Programmable Packet Scheduling

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

ABSTRACT

Switches today provide a small set 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 builds on the observation that scheduling algorithms make two decisions: *in what order* to schedule packets and *when* to schedule them. Further, in many scheduling algorithms these decisions can be made when packets are enqueued. We leverage this observation to build a programmable scheduler using a single abstraction: the push-in first-out queue (PIFO), a priority queue that maintains the scheduling order and time for such algorithms.

We show that a programmable scheduler using PIFOs lets us program a wide variety of scheduling algorithms. We present a detailed hardware design for this scheduler for a 64-port 10 Gbit/s shared-memory switch with <4% chip area overhead on a 16-nm standard-cell library. Our design lets us program many sophisticated algorithms, such as a 5-level hierarchical scheduler with programmable scheduling algorithms at each level.

1. INTRODUCTION

Today's line-rate switches provide a menu of scheduling algorithms: typically, a combination of Deficit Round Robin [34], strict priority scheduling, and traffic shaping. A network operator can configure parameters in these algorithms. However, an operator cannot change the core algorithmic logic in an existing scheduling algorithm, or program a new one, without building new switch hardware.

By contrast, with a *programmable* packet scheduler, network operators would be able to deploy custom scheduling algorithms to better meet application requirements, e.g., minimizing flow completion times [9] using Shortest Remaining Processing Time [33], flexible bandwidth allocation across flows or tenants [31, 26] using Weighted Fair Queueing [17], or minimizing tail packet delays [16] using Least Slack Time First [28]. With a programmable packet scheduler, switch designers would implement scheduling algorithms as programs atop a programmable substrate. Moving scheduling algorithms into software makes it much easier to build and verify algorithms in comparison to implementing the same algorithms as rigid hardware IP.

This paper presents a design for programmable packet scheduling in line-rate switches. Our design is motivated by the observation that all scheduling algorithms make two key decisions: first, in what order should packets be scheduled, and second, at what time should each packet be scheduled. Furthermore, in many scheduling algorithms, these two decisions can be made when a packet is enqueued. This observation was first made in a recent position paper [36]. The same paper also proposed the *push-in first-out queue (PIFO)* [15] abstraction for maintaining the scheduling order or scheduling time for packets, when these can be determined on enqueue. A PIFO is a priority queue data structure that allows elements to be pushed into an arbitrary location based on an element's *rank*, but always dequeues elements from the head.

Building on the PIFO abstraction, this paper presents the detailed design, implementation, and analysis of feasibility of a programmable packet scheduler. To program a PIFO, we develop the notion of a *scheduling transaction*—a small program to compute an element's rank in a PIFO. We present a rich programming model built using PIFOs and scheduling transactions (§2) and show how to program a diverse set of scheduling algorithms in the model (§3): Weighted Fair Queueing [17], Token Bucket Filtering [7], Hierarchical Packet Fair Queueing [10], Class-Based Queueing [19, 20], Least-Slack Time-First [28], Stop-and-Go Queueing [22], the Rate-Controlled Service Disciplines [40], and fine-grained priority scheduling (e.g., Shortest Job First, Shortest Remaining Processing Time, Least Attained Service, and Earliest Deadline First).

Until now, all line-rate implementations of these scheduling 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 can't be programmed using a PIFO.

We present a detailed hardware design for a programmable scheduler using PIFOs (§4). We have implemented this design and synthesized it to an industry-standard 16 nm standard-cell library (§5). We find, contrary to conventional wisdom [34, 29], that transistor technology has scaled to a point where the sorting operation at the core of a PIFO is surprisingly cheap. As a consequence, we show that it is feasible to build a programmable scheduler, which

1. supports 5-level hierarchical scheduling, where the scheduling algorithms at each level are programmable.
2. runs at a clock frequency of 1 GHz—sufficient for a 64-port shared-memory switch with a 10 Gbit/s line rate per port.
3. incurs <4% chip area overhead relative to a shared-memory switch supporting a small set of scheduling algorithms.
4. handles the same buffering requirements as a typical shared-memory switch today [1] (about 60K packets and 1K flows).

2. A PROGRAMMING MODEL FOR PACKET SCHEDULING

This section introduces the basic abstractions and programming model we use to express packet scheduling algorithms. The key idea underlying this programming model is that any scheduling algorithm makes two decisions: the *order* in which packets are scheduled, and the *time* at which they are scheduled. These two decisions capture work-conserving and non-work-conserving scheduling algorithms respectively. Further, in many practical scheduling algorithms, the order and time can be determined when a packet is enqueued into the packet buffer.

