PIPO: Efficient Programmable Scheduling for Time Sensitive Networking

Chuwen Zhang*¶, Zhikang Chen*, Haoyu Song†, Ruyi Yao‡, Yang Xu‡¶, Yi Wang§¶, Ji Miao*, Bin Liu*¶.
*Tsinghua University, China. †Futurewei, USA. ‡Fudan University, China.
§Southern University of Science and Technology, China. ¶Peng Cheng Lab, China.

2021 IEEE 29th International Conference on Network Protocols (ICNP) | 978-1-6654-4131-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICNP52444.2021.9651944

***Abstract*—Time Sensitive Networking (TSN) is an emerging Ethernet technology for real-time systems. To address different Quality-of-Service (QoS) requirements of applications, IEEE 802.1 TSN Task Group has standardized several packet scheduling and shaping algorithms. The software implementation of these algorithms is hard to meet the performance requirements, while the hardware implementation in Application-Specific Integrated Circuit (ASIC) is inflexible. A hardware-programmable scheduler is necessary to deal with this dilemma. Among the existing primitives, the most expressive one is Push-In-Extract-Out (PIEO), but its complexity makes the implementation very expensive. A relatively lower-cost implementation of PIEO cannot guarantee the scheduling correctness for the most critical Time-Triggered (TT) traffic in TSN. As a remedy, in this paper we propose a new Push-In-Pick-Out (PIPO) primitive under a TSN programmable scheduling framework. Composed of simple priority queues, PIPO can express all existing TSN scheduling and shaping algorithms, and is flexible enough to support future ones. Our PIPO implementation guarantees the TT traffic scheduling correctness. The simulation results corroborate the theoretical analysis that the low-cost PIPO can closely approximate PIEO and sustain a high bandwidth utilization. The prototype on Xilinx FPGA shows that, with 2,048 inputs, the PIPO-based scheduler achieves a throughput of 70 Mpps, which is 1.64x higher than the PIEO-based one, but using only 14.7% Look-Up Tables (LUTs) and 40.5% Block RAMs of the latter.**

I. Introduction

Today, an increasing number of time-critical applications, such as Industry Internet of Things (IIoT), Vehicle Adhoc Networks (VANET), and Cyber Physical System (CPS), require bounded and ultra-low latency for end-to-end communication in distributed systems. In response, Time Sensitive Networking (TSN) is proposed by IEEE 802.1 TSN Task Group to ensure Quality of Service (QoS) for time-critical traffic while maintaining the best-effort service for non-time-critical traffic in the same time-synchronized Ethernet infrastructure.

A packet scheduler, which manages the sending time and order of the queued packets, plays a vital role in TSN. Due to different QoS requirements for diverse applications, IEEE 802.1 TSN Task Group has standardized several packet scheduling and shaping algorithms. For example, to guarantee the latency for Time-Triggered (TT) traffic, Time-Aware Shaping (TAS) in IEEE 802.1Qbv [1] and Cyclic Queue Forwarding (CQF) in IEEE 802.1Qch [2] are standardized. With the development of TSN, new algorithms are expected to be proposed to meet the emerging requirements, such as the Asynchronous Time Scheduling (ATS) algorithm in the on-going IEEE 802.1Qcr standard [3]. Such diversity requires the TSN packet scheduler to be programmable and adaptive to customized algorithms. The scheduler also needs to achieve high throughput for line-rate forwarding and low resource consumption for cost efficiency.

A software-based solution (i.e., using CPU+DRAM) is indeed flexible to express various packet scheduling algorithms but hard to achieve high throughput and deterministic scheduling delay. The state-of-the-art software scheduler Eiffel [4], with a nominal $O(1)$ complexity for a scheduling decision, is still much slower than a hardware one which takes only a fixed and small number of clock cycles. Given the stringent requirements of TSN, a hardware scheduler is necessary. However, an ASIC-based hardware implementation can only support one or a few algorithms and is inflexible to adapt to future changes, whereas in TSN, the scheduling programmability is equally, if not more important than its performance.

In recent years, to support an expanding range of scheduling algorithms using the same hardware structure, researchers have proposed several general packet scheduling primitives, such as First-In-First-Out (FIFO), Push-In-First-Out (PIFO), and Push-In-Extract-Out (PIEO). FIFO requires packets in the same queue to be dequeued in their enqueuing order, so it has limited expression ability; PIFO [5], essentially a priority queue sorted in ascending order of the customized *rank* field, can express many packet scheduling algorithms such as WFQ [6]. Further, PIEO [7] supports to first filter eligible packets using some *predicate* and then dequeue the smallest-ranked eligible packet, so it can express even more scheduling and shaping algorithms than PIFO.

In this paper, our first original contribution is a programmable packet scheduling framework for TSN with extensive expressibility for TSN standard scheduling algorithms. Although the PIEO primitive can be used in this framework, its implementation cost is beyond affordability. Moreover, the alternative lower-cost implementation proposed in [7] cannot guarantee the scheduling correctness, which is unacceptable for the critical TT traffic in TSN. As a remedy, our second major contribution is a simpler Push-In-Pick-Out (PIPO) primitive as the approximation of PIEO. Different from PIEO

This work is supported by Guangdong Basic and Applied Basic Research Foundation (2019B1515120031), Key-Area Research and Development Program of Guangdong Province (2021B0101400001), NSFC grant (62032013, 61872213, 61432009, 62172108), NSFC-RGC (62061160489), PCL2021A08, and Shanghai Pujiang Program (2020PJD005). Co-corresponding Authors: Yi Wang (wy@ieee.org) and Bin Liu (lmyujie@gmail.com).

which strictly selects the *smallest*-ranked eligible packet, PIPO just picks a *small*-ranked eligible packet for non-TT traffic, without guaranteeing the extremum. This relaxed requirement significantly reduces the implementation complexity of PIPO, which can be implemented with pHeap [8], the efficient hardware priority queues. We show that the TT traffic can be correctly scheduled by our PIPO implementation, and the small inaccuracy on scheduling non-TT traffic is controllable and acceptable. The theoretical analysis is confirmed by the simulation results. The prototype on Xilinx FPGA shows that the PIPO implementation can achieve more than 70 Mpps throughput with 2,048 inputs. It consumes only 14.7% Look-Up Tables (LUTs) and 40.5% Block RAMs (BRAM) of a PIEO-based scheduler, but boosts the throughput by 1.64x.

The rest of the paper is organized as follows. Section II introduces the background of packet scheduling algorithms and related TSN standards. Section III explains the programmable packet scheduling framework and uses it to express the standard algorithms. Section IV describes the PIPO primitive. Section V evaluates the performance of PIPO by simulation and implementation. Finally, Section VI concludes the paper.

II. Background

In this section, we first introduce two classes of packet scheduling algorithms, and then describe the three existing primitives: FIFO, PIFO, and PIEO. Last, we introduce the related TSN standards on packet scheduling algorithms.

A. Packet Scheduling Algorithms

A packet scheduling algorithm specifies when and in what order the queued packets should be transmitted. Most scheduling algorithms can be abstracted to a process which selects the smallest-ranked eligible packet when the link is idle [7]. These algorithms can be divided into two classes in terms of whether an eligibility predicate takes effect.

Work-conserving algorithms do not let a link idle if there exists a packet waiting to be scheduled on it. The representative work-conserving algorithms include Strict Priority (SP), Deficit Round Robin (DRR) [9], WFQ, Worst-case Fair Weighted Fair Queuing (WF^{2}Q) [10], and Stochastic Fairness Queuing (SFQ) [11].

Non-work-conserving algorithms allow a link to be idle rather than sending an ineligible packet. The typical predicate can be checking the eligible sending time against the current time. As a result, non-work-conserving algorithms can limit the flow rates and thus express traffic shaping such as [12], [13]. The representative non-work-conserving scheduling algorithms include Token Bucket (TB) [14], TAS, and CQF.

B. Packet Scheduling Primitives

Although it is proved that no universal algorithm can express all scheduling algorithms [15], researchers strive to find some general scheduling primitives to cover as many scheduling algorithms as possible. We compare the definitions, expression ability, and hardware structure of the three basic primitives: FIFO, PIFO, and PIEO.

