Programmable Calendar Queues for High-speed Packet Scheduling

Naveen Kr. Sharma[*] *Chenxingyu Zhao*[*] *Ming Liu*[*] *Pravein G Kannan*[†] *Changhoon Kim*[‡]
Arvind Krishnamurthy[*] *Anirudh Sivaraman*[§]

Abstract

Packet schedulers traditionally focus on the prioritized transmission of packets. Scheduling is often realized through coarse-grained queue-level priorities, as in today's switches, or through fine-grained packet-level priorities, as in recent proposals such as PIFO. Unfortunately, fixed packet priorities determined when a packet is received by the traffic manager are not sufficient to support a broad class of scheduling algorithms that require the priorities of packets to change as a function of the time it has spent inside the network. In this paper, we revisit the Calendar Queue abstraction and show that it is an appropriate fit for scheduling algorithms that not only require prioritization but also perform dynamic escalation of packet priorities. We show that the calendar queue abstraction can be realized using either dataplane primitives or control-plane commands that dynamically modify the scheduling status of queues. Further, when paired with programmable switch pipelines, we can realize programmable calendar queues that can emulate a diverse set of scheduling policies. We demonstrate the power of this abstraction using three case studies that implement variants of LSTF, Fair Queueing, and pFabric in order to provide stronger delay guarantees, burst-friendly fairness, and starvation-free prioritization of short flows, respectively. We evaluate the benefits associated with these scheduling policies using both a custom simulator and a small-scale testbed.

1 Introduction

Many network scheduling algorithms today require a notion of *dynamic priority*, where the priority of individual packets within a flow varies over the lifetime of the flow. These packet priorities generally change with either the amount of bytes sent by the flow, how fast the flow is transmitting, or time spent by the packet inside the network. Such scheduling algorithms enable richer application-level prioritization and performance guarantees, such as shortest-job first to minimize average flow completion time, earliest deadline first to enable timely delivery of all messages, or fair-queueing, as illustrated by a long list of proposed scheduling algorithms (e.g., pFabric [4], PIAS [5], WFQ [14], FIFO+ [13], LSTF [17], and EDF [18]).

Switch-level support for multiple fine-grained priority levels (as in PIFO [28], pHeap [7]) can aid the realization of these scheduling algorithms. However, there are still several challenges in faithfully implementing these algorithms efficiently and at line rate. First, implementing strict and fine-grained priority levels is expensive, especially at scale involving high bandwidths and hundreds or thousands of unique flows at multiple terabits per second. Second, and more crucially, existing switch support for priorities does not allow for dynamic changes to the priority of a packet during its stint inside the switch buffer. Fixed packet priorities cannot effectively emulate the *ageing* property required by many of the scheduling algorithms, wherein the priority of a packet increases with the time it spends inside a queue.

Consider, for instance, the Least Slack Time First (LSTF [17]) scheduling discipline wherein each packet maintains a delivery deadline, and the switch emits from its buffer the packet with the least slack at a given instant. LSTF cannot be realized using fixed packet priorities that are determined when the packets are inserted into a priority queue. Notably, given a packet that has a deadline of $current\_time + slack$, a switch scheduler that maps this deadline to a priority level would quickly exhaust a *finite* number of priority levels. As long as there are packets buffered inside the switch, packets received with later deadlines would have to be progressively assigned lower priority levels, and the switch will eventually run out of priority levels to use for incoming packets.

In this paper, we argue that what is needed is a scheduling mechanism that supports both prioritization and implicit escalation of a packet's priority as it spends more time inside the switch buffer. We observe that a mechanism similar to Calendar Queues [10] would be a more appropriate fit for implementing these scheduling algorithms. Calendar Queues allow events (or packets) to be enqueued at a priority level or rank corresponding to a future time, and this rank gradually changes as time moves forward. A scheduling algorithm simply decides how far in the future a packet must be processed and then controls how time is advanced, say based on the switch's physical clock (i.e., *physical time*) or the number of communication rounds across flows (i.e., *logical time*).

Calendar queues have certain properties that make them amenable to efficient hardware realization, especially on upcoming programmable switch hardware. A calendar queue consists of multiple physical switch queues, and at any point in time, only one of the queues in a calendar queue is active. Further, a calendar queue imposes a fixed rotation order for activating queues. We describe how the activation of queues in a fixed order can be achieved by periodically modifying the priority and active status of queues, either through dataplane primitives expected in future programmable switches or through today's control-plane operations, albeit at a higher

[*]University of Washington
[†]School of Computing, National University of Singapore
[‡]Barefoot Networks
[§]NYU

latency. When combined with the stateful and flexible packet processing capabilities of a programmable switch (such as Barefoot's Tofino and Cavium's Xpliant), we can customize the calendar queue abstraction to realize a broad class of scheduling algorithms that capture both physical and logical notions of time.

We demonstrate the power and flexibility of the calendar queue abstraction using three case studies. First, we use a calendar queue to perform deadline-aware scheduling of aggregate flows (or co-flows) and further use the same underlying calendar queue to implement fair queuing for the background traffic. The programmable switch pipeline performs the accounting operations required to keep track of the slack available for a packet and computes its rank as it traverses a network path. In our second case study, we use calendar queues to implement fair queueing and its variants that can tolerate a limited amount of burstiness, thereby providing a configurable balance between fairness and burst-friendliness. Here, the programmable switch pipeline maintains the flow state to perform the accounting operations required for scheduling. For our third case study, we realize pFabric and its variant that can provide a configurable amount of starvation prevention. We use a programmable pipeline to ensure the in-order transmission of packets even as the switch attributes higher priorities to later packets transmitted within a flow. We use a small-scale testbed comprising of Barefoot Tofino switches and a custom event-driven simulator to evaluate the benefits of these different scheduling algorithms.

2 Background

In this section, we discuss background material related to reconfigurable switches, the structure of the traffic manager (which is the same for both fixed-function and today's reconfigurable switches), and prior work on programmable packet scheduling.

2.1 Reconfigurable Switches

In this work, we assume that this programmable scheduling is used in conjunction with a reconfigurable switch, such as the Reconfigurable Match Table (RMT) model described in [8, 9]. A reconfigurable switch can operate at terabits/s by providing a restricted match+action (M+A) processing model: match on arbitrary packet header fields and perform simple packet processing actions. When a packet arrives at the switch, relevant headers fields are extracted via a user-defined parser and passed into a pipeline of user-defined M+A stages. Each stage matches on a subset of extracted headers and performs simple processing primitives or actions on any header. After traversing the ingress pipeline stages, packets are deposited in one of the multiple queues, typically 32–64, associated with the egress port for future transmission. On transmission, the packet goes through a similar egress pipeline undergoing further modifications.

In addition to a programmable parser and pipeline stages, these switches provide several hardware features to implement more complex use-cases: (1) a limited amount of stateful memory, such as counters and meters, which can be read and updated to maintain state across packets, (2) computation primitives, such as simple arithmetic operations and hashing, which can perform a limited amount of processing on header fields and data retrieved from stateful memory, and (3) the ability to recirculate or generate special datapath packets using *timers* that can be used to modify Traffic Manager status as well as synchronize the ingress and egress pipelines. Further, switch metadata, such as queue lengths, congestion status, and bytes transmitted, can also be used in packet processing. The complete packet processing, including parsing and match+action stages, can be configured using a high-level language, such as P4 [8]. A number of such switches, e.g., Cavium XPliant [11], Barefoot Tofino [6] and Intel Flexpipe [20], are available today.

A single pipeline's throughput is limited by the clock frequency achievable using today's transistor technology (typically about 1 GHz). To scale to higher packet-processing rates, which is required for switches with aggregate switch capacity in the Tbit/s range, the switch consists of multiple identical pipelines where the programmable stateful memory is local to each pipeline.

2.2 Traffic Manager

We now briefly describe the architecture of the traffic manager (TM) on merchant-silicon switches (e.g., Barefoot's Tofino and Broadcom's Trident series). The TM is responsible for two tasks: (1) buffering packets when more than one input port is trying to send packets to the same output port simultaneously, and (2) scheduling packets at each output port when the link attached to the port is ready to accept another packet.

Buffering: The TM is organized as a fixed number of first-in, first-out (FIFO) queues per output port. The ingress pipeline of the switch is responsible for determining both the FIFO queue and the output port that the packet should go to. Once the packet exits the ingress pipeline, the TM checks if there is sufficient space in the packet buffer to admit the new packet. Once the packet has been admitted to the buffer, it can not be dropped later because dropping a previously enqueued packet requires additional memory accesses to the packet buffer, which is expensive at line rate.

Scheduling: Packets are eventually dequeued from the buffer when the link attached to an output port goes idle and requests a new packet. During dequeue, the TM has to pick a particular FIFO at that output port, remove the earliest packet at that queue, and transmit it. The TM uses a combination of factors to determine which queue to dequeue from. First, each queue has a priority; queues with higher priority are strictly preferred to those with lower priorities. Next, within a priority level, the queues are scheduled in weighted round-round robin order (using an algorithm like DRR [26]). Lastly, each queue can be limited to a maximum rate and is paused and removed

from consideration for scheduling if it has exceeded this rate over some time interval. Queues can also be paused because of a PFC [16] pause frame from a downstream switch.