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

1. The *push-in first-out queue (PIFO)* [15] data structure that maintains the scheduling order or scheduling time for algorithms where these can be determined at enqueue. A PIFO is a priority queue that allows elements to be pushed into an arbitrary location on enqueue based on an element's *rank*, but dequeues elements from the head.[1] A PIFO breaks ties between elements with the same rank in the order in which they were enqueued.
2. A set of operations on the PIFO data structure called *transactions* that compute an element's rank before enqueuing it into a PIFO.

We now describe the three main abstractions of 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 *tree* of scheduling transactions 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).

[1] We use the term rank instead of priority to avoid confusion with strict priority scheduling. Throughout this paper, lower ranks are dequeued first from the PIFO.

```
f = flow(p) // compute flow from packet p
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
```

Figure 1: Scheduling transaction for STFQ

2.1 Scheduling transactions

A scheduling transaction is a block of code that is executed for each packet before enqueueing it into a PIFO. The scheduling transaction computes a rank for the packet. This rank then determines the position in the PIFO where the packet is enqueued. Scheduling transactions are an instance of packet transactions [35] — blocks of code that are atomic and isolated from other such transactions. Packet transactions guarantee that any visible state is equivalent to a serial execution of these transactions across consecutive packets.

Scheduling transactions can be used to program work-conserving scheduling algorithms. In particular, a single scheduling transaction (and PIFO) is sufficient to program any scheduling algorithm where the relative scheduling order of packets already in the buffer does not change with the arrival of future packets.

Take Weighted Fair Queueing (WFQ) [17] as an example. WFQ provides weighted max-min allocation of link capacity across flows[2] sharing a link. Practical approximations to WFQ include Deficit Round Robin (DRR) [34], Stochastic Fairness Queueing (SFQ) [29], and Start-Time Fair Queueing (STFQ) [23]. We consider STFQ here.

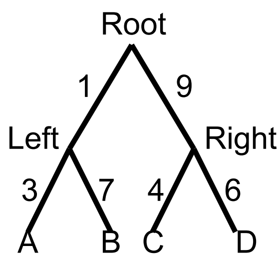
Before a packet is enqueued, STFQ computes a *virtual start time* for that packet as the maximum of the *virtual finish time* of the previous packet in that packet's flow and the current value of the *virtual time* (a single state variable that tracks the virtual start time of the last dequeued packet). Packets are then scheduled in order of increasing virtual start times. To program STFQ using a PIFO, we use the scheduling transaction shown in Figure 1.

Across all transactions in the paper, we use the notation `p.x` to refer to a packet field and set `p.rank` to the desired value at the end of the transaction based on the computations in the transaction.

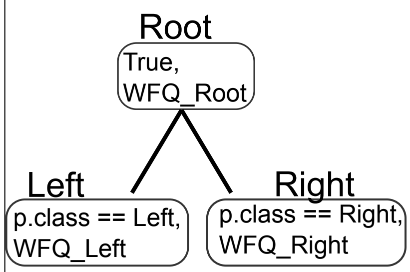
2.2 Tree of scheduling transactions

Scheduling algorithms that require changing the relative scheduling order of already buffered packets with the arrival of new packets cannot be implemented with a single scheduling transaction and PIFO. An important class of such algorithms are *hierarchical* schedulers that compose multi-

[2] We use the term 'flow' to generically describe a set packets with a common attribute. For example, a flow could be packets destined to the same subnet, or video packets, or a TCP connection.



