Programmable Packet Scheduling at Line Rate

Anirudh Sivaraman*, Suvinay Subramanian*, Mohammad Alizadeh*, Sharad Chole‡, Shang-Tse Chuang‡, Anurag Agrawal†, Hari Balakrishnan*, Tom Edsall‡, Sachin Katti+, Nick McKeown+
*MIT CSAIL, †Barefoot Networks, ‡Cisco Systems, +Stanford University

ABSTRACT

Switches today provide a small menu of scheduling algorithms. While we can tweak scheduling parameters, we cannot modify algorithmic logic, or add a completely new algorithm, after the switch has been designed. This paper presents a design for a *programmable* packet scheduler, which allows scheduling algorithms—potentially algorithms that are unknown today—to be programmed into a switch without requiring hardware redesign.

Our design uses the property that scheduling algorithms make two decisions: *in what order* to schedule packets and *when* to schedule them. Further, we observe that in many scheduling algorithms, definitive decisions on these two questions can be made when packets are enqueued. We use these observations to build a programmable scheduler using a single abstraction: the push-in first-out queue (PIFO), a priority queue that maintains the scheduling order or time.

We show that a PIFO-based scheduler lets us program a wide variety of scheduling algorithms. We present a hardware design for this scheduler for a 64-port 10 Gbit/s shared-memory (output-queued) switch. Our design costs an additional 4% in chip area. In return, it lets us program many sophisticated algorithms, such as a 5-level hierarchical scheduler with programmable decisions at each level.

CCS Concepts

•**Networks** → **Programmable networks;**

Keywords

Programmable scheduling; switch hardware

SIGCOMM '16, August 22 - 26, 2016, Florianopolis , Brazil

 ISBN 978-1-4503-4193-6/16/08…$15.00

DOI: http://dx.doi.org/10.1145/2934872.2934899

1. INTRODUCTION

Today's fastest switches, also known as *line-rate switches*, provide a small menu of scheduling algorithms: typically, a combination of Deficit Round Robin [36], strict priority scheduling, and traffic shaping. A network operator can change parameters in these algorithms, but cannot change the core logic in an existing algorithm, or program a new one, without building new switch hardware.

By contrast, with a *programmable* packet scheduler, network operators could customize scheduling algorithms to application requirements, e.g., minimizing flow completion times [12] using Shortest Remaining Processing Time [35], allocating bandwidth flexibly across flows or tenants [26, 33] using Weighted Fair Queueing [20], minimizing tail packet delays using Least Slack Time First [29], etc. Moreover, with a programmable packet scheduler, switch vendors could implement scheduling algorithms as programs running on a programmable switching chip, making it easier to verify and modify these algorithms compared to baking in the same algorithms into a chip as rigid hardware.

This paper presents a design for programmable packet scheduling in line-rate switches. All scheduling algorithms make two basic decisions: in *what order* packets should be scheduled and *when* they should be scheduled, corresponding to work-conserving and non-work-conserving algorithms respectively. Furthermore, for many scheduling algorithms, these two decisions can be made when a packet is enqueued. This observation suggests a natural hardware primitive for packet scheduling: a *push-in first-out queue (PIFO)* [19, 38]. A PIFO is a priority queue that allows elements to be pushed into an arbitrary position based on an element's *rank* (the scheduling order or time),[1] but always dequeues elements from the head.

We develop a programming model for scheduling (§2) based on PIFOs with two key ideas. First, we allow users to set a packet's rank in a PIFO by supplying a small program for computing packet ranks (§2.1). Coupling this program with a single PIFO allows the user to program any scheduling algorithm where the relative scheduling order of buffered packets does not change with future packet arrivals. Second, users can compose PIFOs together in a tree to program

[1]When the rank denotes the scheduling time, the PIFO implements a calendar queue; we distinguish between PIFOs and priority queues for this reason.

hierarchical scheduling algorithms that violate this relative ordering property (§2.2 and §2.3).

We find that a PIFO-based scheduler lets us program many scheduling algorithms (§3), e.g., Weighted Fair Queueing [20], Token Bucket Filtering [10], Hierarchical Packet Fair Queueing [13], Least-Slack Time-First [29], the Rate-Controlled Service Disciplines [42], and fine-grained priority scheduling (e.g., Shortest Job First). Until now, any line-rate implementations of these algorithms—if they exist at all—have been hard-wired into switch hardware. We also describe the limits of the PIFO abstraction (§3.5) by presenting examples of scheduling algorithms that cannot be programmed using PIFOs.

To evaluate the hardware feasibility of PIFOs, we implemented the design (§4) in Verilog [9] and synthesized it to an industry-standard 16-nm standard-cell library (§5). The main operation in our design is sorting an array of PIFO elements at line rate. To implement this sort, traditionally considered hard [30, 36], we exploit two observations. One, most scheduling algorithms schedule across flows, with packet ranks increasingly monotonically within each flow. Hence, we only need to sort the head packets of all flows to dequeue from a PIFO. Two, transistor scaling now makes it feasible to sort these head packets at line rate.

As a result, we find (§5) that is feasible to build a programmable scheduler, which

- supports 5-level hierarchical scheduling, where the scheduling algorithms at each level are programmable;
- runs at a clock frequency of 1 GHz—sufficient for a 64-port 10 Gbit/s shared-memory switch;
- uses only 4% additional chip area compared to a shared-memory switch that supports only a small menu of scheduling algorithms; and
- has the same buffer size as a typical shared-memory switch in a datacenter (~60K packets, ~1K flows) [3].

While we have not produced a chip supporting PIFOs, our synthesis results are promising and make a strong technical case for switching chip manufacturers to invest in hardware for programmable schedulers. To that end, C++ code for a hardware reference model of our programmable scheduler and Verilog code for our hardware design are available at http://web.mit.edu/pifo/.

2. A PROGRAMMING MODEL FOR PACKET SCHEDULING

For work-conserving scheduling algorithms, the characteristic feature is the order in which packets are scheduled; for non-work-conserving ones, it is the time at which each packet is sent. Moreover, for most algorithms used in practice, these two decisions can be determined definitively when a packet is enqueued into the packet buffer [38].

Our programming model is built around this observation and has two underlying components:

```
f = flow(p) # compute flow from packet p
if f in last_finish:
 p.start = max(virtual_time, last_finish[f])
else: # p is first packet in f
 p.start = virtual_time
last_finish[f] = p.start + p.length/f.weight
p.rank = p.start
```

Figure 1: Scheduling transaction for STFQ. `p.x` refers to a packet field `x` in packet `p`. `y` refers to a state variable that is persisted on the switch across packets, e.g., `last_finish` and `virtual_time` in this snippet. `p.rank` denotes the packet's computed rank.

1. The *push-in first-out queue (PIFO)* [19], which maintains the scheduling order or scheduling time for enqueued elements. A PIFO is a priority queue that allows elements to be enqueued into an arbitrary position based on the element's *rank*, but dequeues elements from the head. Elements with a lower rank are dequeued first; if two elements have the same value, the element enqueued earlier is dequeued first.
2. The computation of an element's rank before it is enqueued into a PIFO. We model this computation as a *packet transaction* [37], an atomically executed block of code that is executed once for each element before enqueuing it in a PIFO.[2]

We note that scheduling with the PIFO abstraction does not require packets to be stored in per-flow queues.

We now describe the three main abstractions in our programming model. First, we show how to use a *scheduling transaction* to program simple work-conserving scheduling algorithms using a single PIFO (§2.1). Second, we generalize to a *scheduling tree* to program hierarchical work-conserving scheduling algorithms (§2.2). Third, we augment nodes of this tree with a *shaping transaction* to program non-work-conserving scheduling algorithms (§2.3).

2.1 Scheduling transactions

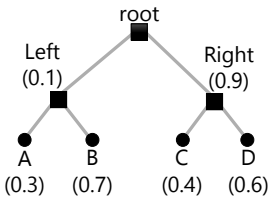
A *scheduling transaction* is a block of code associated with a PIFO that is executed once for each packet before the packet is enqueued. The scheduling transaction computes the packet's rank, which determines its position in the PIFO. A single scheduling transaction and PIFO are sufficient to specify any scheduling algorithm where the relative scheduling order of packets already in the buffer does not change with the arrival of future packets.

Weighted Fair Queueing (WFQ) [20] is one example. It achieves weighted max-min allocation of link capacity across flows[3] sharing a link. Approximations to WFQ[4] include Deficit Round Robin (DRR) [36], Stochastic Fair-

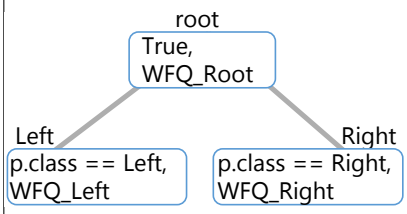
[2]Any state visible on the switch after processing N consecutive packets is identical to a serial execution of the transactions across the N packets in order of packet arrival [37].

[3]In this paper, a flow is any set of packets sharing common values for specific packet fields.