To perform buffering and scheduling, the TM maintains a per-queue priority level, a per-queue pause status flag, and counters that track buffer occupancy on a per-input-port, per-PFC-class, per-output-port, and per-output-FIFO basis. These are required both to decide when and whether to admit packets and which queues to schedule. Because the status flag and counters support limited operations (e.g., toggling or increment/decrement), they can be implemented very efficiently in hardware, allowing simultaneous access from multiple ingress and egress switch pipelines in a single clock cycle (unlike state within the pipeline that can only be accessed once per clock cycle). It is important to note that these hardware mechanisms appear in traffic managers in both fixed-function and programmable switches.

2.3 Programmable Scheduling

The discussion above focused on a fixed-function TM that supports a small menu of scheduling algorithms (typically priorities, weighted round-robin, and traffic shaping). Recent proposals for programmable scheduling [2, 19, 23, 27–29] propose additional switch hardware in the TM to make the scheduling decision programmable, assuming the existence of a programmable ingress and egress switch pipeline like RMT. Of the proposals for programmable scheduling, we describe the PIFO work because it targets a switch similar to our paper, and it is representative of the hardware considerations associated with programmable scheduling. We defer a detailed comparison of both expressiveness and feasibility with prior work to later sections.

PIFOs enable programmable scheduling by using a programmable priority queue to express custom scheduling algorithms. Some external computation (either on the end host or a programmable switch's ingress pipeline) sets a rank for the packet. This rank determines the packet's order in the priority queue. By writing different programs to compute different kinds of packet ranks (e.g., deadlines or virtual times), different scheduling algorithms can be expressed using PIFOs.

While PIFOs are flexible, they have two shortcomings. PIFOs not only require the development of new hardware blocks that can scale with line rate (as we discuss below), but they are also limited given their support for a finite priority range (as we discuss in the next section). The scaling challenge arises out of needing to maintain a priority queue. The PIFO paper assumes that ranks within flows are naturally in strictly increasing order (i.e., flows are FIFOs), requiring the switch to only find the minimum rank among the head packets across all flows. While this reduces the sorting/ordering requirement of PIFO, sorting the total number of flows in the buffer is still challenging. The PIFO work provides a custom hardware primitive, the flow scheduler, which maintains a sorted array of a few thousand flows and can process tens of flows per output port across about 64 output ports on a single pipeline for an aggregate throughput of 640 Gbit/s. Scaling this primitive to higher speeds and a multi-pipeline switch can be challenging. Thus, a key goal of our programmable scheduling proposal is that it should be realized without increasing the temporal and spatial complexity of existing TM implementations.

3 Packet Scheduling using Programmable Calendar Queues

In this section, we begin by describing the Calendar Queue concept as introduced by Randy Brown in 1988 [10]. We then consider the abstraction of a Programmable Calendar Queue that combines the calendar queue scheduling mechanism with programmable packet processing pipelines. This combination allows for flexibility and extensibility, and we show how we can instantiate different variants of Programmable Calendar Queues to emulate different scheduling algorithms that appear in the literature.

3.1 Motivating Calendar Queues

Calendar Queues: The Calendar Queue was first introduced for organizing the pending event set in a discrete event simulation. It is a type of priority queue implementation that has low insertion and removal costs for certain priority distributions. Calendar Queues (CQs) are analogous to desk calendars used for storing future events for the next year in an ordered manner. A CQ consists of an array of buckets or queues, each of which stores events for a particular *day* in sorted order. Events can be scheduled for a future date by inserting the event in the bucket corresponding to the date. At any point, events are dequeued and processed from the current day in sorted order. Once all events are processed from the current day, we stop processing events for the current day and move onto the next day. The emptied bucket is then used to store tasks that need to be performed a year from now.

Drawbacks of existing priority queueing schemes: Prior work has made the observation that, scheduling algorithms make two decisions: in what order packets should be scheduled (in the case of work-conserving algorithms) or when they should be scheduled (in the case of non-work-conserving algorithms). For most scheduling algorithms, these decisions can be made at packet enqueue time. Comparison-based fine-grained priority queueing schemes, such as pHeap [7] or PIFO [28], can realize some of these algorithms by computing an immutable *rank* for a packet at packet enqueue time and dequeuing packets in increasing rank order. Eiffel [23] further observes that packet ranks often have a specific range (which can be expressed as integers) and that a large number of packets share the same rank. These characteristics make a bucket-based priority queue an efficient and feasible solution for implementing various scheduling algorithms.

We observe that many scheduling algorithms cannot be realized using fine-grained priority queuing schemes if the

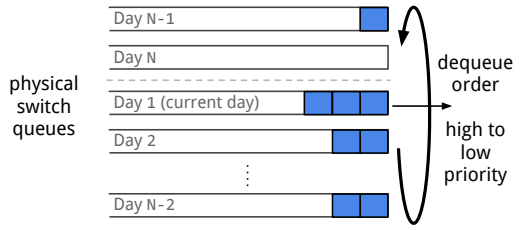

Figure 1: Example of a Programmable Calendar Queue.

computed rank needs to fall within a finite range. Consider the example of fair queuing (as in WFQ or STFQ), where for each arriving packet, a finishing round is computed based on the current round number and the finishing round of the previous packet in that packet's flow. Packets are then transmitted in order of increasing finishing round numbers. Further, the algorithm periodically increases the current round number. One could attempt to realize fair queuing using a fine-grained priority queuing scheme by mapping a packet's finishing round number to an immutable rank. Since the ranks of buffered packets cannot be changed, the mapping function needs to be monotonic, i.e., it needs to map higher finishing rounds to higher ranks. The mapping function would then exhaust any finite range of ranks, and the switch would then not be able to attribute a meaningful rank for incoming packets.

Similarly, in the case of the earliest deadline first (EDF), each packet in a flow is associated with a wall-clock deadline, and packets need to be scheduled in increasing order of deadlines. If one were to compute the rank of a packet as a monotonic function of the packet's deadline, the switch would exhaust the rank space as the wall-clock time progresses.

It is worth noting that, when the switch has no buffered packets, the mapping function could execute a "reset" and start reusing lower ranks. However, an implementation cannot assume that the switch would ever enter such a state, let alone periodically (i.e., within a bounded period of time before the rank space is exhausted).

Utility of Calendar Queues: We propose to use the Calendar Queues abstraction as a mechanism to realize scheduling algorithms such as EDF and fair queuing. A CQ is an attractive option for implementing these algorithms as it allows for implicit and en-masse escalation of the priorities of buffered packets when the CQ moves from one day to another. For instance, when a CQ completes processing the events for Day k and performs a *rotation* to Day $k+1$, it implicitly increases the priority of all days except Day k, which now occupies the lowest priority in the priority range. This rotation mechanism allows scheduling algorithms to escalate the priorities of buffered packets with time (as is the case with EDF and fair queuing) and reuse emptied buckets for incoming packets with low priority.

3.2 Programmable Calendar Queues (PCQs)

We now describe Programmable Calendar Queues in the context of reconfigurable switches. The programmable packet processing pipelines on these switches allow us to customize not only the rank computation but also the CQ rotation process. Just like a calendar has 365 days, we assume our Calendar Queue abstraction has a fixed number of buckets or FIFO queues, say N, each of which stores packets scheduled for next N periods (see Figure 1). Any network scheduling algorithm using CQs must then make the following key decisions. First, the scheduling algorithm must decide how far in the future the incoming packet should be scheduled, i.e., choose a future period from [0, N-1] to enqueue the packet into. This is similar to rank computation in PIFO. Second, it must periodically decide when and how to advance time, i.e., decide when a period is over and move onto the next period. This stops the enqueueing of packets in the current period and allows the reuse of the corresponding queue resource for the period that is N periods into the future. Third, when the CQ advances to the next period, the pipeline state has to be suitably modified to ensure the appropriate computation of ranks for incoming packets.

The advancing of time can be done using a physical clock, i.e., the CQ moves onto the next queue after a fixed time interval periodically; we call this a *Physical* Calendar Queue. Alternatively, the CQ can advance to the next queue whenever the current queue is empty, i.e., it happens logically depending on metrics such as bytes sent or number of communication rounds; we call this a *Logical* Calendar Queue. A Physical Calendar Queue lets us implement both work-conserving schemes, such as EDF, and non-work-conserving schemes, such as Leaky Bucket Filter, Jitter-EDD, and Stop-and-Go, whereas a Logical Calendar Queue can implement work-conserving schemes, such as LSTF, WFQ, and SRPT.

We now list the interface methods exposed to the packet processing pipelines that enable these forms of customization.