(a) Algorithm



(b) Tree of PIFOs

```
f = flow(p)
// compute flow from packet p:
// (Left / Right for WFQ_Root)
// (A / B for WFQ_Left)
// (C / D for WFQ_Right)
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 3: Programming HPFQ using PIFOs

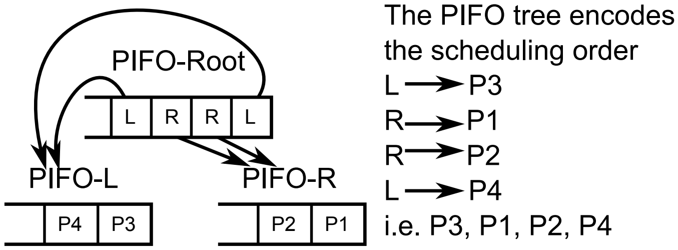


Figure 2: PIFO trees encode instantaneous scheduling order.

ple scheduling policies at different levels of hierarchy. We introduce the idea of a *tree of scheduling transactions* to program such algorithms.

To illustrate this idea, consider Hierarchical Packet Fair Queueing (HPFQ) [10]. HPFQ first apportions link capacity between classes, then recursively between sub classes belonging to each class, all the way down to the leaf nodes. Figure 3a provides an example scheduling hierarchy, the numbers on the edges indicating the relative weights of child nodes with respect to their parent's fair scheduler. 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 (see Section 2.2 of the HPFQ paper [10] for 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 simply executes some variant of WFQ at each level of the hierarchy, with each node using WFQ to pick among its children. As we showed in §2.1, a single PIFO encodes the instantaneous scheduling order for WFQ, i.e. the scheduling order if there are no further arrivals. Similarly, a tree of PIFOs (Figure 2), where each PIFO's elements are either packets or references to other PIFOs can be used to encode the instantaneous scheduling order of HPFQ (and other hierarchical scheduling algorithms) as follows. First, inspect the root PIFO to determine the next child PIFO to schedule. Then, recursively inspect the child PIFO to determine the next grand child PIFO to schedule, until we reach a leaf PIFO that determines the next packet to schedule. Figure 2 shows this encoding.

The instantaneous scheduling order of this PIFO tree can be modified as packets are enqueued, by providing a scheduling transaction for each node in the PIFO tree. This is our next programming abstraction: a tree of such scheduling transactions. Each node in this tree is a tuple with two attributes (Figure 3b). First, a packet predicate specifies which packets execute that node's scheduling transaction before the packet or a reference to another PIFO is enqueued into that node's PIFO. Second, a scheduling transaction specifies how the rank is computed for elements (packet or PIFO reference) that are enqueued into the node's PIFO.

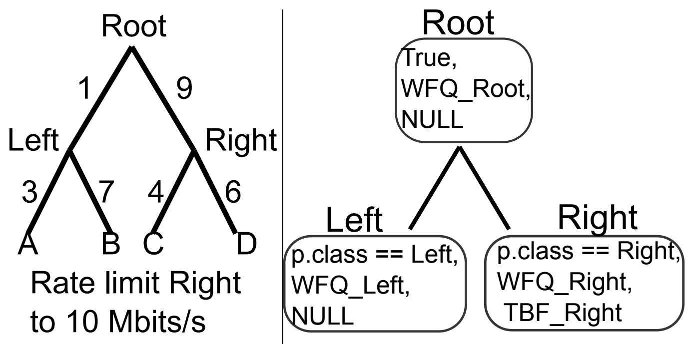
When a packet is enqueued into a PIFO tree, it executes one transaction at each node whose packet predicate matches the arriving packet. These nodes form a path from a leaf to the root of the tree and the transaction at each node on this path updates the scheduling order at that node. Notice that for each packet, 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 another PIFO in the tree that eventually points to the packet. Packets are dequeued in the order encoded by the PIFO tree (Figure 2). Figure 3 shows how a network operator would program HPFQ using this tree abstraction.

2.3 Shaping transactions

So far, we have only considered work-conserving scheduling algorithms. Our final abstraction, *shaping transactions*, allows 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 scheduled as opposed to just the scheduling *order*. As an example, consider the scheduling algorithm shown in Figure 4a, which extends the algorithm in Figure 3a with the additional requirement that the traffic in the Right class be limited to 10 Mbit/s, regardless of the offered load. We refer to this example throughout the paper as *Hierarchies with Shaping*.

To motivate our abstraction for this and other non-work-conserving algorithms, recall that a PIFO tree encodes the instantaneous scheduling order, by walking down the tree from the root PIFO to a leaf PIFO to schedule the next



(a) Algorithm

(b) Tree of PIFOs

```
tokens = min(tokens + r * (now - last_time), 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 scheduling transactions for WFQ_Right, WFQ_Left, and WFQ_Root are identical to Figure 1.

packet. With this encoding, an element (packet or 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 element. To program non-work-conserving scheduling algorithms, we provide the ability to defer when a packet or PIFO reference is enqueued into a PIFO and hence is visible for scheduling.

To defer enqueues into PIFOs, we augment nodes of the tree with a *shaping transaction* that is executed on all packets matched by the node's packet predicate. The shaping transaction determines when a packet (or a reference to a node's PIFO) is available for scheduling in the node's parent's PIFO. It is only after the time set by the shaping transaction that the shaped packet or PIFO reference is actually released to the parent node, where it is scheduled by executing the parent's scheduling transaction and enqueuing it in its PIFO.