[4]An approximation is required because the original WFQ algorithm [20] has a complex virtual time calculation.



(a) Algorithm



(b) Scheduling tree

```
f = flow(p) # see caption below
if f in last_finish:
  p.start = max(virtual_time, last_finish[f])
else:
  p.start = virtual_time
last_finish[f] = p.start + p.length / f.weight
p.rank = p.start
```

(c) Scheduling transaction for WFQ_Root, WFQ_Left, and WFQ_Right.

Figure 2: Programming HPFQ using PIFOs. "Left" and "Right" are classes. A, B, C, and D are flows. Within each tree node in the scheduling tree, the first line is the packet predicate and the second is the scheduling transaction. All three nodes execute the same code for the scheduling transaction except for their flow() function, which returns a packet's flow/class. For WFQ_Root, it returns the packet's class: Left/Right. For WFQ_Left and WFQ_Right, it returns the packet's flow: A/B or C/D.

ness Queueing (SFQ) [30], and Start-Time Fair Queueing (STFQ) [25]. We consider STFQ here, and show how to program it using the scheduling transaction in Figure 1.

Before a packet is enqueued, STFQ computes a *virtual start time* for that packet (`p.start` in Figure 1) as the maximum of the *virtual finish time* of the previous packet in that packet's flow (`last_finish[f]` in Figure 1) and the current value of the *virtual time* (`virtual_time` in Figure 1), a state variable that tracks the virtual start time of the last dequeued packet across all flows (§5.5 discusses how this state variable can be accessed on enqueue). Packets are scheduled in order of increasing virtual start times, which is the packet's rank in the PIFO.

2.2 Scheduling trees

Scheduling algorithms that require changing the relative order of buffered packets when a new packet arrives cannot be programmed using a single scheduling transaction and PIFO. An important class of such algorithms are *hierarchical* schedulers that compose multiple scheduling policies at different levels of the hierarchy. We introduce a *scheduling tree* for such algorithms.

To illustrate a scheduling tree, consider Hierarchical Packet Fair Queueing (HPFQ) [13]. HPFQ first divides link capacity between classes, then recursively between sub classes in each class, all the way down to the leaf nodes. Figure 2a provides an example; the number on each child indicates its weight relative to its siblings. HPFQ cannot be realized using a single scheduling transaction and PIFO because the relative scheduling order of packets that are already buffered can change with future packet arrivals (Section 2.2 of the HPFQ paper [13] provides an example).

HPFQ *can*, however, be realized using a tree of PIFOs, with a scheduling transaction attached to each PIFO in the tree. To see how, observe that HPFQ executes WFQ at each level of the hierarchy, with each node using WFQ among its children. As discussed in §2.1, a single PIFO encodes the current scheduling order for WFQ, i.e., the scheduling order if there are no further arrivals. Similarly, a tree of PIFOs (Figure 3), where each PIFO's elements are either packets or references to other PIFOs, can be used to encode the current scheduling order of HPFQ and other hierarchical scheduling algorithms. To determine this scheduling order, inspect

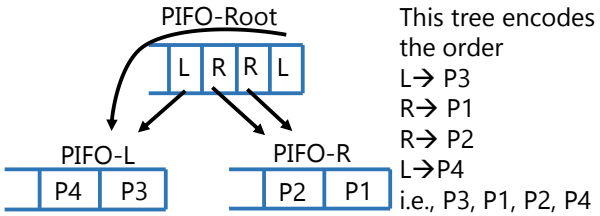


Figure 3: PIFO trees encode current scheduling order.

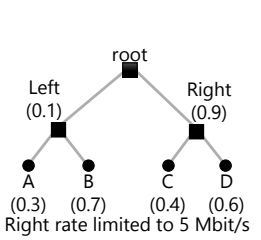
the root PIFO to determine the next child PIFO to schedule. Then, recursively inspect the child PIFO to determine the next grandchild PIFO to schedule, until reaching a leaf PIFO that determines the next packet to schedule.

The current scheduling order of the PIFO tree can be modified as packets are enqueued, by executing a scheduling transaction at each node in the PIFO tree. This is our second programming abstraction: a *scheduling tree*. Each node in this tree is a tuple with two attributes. First, a packet predicate that specifies which packets execute that node's scheduling transaction before enqueuing an element into that node's PIFO; this element is either a packet or a reference to a child PIFO of the node. Second, a scheduling transaction that specifies how the rank is computed for elements (packet or PIFO references) that are enqueued into the node's PIFO. Figure 2b shows an example for HPFQ.

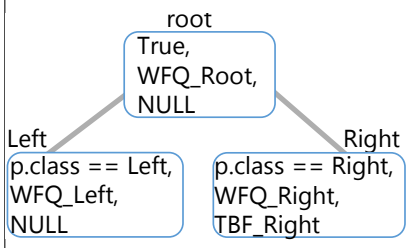
When a packet is enqueued into a scheduling tree, it executes one transaction at each node whose packet predicate matches the arriving packet. These nodes form a path from a leaf to the root of the tree and the transaction at each node on this path updates the scheduling order at that node. One element is enqueued into the PIFO at each node on the path from the leaf to the root. At the leaf node, that element is the packet itself; at the other nodes, it is a reference to the next PIFO on the path towards the leaf. Packets are dequeued in the order encoded by the tree of PIFOs (Figure 3).

2.3 Shaping transactions

So far, we have only considered work-conserving scheduling algorithms. *Shaping transactions* allow us to program non-work-conserving scheduling algorithms. Non-work-conserving algorithms differ from work-conserving algorithms in that they decide the *time* at which packets are



(a) Algorithm



(b) Scheduling tree

```
tokens = tokens + r * (now - last_time)
if (tokens > B):
  tokens = B
if (p.length <= tokens):
  p.send_time = now
else:
  p.send_time = now + (p.length - tokens) / r
tokens = tokens - p.length
last_time = now
p.rank = p.send_time
```

(c) Shaping transaction for TBF_Right.

Figure 4: Programming Hierarchies with Shaping using PIFOs. The third line within each tree node in the scheduling tree is the shaping transaction. The scheduling transactions for WFQ_Right, WFQ_Left, and WFQ_Root are identical to Figure 2.

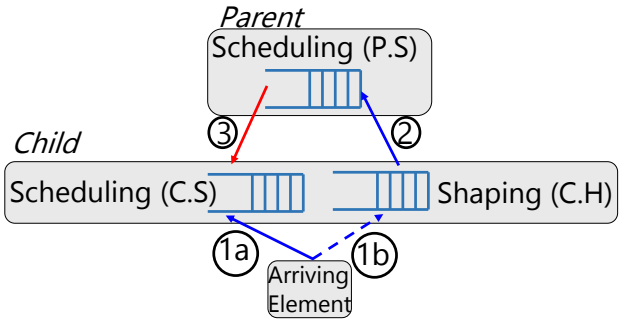
scheduled as opposed to the scheduling *order*. As an example, consider the algorithm shown in Figure 4a, which extends the previous HPFQ example with the requirement that the Right class be limited to 5 Mbit/s. We refer to this example throughout the paper as *Hierarchies with Shaping*.

To motivate our abstraction for non-work-conserving algorithms, recall that a PIFO tree encodes the current scheduling order, by walking down the tree from the root PIFO to a leaf PIFO to schedule packets. With this encoding, a PIFO reference can be scheduled only if it resides in a PIFO and there is a chain of PIFO references from the root PIFO to that PIFO reference. To program non-work-conserving scheduling algorithms, we provide the ability to *defer* when a PIFO reference is enqueued into the PIFO tree, and hence is available for scheduling.

To defer enqueues into the PIFO tree, we augment nodes of the scheduling tree with an optional third attribute: a *shaping transaction* that is executed on all packets matched by the node's packet predicate. The shaping transaction on a node determines when a reference to the node's PIFO is available for scheduling in the node's *parent's* PIFO. The shaping transaction is implemented using a *shaping PIFO* at the child—distinct from the scheduling PIFO at all nodes—that holds references to the child's scheduling PIFO until they are released to the parent's scheduling PIFO. The shaping transaction uses the wall-clock departure time as the rank for the shaping PIFO, unlike the scheduling transaction that uses the relative scheduling order as the rank.

Once a reference to the child's scheduling PIFO has been released to the parent's scheduling PIFO from the child's shaping PIFO, it is scheduled by executing the parent's scheduling transaction and enqueuing it in the parent's scheduling PIFO. If a node has no shaping transaction, references to that node's scheduling PIFO are immediately enqueued into its parent's scheduling PIFO with no deferral. The dequeue logic during shaping still follows Figure 3: dequeue recursively from the root until we schedule a packet.

Figure 4c shows an example of a shaping transaction that defers enqueues based on a Token Bucket Filter (TBF) with a rate-limit of r and a burst allowance B. Here, the packet's wall clock departure time (`p.send_time`), is used as its rank in the shaping PIFO. Figure 4b shows how to use this shaping transaction to program Hierarchies with Shaping:



1a. Time 0: Enqueue element into C.S.
1b. Time 0: Enqueue ref to C.S into C.H.
2. Time T (> 0): C.H dequeues ref, enqueues it into P.S.
3. Time T′ (> T): P.S dequeues from C.S.