- **CQ.enqueue(n)**: Used by the ingress pipeline to schedule the current packet n periods into the future.
- **CQ.dequeue()**: Used by the egress pipeline to obtain a buffered packet, if any, for the current period.
- **CQ.rotate()**: Used by the pipelines to advance the CQ so that it can start transmitting packets for the next period.

We observe that PCQs have certain properties that allow for efficient implementations. (In Section 3.4, we describe how to realize this abstraction in hardware.) When individual CQ periods are mapped to separate physical switch queues, a CQ scheduler needs to maintain state only at the granularity of switch queues (e.g., the queue corresponding to the current period). The scheduler does not require expensive sorting or comparisons to determine packet transmission order. More importantly, a CQ rotation involves a deterministic and predictable transition from one switch queue to another at the end of each period. This transition can be realized either using data-plane primitives in upcoming reconfigurable hardware (as we discuss in Section 3.4) or through the switch's control plane (as is the case with our prototype).

3.3 Programmable Scheduling using PCQs

We now show how various scheduling algorithms can be realized using Calendar Queues in conjunction with a programmable packet processing pipeline. We describe three different algorithms, each of which differs in the way it utilizes CQs. First, an approximate variant of WFQ that uses a *Logical* CQ. Next, we implement approximate EDF using LSTF scheduling that uses a work-conserving *Physical* CQ. Finally, we realize a Leaky Bucket Filter that utilizes a non-work-conserving *Physical* CQ.

3.3.1 Weighted Fair Queueing

Weighted Fair Queueing (WFQ) scheduling achieves max-min fair allocation among flows traversing a link by emulating a bit-by-bit round-robin scheme where each active flow transmits a single bit of data each round. This emulation is realized at packet granularity by assigning each incoming packet a departure *round number* based on the current round number and the total bytes sent by the flow. All buffered packets are dequeued in order of increasing departure round numbers.

```
Packet State
  weight : Packet flow's weight

Switch State
  bytes[f] : Number of bytes sent by flow f
  round    : Current round number
  BpR      : Bytes sent per round for each flow

Rank Computation & Enqueueing
  bytes[f] = max(bytes[f], round * BpR * weight)
  n = (bytes[f] + pkt.size) / (BpR * weight) - round
  CQ.enqueue(n)

Queue Rotation
  if CQ.dequeue() is null
    CQ.rotate()
    round = round + 1
```

Figure 2: WFQ implementation using a Logical CQ.

We implement WFQ using Logical CQs closely following the round number approximation described in [25]. We use coarse-grain rounds that are only incremented after all active flows have transmitted a configurable quantum of bytes. The rank computation is done in such a way that each fair queuing round is mapped to a day (queue) in the Calendar Queue and, whenever a day finishes (i.e., the queue is drained completely), the round number is incremented by one. The complete switch state and computation required is shown in Figure 2. Note that this is an approximation of the WFQ algorithm where the round numbers are not as precise or faithful to the original algorithm, and there can be situations in which packets are transmitted in an *unfair* order. However, this *unfairness* has an upper-bound and is controlled by the BpR variable in the rank computation. As we show later in the evaluation, this approach closely approximates ideal fair queueing.

3.3.2 Earliest Deadline First

In Earliest Deadline First (EDF) scheduling, each packet from a flow is assigned a deadline or expected time of reception. At each network hop, the packet with the closest deadline is transmitted first. We implement EDF using Least Slack Time First scheduling, where each packet carries a *slack* value of the time remaining till its deadline. The slack is initialized to `deadline-arrivalTime` at the source and updated at each hop along the way (i.e., each switch subtracts the time spent at the hop from the slack). The implementation uses a Physical CQ, as shown in Figure 3, which we describe next.

```
Packet State
  slack : Initialize to flow_deadline - arrival_time

Switch State
  dT      : Time interval of each queue
  delta   : Skew between ideal and measured time
  lastRot : Timestamp of last rotation

Rank Computation & Enqueueing
  n = (slack - delta + (currentTime - lastRot)) / dT
  CQ.enqueue(n)

Queue Rotation
  if CQ.dequeue() is null
    CQ.rotate()
    delta = delta + (dT - (currentTime - lastRot))
    lastRot = currentTime
```

Figure 3: EDF using a work-conserving Physical CQ.

We choose a fixed time interval for each *day* or queue for our Physical CQ, say dT. Packets with an effective slack of $0 - dT$ are assigned to queue 1, slack of $dT - 2 \cdot dT$ are assigned to queue 2, and so on. This assignment ensures that packets with closer deadlines are prioritized. Queue rotation occurs when the current queue becomes empty. Since we can spend a longer or shorter time than dT in any queue depending on the traffic pattern, we require some additional state to ensure that new packets are inserted in the correct queue with respect to the deadlines of already enqueued packets. The `delta` variable keeps track of how far ahead the CQ is compared to the ideal time. If we spend less than dT for a queue, *delta* increases, and if we spend more than dT, it decreases. The *delta* is then incorporated in the rank computation and is reset to 0 whenever there are zero buffered packets. Note that the programmable switch pipeline allows us to perform not only the rank computation but also decide when to perform the CQ rotation and how to update switch state after a rotation.

3.3.3 Leaky Bucket Filter

A Leaky Bucket Filter (LBF) is a non-work-conserving scheduling algorithm that rate limits a flow to a specified bandwidth and a maximum backlog buffer size. An LBF can be realized using a Physical Calendar Queue by storing a fixed quantum of bytes per flow in each queue and rotating

```
Packet State
  rate : Output rate limit
  size : Maximum bucket size

Switch State
  dT       : Time interval size of each queue
  bytes[f] : Bytes sent by flow f
  round    : Current round number

Rank Computation & Enqueueing
  bytes[f] = max(bytes[f], round * rate * dT)
  n = bytes[f] / (rate * dT) - round
  if n > size / (rate * dT)
    drop packet
  else
    CQ.enqueue(n)

Queue Rotation
  if dT time has elapsed
    CQ.rotate()
    round = round + 1
```

Figure 4: A Leaky Bucket Filter using a non-work-conserving Physical CQ.

queues at fixed time intervals, very similar to the WFQ example discussed earlier. However, we do not dequeue packets from the next queue even if the current queue is empty, which gives us the desired non-work-conserving behavior. The byte quantum depends on the rate limit set for the flow and the configured time interval dT of each queue. If the number of enqueued bytes for a flow exceeds the bucket size, we simply drop the packet. This scheme lets us realize multiple filters using the same underlying CQ, as shown in Figure 4.

We assume the configured `rate` and `size` parameters for the filter are in the packet header, but they can be stored at the switch as well. For each flow, we keep track of bytes sent so far and compute the queue id by dividing it with the quantum, which is the configured filter rate times the queue interval dT. We assume that the cumulative rate of all flows does not exceed the line rate at the switch; if that happens, all flows will be slowed down in proportion to their configured rates equally.

3.4 Implementing PCQs in Hardware

We now describe how Calendar Queues can be implemented on programmable switches. We assume an RMT model switch (as described in Section 2) with an ingress pipeline, followed by the traffic manager, which maintains multiple packet queues, and finally an egress pipeline. Implementing a CQ in this model is non-trivial because the packet enqueue decision (i.e., which queue to insert the packet into) is made in the ingress pipeline, but the queue status (i.e., occupancy, depth) is available in the egress pipeline after the packet traverses the traffic manager. Since these modules are implemented as separate hardware blocks, we need to *synchronize* state among them to achieve the CQ abstraction.

We can realize CQs on programmable switches using mutable switch state, multiple FIFO queues, the ability to create and recirculate packets selectively, and the ability to pause/resume queues or alter queue priorities directly in the data plane. All these capabilities are either already available in today's programmable switches except for the data-plane-based queue pausing, resuming, and priority updating. However, we confirmed with experts in switching ASIC design that adding such a capability is relatively straightforward because doing so doesn't change the order of temporal or spatial complexity of existing TM implementations. Existing TMs already support several per-packet metadata to expose controllable features or statistics. Moreover, PFC already requires a similar queue pausing/resuming capability triggered by certain protocol messages in the data plane anyway. This new feature exposes a similar functionality only in a programmatic way by exposing such knobs to the programmable pipeline. Moreover, even in the absence of data-plane support for priority changes, we can still approximate this functionality using the control plane (as we do in our testbed).

Implementation Overview: We first provide a high-level description of our scheme. Each *period* in the CQ is mapped to a single FIFO queue within a set of queues associated with the outgoing port. The ingress pipeline computes which *period* or queue each incoming packet is enqueued into. We assume a TM that allows the pipeline to enqueue incoming packets into any of the available FIFO queues. At any given time, the queue settings satisfy the following properties: (a) The queue corresponding to the current period has the highest priority level, so that its packets can be transmitted immediately. We refer to this queue as the head queue. (b) The queue corresponding to the next period has a lower priority level and is active/unpaused. (c) All other queues corresponding to future periods are at the lowest priority level and are paused. This specific configuration has two desirable properties. First, since the next period's queue is also active, the switch can start transmitting packets from the next period's queue as soon as the head queue becomes empty. Second, when we perform a CQ rotation, we need to perform only a small number of changes to queue priorities and active statuses.