Figure 4c shows an example of a shaping transaction that implements a Token Bucket Filter (TBF) with a rate-limit of r and a burst allowance B. Figure 4 shows how an operator would use this shaping transaction to program Hierarchies with Shaping. Here the TBF shaping transaction (TBF_Right) determines when the PIFO references for class Right are released to its parent node (Root). Until PIFO references for class Right are released to its parent, they are held in a separate shaping PIFO, distinct from the node's scheduling PIFO. The shaping PIFO uses the wall-clock departure time of an entry as its rank and pushes an entry to the parent's scheduling PIFO when the entry's wall clock time arrives.

The semantics of shaping transactions.

We explain the semantics of shaping transactions within a tree of scheduling and shaping transactions using the two nodes $Child$ and $Parent$ shown in Figure 5. When a packet is enqueued, it executes a scheduling transaction at the leaf node whose predicate matches this packet. It then continues upward towards the root, as before, executing scheduling transactions along the path, until it reaches the first node $Child$ that also has a shaping transaction attached to it.

At this point, we execute two transactions at $Child$: the original scheduling transaction to push an entry into $Child$'s scheduling PIFO and a shaping transaction to push an element R, which is a reference to $Child$'s scheduling PIFO, into $Child$'s shaping PIFO. After R is pushed into $Child$'s shaping PIFO, transactions are now suspended: no further transactions are executed when the packet is enqueued.

Say R had a wall-clock time T as its rank when it was pushed into $Child$'s shaping PIFO. At time T, R will be dequeued from $Child$'s shaping PIFO and enqueued into $Parent$'s scheduling PIFO, making $Child$'s scheduling PIFO now visible to $Parent$. The rest of the path to the root is now resumed starting at $Parent$. Note that 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. The dequeue logic remains identical to before, i.e., starting from the root dequeue recursively until we schedule a packet.

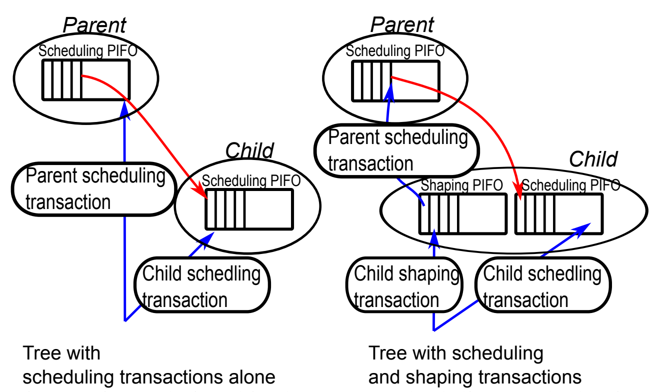


Figure 5: Relationship between scheduling and shaping transactions. The child shaping transaction suspends the execution of the parent scheduling transaction until the wall-clock time computed by the shaping transaction.

3. THE EXPRESSIVENESS OF PIFOS

In addition to the three detailed examples from §2, we

```
p.slack = p.slack - p.prev_wait_time;
p.rank  = p.slack;
```

Figure 6: Scheduling transaction for LSTF

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

Figure 7: Shaping transaction for Stop-and-Go Queueing

now provide three more examples (§3.1, §3.2, §3.3) to demonstrate that our programming model built using the PIFO abstraction is expressive. We also list several other examples that can be programmed using PIFOs (§3.4).

3.1 Least Slack-Time First

Least Slack-Time First (LSTF) [28, 30] 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 and decremented by the wait time at each switch's queue. We can program LSTF at a PIFO-enabled switch using the scheduling transaction in Figure 6. To compute wait times at upstream switches, we tag packets with their timestamps before and after they enter the queue, and compute the difference between the two.[3]

3.2 Stop-and-Go Queueing

Stop-and-Go Queueing [22] 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. To program Stop-and-Go Queueing, we use the shaping transaction in Figure 7. `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 (§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 all such guarantees doesn't exceed the link capacity. A minimum rate guarantee can be programmed using PIFOs by using a two-level PIFO tree, where the root of the tree implements strict priority scheduling across flows: those flows below their minimum rate are scheduled preferentially to flows above their minimum rate. Then, at the next level of the tree, flows implement the first-in first-out discipline across their packets.