Figure 5: *Child's* shaping transaction (1b) *defers* enqueue into *Parent's* scheduling PIFO (2) until time T. Blue arrows show enqueue paths. Red arrows show dequeue paths.

the TBF shaping transaction (TBF_Right) determines when PIFO references for Right's scheduling PIFO are released to Root's scheduling PIFO.

Timing of operations during shaping. When a packet is enqueued in a tree of PIFOs, it executes a scheduling transaction at the leaf node whose predicate matches this packet. It then continues upward towards the root executing scheduling transactions along the path, until it reaches the first node that also has a shaping transaction attached to it. Figure 5 shows the operations that occur at this node, *Child*, and its parent, *Parent*, to implement shaping.

Two transactions are executed at *Child*: the original scheduling transaction to push an element into *Child*'s scheduling PIFO (step 1a in Figure 5) and a shaping transaction to push an element R (step 1b), which is a reference to *Child*'s scheduling PIFO, into *Child*'s shaping PIFO. After R is pushed into *Child*'s shaping PIFO, further transactions for this packet are suspended until R's rank, the wall-clock time T, is reached.

At T, R will be dequeued from *Child*'s shaping PIFO and enqueued into *Parent*'s scheduling PIFO (step 2), making it available for scheduling at *Parent*. The rest of the packet's path to the root is now resumed starting at *Parent*. This suspend-resume process can occur multiple times if there are

multiple nodes with shaping transactions along a packet's path from its leaf to the root.

3. THE EXPRESSIVENESS OF PIFOS

In addition to the three examples from §2, we now provide several more examples (§3.1 through §3.4) and also describe the limitations of our programming model (§3.5).

3.1 Least Slack-Time First

Least Slack-Time First (LSTF) [29, 31] schedules packets at each switch in increasing order of packet slacks, i.e., the time remaining until each packet's deadline. Packet slacks are initialized at an end host or edge switch and are decremented by the wait time at each switch's queue. We can program LSTF using a simple scheduling transaction:

```
p.rank  = p.slack + p.arrival_time
```

The addition of the packet's arrival time to the slack already carried in the packet ensures that packets are dequeued in order of their slack at the time of dequeue, not enqueue. Then, after packets are dequeued, we subtract the time at which the packet is dequeued from the packet's slack, which has the effect of decrementing the slack by the wait time at the switch's queue. This subtraction can be achieved by programming the egress pipeline of a programmable switch [17] to decrement one header field by another.

3.2 Stop-and-Go Queueing

```
if (now >= frame_end_time):
  frame_begin_time = frame_end_time
  frame_end_time   = frame_begin_time + T
p.rank = frame_end_time
```

Figure 6: Shaping transaction for Stop-and-Go Queueing.

Stop-and-Go Queueing [24] is a non-work-conserving algorithm that provides bounded delays to packets using a framing strategy. Time is divided into non-overlapping frames of equal length `T`, where every packet arriving within a frame is transmitted at the end of the frame, smoothing out any burstiness in traffic patterns induced by previous hops.

The shaping transaction in Figure 6 specifies the scheme. `frame_begin_time` and `frame_end_time` are two state variables that track the beginning and end of the current frame in wall-clock time. When a packet is enqueued, its departure time is set to the end of the current frame. Multiple packets with the same departure time are sent out in first-in first-out order, as guaranteed by a PIFO's semantics for breaking ties with equal ranks (§2).

3.3 Minimum rate guarantees

A common scheduling policy on many switches today is providing a minimum rate guarantee to a flow, provided the sum of such guarantees does not exceed the link capacity. A minimum rate guarantee can be programmed using PIFOs with a two-level PIFO tree, where the root of the tree implements strict priority scheduling across flows. Flows below their minimum rate are scheduled preferentially to flows above their minimum rate. Then, at the next level of the tree, the PIFOs implement the FIFO discipline for each flow.

```
# Replenish tokens
tb = tb + min_rate * (now - last_time)
if (tb > BURST_SIZE):
  tb = BURST_SIZE

# Check if we have enough tokens
if (tb > p.size):
  p.over_min = 0 # under min. rate
  tb = tb - p.size
else:
  p.over_min = 1 # over min. rate

last_time = now
p.rank = p.over_min
```

Figure 7: Scheduling transaction for min. rate guarantees.

When a packet is enqueued, we execute a FIFO scheduling transaction at its leaf node, setting its rank to the wall-clock time on arrival. At the root, a PIFO reference (the packet's flow identifier) is pushed into the root PIFO using a rank that reflects whether the flow is above or below its rate limit after the arrival of the current packet. To determine this, we run the scheduling transaction in Figure 7 that uses a token bucket (the state variable `tb`) that can be filled up until `BURST_SIZE` to decide if the arriving packet puts the flow above or below `min_rate`.

Note that a single PIFO node with the scheduling transaction in Figure 7 is not sufficient. It causes packet reordering within a flow: an arriving packet can cause a flow to move from a lower to a higher priority and, in the process, leave before low priority packets from the same flow that arrived earlier. The two-level tree solves this problem by attaching priorities to transmission opportunities for a specific flow, not specific packets. Now if an arriving packet causes a flow to move from low to high priority, the next packet scheduled from this flow is the earliest packet of that flow chosen in FIFO order, not the arriving packet.

3.4 Other examples

We now briefly describe several more scheduling algorithms that can be programmed using PIFOs.

1. **Fine-grained priority scheduling.** Many algorithms schedule the packet with the lowest value of a field initialized by the end host. These algorithms can be programmed by setting the packet's rank to the appropriate field. Examples of such algorithms and the fields they use are strict priority scheduling (IP TOS field), Shortest Flow First (flow size), Shortest Remaining Processing Time (remaining flow size), Least Attained Service (bytes received for a flow), and Earliest Deadline First (time until a deadline).
2. **Service-Curve Earliest Deadline First (SC-EDF) [34]** schedules packets in increasing order of a deadline computed from a flow's service curve, which specifies the service a flow should receive over any given time interval. We can program SC-EDF using a

scheduling transaction that sets a packet's rank to the deadline computed by the SC-EDF algorithm.

3. **Rate-Controlled Service Disciplines (RCSD) [42]** such as Jitter-EDD [41] and Hierarchical Round Robin [27] are a class of non-work-conserving schedulers that can be implemented using a combination of a rate regulator to shape traffic and a packet scheduler to schedule the shaped traffic. An RCSD algorithm can be programmed using PIFOs by setting the rate regulator using a shaping transaction and the packet scheduler using a scheduling transaction.
4. **Incremental deployment.** Operators may wish to use programmable scheduling only for a subset of their traffic. This can be programmed as a hierarchical scheduling algorithm, with one FIFO class dedicated to legacy traffic and another to experimental traffic. Within the experimental class, an operator could program any scheduling tree.

3.5 Limitations

Changing the scheduling order of all packets of a flow. Although a tree of PIFOs can enable algorithms where the relative scheduling order of buffered packets changes in response to new packet arrivals (§2.2), it does not permit arbitrary changes to the scheduling order of buffered packets. In particular, it does not support changing the scheduling order for *all* buffered packets of a flow when a new packet from that flow arrives.

An example of an algorithm that needs this capability is pFabric [12], which introduces "starvation prevention" to schedule the packets of the flow with the shortest remaining size in FIFO order, in order to prevent packet reordering. To see why this is beyond the capabilities of PIFOs, consider the sequence of arrivals below, where pi(j) represents a packet from flow i with remaining size j, where the remaining size is the number of unacknowledged bytes in a flow.

1. Enqueue p0(7).
2. Enqueue p1(9), p1(8).
3. The scheduling order is: p0(7), p1(9), p1(8).
4. Enqueue p1(6).
5. The new order is: p1(9), p1(8), p1(6), p0(7).

Specifying these semantics are beyond the capabilities of the PIFO abstractions we have developed.[5] For instance, adding a level of hierarchy with a PIFO tree does not help. Suppose we programmed a PIFO tree implementing FIFO at the leaves and picking among flows at the root based on the remaining flow size. This would result in the scheduling order p1(9), p0(7), p1(8), p1(6), after enqueuing p1(6). The problem is that there is no way to change the scheduling order for *multiple* references to flow 1 in the root PIFO by enqueuing only one reference to flow 1.

A single PIFO *can*, however, implement pFabric without starvation prevention, which is identical to the Shortest Remaining Processing Time (SRPT) discipline (§3.4). It can also implement the Shortest Flow First (SFF) discipline (§3.4), which performs almost as well as pFabric [12].

Traffic shaping across multiple nodes in a scheduling tree. Our programming model attaches a single shaping and scheduling transaction to a tree node. This lets us enforce rate limits on a single node, but not across multiple nodes.