The CQ cycles or rotates through available queues one at a time, making each queue the head queue. The rotation is triggered when the head queue is empty (in case of a logical CQ) or the CQ time interval has elapsed (in case of a physical CQ). When a rotation happens, we need to make sure all packets from the head queue are drained completely and that it is empty before changing its priority to lowest. Once the priority is set, the head queue can be reused to store packets for future periods.

Implementation Details: We break down the implementation of CQs into the following three steps. Appendix A provides additional details.

Step 1 - Initiate Rotation: This step detects when a rotation needs to happen and initiates the rotation by informing the ingress pipeline using a recirculation packet. In the case of a logical CQ, we initiate rotation when the head queue is empty. This can be detected in two ways. First, in some

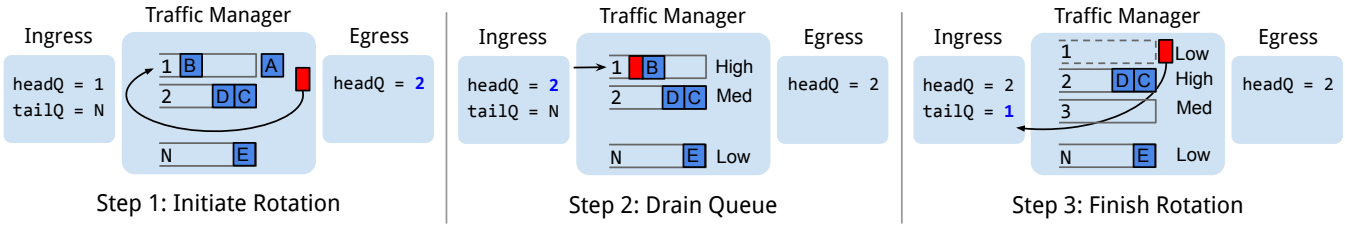

Figure 5: Step 1, the egress pipeline tries to dequeue from the current headQ, and if it is the last packet, it initiates rotation by creating and recirculating a `rotate` (red packet) to the ingress pipeline. Step 2, the ingress pipeline on receiving this packet, updates the headQ and enqueues a `marker` packet as the last packet into the old headQ. Finally in Step 3, when the egress pipeline receives the `marker` packet, it updates the queue priority and notifies the ingress pipeline that it is safe to reuse the queue for future packets by updating the tailQ.

programmable switches, the traffic manager metadata could provide the egress pipeline information regarding the depth of the queue from which the packet was dequeued. If it is zero, this is the last packet from the head queue, and we initiate rotation. Second, we can check the queue id from which the packet was dequeued to infer whether the head queue is empty. Since the successor head queue is also unpaused with a lower priority, a packet dequeued from it implies the head queue is empty. We use the latter method in our prototype as it imposes fewer requirements on the traffic manager metadata available to the switch pipeline. In the case of a physical CQ, the rotation happens at fixed time intervals and is configured through timers or packet generators. When a rotation begins, we recirculate a special `rotate` packet to the ingress pipeline so that it stops enqueuing packets in the head queue and begins draining it, which happens in Step 2.

Step 2 - Drain Queue: This step ensures that the head queue is completely drained, and no more packets are enqueued into it till the rotation finishes. On receiving the `rotate` recirculation packet, the ingress pipeline advances the head queue pointer, essentially stopping any new packets from being enqueued into it. But, there could still be some packets in the pipeline currently making their way into the head queue. To make sure these are transmitted in the right order, the ingress enqueues a special `marker` packet into the head queue after updating the head of the calendar queue. This packet is the last packet to be enqueued into the head queue, and its arrival at the egress pipeline means the queue is completely drained, and we can proceed with finishing the rotation described in Step 3.

Step 3 - Finish Rotation: The `marker` packet is recirculated back to the ingress pipeline, and this informs the ingress pipeline that it is safe to reuse the queue for future periods. The ingress changes the priority of the just emptied queue to lowest and also pauses it, essentially pushing the queue to the end of the CQ. The ingress also unpauses the queue associated with the next period to ensure that there are no transmission stalls after the current period ends. The queue configuration change can be achieved in two ways depending on the underlying hardware support. The `marker` packet can be pushed up to the control plane CPU, which can alter the queue configurations using traffic manager APIs. This approach incurs a latency overhead before the drained queue can be used for packets associated with future periods. Alternatively, if the hardware supports priority change in the datapath, the processing of the `marker` packet with the appropriate metadata tags affects the configuration change almost immediately.

We now briefly highlight some of the attributes of PCQ that aid in efficient hardware realization. First, CQs maintain state at the granularity of physical switch queues instead of individual packets or flows. Second, at any given point in time, there is a designated head queue that is responsible for providing the packets that are to be transmitted. Third, the rotation operation involves changing just the metadata of queue and that too of at most three queues. This combination of factors allows us to bolt-on the PCQ abstraction on to a traditional TM.

3.5 Analysis and Extensions

We now analyze our PCQ abstraction and compare it to both fine-grained priority queuing schemes and an ideal Calendar Queue along different dimensions. We also provide an extension that expands the scheduling capability of the PCQ.

Expressiveness: From a theoretical perspective, a static priority mechanism (e.g., PIFO) with infinite priority levels is equivalent to a Calendar Queue with infinite buckets, and most scheduling algorithms can be expressed in both these hypothetical schemes. However, a practical PIFO has finite priority levels, and a practical CQ has finite buckets, which affects the feasibility and the fidelity of scheduling algorithms. An algorithm can be implemented using PIFOs if all packets throughout the "lifetime" of the algorithm have ranks strictly in the priority queue range. This is true for algorithms like pFabric where packet rank is solely a function of flow size, but not true for WFQ or EDF where packet ranks are computed based on a round number or current time, which is monotonically increasing. As discussed earlier, the priority-level space will roll over eventually, and the ordering of enqueued packets will be violated. On the other hand, our realization of PCQs requires that the enqueued packets' ranks at any instant fit within the available buckets or queues, which makes it challenging to implement algorithms that require both a large

packet rank range as well as high fidelity in distinguishing between the packet ranks; we can extend the range of a CQ by bucketing several packet ranks together, but that introduces approximations which we discuss next.

Approximations: There are two sources of approximations that arise in a PCQ. First, inversions within a FIFO queue. The original Calendar Queue [10] maintains events in a single bucket in sorted order, whereas we simply use a FIFO queue. This can lead to inversions if multiple ranks are assigned to the same bucket, i.e., a higher priority packet is scheduled after a lower priority packet. This presents a feasibility vs. accuracy trade-off for the scheduling algorithm, and if the bucket intervals are chosen carefully, the approximation is acceptable as we show later in the evaluation. It is worth noting that one could borrow some of the mechanisms from the SP-PIFO work [2] to reduce rank inversions, but we leave that to future work. Second, the PCQ imposes a limit on the range of the CQ. Since the number of FIFO queues is limited, there is a possibility that packets will arrive with a rank beyond the range of the CQ. One can theoretically increase the bucket size to include a larger priority schedule such packets that are very far in the future. However, this will lead to an increase in inversions and reduce the accuracy of the priority queuing mechanism. Another option is to store overflowing packets into a separate queue and recirculate them into their appropriate queue when they get close to their service time. Furthermore, the range of a CQ can be significantly increased by employing a hierarchical structure, and we describe this next.

Hierarchical Calendar Queues: One way to extend the range of a CQ is to employ a hierarchical structure among the available FIFO queues, similar to hierarchical timing wheels, at the cost of recirculating some data packets. To create a 2-level hierarchical calendar queue (HCQ), we split the N FIFO queues into two groups of sizes n_1 and n_2, respectively. The two groups run independent calendar queues CQ_1 and CQ_2 on top of them, however with different bucket intervals: CQ_1 having an interval of one time period and CQ_2 having an interval of n_1 time periods, as shown in Figure 6. The idea is that a single queue of CQ_2 has an interval equivalent to the full rotation of all n_1 queues of CQ_1. A packet with a scheduled time between 1 to n_1 is inserted into the appropriate queue in CQ_1, packets with time between n_1+1 to $2\times n_1$ are inserted into the first queue of CQ_2, packets with $2\times n_1+1$ to $3\times n_1$ are inserted into the second queue of CQ_2, and so on. This approach provides a total range of $n_1\times n_2$ time periods, whereas just using a single CQ over N queues would give a range of just n_1+n_2.