FIFO, essentially a first-in and first-out queue, is the basic and simplest scheduling primitive. Although the simplicity makes it fast, scalable, and easy to be implemented in hardware, it is incapable of expressing some popular packet scheduling algorithms. Using multiple FIFOs can help to some extent, e.g., realizing SP and some Round Robin based scheduling algorithms, or approximating PIFO as in [16], [17], but this also increases the hardware resource overhead.

PIFO is essentially a priority queue, in which each packet is assigned a rank to denote its priority, and the packet with the smallest rank (i.e., the highest priority) is always dequeued first. Many hardware implementations of the priority queue have been proposed. Binary tree [18] needs $O(\log n)$ steps for the dequeue operation, rendering the throughput too low to support large-scale inputs. Pipelined binary heaps [8], [19] offer scalability by using SRAM and achieve high throughput by using dedicated pipelines. Shift registers [18] and systolic arrays [18] both adopt the *compare-and-shift* process to maintain an ordered list by the rank. The main difference is that the former is parallel and the latter is pipelined. Sivaraman et al. propose the PIFO structure based on shift registers, achieving $O(1)$ operations for each enqueue and dequeue operation [5]. With the priority queue, PIFO is much more expressive than FIFO and capable of expressing most of the work-conserving packet scheduling algorithms.

PIEO is a priority queue with constraints, in which each packet is assigned not only a rank but also an eligibility predicate. The enqueue operation is similar to that of PIFO, but the smallest-ranked packet is dequeued only when it becomes eligible. Shrivastav first proposes PIEO and implements it with the *compare-and-shift* architecture for enqueue and *compare-and-encoder* for dequeue [7]. The latter needs to verify all the eligibility predicates through parallel comparison first; then the comparison result, i.e., a bitmap, goes through a priority encoder to get the index of the smallest-ranked eligible element. The PIEO implementation with only registers and logic gates is fast but not scalable, so a two-level structure which needs at most a $2\sqrt{n}$-bit shift register and $2\sqrt{n}$ BRAMs is proposed, where n is the ordered list length. However, the two-level structure needs four steps for each enqueue or dequeue operation, which cannot be pipelined well. Even worse, this implementation cannot realize the PIEO semantics for multi-dimensional predicates, and some scheduling failures may happen, making it inapplicable to TSN in which multi-dimensional predicate functions are required and the TT traffic cannot tolerate any scheduling inaccuracy. PCQ [20] uses the calendar queue [21] to approximate PIEO with low overhead, but it uses FIFO to store packets in the same bucket, resulting in non-negligible jitters for TT packets. Albeit the concerns on performance and cost, PIEO is more expressive than PIFO, and it can express many non-work-conserving algorithms, such as TB and Rate-controlled Static-Priority Queuing (RCSP) [22].

C. TSN Standards on Packet Scheduling

There are tree typical types of traffic in TSN: TT, Audio/Video Bridging (AVB), and Best-Effort (BE). TT traffic

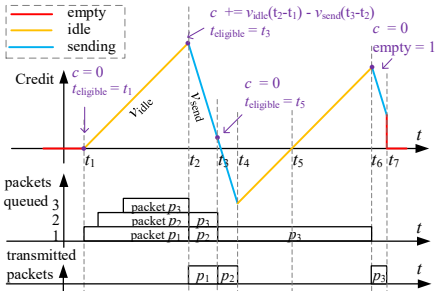


Fig. 1: Work process of CBS.

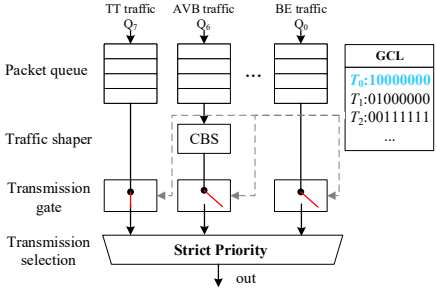


Fig. 2: Scheduler for IEEE 802.1Qbv.

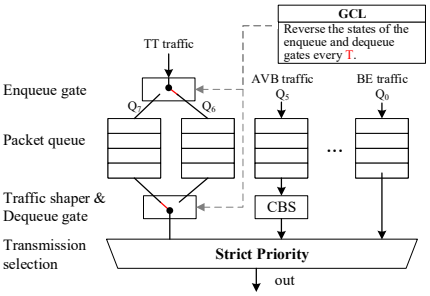


Fig. 3: Scheduler for IEEE 802.1Qch.

refers to the periodic and urgent traffic, which usually requires the latency and jitter to be within microseconds. AVB traffic refers to the traffic that demands guaranteed bandwidth and millisecond latency, such as the voice call. BE traffic refers to the ordinary traffic (e.g., the Web browsing) that should utilize the residual band and has the lowest requirement on latency. The TSN Task Group has proposed several standards for packet scheduling, such as Credit-Based Shaper (CBS) in IEEE 802.1Qav [23] for AVB traffic, TAS in 802.1Qav and CQF in 802.1Qch for TT traffic, and frame preemption in 802.1Qbu [24] and 802.3br [25] for BE traffic.

CBS is a non-work-conserving algorithm, which behaves like TB, i.e., a packet must wait in a queue until enough credit (tokens) is accumulated. However, CBS allows negative credit and a packet can be sent once the credit becomes positive, while TB requires the tokens to be greater than the packet size. An example of CBS process is shown in Fig. 1. During the time interval $[t_1, t_2]$ and $[t_4, t_6]$, the credit grows due to the other higher priority packets or negative credit, and during the time interval $[t_2, t_4]$ and $[t_6, t_7]$, the credit drops due to the packet transmission.

Fig. 2 shows an example of the TSN scheduler structure with 802.1Qbv, which possesses eight per-class packet queues: Q_7 for TT traffic, Q_6 for AVB traffic with CBS gates, and the others for BE traffic. Each queue is equipped with a TAS gate. The scheduled packet is selected among all available queues whose CBS and TAS gates are both opened according to SP. Qbv regulates the open or close state of the TAS gates by a Gate Control List (GCL) under a globally synchronized clock. As shown in Fig. 2, during the time interval $[T_0, T_1]$, only the TAS gate for queue Q_7 is '1' according to the GCL, so the TT traffic engrosses the port. In this way, the time line is split into many non-interfering time windows, resembling a special Time Division Multiple Access (TDMA) in a broad sense. The bounded end-to-end latency and jitters can be achieved if such windows are set properly. Existing work [26]–[28] has striven to get a feasible GCL for predefined TT traffic patterns, so in this paper we take the availability of a GCL for granted.

If an AVB or a BE packet is occupying the link at the start of a TT packet's window, the TT packet will be delayed. Hence, before each TT window, 802.1Qbv sets a large enough guard band during which no new packet is scheduled at the cost of bandwidth waste. Frame preemption solves this problem by a dedicated and complex MAC sub-layer for splitting and reassembling AVB and BE packets, with a guard band up to 143 bytes [29]. Furthermore, a packet scheduling algorithm, PAS [30], is proposed to utilize the guard band efficiently by modeling the problem as a Knapsack Problem [31].

Fig. 3 shows an example of the TSN scheduler structure for 802.1Qch, another standard for TT traffic. The states of enqueue and dequeue gates are opposite and exchanged every cycle T, so TT packets enter and leave Q_7 and Q_6 alternately. In the so-called ping-pong manner, any TT packet received in the cycle i must be sent out and received by its next-hop switch in the cycle $i+1$. As a result, if the ping-pong manner is not violated, the delay at each hop is bounded, and the end-to-end delay is in the range of $(h-1)\cdot T$ to $(h+1)\cdot T$, where h is the number of hops. Similarly, to meet the latency requirements of TT traffic, network operators need to derive a suitable T and a sending offset for each TT flow [32].

III. Programmable Packet Scheduling Framework

In this section, we first introduce a programmable packet scheduling framework for TSN, and then verify its powerful programmability by expressing typical TSN packet scheduling and shaping algorithms.

A. Framework Overview

As shown in Fig. 2, the TSN scheduler for 802.1Qbv relies on a complex and rigid structure to realize TAS, CBS, and SP. This scheduler lacks programmability in that, 1) it cannot realize the other non-work-conserving algorithms such as PAS, 2) it is not flexible to run customized work-conserving algorithms, and 3) its class-level scheduling granularity is too coarse to meet the diverse QoS requirements. Ideally, we need a flexible scheduling framework which is able to program more algorithms at flow-level granularity.