As an example, PIFOs cannot express the following policy: WFQ on a set of flows A, B, and C, with the additional constraint that the aggregate throughput of A+B doesn't exceed 10 Mbit/s. One work around is to implement this as HPFQ across two classes C1 and C2, with C1 containing A and B, and C2 containing C alone. Then, we enforce the rate limit of 10 Mbit/s on C1 as in Figure 4. However, this isn't equivalent to our desired policy. More generally, our programming model for programmable scheduling establishes a one-to-one relationship between the scheduling and shaping transactions, which is constraining for some algorithms.

Output rate limiting. The PIFO abstraction enforces rate limits using a shaping transaction, which determines a packet or PIFO reference's scheduling time *before* it is enqueued into a PIFO. The shaping transaction permits rate limiting on the *input* side, i.e., before elements are enqueued. An alternate form of rate limiting is on the *output*, i.e., by limiting the rate at which elements are scheduled.

To illustrate the difference, consider a scheduling algorithm with two priority queues, `LO` and `HI`, where `LO` is to be rate limited to 10 Mbit/s. To program this using input side rate limiting, we would use a shaping transaction to impose a 10 Mbit/s rate limit on `LO` and a scheduling transaction to implement strict priority scheduling between `LO` and `HI`. Now, assume packets from `HI` starve `LO` for a long period of time. During this time, packets from `LO`, after leaving the shaping PIFO, accumulate in the PIFO shared with `HI`. Now, if there are suddenly no more `HI` packets, all packets from `LO` are transmitted at line rate, and no longer rate limited to 10 Mbit/s for a transient period of time, i.e., until all instances of `LO` are drained out of the PIFO shared with `HI`. Input rate limiting still provides long-term rate guarantees, while output rate limiting provides short-term guarantees as well.

4. DESIGN

We now present a hardware design for a programmable scheduler based on PIFOs. We target shared-memory switches such as Broadcom's Trident II [3] (Figure 8). In these switches, a parser feeds packets from all ports into a shared ingress pipeline, after which they enter a shared scheduler and a similarly shared egress pipeline. To reduce chip area, combinational logic and memory for packet processing are shared across ports, both in the pipelines and in the scheduler. As a result, digital circuits on the switch must handle the aggregate processing requirements of all output ports at minimum packet size, e.g., 64 10 Gbit/s ports each transmitting 64 byte packets. This translates into ~1 billion packets per second, after accounting for minimum inter-packet gaps, or a 1 GHz clock frequency.

We first describe how scheduling and shaping transactions can be implemented (§4.1). Then, we show how a tree of PI-

[5] This is ironic because we started this project to implement pFabric in a programmable manner, and have ended up being able to do almost everything but that!

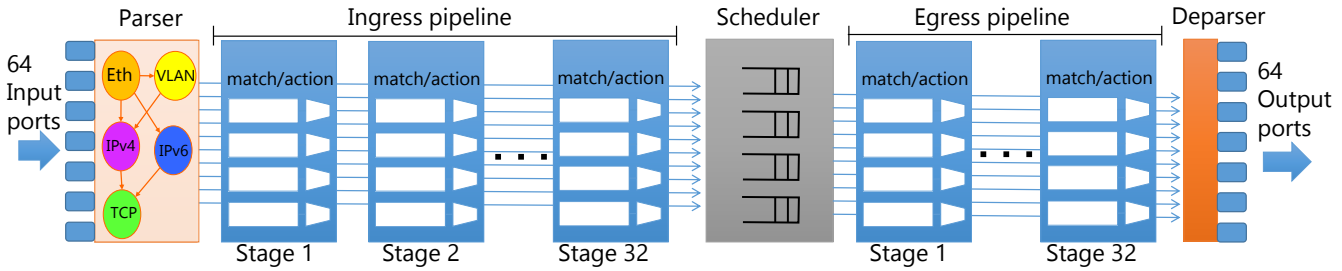


Figure 8: A 64-port shared memory switch. Combinational logic and memory are shared across ports, both in the pipelines and in the scheduler. The switch runs at a clock frequency of 1 GHz.

FOs can be realized using a full mesh of PIFO blocks by appropriately interconnecting these blocks (§4.2). We also describe how a compiler (§4.3) could automatically configure this mesh from a scheduling tree.

4.1 Scheduling and shaping transactions

To program and implement scheduling and shaping transactions, we use Domino [37], a recent system to program stateful data-plane algorithms at line rate. Domino introduces hardware primitives (atoms), and software abstractions (packet transactions) to program stateful algorithms on programmable switches [1, 5, 11, 17]

Atoms are processing units representing a programmable switch's instruction set, while a packet transaction is a block of code that is guaranteed to execute atomically. The Domino compiler compiles a scheduling or shaping packet transaction into an atom pipeline that executes the transaction atomically, rejecting the transaction if it can't run at line rate. Transactions may be rejected for two reasons; either because there are not enough atoms to execute the transaction, or because the transaction requires computation beyond the atoms' capabilities.

Domino proposes atoms that are expressive enough to support many data-plane algorithms and small enough to implement at 1 GHz . For instance, even the most expressive of these atoms, called Pairs, occupies only 6000 μm^2 in a 32 nm standard-cell library [37]; a 200 mm^2 switching chip [23] can support 300 Pairs atoms with < 2% area overhead. These 300 atoms are sufficient for many data-plane algorithms [37]. The Domino paper also shows how the STFQ transaction (Figure 1) can be run at 1 GHz on a switch pipeline with the Pairs atom.

Similarly, we could use the Domino compiler to compile other scheduling and shaping transactions to an atom pipeline. For example, the transactions for Token Bucket Filtering (Figure 4c), minimum rate guarantees (§3.3), Stop-and-Go queueing (§3.2), and LSTF (§3.1), can all be expressed as Domino programs. An important restriction in Domino is the absence of loops, which precludes rank computations containing a loop with an unbounded iteration count. We have not, however, encountered a scheduling or shaping transaction requiring this capability.

4.2 The PIFO mesh

We lay out PIFOs physically as a full mesh (Figure 9) of

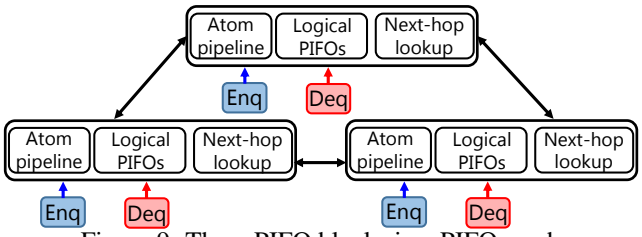


Figure 9: Three PIFO blocks in a PIFO mesh

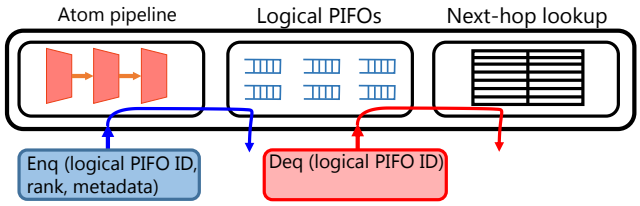


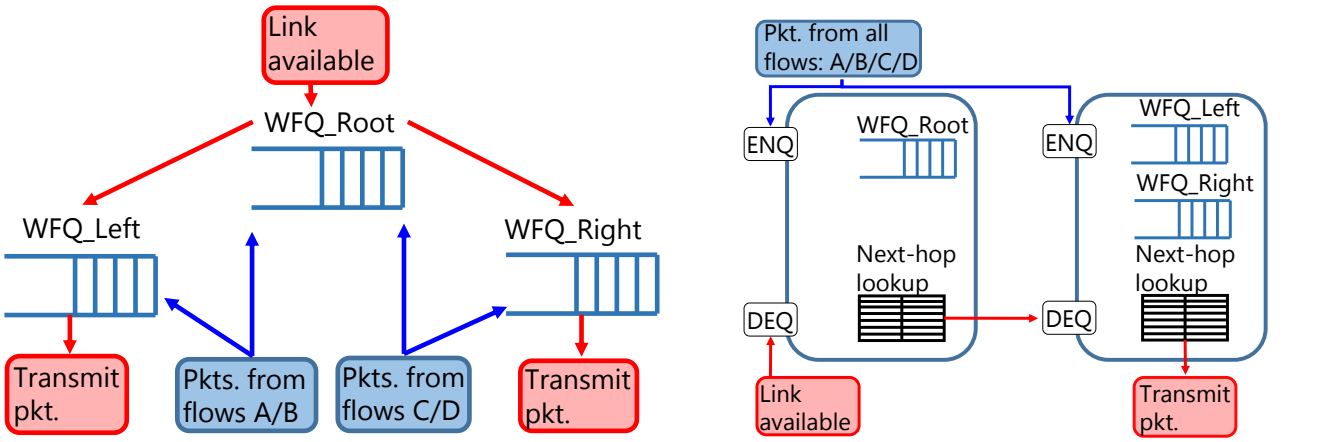
Figure 10: A single PIFO block. Enqueue operations execute transactions in the atom pipeline before enqueuing elements into a logical PIFO. Dequeue operations dequeue elements from logical PIFOs before looking up their next hop.