However, this comes at the cost of recirculating any data packet that is enqueued in CQ_2. When a full rotation of all n_1 queues in CQ_1 finishes, all packets from the head queue of CQ_2 are recirculated and deposited into appropriate queues in CQ_1. Note that this approach is still significantly better

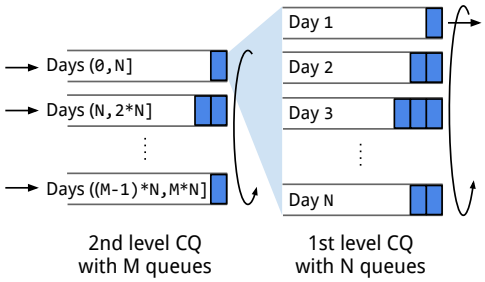

Figure 6: Example of a 2-level Hierarchical CQ. Packets are enqueued into the higher level CQ if they are too far into the future and recirculated into the finer level CQ periodically.

than the approach described in [10], where packets scheduled too far in the future are simply enqueued in the scheduled queue *modulo* N, and recirculated if the scheduled time has not arrived; Brown's scheme can recirculate a packet multiple times, whereas an HCQ recirculates a packet only once leading to more efficient use of bandwidth. Implementing HCQs also requires storing and managing extra state for both CQs and more complex computations when determining the destination queue for a packet. With 32 FIFO queues, a 2-level HCQ can be implemented with 16×16 queues to achieve a reach of 256 time periods or a 3-level HCQ with $16\times 8\times 8$ queues with a total reach of 1024, which is significantly larger than 32.

Limitations: Similar to PIFOs, CQs compute the enqueue rank only on packet arrival. Therefore, the relative order of already buffered packets cannot be changed after enqueuing. This limitation prevents CQs from realizing mechanisms such as pFabric's starvation prevention technique [4], where the dequeue order of multiple previously received packets changes on an enqueue; a later packet within a flow would signal that there are fewer remaining bytes within the flow and, thereby, increase the priority of the flow's previously enqueued packets. CQs do allow us to realize other, arguably stronger, forms of starvation prevention, as we will see in Section 4.5. Another limitation of CQs is that we can schedule packets only in the future. If the computed rank is before the current CQ time, the resulting packet schedule will be different from the desired ordering. Essentially, this means that CQs cannot correctly order packets that have ranks in the past. One could address this limitation by not immediately reusing a queue for a future period as soon as we perform a CQ rotation and allowing some number of queues from the past to be active. Finally, CQs have an upper limit on the range of ranks they can enqueue at a time. If a scheduling algorithm requires a large range, it cannot be implemented accurately using CQs; the hierarchical scheme outlined above can increase a CQ's range, but it comes with an approximation cost.

4 Evaluation

We evaluate the practical feasibility, expressiveness, and performance of Calendar Queues by implementing them on a

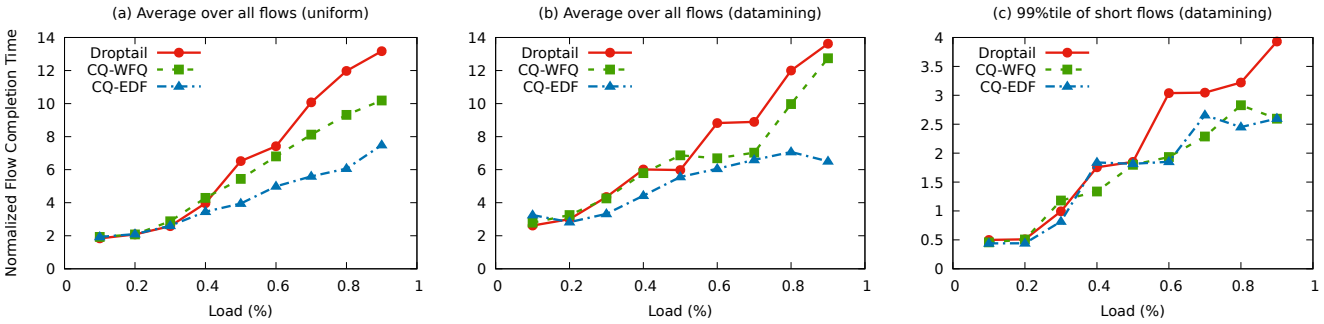

Figure 7: Average and tail latencies for both synthetic and datamining workloads with WFQ and EDF policies implemented using Calendar Queues on top of Barefoot Tofino switch in the hardware testbed.

programmable Barefoot Tofino switch and realizing two classical scheduling algorithms using CQs. Next, using large scale packet-level simulations, we demonstrate the flexibility of the Calendar Queue abstraction with three case studies. First, we instantiate a physical calendar queue that performs deadline-aware scheduling of both aggregate flows (or coflows) and individual flows. Second, we instantiate a logical calendar queue that implements a variant of fair-queueing that can tolerate a limited amount of burstiness, thereby providing a configurable balance between fairness and flow completion time for short flows. Finally, we implement a variant of pFabric that prevents long flows from starving by gradually increasing the priority of all enqueued packets. None of these scheduling algorithms can be realized using traditional priority queuing schemes such as PIFO.

4.1 Hardware Prototype Implementation

We implement and evaluate programmable calendar queues on the Barefoot Tofino 100BF-32X switch. As the current Tofino switch does not support updating a queue's priority on the datapath, we implement CQs using a combination of in-built packet generator, packet re-circulation, and control plane operations to drain packets in the correct order.

First, we use the in-built packet generator on the switch to periodically generate probe packets and detect when a queue rotation needs to be performed. The egress pipeline tracks the current head of the CQ, and when a packet is dequeued from the next queue (signifying the current queue is empty), it re-circulates the probe packet back to the ingress to initiate a rotation. Next, the ingress pipeline updates the current head of the CQ and enqueues the probe packet into the queue being rotated out to drain it fully, and no more packets are inserted into it. When the egress pipeline receives the probe packet again, it is safe to update the priority of the drained queue, and we achieve this by setting a flag in the egress pipeline. Finally, the control CPU polls on this flag variable, and when set, it makes an API call to update the queue's priority and notifies the ingress pipeline that it is safe to use this queue to store the future packets.

Testbed and Workload We implement two scheduling algorithms, WFQ [14] and EDF [18] using Calendar Queues and compare them against standard FIFO droptail scheduling in a 2-level fat-tree topology, consisting of 2 ToR switches, 2 aggregation switches, and 4 servers by using loopback links with ingress-port based virtualization to divide the 32 physical ports into multiple switches. All links in the network are 40Gbps with 80µs end-to-end latency. Each server opens 80 concurrent long-running connections to other servers and generates flows according to a Poisson process at a rate configured to achieve the desired network load. We tested a synthetic workload that draws flow sizes at uniform with a max size of 12.5 MB and the data mining workload from [4].

Performance Figure 7 shows the average and 99^{th} percentile latencies. Across both workloads, we can see the WFQ and EDF implementations performing better than simple droptail queues. The difference is more significant at higher network load when queues build up at the switch due to bursty arrivals. This is when the prioritization and correct scheduling of packets leads to a visible difference in FCTs. However, the important point here is that we were able to realize these scheduling policies at a line-rate of 40 Gbps without being limited by the number of flows. We further measured the extra physical switch resources consumed by our implementation of WFQ and EDF using CQs, and report them in Appendix B.

4.2 Packet-level Simulations

We study our use-cases in a large-scale cluster deployment using an event-driven, packet-level simulator. We extend the mptcp-htsim simulator [21] to implement Calendar Queues and several other comparison schemes. The simulation consists of 256 servers connected in a 3-level fat-tree topology consisting of 8 ToR switches, 8 aggregation switches, and 4 core switches. Each ToR switch is connected to 32 servers using 10 Gbps links, and all other switches are connected to each other using 40 Gbps links. Next, we describe each case-study and evaluate them.

4.3 Use Case 1: Coflow scheduling using Least Slack Time First (LSTF) scheduling

Distributed applications running inside datacenters generate network traffic patterns that require optimizing the perfor-

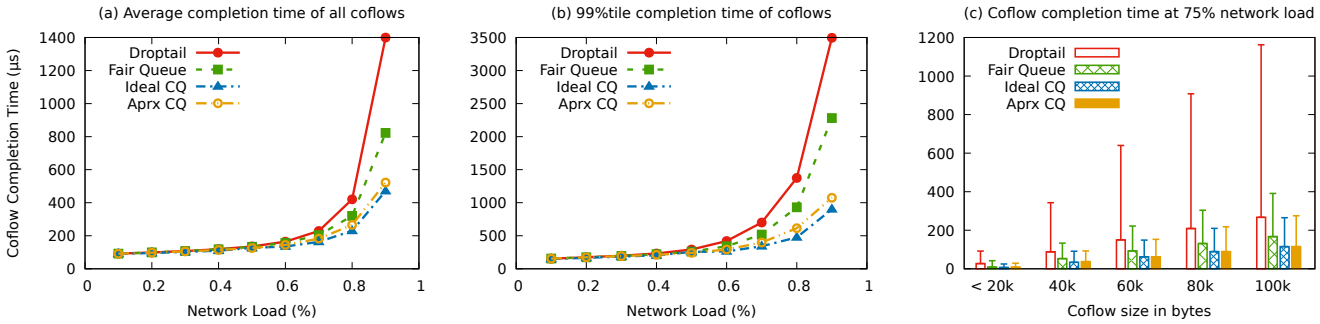

Figure 8: Coflow completion time when running a mix of background and coflow traffic with coflows prioritized over background flows. (a) average CCT for all coflows, (b) 99th percentile CCT for all coflows, and (c) average and 99th percentile (using error bar) for various coflow size buckets at 75% network load.

mance on a collection of flows [12], called coflows, rather than individual flows, e.g., partition-aggregate or bulk synchronous programming tasks such as multi-get Memcached queries and MapReduce jobs. The performance of such applications depends on the last finishing flow among the collection and prior work [1] has shown that near-optimal performance can be achieved by ordering coflows using a Shortest Remaining Processing Time (SRPT) first mechanism and ensuring that any packet from any flow of a coflow X ordered before coflow Y is transmitted before any packet from coflow Y.