Most scheduling algorithms can be abstracted as a shaping and selection process. The former filters out non-eligible packets according to some predefined predicate function, and the latter selects the packet with the highest priority in the remaining packets. Based on the output-triggered model in [7], our packet scheduling framework supports scheduling algorithms with two prerequisites: 1) the packet descriptors which represent the necessary packet properties (e.g., rank) are stable and 2) the predicate functions only involve the packet properties and external environment variables (e.g., current time). As shown in Fig. 4, the framework consists of the packet buffer in per-flow FIFO queues, and the scheduler which handles the descriptor of each queue's head packet,

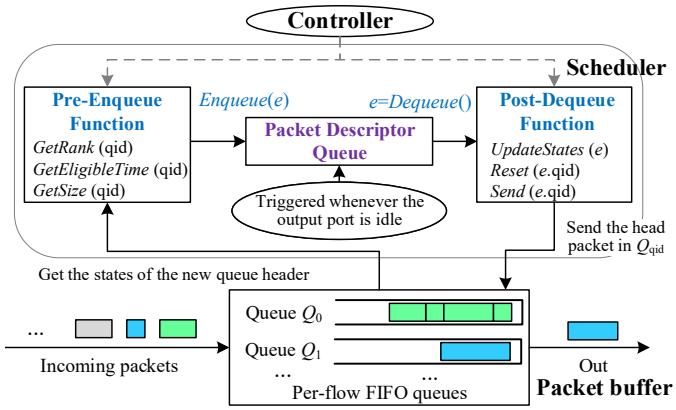


Fig. 4: The programmable scheduling framework for TSN.

so that packets from the same flow always leave in order. There is a descriptor queue in the scheduler, which stores the descriptor from the *Pre-Enqueue* function and sends the scheduled descriptor to the *Post-Dequeue* function. The enqueue and dequeue operations of the scheduler work as follows:

Enqueue: whenever an empty queue receives a new packet or a non-empty queue receives a call from the *Post-Dequeue* function, the *Pre-Enqueue* function (Line 1∼4 in Algorithm 1) generates a new descriptor e of the queue's head packet with four necessary properties: eligible sending time, rank, packet size, and queue id. The scheduler inserts this descriptor to a proper position in the packet descriptor queue.

Dequeue: when the link is idle, the descriptor queue first verifies the eligibility of all descriptors with two predicate functions: 1) eligible time predicate, which is true when the current time t_{now} reaches the eligible time, i.e., $t_{\text{eligibility}} \leq t_{\text{now}}$, and 2) packet size predicate, which is true when the packet size s does not exceed the residual band r, i.e., $s \leq r$. Then, the scheduler dequeues the smallest-ranked eligible descriptor e. At last, the *Post-Dequeue* function (Line 5∼11 in Algorithm 1) extracts the useful information from the dequeued descriptor, updates the flow-based states, demands the corresponding packet queue to transmit its head packet, and calls *Pre-Enqueue* to prepare for the next turn.

Some other frameworks based on PIFO and SP-PIFO execute the *Pre-Enqueue* function before buffering a packet and do not have a *Post-Dequeue* function, so they are easier to deploy but unable to express some algorithms such as WF^{2}Q and CBS. The state-of-the-art framework based on PIEO supports only one predicate of send time, failing to express PAS. The programmability of our framework is reflected in the transaction of the *Pre-Enqueue* and *Post-Dequeue* functions. We detail the expressiveness of our framework next.

B. Expressiveness

The scheduling algorithms in TSN can be divided into four categories: 1) packet-size-constraint algorithms such as PAS, 2) eligible-time-constraint algorithms such as TAS and CQF, 3) bandwidth-constraint algorithms such as CBS, and 4) work-conserving algorithms such as WFQ. We detail their expressions by the programmable framework.

Algorithm 1: Pre-Enqueue and Post-Dequeue function

```
1  Function Pre-Enqueue (q)
2      e.t_eligible ← GetEligibleTime (q)
3      e.rank ← GetRank (q) e.q_id ← q, e.s ← Size (Q_q.head)
4      e_queue.Enqueue (e)
5  Function Post-Dequeue (e)
6      UpdateStates (e)
7      Send (e.q_id)
8      if the packet queue of e.q_id is not empty then
9          Pre-Enqueue (e.q_id)
10     else
11         Reset () // Clear the states if needed
```

Algorithm 2: Transaction for TAS

```
1  Function GetEligibleTime (q)
2      return GetWindowStartTime(q) // from GCL
3  Function GetRank (q)
4      return 0 // the highest priority
```

1) Expressing PAS: The ideal PAS algorithm models the problem of utilizing residual band as a special Knapsack Problem, but its complexity is overwhelming for a hardware scheduler. Instead, the framework adopts the greedy PAS which selects the smallest-ranked packet whose size is no more than the residual band. The greedy PAS is supported by the framework naturally, as long as the packet size s can be obtained and the residual band r is maintained.

2) Expressing TAS and CQF: Essentially, both TAS and CQF specify the sending time window of each packet. A TT packet should be sent in its planned time window (which can be obtained from the GCL) for TAS and in the next cycle for CQF. To express TAS or CQF correctly, the following three requirements at the beginning of the interval must be met: 1) no other packet is being transmitted; 2) the TT packet is in the packet buffer; and 3) the TT packet has the highest priority for scheduling. The first requirement is met as PAS is supported; and if no forwarding misbehavior happens, the second one holds naturally. Ideally, all the packets of a TT flow should be sent back by back in their sending window, so we only need to set their $t_{eligible}$ to be the start time of their expected time window and rank to be the smallest (i.e., 0). The transactions of TAS and CQF are shown in Algorithm 2 and 3, respectively.

Using the framework to express TAS and CBS brings the following three benefits: 1) flexibility, as the complex modeling [26]–[28] for ensuring the TT packet order is not necessary, 2) efficiency, as the physical exclusive time windows disappear, and 3) stability, as the slightly delayed TT packet will be transmitted as soon as possible rather than waiting for the subsequent TT window and even violating the end-to-end latency requirement.

3) Expressing CBS: CBS deems a packet eligible for transmission only if the credit of its flow is non-negative. The detailed workflow is as follows: 1) while a packet is waiting

Algorithm 3: Transaction for CQF

Function *GetEligibleTime* (q)
  return *GetNextCycleStartTime*()
Function *GetRank* (q)
  return 0 // the highest priority

Algorithm 4: Transaction for CBS

Function *GetEligibleTime* (q)
  if empty$[q]$ **then**
    last$[q] \leftarrow t_{\text{now}}$
    return t_{now}
  credit$[q] \leftarrow$ credit$[q] + v_{\text{idle}}(t_{\text{now}} -$ last$[q]) - v_{\text{send}}$ delay$[q]$ // Update credit
  last$[q] \leftarrow t_{\text{now}} +$ delay$[q]$
  if credit$[q] \geq 0$ **then**
    return last$[q]$
  last$[q] \leftarrow$ last$[q] - \frac{\text{credit}[q]}{v_{\text{idle}}}$, credit$[q] \leftarrow 0$
  return last$[q]$
Function *GetRank* (q)
  return 1 // The second highest priority
Function *UpdateStates* (e)
  delay$[e.q_{\text{id}}] \leftarrow \frac{e.s}{v_{\text{link}}}$
Function *Reset* (q)
  credit$[q] \leftarrow 0$, empty$[q] \leftarrow 1$

in a queue, its flow's credit grows at a rate of v_{idle}; 2) while a flow is being transmitted, its credit drops at a rate of v_{send}; 3) when a queue is empty, the credit is cleared to zero. For each flow, we should set its v_{idle} equal to its required bandwidth and $v_{\text{send}} = v_{\text{idle}} - v_{\text{link}}$. The challenge of expressing CBS with the framework is that we cannot directly set the predicate to $Credit \geq 0$. This is because the credit may have changed since the descriptor is inserted into the descriptor queue, beyond the framework's requirement for stable properties. To solve this problem, we use the eligible time predicate instead. The eligible time, which can be calculated according to the current credit and v_{idle}, constrains when the packet should be scheduled if it is not blocked by any other packet with a higher priority. Each flow maintains the states of its last sending time and credit, allowing the *GetEligibleTime* function to update the current credit if the last sent packet is delayed.