[3] This can be achieved through proposals such as Tiny Packet Programs [25] and In-Band Network Telemetry [3].

```
// 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 8: Scheduling transaction for min. rate guarantees.

To program minimum rate guarantees, when a packet is enqueued, it executes a FIFO scheduling transaction at its leaf node, setting its rank to the wall-clock time on arrival. At the root level, 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 8 that uses a token bucket (tb) that can be filled up until BURST_SIZE to decide if the arriving packet puts the flow below or above a particular flow rate.

Note that "collapsing" this tree into a single node that implements the scheduling transaction in Figure 8 does not work because 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 flow that arrived earlier. The 2-level tree solves this problem by attaching priorities to transmission opportunities for a specific flow and not specific packets. Now if an arriving packet causes a flow to move from low to high priority, the next packet that is scheduled from this flow is the earliest packet from that flow—not the arriving packet.

3.4 Other examples

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

1. **Fine-grained priority scheduling:** Many algorithms implement fine-grained priority scheduling and schedule the packet with the lowest value of a field initialized by the end host. These algorithms can be programmed by using a scheduling transaction to set 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 Job First (flow size), Shortest Remaining Processing Time (remaining flow size), Least Attained Service (service received in bits for a flow), and Earliest Deadline First (time until a deadline).
2. **Service-Curve Earliest Deadline First (SC-EDF) [32]** is a scheduling algorithm that schedules

packets in increasing order of a deadline computed from a flow's service curve (a specification of 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. **First-In First-Out** can be programmed by using a scheduling transaction that sets a packet's rank to the wall-clock time on arrival.
4. **Rate-Controlled Service Disciplines (RCSD) [40]** are a class of non-work-conserving scheduling algorithms that can be implemented using a combination of a rate regulator to shape traffic and a packet scheduler to schedule traffic. An algorithm from the RCSD framework can be programmed using PIFOs by expressing the rate regulator using a shaping transaction and the packet scheduler using a scheduling transaction. Examples in this class include Jitter-EDD [39] and Hierarchical Round Robin [27].
5. **Class-Based Queueing (CBQ) [19, 20]** is a hierarchical scheduling algorithm that first schedules among classes based on a priority field assigned to each class, and then uses fair queueing to schedule packets within a class. CBQ can be programmed by a using a two-level PIFO tree to realize inter-class and intra-class scheduling.

3.5 Limitations

We conclude by describing some algorithms that cannot be programmed using the PIFO abstraction.

Changing the scheduling order of all packets in a flow.

A PIFO allows an arriving element (packet or PIFO reference) to determining its own scheduling order relative to all other elements currently in the PIFO. A PIFO does not allow the arriving element to change the scheduling order of all elements belonging to that element's flow that are already present in the PIFO.

An example of an algorithm that needs this capability is pFabric [9], which schedules the earliest packet from the flow with the shortest remaining processing time, to prevent packet reordering. To illustrate why this is beyond a PIFO's capabilities, consider the sequence of arrivals below, where pi(j) represents a packet from flow i with remaining processing time j.

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

The order of all packets from flow 1 needs to change in response to the arrival of a single packet p1(6) from flow 1, which is beyond a PIFO's capabilities.

Traffic shaping across multiple nodes in a tree.

Our programming model for scheduling attaches a single shaping and scheduling transaction to a node. This lets us enforce rate limits on a single node, but not across multiple nodes in the tree. As an illustration, 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 and B together 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. However, this isn't equivalent to our desired policy. More generally, our programming model for programmable scheduling establishes a 1-to-1 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.