PIFO blocks (Figure 10). Each PIFO block supports multiple logical PIFOs. These logical PIFOs correspond to PIFOs for different output ports or different classes in a hierarchical scheduling algorithm, which share the combinational logic required for a PIFO. We expect a small number of PIFO blocks in a typical switch (e.g., fewer than five) because each PIFO block corresponds to a different level of a hierarchical scheduling tree and most practical hierarchical scheduling algorithms we know of do not require more than a few levels of hierarchy. As a result, a full mesh between these blocks is feasible (§5.3 has more details).

PIFO blocks run at 1 GHz and contain an atom pipeline to execute scheduling and shaping transactions before enqueuing into a logical PIFO. In every clock cycle, each PIFO block supports one enqueue and dequeue operation on a logical PIFO residing within that block (shaping transactions require more than one operation per clock cycle and are discussed in §4.4).

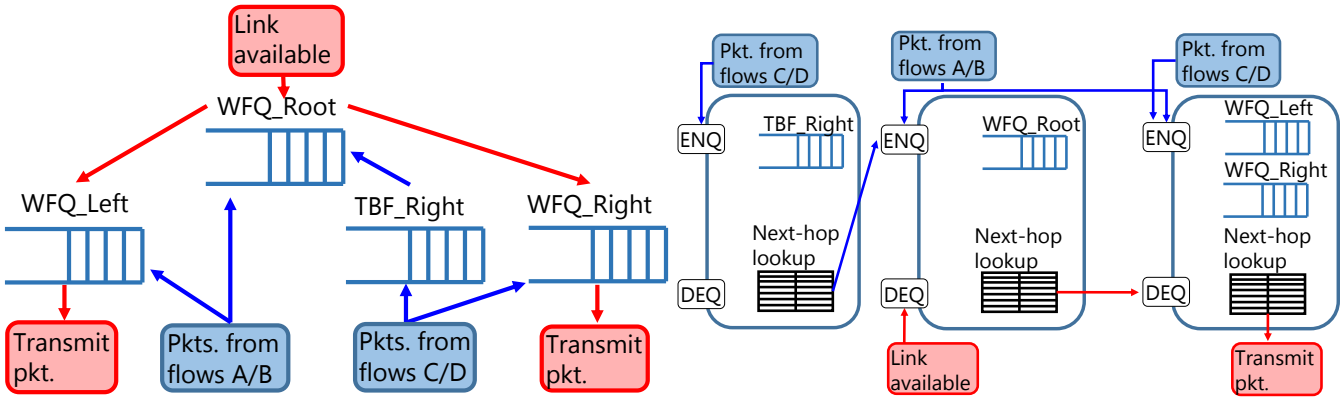
The interface to a PIFO block is:

1. Enqueue an element (packet or reference to another PIFO) given a logical PIFO ID, the element's rank, and some metadata that will be carried with the element such as the packet length required for STFQ's rank computation. The enqueue returns nothing.



(a) Logical PIFO tree for HPFQ (b) Physical PIFO mesh for HPFQ

Figure 11: Compiling HPFQ (Figure 2) to a PIFO mesh. On the left, the logical PIFO tree captures relationships between PIFOs: which PIFOs dequeue or enqueue into which PIFOs. Red arrows indicate dequeues, blue indicates enqueues. On the right, we show the physical PIFO mesh for the logical PIFO tree on the left, following the same notation.



(a) Logical PIFO tree for Hierarchies with Shaping (b) Physical PIFO mesh for Hierarchies with Shaping

Figure 12: Compiling Hierarchies with Shaping (Figure 2) to a PIFO mesh. Same comments as Figure 11 apply.

2. Dequeue from a specific logical PIFO ID within the block. The dequeue returns either a packet or a reference to another PIFO.

After a dequeue, besides transmitting a packet, a PIFO block may communicate with another for two reasons:

1. To dequeue a logical PIFO in another block, e.g., when dequeuing a sequence of PIFOs from the root to a leaf of a scheduling tree to transmit packets.
2. To enqueue into a logical PIFO in another block, e.g., when enqueuing a packet that has just been dequeued from a shaping PIFO.

We configure these post-dequeue operations using a small lookup table, which looks up the "next hop" following a dequeue. This lookup table specifies an operation (enqueue, dequeue, transmit), the PIFO block for the next operation, and any arguments the operation needs.

4.3 Compiling from a scheduling tree to a PIFO mesh

A programmer should not have to manually configure a PIFO mesh. Instead, a compiler translates from a scheduling tree to a PIFO mesh configuration implementing that tree. While we haven't prototyped this compiler, we illustrate how it would work using HPFQ (Figure 2) and Hierarchies with Shaping (Figure 4).

The compiler first converts the scheduling tree to a logical PIFO tree that specifies the enqueue and dequeue operations on each PIFO. Figures 11a and 12a show this tree for Figures 2 and 4 respectively. It then overlays this tree over a PIFO mesh by assigning every level of the tree to a PIFO block and configuring the lookup tables to connect PIFO blocks as required by the tree. Figure 11b shows the PIFO mesh for Figure 2, while Figure 12b shows the PIFO mesh for Figure 4.

If a particular level of the tree has more than one enqueue or dequeue from another level, which arises in the presence of shaping transactions (§4.4), we allocate new PIFO blocks to respect the constraint that any PIFO block provides one enqueue and dequeue operation per clock cycle, e.g., Figure 12b has an additional PIFO block containing TBF_Right alone. Finally, we use the Domino compiler to compile scheduling and shaping transactions.

4.4 Challenges with shaping transactions

Each PIFO block supports one enqueue and dequeue operation per clock cycle. This suffices for any algorithm that only uses scheduling transactions (work-conserving algorithms) because, for such algorithms, each packet needs at most one enqueue and one dequeue at each level of its scheduling tree, and we map the PIFOs at each level to a different PIFO block.

However, shaping transactions pose challenges. Consider Hierarchies with Shaping (Figure 12a). When the shaping transaction enqueues elements into TBF_Right, these elements will be released into WFQ_Root at a future time T. The external enqueue into WFQ_Root may also happen exactly at T, if a packet arrives at T. This creates a conflict because there are two enqueue operations in the same cycle. Conflicts may also occur on dequeues. For instance, if TBF_Right shared its PIFO block with another logical PIFO, dequeue operations to the two logical PIFOs could occur at the same time because TBF_Right can be dequeued at any arbitrary wall-clock time.

In a conflict, only one of the two operations can proceed. We resolve this conflict in favor of scheduling PIFOs. Shaping PIFOs are used for rate limiting to a rate lower than the line rate. Therefore, they can afford to be delayed by a few clocks until there are no conflicts. By contrast, delaying scheduling decisions of a scheduling PIFO would mean that the switch would idle and not satisfy its line-rate guarantee.

As a result, shaping PIFOs only get best-effort service. There are workarounds to this. One is overclocking the pipeline at (say) 1.25 GHz instead of 1 GHz, providing spare clock cycles for such best-effort processing. Another is to provide multiple ports to a PIFO block to support multiple operations every clock. These techniques are commonly used in switches for background tasks such as reclaiming buffer space, and can be applied to the PIFO mesh as well.

5. HARDWARE IMPLEMENTATION

This section describes the hardware implementation of our programmable scheduler. We discuss performance requirements (§5.1), the implementation of a PIFO block (§5.2), and the full-mesh interconnect between them (§5.3). Finally, we estimate the area overhead of our design (§5.4).

5.1 Performance requirements

Our goal is a programmable scheduler competitive with common shared-memory switches, such as the Broadcom Trident II [3], used in many datacenters today. Based on the Trident II, we target 1000 flows that can be flexibly allocated across logical PIFOs and a 12 MByte packet buffer size [6] with a cell size[6] of 200 bytes. In the worst case, every packet is a single cell. Hence, up to 60K packets/elements per PIFO block can be spread out over multiple logical PIFOs.

Based on these requirements, our baseline design targets a PIFO block that supports 64K packets and 1024 flows that can be shared across 256 logical PIFOs. Further, we target a 16-bit rank field and a 32-bit metadata field (e.g., `p.length` in Figure 1) for our PIFO block. We put 5 such blocks together into a 5-block PIFO mesh that can support up to 5 levels of hierarchy in a scheduling algorithm—sufficient for most practical hierarchical schedulers we know of.

[6]Packets in a shared-memory switch are allocated in small units called cells.

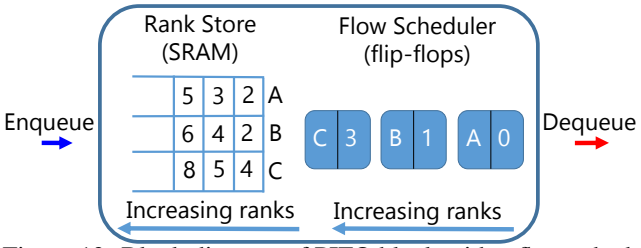


Figure 13: Block diagram of PIFO block with a flow scheduler and a rank store. Logical PIFOs and metadata are not shown for simplicity.