The transaction for CBS is shown in Algorithm 4. For simplicity, we set the rank of AVB traffic to 1, i.e., the second highest priority. Fig. 1 reflects the key operations of Algorithm 4. We omit the description due to space limitation.

4) Expressing WFQ: For BE traffic, we can use various work-conserving algorithms together with the greedy PAS. We take the Start-Time Fair Queueing (STFQ) [33], an approximation of WFQ, as an example. We need to maintain a global virtual time t_{virtual} for all flows adopting STFQ. The sending time t_{start} (i.e., the rank in this case) is the maximum of the virtual finishing time of the flow's last packet and current t_{virtual}. When a packet is scheduled, it needs to update t_{virtual}

Algorithm 5: Transaction for STFQ

Function *GetEligibleTime* (q)
  return t_{now}
Function *GetRank* (q)
  $t_{\text{start}} \leftarrow \max(\text{last}[q], t_{\text{virtual}})$
  $\text{last}[q] \leftarrow t_{\text{start}} + \frac{Size(Q_q.\text{head})}{\text{weight}[q]}$
  return t_{start}
Function *UpdateStates* (e)
  $t_{\text{virtual}} \leftarrow t_{\text{virtual}} + \frac{e.s}{\text{weight}[e.q_{\text{id}}]}$

to be the packet's rank. Algorithm 5 shows the transaction for STFQ. Note that though the *GetEligibleTime* function always returns t_{now} to satisfy the eligible time predicate, BE packets are actually non-work-conserving due to the packet size predicate. Besides, t_{virtual} is updated according to the packet size and flow weight, rather than the rank of the scheduled packet, because the packet size predicate may make the BE packets out of the rank order.

C. Limitations

Although our programmable framework can express most scheduling algorithms, including the current standards in TSN, it has some limitations. First, since the enqueued descriptors cannot be modified, some descriptors for non-TT packets may suffer starvation and the scheduling algorithms with dynamic priority like pFabric [34] cannot be expressed by the framework. To solve the starvation problem, we propose a possible approach by considering the delayed time of each descriptor (§IV-B). Second, the per-flow packet queues may raise the concern of scalability. Fortunately, as discovered in [35], the number of active flows is typically small and a dynamic queue allocation mechanism can be used to solve the problem. Finally, the properties are stored in fixed and limited bits, but some algorithms require their values to increase monotonically (e.g., rank in STFQ), so assigning property values when reaching the upper bound is a problem. To alleviate this problem, we can reserve more bits for the properties and restore the initial state when the flow queue is empty.

IV. Push-In-Pick-Out Queue

For the programmable packet scheduling framework, the *Pre-Enqueue* and *Post-Dequeue* functions can be realized using the Domino [36] language on programmable switches or the Verilog language on FPGA-based switches [7]. We focus on the implementation of the packet descriptor queue in this section. We first analyze the issues of the PIEO primitive and its implementation[1]. Then, we detail the design of PIPO as the approximation of PIEO. Finally, we analyze PIPO's performance and its degree of approximation to PIEO.

[1]The original PIEO only supports the predicate of eligible time. For fair comparison, we modify the PIEO structure to also support the predicate of packet size. The term of PIEO refers to the modified structure from now on.

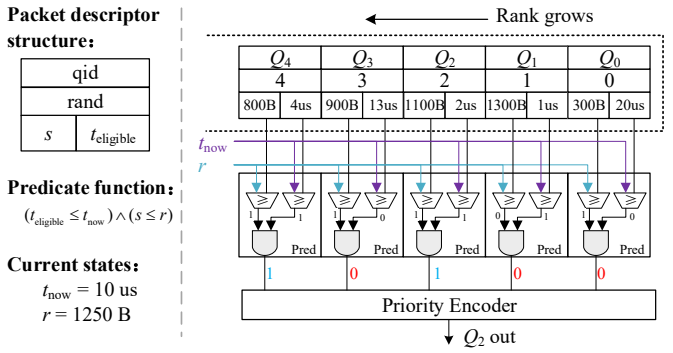


Fig. 5: An example of PIEO implementation with the one-level structure: the descriptors of Q_0 to Q_4 are sorted by the rank. Both Q_2 and Q_4 are eligible, but Q_2 is out as it has a smaller rank than Q_4.

A. Issues of PIEO Primitive and Implementation

Using PIEO, the packet descriptor queue always dequeues the smallest-ranked eligible packet, so PIEO can be used as the descriptor queue for TSN. Fig. 5 shows a straightforward implementation of the PIEO primitive with the one-level structure, where each packet descriptor stores its eligible time t_{eligible} and packet size s for the required predicate $(t_{\text{eligible}} \le t_{\text{now}} \wedge s \le r)$. It adopts the *compare-and-encoder* structure for dequeue operation. The five descriptors in Fig. 5 are ordered by their ranks in the shift registers. The predicate is realized by the parallel comparators, and the resulting bitmap is '00101'. Since the first bit '1' corresponds to the descriptor of Q_2, the priority encoder guides the descriptor of Q_2 to be dequeued, which satisfies the principle of the smallest-ranked-eligible-packet-first.

Unfortunately, such an implementation does not scale well. As the number of descriptors increases, the consumption of the shift registers increases linearly and can eventually exhaust the available resource. Moreover, obtaining the residual band r is not easy as the structure must know the start time of the next TT window with extra configuration cost. Trying to improve the cost, the PIEO implementation with a two-level structure [7] splits the priority queue of length n into $2\sqrt{n}$ sub-lists each with length $\sqrt{n}$. Only the sub-list table, which saves the digests of each sub-list, is stored in registers and the sub-lists are stored in BRAMs.

However, the two-level structure cannot realize PIEO's semantics (i.e., guarantee scheduling the smallest-ranked eligible packet) for multi-dimensional predicate functions and the scheduling failures may happen if some compromise is taken. As shown in the left of Fig. 6, if both the smallest size and eligible time are stored as predicates in register-based sub-list table as suggested in [7], for the scheduling example, sub-list 0 will be fetched but no eligible descriptor can be found in it. This is because the eligible predicate for each dimension may belong to different descriptors. In order to at least find an eligible descriptor, as shown in the right of Fig. 6, we need to maintain a table for the largest size and eligible time for each sub-list in place of the smallest size and eligible time,

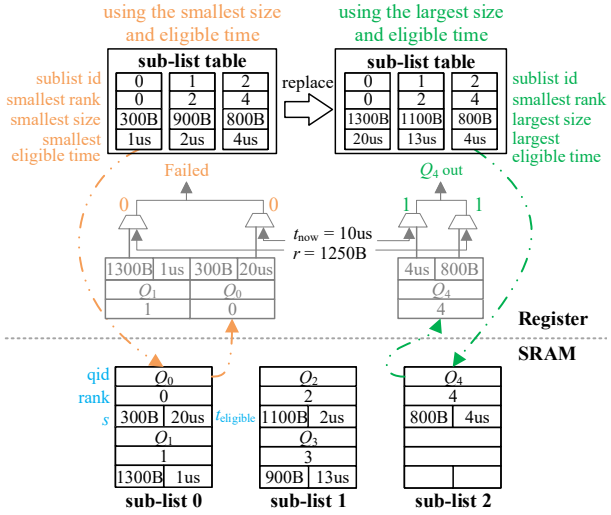


Fig. 6: An example of PIEO implementation with the two-level structure: the length of each sub-list is two; Q_0 and Q_1 is stored in sub-list 0; Q_2 and Q_3 is stored in sub-list 1; Q_4 is stored in sub-list 2. It fails to find a feasible solution with the smallest size and eligible time as both Q_0 and Q_1 in the selected sub-list 0 is not eligible. However, it can obtain an eligible packet Q_4 from the table recording the largest size and eligible time, although it may not be the smallest-ranked one (e.g., Q_2 in sub-list 1 is better).

so the selected packet based on this table is guaranteed to be eligible, although it may not have the smallest rank.