As an example, consider a scheduling algorithm with two priority queues, low and high, where low 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 low and a scheduling transaction to implement strict priority scheduling between low and high. Now, assume packets from high starve low for an extended period of time. During this time, packets from low get rate limited through the shaping transaction and accumulate in the PIFO shared with high. Now, if there are suddenly no more high packets, all packets from low would get scheduled out at line rate, and no longer be rate limited to 10 Mbit/s over a transient period of time (until all instances of low are drained out of the PIFO shared with high). Input rate limiting still provides long-term guarantees on the rate, while output rate limiting provides these guarantees on shorter time scales as well.

4. DESIGN

We now present a hardware design for a programmable scheduler based on PIFOs. For concreteness, we target line-rate switches with an aggregate capacity of up to a billion packets/s, i.e., a 1 GHz clock frequency for the switch pipeline. Examples of such switch architectures include the RMT architecture [13] and Intel's FlexPipe [4], both of which provide 64 10 Gbit/s ports, which at the minimum packet size of 64 bytes translates to a billion packets/s.

For our hardware design, we first describe how scheduling and shaping transactions can be implemented (§4.1). Then, we show how a tree of PIFOs 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) can automatically configure this mesh given the description of the scheduling algorithm as a tree of scheduling and shaping transactions.

4.1 Scheduling and shaping transactions

Scheduling and shaping transactions compute an element's rank and execute atomically. By this, we mean that the state of the system (both the PIFO and any auxiliary state used in the transaction) after N enqueues is equivalent to serially executing N transactions one after the other, with no overlap between them. For our purpose, we need to execute these transactions atomically at line rate, i.e., the rate at which the switch receives packets (e.g., a billion packets/s).

To implement scheduling and shaping transactions, we use Domino [35], a recent system to program data-plane algorithms at line rate using packet transactions. Domino extends work on programmable line-rate switches [13, 4, 8] by providing hardware primitives, called atoms, and software abstractions, called packet transactions, to program data-plane algorithms at line rate. Atoms are small processing units that represent a programmable switch's instruction set. The Domino compiler compiles a packet transaction into a pipeline of atoms that executes the transaction atomically, rejecting the packet transaction if it can't run at line rate. Scheduling and shaping transactions are, in fact, packet transactions written in the Domino language.

The Domino paper proposes atoms that are rich enough to support many data-plane algorithms and small enough that they can be implemented at 1 GHz with modest chip area overhead. The largest of these atoms, called Pairs, occupies an area of 6000 μm^2 in a 32 nm standard-cell library; a switch with a chip area of 200 mm^2 [21] can support 300 of these with less than 1% area overhead. The Domino paper further shows how the transaction in Figure 1 can be run at 1 GHz on a switch pipeline with the Pairs atom.

In a similar manner, we could use the Domino compiler to compile scheduling and shaping transactions to a pipeline of atoms for other scheduling and shaping transactions. For example, the transactions for Token Bucker Filtering (Figure 4c), minimum rate guarantees (Figure 8), Stop-and-Go queueing(Figure 7), and LSTF (Figure 6), can all be expressed as Domino programs.

4.2 The PIFO mesh

The next component of the hardware is the actual PIFO itself. We lay out PIFOs physically as a full mesh of *PIFO blocks* (Figure 9), where each block implements multiple logical PIFOs. We expect a small number of PIFO blocks in a typical switch (e.g., less than five). The PIFO blocks correspond to different levels of a hierarchical scheduling tree. Most practical scheduling algorithms do not require more than a few levels of hierarchy, implying the required number of PIFO blocks is small as well. As a result, a full mesh between these blocks is feasible (see §5.4 for more details).

