploits the memory hierarchy available in hardware to efficiently distribute storage and processing across SRAM and flip-flops. Finally, even with 2× SRAM overhead (Invariant 1), the total SRAM consumption for PIEO's implementation is fairly modest (Fig. 9).

6.2 Scheduling rate

In this section, we evaluate the rate at which PIEO scheduler can make the scheduling decisions. Scheduling rate is typically a function of: (a) the clock rate of the scheduler circuit, and (b) number of cycles needed to execute each primitive operation[6]. Each primitive operation in PIEO takes 4 clock cycles and Fig. 10 shows the clock rate of PIEO circuit against increasing PIEO size. The clock rate naturally decreases with increasing circuit complexity, but even at 80 MHz and assuming a non-pipelined design, one can execute a PIEO primitive operation every 50 ns, which is sufficient to schedule MTU-sized packets at 100 Gbps line rate.

PIEO's scheduling rate can be further improved by pipelining the primitive operations. In a fully pipelined design, one could execute one primitive operation every clock cycle. However, PIEO's design is limited by the number of SRAM access ports. As such, the memory stages of each primitive operation (cycle 2 and 4) uses both the available access ports of dual-port SRAM to read/write two sublists. Hence, memory stages of different primitive operations cannot be executed in parallel, thus preventing a fully pipelined design. In principle, by carefully scheduling the primitive operations, one can still achieve some degree of pipelining, resulting in a better scheduling rate than a non-pipelined design. However, for simplicity, our prototype only implements the non-pipelined design, and all the analysis and results in the paper are for a non-pipelined design.

Further, the clock rates achieved by PIEO is a function of both the PIEO design and the hardware device used to run the design. We expect our design to run at much higher clock rates on more powerful FPGAs [18], but even more importantly, on an ASIC hardware, as ASIC based implementations tend to be more performant than an equivalent FPGA based implementation of the same design [20]. To back this using a concrete example, we note that PIFO's design on top of our FPGA was clocked at 57 MHz, as opposed to 1 GHz on an ASIC hardware as shown in [37]. At 1 GHz clock rate, each primitive operation in PIEO would only take 4 ns.

PIEO vs. PIFO trade-offs. Unlike PIEO, PIFO's hardware design can be fully pipelined, partly because it does not access SRAM at all, and hence PIFO scheduler can schedule at a higher rate than PIEO. This is the price we pay in PIEO's hardware design to achieve order of magnitude more scalability than PIFO. Alternatively, one can also implement PIEO primitive using PIFO's hardware design[7] and achieve more expressibility than PIFO without compromising on scheduling rate, albeit at a much smaller scale. We, however, argue that the trade-off made in PIEO's hardware design is a good trade-off to make, as PIEO's design is still extremely fast and can schedule at tens of nanosecond timescale (even less at higher clock rates), while also scale to tens of thousands of flows, which is critical in today's multi-tenant cloud networks [31, 32].

[6]For output-triggered model (§3.2.1), the complexity of Pre-Enqueue function also affects the scheduling rate, as explained in §3.2.1.

[7] Porting PIEO primitive to PIFO's hardware design is trivial despite PIEO supporting a more complex dequeue function, because the kind of predicates used in PIEO implementation can be evaluated in parallel in flip-flops in one clock cycle.

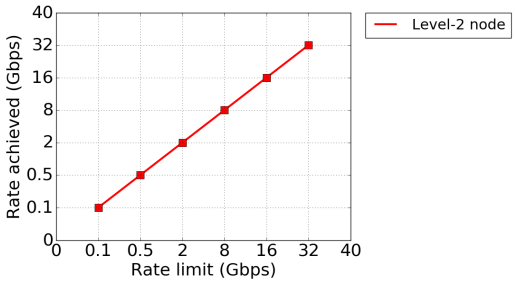

Figure 11: Rate-limit enforcement in PIEO prototype.