Moreover, the two-level PIEO structure cannot achieve high throughput. Take its dequeue operation as an example. In the first cycle, it gets the sub-list index which stores the smallest-ranked eligible descriptor, by comparing the current time and residual band with the largest eligible time and largest packet size in each sub-list, respectively. In the second cycle, it reads sub-list S from SRAM. In the third cycle, it finds the position of the smallest-ranked eligible descriptor in S and does other necessary operations. In the last cycle, it dequeues the selected descriptor and writes the sub-list back to SRAM. The above four cycles cannot be pipelined for two reasons: 1) the second and fourth cycles involve SRAM read/write, so the structural hazard requires enough bubbles to be placed in the pipeline; 2) the packet scheduled in the fourth cycle reduces the residual bandwidth capacity for the first cycle, i.e., it may cause the true predicates to be false in the previous stages, leading to a data hazard. The data hazard is difficult to solve, as the length of the dequeued packet cannot be obtained until the last cycle. In brief, the above PIEO implementations cannot achieve high throughput, low cost, or the required semantics, making them unsuitable for TSN.

B. PIPO Primitive and Structure

We propose a new PIPO primitive which also assigns each descriptor an eligibility predicate and a rank like PIEO. It, however, relaxes the requirement of dequeuing the smallest-ranked eligible descriptor only. Instead, it allows to pick one

of the small-ranked eligible descriptors. The relaxation makes PIPO more friendly for hardware implementation.

TAS, CQF, and CBS need the eligible-time predicate, for which once a descriptor becomes eligible, it can never turn ineligible again. This inspires us to use two priority queues to stores the ineligible descriptors ordered by the eligible time, and the eligible descriptors ordered by the rank. When a new descriptor is generated, it is pushed into the first queue if it is ineligible, otherwise the second queue; when the head descriptor of the first queue becomes eligible, it is moved into the second queue; when the link is idle, the head packet of the second queue is dequeued. For a TT packet, in the extreme case that its descriptor is located in the first queue, but some descriptors of the AVB flows are ahead of it, the two priority queues may not be able to send it on time. For example, assume there is a descriptor e_{TT} for a TT flow and $n-1$ e_{AVB} for AVB flows ahead of it in the first queue, where n is the length of the queue. Hence, e_{TT} has to wait $n-1$ move operations, which may influence its deterministic forwarding requirement. Although the probability of this reorder issue is relatively low and the moving operation is fast, such delay is intolerable for TT packets in TSN. To solve this problem, we set a dedicated priority queue for TT packets ordered by the eligible time and ensure dequeuing its head at the eligible time for deterministic forwarding.

PAS needs the packet size predicate, for which the eligibility of a descriptor will change as the environment changes with a limited state space. For example, the packet size is between 84 and 1,538 bytes, composed of an Ethernet frame (64 to 1,518 bytes), a Preamble (1 byte), an SFD (7 bytes), and an IFG (12 bytes). Since the space of packet size is limited, we can set a small number of packet-size groups (e.g., four groups covering packet size in 64∼127, 128∼511, 512∼1,023, and 1,024∼1,538 bytes, respectively). In this way, packets can be selected among the eligible groups whose upper limits satisfy the packet size predicate, instead of examining the predicate of all descriptors. Further, if each group is ordered by the rank (i.e., using a priority queue), the smallest-ranked head descriptor among all the eligible groups is more likely to be the smallest-ranked eligible one globally. The detailed analysis is in the next subsection.

The multiple-priority-queue structure which implements the PIPO primitive with the predicate $(t_{\text{eligible}} \le t_{\text{now}} \wedge s \le r)$ is shown in Fig. 7. TT flows use a separate priority queue pq_{TT} ordered by the eligible time. This queue has the highest scheduling priority once the eligible time is reached. For non-TT flow, PIPO adopts the two-level priority queues: the first level is a priority queue pq_0 ordered by the eligible time for descriptors whose eligible time has not arrived yet, the second level comprises m (e.g., $m = 4$ in Fig. 7) priority queues ordered by the rank for dividing descriptors according to packet size. The enqueue and dequeue operations for non-TT packets can be pipelined with two stages as follows.

Stage 1. The newly generated descriptor of a non-TT flow is enqueued to pq_0. If the head descriptor of pq_0 or the new descriptor satisfies $t_{\text{eligible}} \le t_{\text{now}}$, the one with a lower t_{eligible} is dequeued from pq_0.

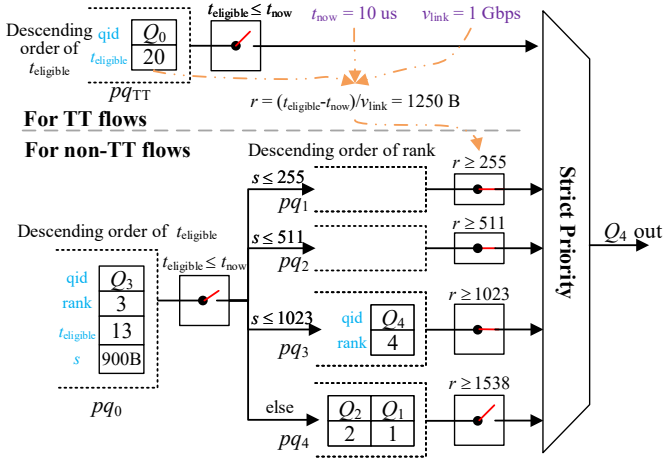


Fig. 7: PIPO structure example: the priority queue pq_{TT} and pq_0 store Q_0 and Q_3, respectively, as their t_{eligible} properties have not arrived yet; and pq_1 to pq_4 store descriptors according to their size. Q_4 is out, and the globally optimal descriptor Q_2 is blocked by Q_1 and pq_4's gate.

Stage 2. The descriptor dequeued from pq_0 is pushed into one of the priority queues among pq_1∼pq_4 according to the packet size property. Meanwhile, the scheduler selects the head descriptor with the smallest rank from the priority queues whose upper limits are not larger than the residual band.

As shown in Fig. 7, since the packet size and eligible time properties of non-TT packets are no longer used in pq_1 to pq_4, they can be omitted to save resources, and so do the packet size and rank properties of TT packets in pq_{TT}. Besides, the residual band r is obtained from the eligible time t_{eligible} of the head descriptor in pq_{TT}, current time t_{now}, and the link rate v_{link}, which is more flexible than the PIEO structure in Fig. 5.

For the PIPO structure, each priority queue can be implemented as a pHeap [8], which supports retrieving the smallest-ranked packet in one cycle and executing enqueue/dequeue operation every two cycles using pipeline. As the pHeap scale increases, we can use BRAMs to store the heap structure to save the register and logic gate resources.

In pHeap, a descriptor is swapped up if it has a lower rank than its counterpart. A possible approach to avoid the starvation problem is to compute a temporary rank for comparison in the heap operation. The temporary rank is based on the original rank and decreases as the delayed time (i.e., $\max\{0, t_{\text{now}} - t_{\text{eligible}}\}$) grows. As a result, the longer a descriptor is delayed, the more likely it is swapped up and scheduled, so the starvation problem is avoided. Note that the eligible time property cannot be omitted in this approach. We leave the detailed design and evaluation for future work.

C. Analysis of PIPO's Approximation Ability

Since PIPO adopts a relaxed scheduling semantics, it cannot guarantee the smallest-ranked eligible scheduling. For the PIPO structure in Fig. 7, the deterministic forwarding of TT packets is guaranteed by an extra highest-priority ordered

queue and the real-time updated residual band r, so we analyze its approximation to PIEO for non-TT packets.

The bandwidth utilization is the key performance indicator for non-TT packets, e.g., AVB, so we focus on the PIPO's approximation degree and bandwidth utilization to the PIEO primitive. We first define the approximation degree α_r^m as the probability that the PIPO structure picks up the same AVB or BE packet as the PIEO primitive when the given residual band is r and the number of priority queues in the second level is m. Let a random variable S denote the packet size in the range of $[S_{min}, S_{max}]$. We limit r in the range of $[0, S_{max}]$, since the packet size predicate does not take effect when $r \geq S_{max}$. The lower and upper bounds of pq_i are L_i and U_i, satisfying $L_1 = S_{min}$, $L_i = U_{i-1} + 1$, and $U_m = S_{max}$. Assume the rank is independent with the packet size.