5.2 A single PIFO block

A PIFO block supports two operations: an enqueue that inserts an element into a logical PIFO and a dequeue to remove the head of a logical PIFO. We first describe an implementation of a PIFO block with a single logical PIFO and then extend it to multiple logical PIFOs in the same block.

One naive implementation is a single sorted array. An incoming element is compared against all array elements in parallel to determine a location for the new element, and then inserted there by shifting the array. However, each comparison needs one comparator circuit, and supporting 64K of these is infeasible.

At the same time, nearly all practical scheduling algorithms group packets into flows or classes,[7] e.g., based on traffic type, ports, or addresses. They then schedule a flow's packets in FIFO order because packet ranks increase across a flow's consecutive packets. This motivates a design with two parts (Figure 13):

1. A *flow scheduler* that picks the element to dequeue based on the rank of the *head* (earliest) elements of each flow. The flow scheduler is effectively a PIFO consisting of the head elements of all flows.
2. A *rank store*, a FIFO bank that stores the ranks of elements beyond the head for each flow in FIFO order.

This decomposition reduces the number of elements requiring sorting from the number of packets (64K) to the number of flows (1024). During an enqueue, an element (both rank and metadata) is appended to the end of the appropriate FIFO in the rank store. For a flow's first element, we bypass the rank store and directly push it into the flow scheduler. To permit enqueues into this PIFO block, we also supply a flow ID argument to the enqueue operation.

The FIFO bank needed for the rank store is a well-understood hardware design. Such FIFO banks are used to buffer packet payloads in switches and much engineering effort has gone into optimizing them. As a result, we focus our

[7]Last-In First-Out (LIFO) is a counterexample. We can handle LIFO by creating a new flow for every packet, if there are fewer than 1024 packets (flows) in the buffer at any time.

implementation effort on the flow scheduler alone.

The Flow Scheduler. The flow scheduler sorts an array of flows using the ranks of the head elements in each flow. It supports one enqueue *and* one dequeue to its enclosing PIFO block every clock cycle, which translates into the following operations on the flow scheduler every clock cycle.

1. Enqueue operation: Inserting a flow when the flow goes from empty to non-empty.
2. Dequeue operation: Removing a flow that empties once it is scheduled, (or) removing and reinserting a flow with the rank of the next element if the flow is still backlogged.

The operations above require the flow scheduler to internally support two primitives every clock cycle.

1. *Push* up to two elements into the flow scheduler: one each for an enqueue's insert and a dequeue's reinsert.
2. *Pop* one element: for the remove from a dequeue.

These primitives access all of the flow scheduler's elements in parallel. To facilitate this, we implement the flow scheduler in flip flops, unlike the rank store, which is in SRAM.

The flow scheduler is a sorted array, where a push is implemented by executing the three steps below (Figure 14).

1. Compare the incoming rank against all ranks in the array in parallel, using a comparator. This produces a bit mask of comparison results indicating if the incoming rank is greater/lesser than an array element's rank.
2. Find the first 0-1 transition in this bit mask, using a priority encoder, to determine the index to push into.
3. Push the element into this index, by shifting the array.

A pop is implemented by shifting the head element out of the sorted array.

So far, we have focused on a flow scheduler implementation handling a single logical PIFO. To handle multiple logical PIFOs, we keep elements sorted by rank, regardless of the logical PIFO they belong to; hence, the push logic doesn't change. To pop from a specific logical PIFO, we compare against all elements to find elements with that logical PIFO ID. Among these, we find the first using a priority encoder, and remove this element by shifting the array. The rank store implementation doesn't change when introducing logical PIFOs; however, we do require that a flow belong to exactly one logical PIFO.

To concurrently issue 2 pushes and 1 pop every clock cycle, we provision three parallel digital circuits (Figure 14). Both the push and pop require 2 clock cycles to complete and need to be pipelined to maintain the required throughput (Figure 15). For pushes, the first stage of the pipeline executes the parallel comparison and priority encoder steps to determine an index; the second stage pushes the element into the array using the index. Similarly, for pops, the first stage executes the equality check (for logical PIFO IDs) and priority encoder steps to compute an index; the second stage pops the head element out of the array using the index.

Our implementation meets timing at 1 GHz and supports up to one enqueue/dequeue operation on a logical PIFO within a PIFO block every clock cycle. Because a reinsert operation requires a pop, followed by an access to the rank

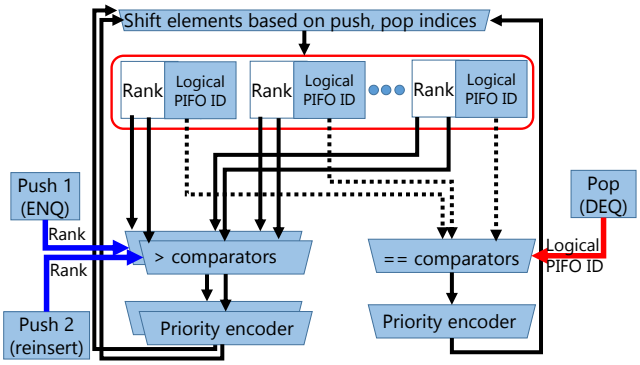


Figure 14: Hardware implementation of flow scheduler. Each element in the flow scheduler is connected to two > comparators (2 pushes) and one == comparator (1 pop).

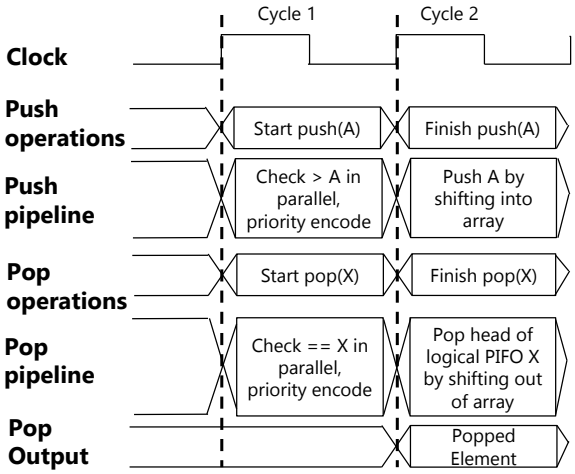


Figure 15: 2-stage pipeline for flow scheduler

store for the next element, followed by a push, our implementation supports a dequeue from the same logical PIFO only once every 4 cycles. This is because if a dequeue is initiated in clock cycle 1, the pop for the dequeue completes in 2, the rank store is accessed in 3, and the push is initiated in 4, making cycle 5 the earliest time to reissue a dequeue. This restriction is inconsequential in practice. A dequeue every 4 cycles from a logical PIFO is sufficient to service the highest link speed today, 100 Gbit/s, which requires a dequeue at most once every 5 clock cycles for a minimum packet size of 64 bytes. Dequeues to distinct logical PIFO IDs are still permitted every cycle.

5.3 Interconnecting PIFO blocks

An interconnect between PIFO blocks allows PIFO blocks to enqueue into and dequeue from other blocks. Because the number of PIFO blocks is small, we provide a full mesh between them. For a 5-block PIFO mesh as in our baseline design, this requires 5*4 = 20 sets of wires between PIFO blocks. Each set carries all the inputs required for specifying an enqueue and dequeue operation on a PIFO block.

For our baseline design (§5.1), for an enqueue, we require a logical PIFO ID (8 bits), the element's rank (16 bits), the element meta data (32 bits), and the flow ID (10 bits). For

a dequeue, we need a logical PIFO ID (8 bits) and wires to store the dequeued element's metadata field (32 bits). This adds up to 106 bits per set of wires, or 2120 bits for the mesh. This is a small number of wires for a chip. For example, RMT's match-action pipeline uses 4000 1-bit wires between a *a pair of pipeline stages* to move its 4K packet header vector between stages [17].

5.4 Area overhead

Because we target a shared-memory switch, the scheduling logic is shared across ports and a single PIFO mesh services an entire switch. Therefore, to estimate the area overhead of a programmable scheduler, we estimate the area overhead of a single PIFO mesh. Our overhead does not have to be multiplied by the number of ports and is the same for two shared-memory switches with equal aggregate packet rates, e.g., a 6-port 100G switch and a 60-port 10G switch.

To determine the area of a PIFO mesh, we compute the area of a single PIFO block and multiply it by the number of blocks because the area of the interconnect is negligible. For a single block's area, we separately estimate areas for the rank store, atom pipelines, and flow scheduler, and ignore the area of the small next-hop lookup tables. We estimate the rank store's area by using SRAM estimates [8], the atom pipeline's area using Domino [37], and the flow scheduler's area by implementing it in Verilog [9] and synthesizing it to gate-level netlist in a 16-nm standard cell library using the Cadence Encounter RTL Compiler [2]. The RTL Compiler also verifies that the flow scheduler meets timing at 1 GHz.

Overall, our baseline design consumes about 7.35 mm^2 of chip area (Table 1). This is about 3.7% of the chip area of a typical switching chip, using the minimum chip area estimate of 200 mm^2 provided by Gibb et al. [23]. In return for this 3.7%, we get a significantly more flexible packet scheduler than current switches, which provide *fixed* two or three-level hierarchical scheduling. Our 3.7% area overhead is similar to the overhead for other programmable switch functions, e.g., 2% for programmable parsing [23] and 15% for programmable header processing [17].