We implement the above approach using LSTF scheduling on top of Calendar Queues to optimize coflow completion times (CCT). However, instead of using the complex BSSI algorithm in [1], which decides priorities based on other coflows in the system, we choose a much simpler heuristic to order coflows as they arrive. We compute a deadline for the whole coflow, assuming the largest sub-flow in the coflow is the bottleneck and will be the last to finish. Therefore, the deadline is simply the largest sub-flow size divided by endhost link speed. All we need to do is assign this deadline to all packets of all flows in the coflow and ensure that packets with the earliest deadline are transmitted first at each switch.

We calculate each packet's *slack* as the time remaining until the deadline of its corresponding coflow. The slack is initialized in the packet header at the endhost, and as the packet traverses the network, each switch enqueues the packet in the Calendar Queue based on this slack value. The higher the slack value, the farther in future the packet is scheduled for transmission. On departure, the switch deducts the time spent at the switch from the slack and updates the packet header. As a result, critical flows with lower slack values and closer deadlines are dynamically prioritized over non-critical flows with larger slack values and farther deadlines. Note that this scheme could have been implemented using an exact priority queue (such as PIFO) with absolute deadlines embedded in the packet header, but that would require clocks to be synchronized. More importantly, the switch would run out of priority levels eventually and would not be able to enforce deadlines. We, therefore, implement the scheme using LSTF on top of Calendar Queues.

We measure the performance of the above coflow scheduling mechanism using event-driven simulations and compare it with the following queueing schemes:

- **Droptail**: Traditional switch with a single FIFO queue that drops packets from the tail when full.
- **Fair Queue**: Bit-by-bit round-robin algorithm from [14] that achieves max-min fairness.
- **Ideal Calendar Queue**: A CQ with *infinite* buckets that also transmits packets in sorted order within each bucket.
- **Approx Calendar Queue**: Our implementation of CQs that uses 32 FIFO queues and 10µs round interval.

In all the above schemes, we use the same end-host flow control protocol, DCTCP, with the additional embedding of *slack* value based on the coflow deadline.

Workload and Performance Metric We use a mix of background traffic, which is the enterprise workload in [3] and a synthetic coflow workload derived from a Facebook trace [1]. Coflows, on average, have ten sub-flows and a total size of 100 KB. The ratio of background traffic to coflow traffic is 3:1. Flows and coflows arrive according to a Poisson process at randomly and independently chosen source-destination server pairs with an arrival rate chosen to achieve the desired level of utilization in the aggregation-core switch links. We evaluate the performance in terms of flow completion time (FCT) or, in case of coflows, the coflow completion time (CCT), which is the maximum FCT of comprising sub-flows and report both mean and 99th percentile numbers.

Since we have a mix of background flows and coflows, we must decide how they co-exist together and are scheduled inside the network. One trivial way is to treat background traffic as a separate class with a lower priority than coflow traffic. This configuration is evaluated in Setup 1. Another option made possible by CQs is to treat background traffic as fair-queued and coflow traffic as deadline-aware using the same underlying CQ to schedule packets belonging to both classes. This demonstrates the flexibility of CQs in realizing multiple scheduling policies at once, and we evaluate this configuration in Setup 2.

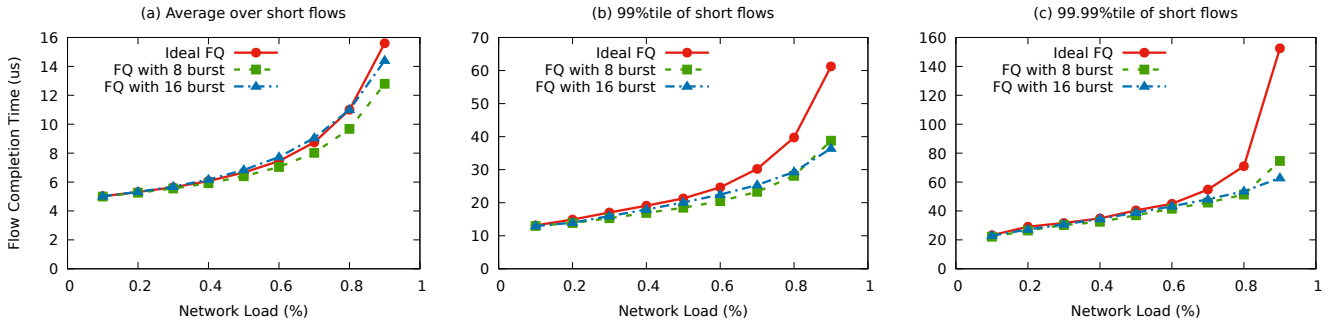

Figure 9: Average and percentile flow completion times for short flows with varying network load and permissible burst size.

Setup 1: Coflows have priority over background traffic. We treat background flows as a different traffic class with lower priority than coflows by using a separate queue with strictly lower priority. As a result, the background traffic performance is not affected by the coflow scheduling policy, and we report coflow completion times in Figure 8. The average CCT improves by up to 3x and 99th percentile CCT by 5x at high network loads. This improvement is both because we are emulating an SRPT policy as well as because we de-prioritize shorter sub-flows within a coflow over other more critical flows. Both droptail and fair-queuing finish short flows within a coflow quickly, although they have a significant *slack* till the deadline. Moreover, our CQ implementation is able to accurately emulate an ideal calendar queue mechanism using a limited number of FIFO queues, and the gap between ideal CQ and our CQ is fairly negligible.

We present the results for Setup 2 in Appendix C.

4.4 Use Case 2: Weighted Fair Queueing with Burst Allowance

First, we implement Fair Queueing using calendar queues as described in Section 3.3.1, which emulates a round-robin scheme wherein each active flow transmits a fixed number of bytes each round (*BpR*). The departure round computed for each packet is mapped to a future day (or queue) in the CQ, which transmits the packets in the correct order. This scheme has been shown to emulate Fair Queuing accurately [25] and while it implements Start Time Fair Queueing at a coarse granularity, it is often desirable to allow a burst of packets from a single flow to be transmitted back-to-back to achieve a better latency for short flows and improve tail latency [13]. This only affects fairness at very short timescales while maintaining fairness at larger timescales.

We modify WFQ to allow short bursts to go through using a simple modification to the enqueuing logic at the ingress. In addition to maintaining a byte counter per flow and computing a packet's round number as byte counter divided by *BpR*, we maintain a permitted *burst size*. While our original WFQ implementation allows a flow to send at most *BpR* bytes per round, the burst-friendly variant lets a flow enqueue up to a fraction of available burst size into a single round, allowing it to exceed its fair share allocation temporarily.

More precisely, instead of computing the round number as `R = bytes[f]/BpR`, where `bytes[f]` is the amount of bytes enqueued by flow `f`, we incorporate burst size into the calculation as follows,

```
R = bytes[f]/max(BpR, BurstSize - bytes[f])
```

where `BurstSize` is the configured permissible burst size. This essentially lets `BurstSize/2` bytes to be enqueued in the current round, `BurstSize/4` in the next round, and so on, as show in Figure 10. Any enqueued bytes exceeding the burst size are assigned the same round number as before.

We implement this burst-friendly fair-queuing scheme with configured burst sizes of 8 and 16 packets (denoted by FQ-8 and FQ-16), along with ideal fair-queueing on top of our Calendar Queues and measure the impact on flow completion times using simulations. We use the same 3-level fat-tree topology and enterprise workload for this use case.

Figure 9 shows the flow completion times of short flows as we increase the network load. At higher network loads, allowing a burst of bytes to go through leads to up to 2-3x reduction in higher percentile latencies. Figure 11 breaks down the latency improvement across different flow size buckets, and it confirms that short flows in the region of burst size show the most improvement. More importantly, larger flows are unaffected by this temporary burst allowance.