For a given residual band r, the smallest-ranked eligible descriptor must exist in $pq_1 \sim pq_{K_r}$, where $U_{K_r-1} \leq r \leq U_{K_r}$. Obviously, if $r < U_1$, PIPO will not select any descriptor (i.e., the approximation degree is zero), and if $r = U_{K_r}$, PIPO will certainly select the smallest-ranked eligible descriptor (i.e., the approximation degree is 100%). Otherwise, only if the smallest-ranked eligible descriptor appears in $pq_1 \sim pq_{K_r-1}$, can PIPO choose the same descriptor as PIEO. We assume each descriptor in $pq_1 \sim pq_{K_r-1}$ has the equal probability to be the smallest-ranked eligible one. Let a random variable N' and N_{K_r} denote the total number of descriptors in $pq_1 \sim pq_{K_r-1}$ and pq_{K_r}, respectively, so we can calculate the theoretical α_r^m as follows,

$$\begin{aligned}\alpha_r^m &= \mathbb{E}\left[\frac{N'}{N' + N_{K_r}}\right] \\ &= \sum_{i=1}^{n}\sum_{j=0}^{n-i} \frac{i}{i+j}\binom{n}{i}\binom{n-i}{j} p_1^i p_2^j p_3^{(n-i-j)},\end{aligned} \tag{1}$$

where n is the total number of descriptors, p_1, p_2, and p_3 denote the probability of the packet size drops in the interval $[S_{min}, U_{K_r-1}]$, $[L_{K_r}, r]$, and $[r+1, S_{max}]$, respectively. Next, we give an approximation of Eq. (1), which is confirmed by the simulation results,

$$\alpha_r^m \approx \frac{p_1}{p_1 + p_2} = \frac{\Pr(S < L_{K_r})}{\Pr(S \leq r)}. \tag{2}$$

From Eq. (2) we deduce some conclusions for the approximation degree. First, as r increases in pq_i's coverage, where $i \geq 1$, α_r^m will drop gradually as the denominator increases. Since $\alpha_r^m = 1$ when $r = U_{K_r}$, the curve of the α_r^m will present a jagged upwards trend with a zero start as shown in the simulation results in Section V. Second, more groups means less drops, i.e., a higher average approximation degree can be reached. If we setup a priority queue for each possible packet size, we can ensure the approximation degree to be 1, at the cost of considerable resource consumption and more complex operations to find the smallest-ranked eligible descriptor.

In terms of the residual band utilization of AVB and BE flows, because both PIEO and PIPO select packet only based on the current state, we can resort to Discrete Time Markov Chain (DTMC) to analyse their performance. Let $\mathcal{V}$ and $\mathbf{P}$ denote the state set and one-step state transition matrix. For any state $v \in \mathcal{V}$, r_v is its residual band. If the current state is allowed to be scheduled, we define it as a transition state, and otherwise an absorption state. All transition states constitute a transition state set $\mathcal{T}$, and all absorption states constitute an absorption state set $\mathcal{A}$, so $\mathcal{V} = \mathcal{T} \cup \mathcal{A}$. The one-step state transition matrix $\mathbf{P}$ records any one-step transition probability from state v to v', denoted by $p_{v,v'} = \Pr(V_{t+1} = v' | V_t = v)$. According to the properties of DTMC, we can use one-step state transition matrix $\mathbf{P}$ to obtain the limit of the n-step transition matrix $\hat{\mathbf{P}} = \lim_{n\to\infty} \mathbf{P}^n$. We can use $\hat{\mathbf{P}}$ to calculate the expectation of the wasted band W under an initial state v as follows

$$\mathbb{E}[W | r_v] = \sum_{v' \in \mathcal{A}} r_{v'} \hat{p}_{v,v'}. \tag{3}$$

The utilization of an given residual band r_v is thus $1 - \frac{\mathbb{E}[W|r_v]}{r_v}$. Next, we only need to clarify the $< \mathcal{V}, \mathbf{P} >$ of PIEO and PIPO.

It is necessary to make it clear that only two cases of state transition exist: case ①, from a transition state v to a transition state v', whose corresponding situation is that the size of the scheduled packet is $r_v - r_{v'}$; case ②, from a transition state v to an absorption state v', whose corresponding situation is that there is no eligible packet, i.e., $r_{v'} = r_v$.

For PIEO, if all the n descriptors' packet size predicates are false in the current state, it corresponds to case ②. Otherwise, it corresponds to case ①, and the probability of finding a packet with size $r_v - r_{v'}$ is $\frac{\Pr(S = r_v - r_{v'})}{\Pr(S \leq r)}$. Therefore, we have

$$p_{v,v'}^{\text{PIEO}} = \begin{cases} \{1 - [\Pr(S > r_v)]^n\} \dfrac{\Pr(S = r_v - r_{v'})}{\Pr(S \leq r_v)}, & \text{if ①;} \\ [\Pr(S > r_v)]^n, & \text{if ②;} \\ 0, & \text{else.} \end{cases} \tag{4}$$

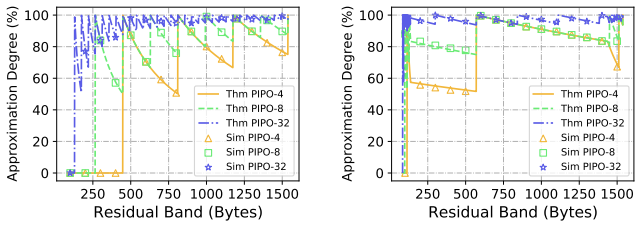
There are two differences between PIPO and PIEO: first, if all packets satisfy $S > y_v$ (where $y_v = r_v$ if $r_v = U_{K_{r_v}}$ and $y_v = U_{K_{r_v}-1}$ otherwise), no packet will be scheduled and thus the next state will be an absorption state; Second, the probability of the dequeued packet with size $r_v - r_{v'}$ is $\frac{\Pr(S = r_v - r_{v'})}{\Pr(S \leq y_v)}$, Therefore, we have

$$p_{v,v'}^{\text{PIPO}} = \begin{cases} \{1 - [\Pr(S > y_v)]^n\} \dfrac{\Pr(S = r_v - r_{v'})}{\Pr(S \leq y_v)}, & \text{if ①;} \\ [\Pr(S > y_v)]^n, & \text{if ②;} \\ 0, & \text{else.} \end{cases} \tag{5}$$

From Eq. (4) and (5), we conclude that PIEO is more likely to trap into the absorption states, and as m increases, $\Pr(S > y_v)$ is approaching $\Pr(S > r_v)$, so does the approximation of PIPO to PIEO.

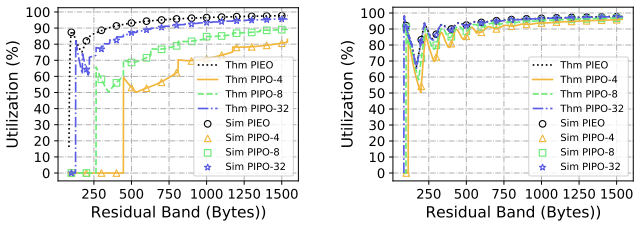
V. Evaluation

In this section, we first show that the PIPO structure can approximate PIEO more closely if it adopts more priority queues. Second, we implement the PIPO structure and compare its performance with other implementation of the PIEO primitive. The results on Xilinx FPGA show that the PIPO prototype has the highest stable scheduling throughput and the lowest resource consumption as the number of descriptors grows.



(a) Uniform distribution (b) Caida distribution

Fig. 8: Theoretical and simulation approximation degree of PIPO-m on different packet-size distributions.



(a) Uniform distribution (b) Caida distribution

Fig. 9: Theoretical and simulation residual band utilization of PIPO-m, and PIEO on different packet-size distributions.

A. Simulation

1) Setup: We first evaluate the approximation degree and bandwidth utilization of PIPO to PIEO, while the PIPO structure adopts m priority groups in its second level, denoted by PIPO-m, where $m = 4, 8, 32$. To reduce the impact of randomness, each result value is the average of 10K tests.

The input set contains enough packets buffered in 256 per-flow queues with random rank and packet size[2]. The rank obeys the uniform distribution in $\{0, 1, \ldots, 1023\}$, and the packet size obeys two typical distributions: 1) uniform distribution in $\{84, 85, \ldots, 1538\}$, and 2) Caida distribution of [37] as shown in Table I.