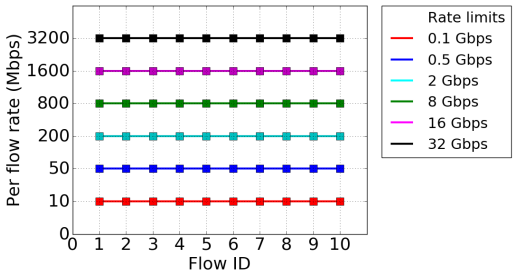

Figure 12: Fair queue enforcement in PIEO prototype.

6.3 Programmability

We show in §4 that one can express a wide range of scheduling algorithms using the PIEO primitive. In this section, we program two such algorithms, namely Token Bucket (§4.2) and WF^{2}Q+ (§4.1), atop our FPGA prototype using System Verilog as the programming language. The two chosen algorithms implement two of the most widely-used scheduling policies in practice, namely rate-limiting and fair queuing.

We program a two-level hierarchical scheduler using our prototype, with ten nodes at level-2 and ten flows within each node, for a total of 100 flows. We implement packet generators, one per flow, on the FPGA to simulate the flows. The link speed is 40 Gbps, and we schedule at MTU granularity. For experiments, we assign varying rate-limit values to each node at level-2 in the hierarchy, and enforce it using the Token Bucket algorithm. The rate-limit value of a particular node at level-2 is then shared fairly across all it's ten flows using WF^{2}Q+ algorithm. In Fig. 11, we sample a random level-2 node, and show that PIEO scheduler very accurately enforces the rate-limit on that node. Further, in Fig. 12, we show that for each rate-limit value assigned to the chosen level-2 node, PIEO scheduler very accurately enforces fair queuing across all the flows within that level-2 node.

7 RELATED WORK

Packet scheduling in hardware. Packet schedulers in hardware have traditionally been fixed-function, that either implement specific scheduling primitives, such as rate-limit, as in [31, 24], and fairness, as in [39, 33], or implement specific scheduling algorithms, such as shortest-job-first, as in [4]. More recently, there have been proposals for programmable packet schedulers, such as PIFO [37], and universal packet scheduling algorithms, such as UPS [27], whose goals align very closely with our work. We discuss the limitations of PIFO and UPS in §2.3. Somewhat complementary to our work, Loom [38] proposes a flexible packet scheduling framework in the NIC, using a new abstraction for expressing scheduling policies, and an efficient OS/NIC interface for scheduling, while leveraging the PIFO scheduler for enforcing the scheduling policies. In principle, systems like Loom can use PIEO scheduler instead of PIFO and achieve more expressibility and scalability.

Datastructures for hardware packet schedulers. Most packet schedulers in hardware rely on FIFO-based datastructures as it enables fast and scalable scheduling, at the cost of limited programmability or accuracy (§2.3). A priority queue allows ordered scheduling of elements, and hence can express a wide range of scheduling algorithms. P-heap [7] is a scalable heap-based implementation of priority queue in hardware. Unfortunately, a heap-based priority queue cannot efficiently implement the "Extract-Out" primitive in PIEO. PIFO [37] also provides a priority queue abstraction, but implements it using an ordered list datastructure, also used to implement PIEO. However, PIFO's hardware implementation of the ordered list is not scalable (Fig. 8). In this paper, we presented both a fast and scalable implementation of an ordered list in hardware.

8 DISCUSSION

PIEO as an Abstract Dictionary Data Type. In general, PIEO primitive can be viewed as an *abstract dictionary data type* [43], which maintains a collection of (*key, value*) pairs, indexed by *key*, and allows operations such as search, insert, delete and update. PIEO presents an extremely efficient implementation of the dictionary data type in hardware, which can do all the above mentioned operations in $O(1)$ time, while also being scalable. Further, it can also very efficiently support certain other key dictionary operations considered traditionally challenging, such as filtering a set of keys within a range, as PIEO implementation described in §5 can be naturally extended to support predicates of the form $a \leq key \leq b$. Dictionary data types are one of the most widely used data types in computer science, and PIEO presents a potential alternative to the traditional hardware implementations of the dictionary data type, such as hashtables and search trees.