4.5 Use Case 3: pFabric with Starvation Prevention

pFabric [4] is a transport layer designed to provide near-optimal flow completion time by essentially emulating Shortest Remaining Processing Time (SRPT) first scheduling at each switch. Each flow packet carries a single number that encodes its priority – in this case, it is set to the remaining flow size when the packet is transmitted. All switches forward the packet with the shortest remaining flow size at any given time. This simple scheme can be implemented using a static fine-grained priority scheme such as PIFO, as it does not require the gradual priority escalation. However, this leads to potential starvation of long flows, which are always deprioritized compared to shorter flows, making it impractical to run in real environments. This is shown in Table 1, where we simulated the same 3-level fat-tree topology and ran the enterprise workload at 80% network load. As flow size increases,

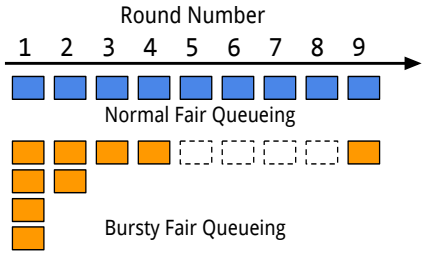

Figure 10: Packet enqueuing behavior of normal fair queueing vs bursty fair queueing.

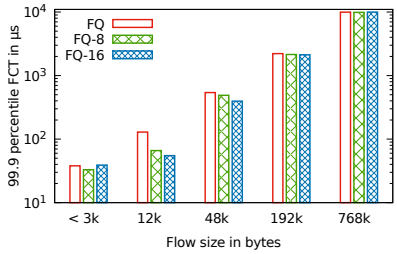

Figure 11: 99.9 percentile latency for various flow size buckets at 90% network load in micro-seconds.

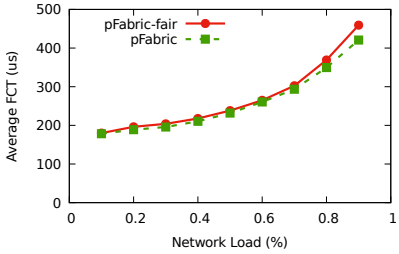

Figure 12: Average FCT for pFabric and pFabric-fair for enterprise workload in our testbed.

Flowsize	10k	100k	1M	10M	100M
pFabric	679	651	670	524	265
pFabric-fair	650	642	638	543	442

Table 1: Average bandwidth in MBps achieved by flows in various bucket sizes.

the average transmission rate decreases resulting in reduced bandwidth available for longer flows.

If we were to think of fair queuing and pFabric as ends of the spectrum, calendar queues would provide us with options in the middle. We implement a fairer version of pFabric, called pFabric-fair, by slightly altering the enqueuing mechanism. A packet with k bytes remaining in the flow is enqueued $f(k)$ periods into the future, where $f(k)$ is a log function. Thus, higher rounds are exponentially bigger, and the CQ can accommodate large flow sizes. Whenever the current head queue is empty, we rotate to the next queue, which ensures that low priority packets from larger flows are not permanently starved, merely deprioritized at enqueue time, and their priorities increase with time. However, we need some additional state to ensure that we do not enqueue a later packet from a flow ahead of the flow's previously received packets, which we achieve by keeping track of the highest queue for each flow.

Figure 12 shows the average and 99th percentile FCT for all flows with varying network loads for pFabric and pFabric-fair. Although pFabric-fair has a slightly higher average FCT at higher network loads, it also provides higher bandwidth to large-sized flows, preventing them from starving.

5 Related Work

Packet schedulers available in switching hardware today are fixed function, supporting specific primitives such as strict priority, rate limits, round-robin fairness, although a vast number of richer scheduling algorithms that provide stronger guarantees exist in the literature, such as WFQ [14], pFabric [4], STFQ [15], SRPT [24], EDF [18].

Several recent proposals aim to provide a programmable packet scheduler that can implement these scheduling algorithms while operating at line rate of terabits per second, such as PIFO [28], PIEO [27], and SP-PIFO [2]. All of these proposals provide the abstraction of *static* and *finite* priority levels, which we argue is insufficient to implement several scheduling algorithms that require monotonically escalating priorities. pHeap [7], PIFO and PIEO provide fine-grained priority levels, which makes it challenging for them to scale to large packet buffers and multi-pipeline switches. SP-PIFO is similar to Calendar Queues as it also provides coarse-grained priority levels using only FIFO queues, and can scale to current line-rate switches. SP-PIFO dynamically adjusts the priority range of individual queues by changing queueing thresholds, which can be explored further in the context of Calendar Queues as well.

Another chain of work proposes efficient packet scheduling in software such as Carousel [22], Loom [29], and Eiffel [23]. These approaches rely on timing wheel data structures or bucketed integer priority queue-like data structures for efficient operation. Calendar Queue borrows ideas from similar data structures while targeting switching hardware that can support today's large buffer and multi-pipeline routers.

6 Conclusion

We propose a flexible packet scheduler designed for line-rate hardware switches, called Programmable Calendar Queues, that enables the efficient realization of several classical scheduling algorithms. It relies on the observation that most algorithms require both prioritization and implicit escalation of a packet's priority. We show how they can be implemented efficiently on today's programmable switches by dynamically changing the priority of queues using either dataplane primitives or control-plane operations. We demonstrate that PCQs can be used to realize interesting variants of LSTF, Fair Queueing, and pFabric to provide stronger delay guarantees, burst-friendly fairness, and starvation-free prioritization of short flows, respectively.

Acknowledgments

We would like to thank the anonymous NSDI reviewers and our shepherd Rachit Agarwal for their valuable feedback. This research was partially supported by NSF Grant CNS-1714508 and Futurewei.

References

[1] AGARWAL, S., RAJAKRISHNAN, S., NARAYAN, A., AGARWAL, R., SHMOYS, D., AND VAHDAT, A. Sincronia: Near-optimal network design for coflows. In *Proceedings of the ACM SIGCOMM Conference* (2018).

[2] ALCOZ, A. G., DIETMÜLLER, A., AND VANBEVER, L. SP-PIFO: Approximating push-in first-out behaviors using strict-priority queues. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (Santa Clara, CA, 2020).

[3] ALIZADEH, M., EDSALL, T., DHARMAPURIKAR, S., VAIDYANATHAN, R., CHU, K., FINGERHUT, A., LAM, V. T., MATUS, F., PAN, R., YADAV, N., AND VARGHESE, G. CONGA: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the ACM SIGCOMM Conference* (2014).

[4] ALIZADEH, M., YANG, S., SHARIF, M., KATTI, S., MCKEOWN, N., PRABHAKAR, B., AND SHENKER, S. pFabric: Minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM Conference* (2013).

[5] BAI, W., CHEN, L., CHEN, K., HAN, D., TIAN, C., AND WANG, H. Information-agnostic flow scheduling for commodity data centers. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Oakland, CA, 2015).

[6] BAREFOOT NETWORKS. Tofino Programmable Switch. `https://www.barefootnetworks.com/technology/`.

[7] BHAGWAN, R., AND LIN, B. Design of a High-speed Packet Switch with Fine-grained Quality-of-Service Guarantees. In *Proceedings of the IEEE International Conference on Communications* (June 2000), vol. 3, pp. 1430–1434 vol.3.

[8] BOSSHART, P., DALY, D., GIBB, G., IZZARD, M., MCKEOWN, N., REXFORD, J., SCHLESINGER, C., TALAYCO, D., VAHDAT, A., VARGHESE, G., AND WALKER, D. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review 44*, 3 (July 2014).

[9] BOSSHART, P., GIBB, G., KIM, H.-S., VARGHESE, G., MCKEOWN, N., IZZARD, M., MUJICA, F., AND HOROWITZ, M. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *Proceedings of the ACM SIGCOMM Conference* (2013), pp. 99–110.

[10] BROWN, R. Calendar queues: A fast 0(1) priority queue implementation for the simulation event set problem. *Communications of the ACM 31* (1988), 1220–1227.

[11] CAVIUM. XPliant Ethernet switch product family. `http://www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html`.

[12] CHOWDHURY, M., ZHONG, Y., AND STOICA, I. Efficient coflow scheduling with varys. In *Proceedings of the ACM SIGCOMM Conference* (2014), SIGCOMM '14, pp. 443–454.

[13] CLARK, D. D., SHENKER, S., AND ZHANG, L. Supporting real-time applications in an integrated services packet network: Architecture and mechanism. In *Proceedings on the ACM SIGCOMM Conference* (1992).

[14] DEMERS, A., KESHAV, S., AND SHENKER, S. Analysis and simulation of a fair queueing algorithm. In *Proceedings on the ACM SIGCOMM Conference* (1989).

[15] GOYAL, P., VIN, H. M., AND CHEN, H. Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks. In *Proceedings of the ACM SIGCOMM Conference* (1996), SIGCOMM '96, pp. 157–168.

[16] IEEE. Priority based flow control. *IEEE 802.11Qbb* (2011).

[17] LEUNG, J. Y.-T. A new algorithm for scheduling periodic, real-time tasks. *Algorithmica 4*, 1-4 (1989), 209.