TABLE I: Caida distribution of the packet size

Size interval	[84, 138)	[138, 1438)	[1438, 1538)	1538
Probability	0.45	0.15	0.2	0.2

2) Approximation Degree: Fig. 8 shows the theoretical and simulation approximation degree of PIPO-m on the two typical packet-size distributions as the given residual band increases. First, the theoretical results based on Eq. (2) match the simulation well. Second, PIPO achieves a higher approximation degree as m increases, which is consistent with our expectation that adopting more priority queues helps to approach PIEO. The curves of PIEO-m display a jagged upward trend, which complies with Eq. (2). Adopting more groups makes the curve more smooth. The distribution of packet size also influences the approximation degree of PIPO-m. Our grouping method makes each priority queue have the equal expected number of descriptors, so more priority queues are used at the two ends of the Caida distribution. As shown in Fig. 8(b), the serrations are flattened and the range of the zero start is reduced significantly on the Caida distribution, achieving a higher approximation degree on average.

Although we cannot guarantee the dequeued descriptor from PIPO to be the smallest-ranked eligible one, we assert it is not worse than that from PIEO for three reasons: first, PIEO expresses the greedy PAS rather than the ideal PAS which achieves the theoretical highest bandwidth utilization for AVB and BE flows, making PIEO's decision not globally optimal; second, PIPO prefers to select smaller packets which are often more critical; third, not selecting the smallest-ranked eligible packet can mitigate the starvation problem to some extent.

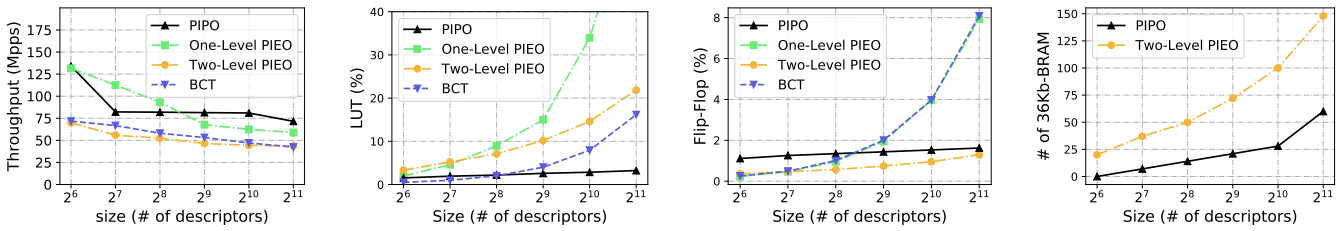
3) Residual Band Utilization: Fig. 9 shows the theoretical and simulation residual band utilization as the given residual band increases. The theoretical models of Eq. (4) and (5) fit the simulation well. PIPO-m still displays a jagged upward trend, but the fluctuation is lighter. Similarly, a larger m helps to realize higher utilization, and the gaps among different m narrow on the Caida distribution due to the higher proportion of small packets. Some curves represented by PIEO have high utilization when the given residual band is small, e.g., 100 Bytes. This comes from two reasons: first, the number of descriptors is 256, so the probability of finding a packet smaller than 100 bytes is high; second, packets must be larger than 84 bytes in our setting, i.e., one selected packet can almost fill the residual band. Third, PIPO-m still performs better on the Caida distribution due to flexible grouping granularity.

Based on the above results, PIPO-4 achieves acceptable performance on the Caida distribution with about 79% approximation degree to PIEO, and 76% bandwidth utilization in contrast to 82% of PIEO, if the given residual band obeys the uniform distribution. It is close to PIPO-32 (about 97% and 81%, respectively) but needs much fewer pHeaps. As the Caida distribution is close to the packet distribution in real networks, we set $m = 4$ in our final implementation of the PIPO structure for a tradeoff between performance and cost.

B. Implementation

We implement PIPO using the hardware priority queue, pHeaps, and compare it with three PIEO implementations: the one-level, the two-level, and the Binary Comparison Tree (BCT) [18]. The PIPO prototype, as shown in Fig. 7, uses two pHeaps pq_{TT} and pq_0 ordered by the eligible time, and four pHeaps pq_1 to pq_4 ordered by the rank with the grouping method of PIPO-4 on the Caida distribution. The one-level PIEO prototype adopts the shift-register-based structure in Fig. 5 without using BRAM. It needs a shift register of length n, three parallel comparators of length n (one for enqueue and two for dequeue), and a priority encoder of length n, where n is the number of descriptors. The two-level PIEO prototype, based on the open-source code from [38], reduces the register scale to $2\sqrt{n}$ but it costs more stages for the enqueue/dequeue operation. To make it comparable to PIPO and guarantee TT

[2]The simulation results show that when the number of per-flow queues exceeds 16, the fluctuation of approximation degree and residual band utilization are controlled within a few tenth of a percent.



(a) Maximum Throughput. (b) Percentage of LUT consumed (c) Percentage of Flip-flop consumed (d) Number of BRAM consumed

Fig. 10: Performance comparisons of the PIPO, one-level PIEO, two-level PIEO and BCT prototype.

flow scheduling correctness, we add a pHeap pq_{TT} to it also and maintain the largest size and eligible time for each sub-list. BCT, another implementation of PIEO, first adopts two n-long parallel comparators to filter the descriptors whose eligible time and packet size predicates are true, and then uses a binary tree with $\log n$ layers to get the final descriptor.

The PIPO prototype is written in Scala [39], comprising 600 LOCs, and then is compiled into Verilog codes by Chisel [40]. The one-level PIEO, two-level PIEO, and BCT are written in System Verilog [41], comprising ∼400, ∼1,000 and ∼400 LOCs, respectively. As the baseline, the bit width of the rank, packet size and eligible time is 16, 12 and 16, respectively. The four prototypes are implemented on a Xilinx Virtex-7 [42] FPGA comprising 712K LUT, 1,424K flip-flops, and 1,880 36Kb dual-port BRAMs.

1) Throughput: The most important performance indicator for a TSN scheduler is throughput. As shown in Fig. 10(a), the PIPO prototype achieves the highest throughput when the number of descriptors is large. It achieves more than 70 Mpps throughput for 2,048 descriptors, easily supporting 40Gbps small-packet link-rate forwarding, which is 1.21, 1.64, and 1.68 times faster than the one-level PIEO, two-level PIEO, and BCT prototypes, respectively. PIPO has a drop when the number of descriptors reaches 128. This is because we implement the first five layers of pHeap (i.e., up to 64 descriptors) in faster Distributed RAMs (i.e., LUTs). Thanks to the scalability of pHeap, the curve of PIPO is flat for more descriptors than 128. Although PIPO can run at the highest clock frequency, pHeap needs two clocks for a new enqueue/dequeue operation in pipeline, which halves the throughput.

The one-level PIEO prototype performs well for a small number of descriptors, but the throughput plunges as the descriptors increase, and it lags behind PIPO after 256 descriptors. This is because the on-chip net delay and priority encoder delay increase as the descriptor size expands. The throughput drop of the two-level PIEO prototype is slow because it shrinks the descriptor size from n to $2\sqrt{n}$. As mentioned above, its four-cycle enqueue/dequeue operation is hard to be pipelined, making the throughput low. The BCT prototype cannot be pipelined either and has a similar throughput trend as the two-level PIEO, though its throughput drop is mainly caused by the increase in the depth of the binary tree.

2) Resource consumption: Fig. 10(b) and 10(c) show the LUT and flip-flop consumption of the four prototypes. The PIPO prototype consumes less than 1% flip-flops than the best BCT and the two-level PIEO when the descriptor scale is small. For more than 512 descriptors, it consumes the fewest LUTs and the second fewest flip-flops, thanks to the fact that some layers of pHeap are implemented in BRAMs. Although the two-level PIEO consumes the fewest flip-flops, its LUT consumption is large, e.g., 21.8% under 2048 descriptors in contrast to 3.2% for the PIPO prototype (i.e., the latter is only 14.7% of the former). This is because it still needs a shift register to store $2\sqrt{n}$ descriptors, which consumes LUTs. Recall that the one-level PIEO outperforms PIPO on throughput when the number of descriptors is no more than 256, but its resource consumption is unscalable. As both LUT and flip-flop are resources in *slice*, the smallest logic unit of Xilinx FPGA, the slice consumption of a two-level PIEO implementation is much higher than that of a PIPO one.

The one-level PIEO and BCT prototypes do not use BRAMs, so their consumption on LUT and filp-flop does not scale well. Fig. 10(d) shows the BRAM consumption of the PIPO and the two-level PIEO prototypes. The BRAM consumption of PIPO is less than half of the two-level PIEO, e.g., 60 and 148 BRAMs under 2,048 descriptors (i.e., the former is only 40.5% of the latter). The BRAM consumption is related to the number of pHeaps, so using more pHeaps to improve the approximation degree costs more BRAMs.