9 CONCLUSION

We presented a new packet scheduling primitive, called *Push-In-Extract-Out (PIEO)*, which could express a wide range of packet scheduling algorithms. PIEO assigns each element a rank and an eligibility predicate, both of which could be programmed based on the choice of the scheduling algorithm, and at any given time, schedules the "smallest ranked eligible" element. We presented a fast and scalable hardware design of PIEO scheduler, and prototyped it on a FPGA. Our prototype could schedule at tens of nanosecond timescale, while also scale to tens of thousands of flows.

Acknowledgments. I thank the anonymous SIGCOMM reviewers and the shepherd, Anirudh Sivaraman, for their valuable comments and suggestions. I also thank my mentors at Cornell, Hakim Weatherspoon and Rachit Agarwal, as well as the entire Microsoft Azure Datapath team for valuable discussions. This work was partially supported by NSF (award No. 1704742).

REFERENCES

[1] IEEE Standard 802.11Q-2005. 2005. Standard for local and metropolitan area networks, Virtual Bridged Local Area Networks. (2005).

[2] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. 2012. Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *NSDI*.

[3] Mina Tahmasbi Arashloo, Monia Ghobadi, Jennifer Rexford, and David Walker. 2017. HotCocoa: Hardware Congestion Control Abstractions. In *HotNets*.

[4] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. 2015. Information-Agnostic Flow Scheduling for Commodity Data Centers. In *NSDI*.

[5] Jon C. R. Bennett and Hui Zhang. 1996. Hierarchical Packet Fair Queueing Algorithms. In *SIGCOMM*.

[6] Jon C. R. Bennett and Hui Zhang. 1996. WF^2Q: Worst-case Fair Weighted Fair Queueing. In *INFOCOMM*.

[7] R. Bhagwan and B. Lin. 2000. Fast and Scalable Priority Queue Architecture for High-Speed Network Switches. In *INFOCOMM*.

[8] Bluespec. 2019. BSV High-Level HDL. (2019). https://bluespec.com/54621-2/

[9] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *SIGCOMM*.

[10] Randy Brown. 1988. Calendar queues: A fast O(1) priority queue implementation for the simulation event set problem. In *Communications of the ACM*.

[11] R. Colwell. 2013. The chip design game at the end of Moore's law. In *HotChips*.

[12] IEEE DCB. 2011. 802.1Qbb - Priority-based Flow Control. (2011). http://www.ieee802.org/1/pages/802.1bb.html

[13] Alan Demers, Srinivasan Keshav, and Scott Shenker. 1989. Analysis and Simulation of a Fair Queueing Algorithm. In *SIGCOMM*.

[14] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark Silicon and the End of Multicore Scaling. In *ISCA*.

[15] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *NSDI*.

[16] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. 2015. Queues Don't Matter When You Can JUMP Them!. In *NSDI*.

[17] Intel. 2015. Stratix V FPGA. (2015). https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-v/stx5_51001.pdf

[18] Intel. 2018. Stratix 10 FPGA. (2018). https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-10/s10-overview.pdf

[19] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. 2015. Silo: Predictable Message Latency in the Cloud. In *SIGCOMM*.

[20] Ian Kuon and Jonathan Rose. 2007. Measuring the Gap Between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26, 2 (Feb. 2007).

[21] He Liu, Matthew K. Mukerjee, Conglong Li, Nicolas Feltman, George Papen, Stefan Savage, Srinivasan Seshan, Geoffrey M. Voelker, David G. Andersen, Michael Kaminsky, George Porter, and Alex C. Snoeren. 2015. Scheduling techniques for hybrid circuit/packet networks. In *CoNEXT*.

[22] Rob McGuinness and George Porter. 2018. Evaluating the Performance of Software NICs for 100-Gb/s Datacenter Traffic Control. In *ANCS*.

[23] P McKenney. 1990. Stochastic Fairness Queuing. In *INFOCOMM*.

[24] Mellanox. 2019. ConnectX-4. (2019). http://www.mellanox.com/page/products_dyn?product_family=204&mtag=connectx_4_en_card

[25] William M. Mellette, Rob McGuinness, Arjun Roy, Alex Forencich, George Papen, Alex C. Snoeren, and George Porter. 2017. RotorNet: A scalable, low-complexity, optical datacenter network. In *SIGCOMM*.