Each PIFO block runs at a clock frequency of 1 GHz and contains an atom pipeline to execute scheduling and shaping transactions. In every clock cycle, each PIFO block supports one enqueue and dequeue operation into any one of the logical PIFOs residing within that block. We address a logical PIFO within a block with a logical PIFO ID.

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.
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 need to communicate with another PIFO block for one of 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 packet).
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).

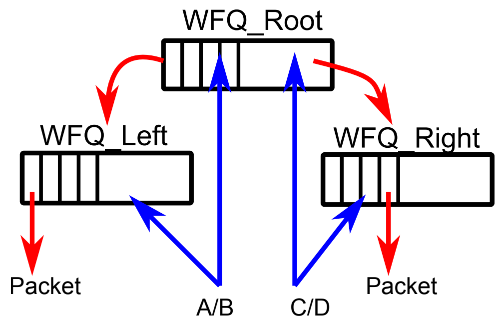
We configure these post-dequeue operations using a small lookup table, which looks up the "next hop" following a dequeue. This lookup table specifies an operation (enqueue, dequeue, transmit), the PIFO block for the next operation, and any arguments the operation needs.

4.3 Compiling to a PIFO mesh

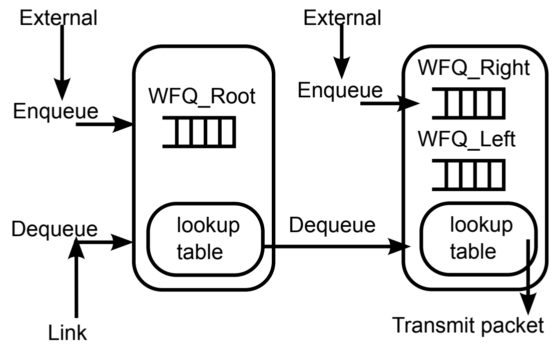
A PIFO mesh is configured by specifying the logical PIFOs within each PIFO block and by populating each PIFO block's next-hop lookup table. A compiler could configure the PIFO mesh from the scheduling algorithm specified as a tree of scheduling and shaping transactions. We illustrate the compilation using examples from Figures 3 and 4. The compiler first converts the tree with scheduling and shaping transactions to an equivalent tree representation that specifies the enqueue and dequeues operations on each PIFO. Figures 10a and 11a show this representation for Figures 3 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 10b shows the PIFO mesh corresponding to Figure 3. If a particular level of the tree has more than one enqueue or dequeue from another level, we allocate new PIFO blocks as required to respect the constraint that any PIFO block provides one enqueue and dequeue operation per clock cycle. This is shown in Figure 11b, which has an additional PIFO block containing TBF_Right[4] alone.

[4]More precisely, the shaping PIFO that the TBF_Right transaction

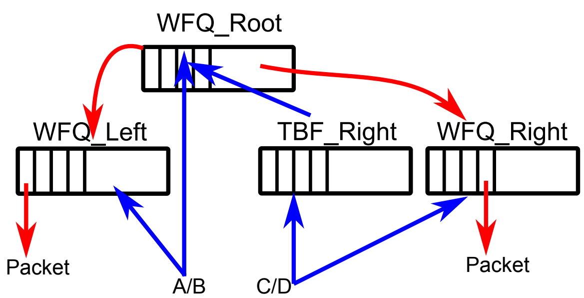


(a) Enqueue and dequeue paths for HPFQ. Downward arrows indicate dequeue paths. Upward arrows indicate enqueue paths.

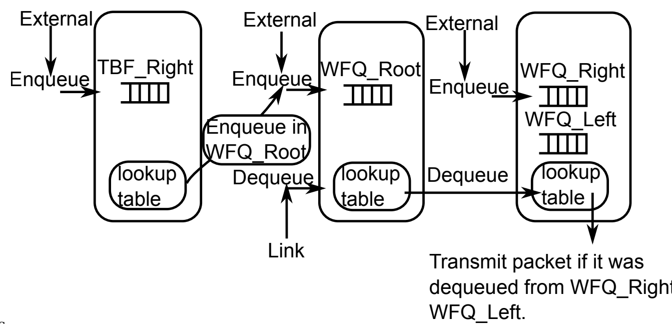


(b) Mesh configuration for HPFQ.

Figure 10: Compiling HPFQ to a PIFO mesh



(a) Enqueue and dequeue paths for hierarchies with shaping. Downward arrows indicate dequeue paths. Upward arrows indicate enqueue paths.



(b) Mesh configuration for Hierarchies with Shaping.

Figure 11: Compiling Hierarchies with Shaping to a PIFO mesh

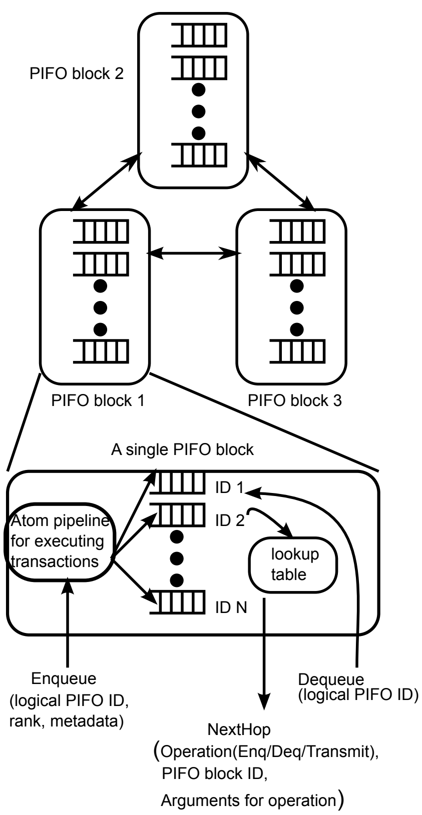


Figure 9: Block diagram of PIFO mesh

Challenges with shaping transactions.

Each PIFO block supports one enqueue and dequeue operation per clock cycle. This suffices to implement any algorithm which only uses scheduling transactions (i.e. work-conserving algorithms) at line-rate. The reason is that 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 the non-work-conserving algorithms in Figure 4. 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, because a packet arrives at that time. This creates a conflict because there are two enqueue operations in the same cycle. Conflicts may also manifest on the dequeue side, e.g., 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 PIFOs where element ranks are computed by scheduling transactions. This reflects the intuition that PIFOs controlled by shaping transactions are used for rate limiting to a rate lower than the line rate at which packets are normally scheduled. As a result, they can

enqueues into. We use a transaction's name to refer both to the transaction and the PIFO it enqueues into.

afford to be delayed by a few clocks until there are no more conflicts. By contrast, delaying scheduling decisions of a PIFO controlled by a scheduling transaction would mean that the switch would idle without transmitting a packet and not satisfy its line-rate guarantee.

Effectively, this means that PIFOs controlled by shaping transactions only get best-effort service. There are workarounds to this undesirable situation. One is to overclock the pipeline at a higher clock rate than required for packet processing, such as 1.25 GHz instead of 1 GHz, providing a few spare clock cycles for 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 today for low priority background tasks such as reclaiming buffer space. They can be applied to the PIFO mesh as well.

5. HARDWARE IMPLEMENTATION