The pHeap settings have a big impact on resource consumption for the PIPO prototype. For example, if the first sixth layers of pHeap are implemented in LUTs and flip-flops, PIPO's LUT and flip-flop consumption will increase by 23% and 69% under 2,048 descriptors, respectively, but its BRAM consumption will drop from 60 to 48; if fewer than five layers are implemented in LUTs and flip-flops, its flip-flop consumption will be smaller than that of the two-level PIEO.

VI. Conclusion

Scheduling is the core component of TSN. The diversified algorithms keep pushing the boundaries of the implementation's expressiveness, performance, and cost. We propose a programmable scheduling framework for TSN and demonstrate its expressiveness on the TSN scheduling algorithms. The new PIPO primitive approximates PIEO and its implementation supports higher throughput and lower cost. Through analysis, simulation, and prototype, we show that the PIPO-based scheduler meets the TSN requirements, and outperforms other approaches on throughput and cost.

REFERENCES

[1] "IEEE standard for local and metropolitan area networks–bridges and bridged networks - amendment 25: Enhancements for scheduled traffic," *IEEE Std*, pp. 1–57, Mar. 2016.

[2] "IEEE standard for local and metropolitan area networks–bridges and bridged networks - amendment 29: Cyclic queuing and forwarding," *IEEE Std*, pp. 1–30, 2017.

[3] "IEEE standard for local and metropolitan area networks - bridges and bridged networks amendment: Asynchronous traffic shaping," *IEEE Std*, Nov. 2018. [Online]. Available: https://1.ieee802.org/tsn/802-1qcr/

[4] A. Saeed, Y. Zhao *et al.*, "Eiffel: Efficient and flexible software packet scheduling," in *Proceedings of USENIX NSDI*, 2019, pp. 17–32.

[5] A. Sivaraman, S. Subramanian *et al.*, "Programmable packet scheduling at line rate," in *Proceedings of ACM SIGCOMM*, 2016, pp. 44–57.

[6] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," in *SIGCOMM Computer Communication Review*, vol. 19, no. 4. ACM, 1989, pp. 1–12.

[7] V. Shrivastav, "Fast, scalable, and programmable packet scheduler in hardware," in *Proceedings of ACM SIGCOMM*, 2019, pp. 367–379.

[8] R. Bhagwan and B. Lin, "Fast and scalable priority queue architecture for high-speed network switches," in *Proceedings of IEEE INFOCOM*, vol. 2, 2000, pp. 538–547.

[9] M. Shreedhar and G. Varghese, "Efficient fair queuing using deficit round-robin," *TON*, no. 3, pp. 375–385, 1996.

[10] J. C. Bennett and H. Zhang, "WF^2Q: Worst-case fair weighted fair queueing," in *Proceedings of IEEE INFOCOM*, vol. 1, 1996, pp. 120–128.

[11] P. E. McKenney, "Stochastic fairness queueing," in *Proceedings of IEEE PINFOCOM*, 1990, pp. 733–740.

[12] S. Radhakrishnan, Y. Geng *et al.*, "SENIC: Scalable nic for end-host rate limiting," in *Proceedings of USENIX NSDI*, 2014, pp. 475–488.

[13] A. Saeed, N. Dukkipati *et al.*, "Carousel: Scalable traffic shaping at end hosts," in *Proceedings of ACM SIGCOMM*, 2017, pp. 404–417.

[14] [Online]. Available: https://en.wikipedia.org /wiki/Token_bucket

[15] R. Mittal, R. Agarwal *et al.*, "Universal packet scheduling," in *Proceedings of USENIX NSDI*, 2016, pp. 501–521.

[16] A. G. Alcoz, A. Dietmüller, and L. Vanbever, "SP-PIFO: Approximating push-in first-out behaviors using strict-priority queues," in *Proceedings of USENIX NSDI*, 2020.

[17] Z. Yu, C. Hu *et al.*, "Programmable packet scheduling with a single queue," in *Proceedings of ACM SIGCOMM*, 2021, pp. 179–193.

[18] S. Moon, J. Rexford, and K. G. Shin, "Scalable hardware priority queue architectures for high-speed packet switches," *TOC*, vol. 49, no. 11, pp. 1215–1227, 2000.

[19] A. Ioannou and M. G. Katevenis, "Pipelined heap (priority queue) management for advanced scheduling in high-speed networks," *ToN*, vol. 15, no. 2, pp. 450–461, 2007.

[20] N. K. Sharma *et al.*, "Programmable calendar queues for high-speed packet scheduling," in *Proceedings of USENIX NSDI*, 2020, pp. 685–699.

[21] R. Brown, "Calendar queues: a fast O(1) priority queue implementation for the simulation event set problem," *Communications of The ACM*, vol. 31, no. 10, pp. 1220–1227, 1988.

[22] H. Zhang and D. Ferrari, "Rate-controlled service disciplines," *Journal of High Speed Networks*, vol. 3, no. 4, pp. 389–412, 1994.

[23] "IEEE standard for local and metropolitan area networks—virtual bridged local area networks amendment 12: Forwarding and queuing enhancements for time-sensitive streams," *IEEE Std*, pp. 1–71, Jan. 2009.

[24] "IEEE standard for local and metropolitan area networks – bridges and bridged networks - amendment 26: Frame preemption," *IEEE Std*, pp. 1–52, Aug. 2016.

[25] "IEEE standard for ethernet amendment 5: Specification and management parameters for interspersing express traffic," *IEEE Std*, pp. 1–58, Oct. 2016.

[26] W. Steiner, "An evaluation of SMT-based schedule synthesis for time-triggered multi-hop networks," in *Proceedings of IEEE Real-Time Systems Symposium*, 2010, pp. 375–384.

[27] S. S. Craciunas, R. S. Oliver, M. Chmelík, and W. Steiner, "Scheduling real-time communication in IEEE 802.1 Qbv time sensitive networks," in *Proceedings of ACM RTNS*, 2016, pp. 183–192.

[28] R. S. Oliver, S. S. Craciunas, and W. Steiner, "IEEE 802.1 Qbv gate control list synthesis using array theory encoding," in *Proceedings of IEEE RTAS*, 2018, pp. 13–24.

[29] D. Thiele and R. Ernst, "Formal worst-case performance analysis of time-sensitive ethernet with frame preemption," in *Proceedings of IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2016, pp. 1–9.

[30] C. Zhang, Y. Wang *et al.*, "Packet-size aware scheduling algorithms in guard band for time sensitive networking," *CCF Transactions on Networking*, pp. 1–17, 2020.

[31] N. Samphaiboon and Y. Yamada, "Heuristic and exact algorithms for the precedence-constrained knapsack problem," *Journal of optimization theory and applications*, vol. 105, no. 3, pp. 659–676, 2000.

[32] J. Yan *et al.*, "Injection time planning: Making cqf practical in time-sensitive networking," in *Proceedings of IEEE INFOCOM*, 2020.

[33] P. Goyal, H. M. Vin, and H. Cheng, "Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks," *IEEE/ACM TON*, vol. 5, no. 5, pp. 690–704, 1997.

[34] M. Alizadeh, S. Yang *et al.*, "pFabric: Minimal near-optimal datacenter transport," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 435–446, 2013.

[35] C. Hu, Y. Tang *et al.*, "Per-flow queueing by dynamic queue sharing," in *Proceedings of IEEE INFOCOM*, 2007, pp. 1613–1621.

[36] A. Sivaraman, A. Cheung *et al.*, "Packet transactions: High-level programming for line-rate switches," in *Proceedings of ACM SIGCOMM*, 2016, pp. 15–28.

[37] Packet size distribution. [Online]. Available: https://www.caida.org/research/traffic-analysis/pkt-size-distribution/graphs.xml

[38] PIEO codes. [Online]. Available: https://github.com/vishal1303/PIEO-Scheduler

[39] Scala. [Online]. Available: https://www.scala-lang.org

[40] Chisel/FIRRTL. [Online]. Available: https://www.chisel-lang.org

[41] [Online]. Available: https://en.wikipedia.org/wiki/SystemVerilog

[42] Virtex-7. [Online]. Available: https://www.xilinx.com/products/silicon-devices/fpga/virtex-7.html