[26] MIT. 2019. PIFO implementation. (2019). https://github.com/programmable-scheduling/pifo-hardware/blob/master/src/rtl/design/pifo.v

[27] Radhika Mittal, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Universal Packet Scheduling. In *NSDI*.

[28] Radhika Mittal, Terry Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-based Congestion Control for the Datacenter. In *Sigcomm*.

[29] Sung-Whan Moon, J. Rexford, and K.G. Shin. 2000. Scalable hardware priority queue architectures for high-speed packet switches. *IEEE Trans. Comput.* 49, 11 (2000), 1215–1227.

[30] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. 2014. Fastpass: A Centralized "Zero-queue" Datacenter Network. In *SIGCOMM*.

[31] Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. 2014. SENIC: Scalable NIC for End-Host Rate Limiting. In *NSDI*.

[32] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. 2017. Carousel: Scalable Traffic Shaping at End Hosts. In *SIGCOMM*.

[33] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. 2018. Approximating Fair Queueing on Reconfigurable Switches. In *NSDI*.

[34] M Shreedhar and G Varghese. 1996. Efficient Fair Queuing using Deficit Round Robin. *IEEE/ACM Transactions on Networking (ToN)* 4, 3 (1996), 375–385.

[35] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. 2019. Shoal: A Network Architecture for Disaggregated Racks. In *NSDI*.

[36] A Sivaraman, A Cheung, M Budiu, C Kim, M Alizadeh, H Balakrishnan, G Varghese, N McKeown, and S Licking. 2016. Packet Transactions: High-Level Programming for Line-Rate Switches. In *SIGCOMM*.

[37] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. 2016. Programmable Packet Scheduling at Line Rate. In *SIGCOMM*.

[38] Brent Stephens, Aditya Akella, and Michael Swift. 2019. Loom: Flexible and Efficient NIC Packet Scheduling. In *NSDI*.

[39] Brent Stephens, Arjun Singhvi, Aditya Akella, and Michael Swift. 2017. Titan: Fair Packet Scheduling for Commodity Multiqueue NICs. In *ATC*.

[40] George Varghese and Anthony Lauck. 1987. Hashed and Hierarchical Timing Wheels: Efficient Data Structures for Implementing a Timer Facility. In *SOSP*.

[41] Bhanu Chandra Vattikonda, George Porter, Amin Vahdat, and Alex C. Snoeren. 2012. Practical TDMA for datacenter ethernet. In *EuroSys*.

[42] Shaileshh Bojja Venkatakrishnan, Mohammad Alizadeh, and Pramod Viswanath. 2016. Costly circuits, submodular schedules and approximate carathеodory theorems. In *SIGMETRICS*.

[43] Wikipedia. 2019. Abstract Dictionary Data Type. (2019). https://en.wikipedia.org/wiki/Associative_array

[44] Wikipedia. 2019. Earliest Deadline First. (2019). https://en.wikipedia.org/wiki/Earliest_deadline_first_scheduling

[45] Wikipedia. 2019. Least Slack Time First. (2019). https://en.wikipedia.org/wiki/Least_slack_time_scheduling

[46] Wikipedia. 2019. openCL. (2019). https://en.wikipedia.org/wiki/OpenCL

[47] Wikipedia. 2019. Shortest Job First. (2019). https://en.wikipedia.org/wiki/Shortest_job_next

[48] Wikipedia. 2019. Shortest Remaining Time First. (2019). https://en.wikipedia.org/wiki/Shortest_remaining_time

[49] Wikipedia. 2019. System Verilog. (2019). https://en.wikipedia.org/wiki/SystemVerilog

[50] Wikipedia. 2019. Token Bucket. (2019). https://en.wikipedia.org/wiki/Token_bucket

[51] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. 2011. Better Never Than Late: Meeting Deadlines in Datacenter Networks. In *SIGCOMM*.

[52] Jun Xu and Richard J. Lipton. 2002. On fundamental tradeoffs between delay bounds and computational complexity in packet scheduling algorithms. In *SIGCOMM*.

[53] H Zhang and D Ferrari. 1994. Rate-Controlled Service Disciplines. *Journal of High Speed Networks* 3, 4 (1994), 389–412.