Varying the flow scheduler's parameters from the baseline. The flow scheduler has four parameters: rank width, metadata width, number of logical PIFOs, and number of flows. Among these, increasing the number of flows has the most impact on whether the flow scheduler meets timing at 1 GHz. This is because the flow scheduler uses a priority encoder, whose size is the number of flows and whose critical path delay increases with the number of flows. With other parameters set to their baseline values, we vary the number of flows to determine the eventual limits of a flow scheduler with today's transistor technology (Table 2), and find that we can scale to 2048 flows while still meeting timing at 1 GHz.

The remaining parameters affect the area of a flow scheduler, but have little effect on meeting timing at 1 GHz. For instance, starting from the baseline design of the flow scheduler that takes up 0.224 mm^2, increasing the rank width to 32 bits increases it to 0.317 mm^2, increasing the number of logical PIFOs to 1024 increases it to 0.233 mm^2, and increasing the metadata width to 64 bits increases it to 0.317 mm^2. In all cases, the flow scheduler continues to meet timing.

Component	Area in mm^2
Switching chip	200–400 [23]
Flow Scheduler	0.224 (from synthesis)
SRAM (1 Mbit)	0.145 [8]
Rank store	64 K * (16 + 32) bits * 0.145 mm^2 / Mbit = 0.445
Next pointers for linked lists in dynamically allocated rank store	64 K * 16 bit pointers * 0.145 = 0.148
Free list memory for dynamically allocated rank store	64 K * 16 bit pointers * 0.145 = 0.148
Head, tail, and count memory for each flow in the rank store	0.1476 (from synthesis)
One PIFO block	0.224 + 0.445 + 0.148 + 0.148 + 0.1476 = 1.11 mm^2
5-block PIFO mesh	5.55
300 atoms spread out over the 5-block PIFO mesh for rank computations	6000 μm^2* 300 = 1.8 mm^2 (§4.1, [37])
Overhead for 5-block PIFO mesh	(5.55 + 1.8) / 200.0 = 3.7 %

Table 1: A 5-block PIFO mesh incurs a 3.7% chip area overhead relative to a baseline switch.

# of flows	Area (mm^2)	Meets timing at 1 GHz?
256	0.053	Yes
512	0.107	Yes
1024	0.224	Yes
2048	0.454	Yes
4096	0.914	No

Table 2: The flow scheduler's area increases with the number of flows. The flow scheduler meets timing until 2048 flows.

5.5 Additional implementation concerns

Coordination between enqueue and dequeue. When computing packet ranks on enqueue, some scheduling algorithms access state modified on packet dequeues. An example is STFQ (§2.1) that accesses the `virtual_time` variable when computing a packet's virtual start time. This enqueue-dequeue coordination can be implemented in two ways. One is shared state that can be accessed on both enqueue and dequeue, similar to queue occupancy counters. Another is to periodically synchronize the enqueue and dequeue views of the same state: for STFQ, the degree of short-term fairness is directly correlated with how up-to-date the `virtual_time` information on the enqueue side is.

Buffer management. Our design focuses on programmable scheduling and does not manage the allocation of a switch's data buffer across flows. Buffer management can use static buffer limits for each flow. The limits can also be dynamic, e.g., RED [22] and dynamic buffer sizing [18].

In a shared-memory switch, buffer management is orthogonal to scheduling, and is implemented using counters that

track flow occupancy in a shared buffer. Before a packet is enqueued into the scheduler, if any counter exceeds a static or dynamic threshold, the packet is dropped. A similar design for buffer management could be used with a PIFO-based scheduler as well.

Priority Flow Control. Priority Flow Control (PFC) [7] is a standard that allows a switch to send a *pause* message to an upstream switch requesting it to cease transmission of packets belonging to particular flows. PFC can be integrated into our hardware design by masking out certain flows in the flow scheduler during the dequeue operation if they have been paused because of a PFC pause message, and unmasking them when a PFC *resume* message is received.

Multi-pipeline switches. The highest end switches today, such as the Broadcom Tomahawk [4], support aggregate capacities exceeding 3 Tbit/sec. At a minimum packet size of 64 bytes, this corresponds to an aggregate packet rate of ~6 billion packets/s. Because a single switch pipeline (Figure 8) typically runs at 1 GHz and processes a billion packets/s, such switches require multiple ingress and egress pipelines that share access to the scheduler subsystem alone.

In multi-pipeline switches, each PIFO block needs to support multiple enqueue and dequeue operations per clock cycle (as many as the number of ingress and egress pipelines) because packets can be enqueued from any of the input ports every clock cycle, and each input port could reside in any of the ingress pipelines. Similarly, each egress pipeline requires a new packet every clock cycle, resulting in multiple dequeues every clock cycle.

A full-fledged design for multi-pipeline switches is beyond this paper, but our current design facilitates a multi-pipeline implementation. A rank store supporting multiple pipelines is similar to the data buffer of multi-pipeline switches today. Building a flow scheduler to support multiple enqueues/dequeues per clock is relatively easy because it is maintained in flip flops, where it is simple to add multiple ports (unlike SRAM).

6. RELATED WORK

The Push-in First-out Queue. PIFOs were first introduced as a proof construct to prove that a combined input-output queued switch could exactly emulate an output-queued switch [19]. We show here that PIFOs can be used as an abstraction for programmable scheduling at line rate.

Packet scheduling algorithms. The literature is replete with scheduling algorithms [12, 13, 24, 25, 29, 35, 36, 42] . Yet, line-rate switches support only a few: DRR, traffic shaping, and strict priorities. As §3 shows, PIFOs allow a line-rate switch to run many of these scheduling algorithms, which, so far, have only been run on software routers.

Programmable switches. Recent work has proposed hardware architectures [1, 5, 11, 17] and software abstractions [16, 37] for programmable switches. While many packet-processing tasks can be programmed on these switches, scheduling isn't one of them. Programmable switches can *assist* a PIFO-based scheduler by providing a programmable ingress pipeline for scheduling and shaping transactions, without requiring a dedicated atom pipeline inside each PIFO block. However, they still need PIFOs for programmable scheduling.

Universal Packet Scheduling (UPS). UPS [31] shares our goal of flexible packet scheduling by seeking a single scheduling algorithm that is *universal* and can emulate any scheduling algorithm. Theoretically, UPS finds that the well-known LSTF scheduling discipline [29] is universal if packet departure times for the scheduling algorithm to be emulated are known up front. Practically, UPS shows that by appropriately initializing slacks, many different scheduling objectives can be emulated using LSTF. LSTF is programmable using PIFOs, but the set of schemes practically expressible with LSTF is limited. For example, LSTF cannot express:

1. Hierarchical scheduling algorithms such as HPFQ, because it uses only one priority queue.
2. Non-work-conserving algorithms. For such algorithms LSTF must know the departure time of each packet upfront, which is not practical.
3. Short-term bandwidth fairness in fair queueing, because LSTF maintains no switch state except one priority queue. As shown in Figure 1, programming a fair queueing algorithm requires us to maintain a virtual time state variable. Without this, a new flow could have arbitrary virtual start times, and be deprived of its fair share indefinitely. UPS provides a fix to this that requires estimating fair shares periodically, which is hard to do in practice.
4. Scheduling policies that aggregate flows from distinct endpoints into a single flow at the switch. An example is fair queueing across video and web traffic classes, regardless of endpoint. Such policies require the switch to maintain the state required for fair queueing because no end point sees all the traffic within a class. However, LSTF cannot maintain and update switch state progammatically.

The restrictions in UPS/LSTF are a result of a limited programming model. UPS assumes that switches are fixed and cannot be programmed to modify packet fields. Further, it only has a single priority queue. By using atom pipelines to execute scheduling and shaping transactions, and by composing multiple PIFOs together, PIFOs express a wider class of scheduling algorithms.

Hardware designs for priority queues. P-heap is a pipelined binary heap scaling to 4-billion entries [14, 15]. However, each P-heap supports traffic belonging to a *single* 10 Gbit/s input port in an input-queued switch and there is a separate P-heap instance for each port [14]. This per-port design incurs prohibitive area overhead on a shared-memory switch, and prevents sharing of the data buffer and binary heap across output ports. Conversely, it isn't easy to overlay multiple logical PIFOs over a single P-heap, which would allow the P-heap to be shared across ports.

7. DISCUSSION

Packet scheduling allocates scarce link capacity across contending flows. This allocation services an application or a network-wide objective, e.g., max-min fairness (WFQ) or minimum flow completion time (SRPT). Past work has demonstrated significant performance benefits resulting from switch support for flexible allocation [12, 21, 32, 39]. However, these benefits have remained unrealized, because today there isn't a pathway to implementing such schemes. PIFOs provide that pathway. They express many scheduling algorithms and are implementable at line rate.