[18] LIU, C. L., AND LAYLAND, J. W. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM 20*, 1 (Jan. 1973), 46–61.

[19] MITTAL, R., AGARWAL, R., RATNASAMY, S., AND SHENKER, S. Universal packet scheduling. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (Santa Clara, CA, 2016).

[20] OZDAG, R. Intel® Ethernet Switch FM6000 Series-Software Defined Networking. `http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ethernet-switch-fm6000-sdn-paper.pdf`.

[21] RAICIU, C. MPTCP htsim simulator. `http://nrg.cs.ucl.ac.uk/mptcp/implementation.html`.

[22] SAEED, A., DUKKIPATI, N., VALANCIUS, V., THE LAM, V., CONTAVALLI, C., AND VAHDAT, A. Carousel: Scalable traffic shaping at end hosts. In *Proceedings of the ACM SIGCOMM Conference* (2017), SIGCOMM '17, pp. 404–417.

[23] SAEED, A., ZHAO, Y., DUKKIPATI, N., ZEGURA, E., AMMAR, M., HARRAS, K., AND VAHDAT, A. Eiffel: Efficient and flexible software packet scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (Boston, MA, 2019), pp. 17–32.

[24] SCHRAGE, L. E., AND MILLER, L. W. The queue m/g/1 with the shortest remaining processing time discipline. *Oper. Res. 14*, 4 (Aug. 1966), 670–684.

[25] SHARMA, N. K., LIU, M., ATREYA, K., AND KRISHNAMURTHY, A. Approximating fair queueing on reconfigurable switches. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)* (Renton, WA, 2018), pp. 1–16.

[26] SHREEDHAR, M., AND VARGHESE, G. Efficient fair queueing using deficit round robin. In *Proceedings on the ACM SIGCOMM Conference* (1995).

[27] SHRIVASTAV, V. Fast, scalable, and programmable packet scheduler in hardware. In *Proceedings of the ACM SIGCOMM Conference* (2019), SIGCOMM '19, pp. 367–379.

[28] SIVARAMAN, A., SUBRAMANIAN, S., ALIZADEH, M., CHOLE, S., CHUANG, S.-T., AGRAWAL, A., BALAKRISHNAN, H., EDSALL, T., KATTI, S., AND MCKEOWN, N. Programmable packet scheduling at line rate. In *Proceedings of the ACM SIGCOMM Conference* (2016).

[29] STEPHENS, B., AKELLA, A., AND SWIFT, M. Loom: Flexible and efficient NIC packet scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (Boston, MA, 2019), pp. 33–46.

A Detailed Implementation Notes for Realizing PCQs in Hardware

We now present the traffic manager API and how it is invoked by the different steps involved in transitioning a CQ from one period to another.

Traffic Manager API

- `tm_enqueue(packet, queue)`: Enqueues a packet in the given queue.
- `tm_pause(queue)`: Stop or pause queue from transmitting any packets until unpause is called on the same queue.
- `tm_unpause(queue)`: Resume queue, allowing it to send out packets until pause is called.
- `tm_setPriority(queue, p)`: Set priority of queue to `p` (one of different levels supported by the TM)
- `tm_dequeue()` (called from egress): Returns a packet from the highest priority unpaused queue, along with the queue id from which it was dequeued.

The exact details of enqueue/dequeue and rotation are described in the pseudocode below. In addition, we store the following state at ingress and egress to keep track of Calendar Queue status and perform queue rotations. The special recirculation packets also contain metadata regarding which queue is being rotated out.

Ingress State

```
currQ: queueId currently at the head of the CQ
prevQ: queueId that was just rotated out and is draining
nextQ: queueId that will be unpaused next
```

Egress State

```
currQ: queueId at the head of the CQ (egress)
```

Packet Enqueue Function

```
packet_enqueue(pkt, X) (called from ingress)
  // According to desired scheduling algorithm.
  x = compute_rank(pkt)
  // Enqueue packet in queue (currQ + x) % N
  tm_enqueue(pkt, (currQ + x) % N)

  // process initiate rotate packet
  if packet == rotate:
     ingress.prevQ = ingress.currQ
     ingress.currQ = ingress.nextQ
     ingress.nextQ = (ingress.nextQ + 1) % N
     tm_enqueue(marker, ingress.prevQ)
     tm_setPriority(prevQ, High)
     tm_setPriority(currQ, Medium)

  // process marker packet
  if packet == marker:
     tm_pause(marker.queueId)
     tm_unpause(nextQ)
     tm_setPriority(nextQ, Low)
```

Packet Dequeue Function

```
packet_dequeue() (called from egress)
  // Returns the highest priority packet
  pkt, queueId = tm_dequeue()

  // Need to initiate rotation
  if queueId != egress.currQ:
     create and circulate a rotate packet on egress.currQ
     egress.currQ++

  // Normal packet dequeue
  if queueId == egress.currQ:
     perform normal packet processing

  if pkt == marker:
     recirculate marker back to ingress
```

Note that, if no more packets remain to be transmitted, then rotate packet is never sent out and currQ remains the same. This avoids unnecessary rotations when the link traffic is less than the link bandwidth.

B Resource Overhead of Implementing CQs

Table 2 shows the additional overhead of implementing CQs along with various scheduling policies, as reported by the P4 compiler. First, since we were able to compile CQs directly onto the Tofino hardware, we can support an arbitrary number of flows at the configured line-rate of 40 Gbps, and are not scale limited in any way. We do require some additional state that is proportional to the number of CQs instantiated across all ports and the number of queues in each CQ. Each scheduling policy also keeps extra state for rank computation, which takes extra resources, e.g., keeping flow byte counters for WFQ results in 50% increase in SRAM usage.

Resource	Baseline	CQ w/ EDF	CQ w/ WFQ	CQ w/ EDF+WFQ
Pkt Header Vector	356	356	356	356
Pipeline Stages	9	9	12	12
Match Crossbar	50	54	63	68
Hash Bits	113	124	140	150
SRAM	27	29	46	48
TCAM	2	2	2	2
ALU Instruction	11	12	13	14

Table 2: Summary of resource usage for a Calendar Queue implementation with 32 physical queues on top of P4 switch.

C Scheduling Deadline-aware and Background Traffic using the same CQ

In Setup 2 described in Section 4.3, we use the same underlying Calendar Queue to schedule background flows as fair-queued traffic and coflows as deadline or slack based traffic. For a background flow packet, which is fair-queued, we compute its departure queue based on bytes enqueued by the flow using the WFQ implementation described in Section 3.3.1. For a deadline-aware coflow packet, we calculate

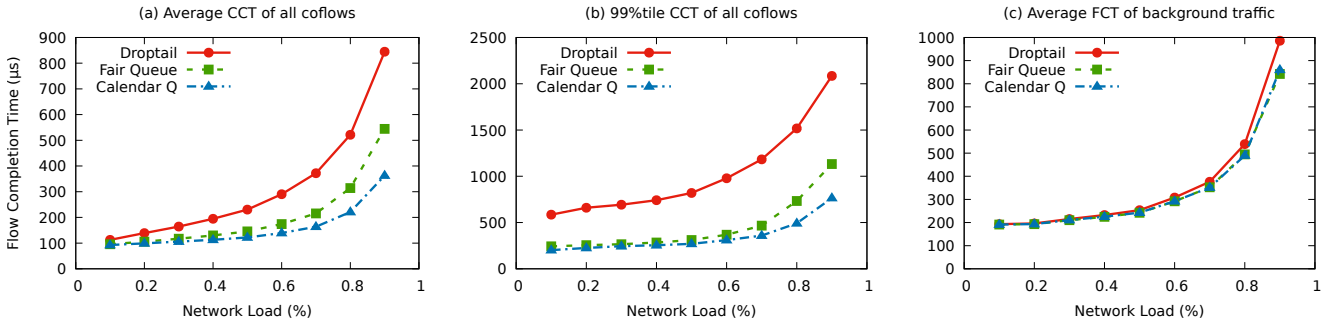

Figure 13: CCT and FCT when scheduling background traffic as fair-queued and coflow traffic as deadline-aware using the same Calendar Queue.

its departure queue using the slack inside the packet header by dividing it with the configured bucket interval as described in Section 3.3.2. We can control the relative priority of background vs. coflow traffic by changing the bucket interval. A higher bucket interval will accommodate more bytes from deadline traffic compared to fair-queued traffic. We use a default value of 10µs as the bucket interval and 1 MSS as the bytes quantum per round for fair queueing.

Figure 13 shows the coflow completion times and FCT of background flows in this setup. The average CCT shows up to 2.5x and 1.5x improvement compared to droptail and fair queuing, respectively. This benefit again comes from the fact that we are able to schedule shorter coflows before longer coflows, as well as de-prioritize shorter sub-flows within a coflow over other more critical sub-flows. The 99th percentile shows a similar improvement of 3x over droptail queues. Moreover, the average FCT of background flows stays roughly the same and is unaffected by the coflow scheduling being done by the Calendar Queue.