How a programmable scheduler will be used is still unclear, but at the very least PIFOs present the network as an additional surface for deploying resource allocation schemes. No longer will transport protocol designers need to restrict themselves to end-host/edge-based solutions to datacenter transport. As an example of a immediate use case for PIFOs, one could run HPFQ for traffic isolation, where the classes corresponds to different tenants in a datacenter and the flows correspond to all source-destination VM pairs belonging to a tenant. This kind of isolation is much harder to provide with end-host/edge-based solutions today.

Looking forward, a programmable scheduler on switches could simplify packet transport. For instance, pFabric [12] minimizes flow completion times by coupling a simple switch scheduler (shortest remaining processing time) with a simple end-host protocol (line rate transmission with no congestion control). Other transport mechanisms [28, 32] leverage fair queueing in the network to simplify transport and make it more predictable.

That said, our current design is only a first step and can be improved in several ways.

1. A scheduling tree is more convenient than directly configuring a PIFO mesh, but it is still a low-level abstraction. Are there higher level abstractions?
2. Now that we have shown it is feasible, how will programmable scheduling be used in practice? This would involve surveying network operators to understand how programmable scheduling could benefit them. In turn, this would provide valuable design guidance for setting various parameters in our hardware design.
3. Beyond a few counter examples, we lack a formal characterization of algorithms that cannot be implemented using PIFOs. For instance, is there a simple, checkable property separating algorithms that can and cannot be implemented using PIFOs? Given an algorithm specification, can we automatically check if the algorithm can be programmed using PIFOs?
4. Our current design scales to 2048 flows. If allocated evenly across 64 ports, we could program scheduling across 32 flows at each port. This permits per-port scheduling across traffic aggregates (e.g., fair queueing across 32 tenants within a server), but not a finer granularity (e.g., 5-tuples). Ideally, to schedule at the finest granularity, our design would support 60K flows: the physical limit of one flow for each packet. We currently support up to 2048. Can we bridge this gap?

8. CONCLUSION

Until recently, it was widely assumed that the fastest switching chips would be fixed-function; a programmable device could not have the same performance. Recent research into programmable parsers [23], fast programmable switch pipelines [17], and languages to program them [16, 40], coupled with recent multi-Tbit/s programmable commercial chips [1, 11] suggests that change might be afoot.

But so far, it has been considered off-limits to program the packet scheduler—in part because the desired algorithms are so varied, and because the scheduler sits at the heart of the shared packet buffer where timing requirements are tightest. It has been widely assumed too hard to find a useful abstraction that can also be implemented in fast hardware.

PIFOs appear to be a very promising abstraction: they include a variety of existing algorithms, and allow us to express new ones. Further, they can be implemented at line rate with modest chip area overhead.

We believe the most exciting consequence will be the creation of many new schedulers, invented by network operators, iterated and refined, then deployed for their own needs. No longer will research experiments be limited to simulation and progress constrained by a vendor's choice of scheduling algorithms. Those needing a new algorithm could create their own, or even download one from an open-source repository or a reproducible SIGCOMM paper.

To get there, we will need real switching chips with programmable PIFO schedulers. The good news is that we see no reason why future switching chips can not include a programmable PIFO scheduler.

Acknowledgements

We are grateful to our shepherd, Jeff Mogul, the anonymous SIGCOMM reviewers, and Amy Ousterhout for many suggestions that greatly improved the clarity of the paper. We thank Ion Stoica for helpful discussions, Robert Hunt for help with the design of the compiler, and Radhika Mittal for helping us understand LSTF. This work was partly supported by NSF grant CNS-1563826 and a gift from the Cisco Research Center. We thank the industrial partners of the MIT Center for Wireless Networks and Mobile Computing (Wireless@MIT) for their support.

9. REFERENCES

[1] Barefoot: The World's Fastest and Most Programmable Networks. https://barefootnetworks.com/media/white_papers/Barefoot-Worlds-Fastest-Most-Programmable-Networks.pdf.

[2] Cadence Encounter RTL Compiler. http://www.cadence.com/products/ld/rtl_compiler.

[3] High Capacity StrataXGS®Trident II Ethernet Switch Series. http://www.broadcom.com/products/Switching/Data-Center/BCM56850-Series.

[4] High-Density 25/100 Gigabit Ethernet StrataXGS Tomahawk Ethernet Switch Series. http://www.broadcom.com/products/Switching/Data-Center/BCM56960-Series.

[5] Intel FlexPipe. http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ethernet-switch-fm6000-series-brief.pdf.

[6] Packet Buffers. http://people.ucsc.edu/~warner/buffer.html.

[7] Priority Flow Control: Build Reliable Layer 2 Infrastructure. http://www.cisco.com/en/US/prod/collateral/switches/ps9441/ps9670/white_paper_c11-542809_ns783_Networking_Solutions_White_Paper.html.

[8] SRAM - ARM. https://www.arm.com/products/physical-ip/embedded-memory-ip/sram.php.

[9] System Verilog. https://en.wikipedia.org/wiki/SystemVerilog.

[10] Token Bucket. https://en.wikipedia.org/wiki/Token_bucket.

[11] XPliant™ Ethernet Switch Product Family. http://www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html.

[12] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pFabric: Minimal Near-Optimal Datacenter Transport. In *SIGCOMM*, 2013.

[13] J. C. R. Bennett and H. Zhang. Hierarchical Packet Fair Queueing Algorithms. In *SIGCOMM*, 1996.

[14] R. Bhagwan and B. Lin. Design of a High-Speed Packet Switch for Fine-Grained Quality-of-Service Guarantees. In *ICC*, 2000.

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

[16] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM CCR*, July 2014.

[17] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *SIGCOMM*, 2013.

[18] A. K. Choudhury and E. L. Hahne. Dynamic Queue Length Thresholds for Shared-memory Packet Switches. *IEEE/ACM Trans. Netw.*, 6(2):130–140, Apr. 1998.

[19] S.-T. Chuang, A. Goel, N. McKeown, and B. Prabhakar. Matching Output Queueing with a Combined Input Output Queued Switch. *IEEE Journal on Selected Areas in Communications*, 17(6):1030–1039, Jun 1999.

[20] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *SIGCOMM*, 1989.

[21] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron. Decentralized Task-aware Scheduling for Data Center Networks. In *SIGCOMM*, 2014.

[22] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, Aug. 1993.

[23] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown. Design Principles for Packet Parsers. In *ANCS*, 2013.

[24] S. J. Golestani. A Stop-and-Go Queueing Framework for Congestion Management. In *SIGCOMM*, 1990.

[25] P. Goyal, H. M. Vin, and H. Chen. Start-time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. In *SIGCOMM*, 1996.

[26] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, A. Greenberg, and C. Kim. EyeQ: Practical Network Performance Isolation at the Edge. In *NSDI*, 2013.

[27] C. R. Kalmanek, H. Kanakia, and S. Keshav. Rate Controlled Servers for Very High-Speed Networks. In *GLOBECOM*, 1990.

[28] S. Keshav. Packet-Pair Flow Control. *IEEE/ACM Transactions on Networking*, 1994.

[29] J.-T. Leung. A New Algorithm for Scheduling Periodic, Real-Time Tasks. *Algorithmica*, 4(1-4):209–219, 1989.

[30] P. McKenney. Stochastic Fairness Queuing. In *INFOCOM*, 1990.

[31] R. Mittal, R. Agarwal, S. Ratnasamy, and S. Shenker. Universal Packet Scheduling. In *NSDI*, 2016.

[32] K. Nagaraj, D. Bharadia, H. Mao, S. Chinchali, M. Alizadeh, and S. Katti. NUMFabric: Fast and Flexible Bandwidth Allocation in Datacenters. In *SIGCOMM*, 2016.

[33] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. FairCloud: Sharing the Network in Cloud Computing. In *SIGCOMM*, 2012.

[34] H. Sariowan, R. L. Cruz, and G. C. Polyzos. SCED: A Generalized Scheduling Policy for Guaranteeing Quality-of-service. *IEEE/ACM Transactions on Networking*, Oct. 1999.

[35] L. E. Schrage and L. W. Miller. The Queue M/G/1 with the Shortest Remaining Processing Time Discipline. *Operations Research*, 14(4):670–684, 1966.

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

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

[38] A. Sivaraman, S. Subramanian, A. Agrawal, S. Chole, S.-T. Chuang, T. Edsall, M. Alizadeh, S. Katti, N. McKeown, and H. Balakrishnan. Towards Programmable Packet Scheduling. In *HotNets*, 2015.

[39] A. Sivaraman, K. Winstein, S. Subramanian, and H. Balakrishnan. No Silver Bullet: Extending SDN to the Data Plane. In *HotNets*, 2013.

[40] H. Song. Protocol-Oblivious Forwarding: Unleash the Power of SDN Through a Future-proof Forwarding Plane. In *HotSDN*, 2013.

[41] D. Verma, H. Zhang, and D. Ferrari. Guaranteeing Delay Jitter Bounds in Packet Switching Networks. In *TRICOMM*, 1991.

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