We now describe the detailed implementation of a PIFO mesh. First, we discuss the implementation of a single PIFO block within the mesh (§5.2). Then, we synthesize this implementation to a 16 nm standard-cell library and evaluate its area overheads (§5.3). Finally, we evaluate the feasibility of interconnecting these PIFO blocks using a full mesh (§5.4).

5.1 Performance requirements

Our goal in implementing a PIFO block is to build a scheduler that is performance-competitive with current shared-memory switches, such as the Broadcom Trident [1], which are commonly used in datacenters today. As concrete performance targets, we target a 1 GHz clock frequency, which supports 64 10 Gbit/s ports. Based on the Broadcom Trident, we target a packet buffer size of 12 MBytes, and a cell size[5] of 200 bytes. In the worst case, every packet is a single cell, implying the need to support up to 60K packets/elements per PIFO block spread over multiple logical PIFOs. Similarly, based on the Broadcom Trident, we set a target of 1000 flows over which scheduling decisions are made at any port.

5.2 A single PIFO block

Functionally, a single PIFO block needs to support two operations: an enqueue operation that inserts an element into a logical PIFO and a dequeue operation to dequeue from the head of a logical PIFO. We first describe an implementation for a single logical PIFO and then extend it to multiple logical PIFOs in the same physical PIFO block.

A naive implementation is a flat sorted array of elements. An incoming element is compared against all elements in parallel to determine a location for the new element. It is then inserted into this location by shifting the array. However, each comparison needs an independent comparator circuit and supporting 60K of these is infeasible.

However, nearly all practical scheduling algorithms naturally group packets into flows or classes (e.g., based on traffic type, ports, or addresses) and schedule packets of a flow in FIFO order. In these algorithms, packet ranks strictly increase across consecutive packets in a flow. This motivates a PIFO design (Figure 12) with two parts: (1) a *flow scheduler* that picks the element to dequeue based on the rank of the *head* element of each flow, i.e., the element that arrived earliest, and (2) a *rank store*, a bank of FIFOs that stores element ranks beyond the head elements. This decomposition reduces the number of elements that need sorting from the number of packets (60 K) to the number of flows (1K). During an enqueue, an element (both rank and metadata) is appended to the end of the appropriate FIFO in the rank store.[6] To permit enqueues into this PIFO block, we also supply a flow ID argument to the enqueue operation.

Besides better scalability, an added benefit of this design is the ability to reuse a significant amount of engineering effort that has gone into building hardware IP for a bank of FIFOs. In a FIFO bank, each FIFO can grow and shrink dynamically as required, subject to an overall limit on the number of entries across the bank. Such banks are commonly used today to buffer packet data in a switch scheduler. As a result, we focus our implementation effort on building the flow scheduler alone.

The Flow Scheduler.

The core operation within the flow scheduler is to sort an array of flows based on the ranks of flow heads. The flow scheduler needs to support one enqueue and one dequeue to the PIFO block every clock cycle, which translates into two operations on the flow scheduler every clock cycle:

1. Inserting a flow when the flow goes from empty to non-empty (for the enqueue operation).
2. Removing a flow that goes empty once it is scheduled, or reinserting a flow with the priority of the next element if it is still backlogged once it is scheduled (for the dequeue operation).

The operations above require the ability to push up to two elements into the flow scheduler every cycle (one each for the insert and reinsert) and the ability to pop one element every cycle (for either the remove or reinsert). These operations require parallel operations on all elements in the flow scheduler. To facilitate this, we implement the flow scheduler data structure in flip flops (unlike the rank store, which is stored in SRAM).

Internally, the flow scheduler is organized as a sorted array, where a push is implemented by:

1. Comparing against all elements in the array in parallel, using a comparator circuit, to produce a bit mask with the comparison results.

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

[6] If this is the first element in the flow, it bypasses the rank store and is directly pushed into the flow scheduler data structure.

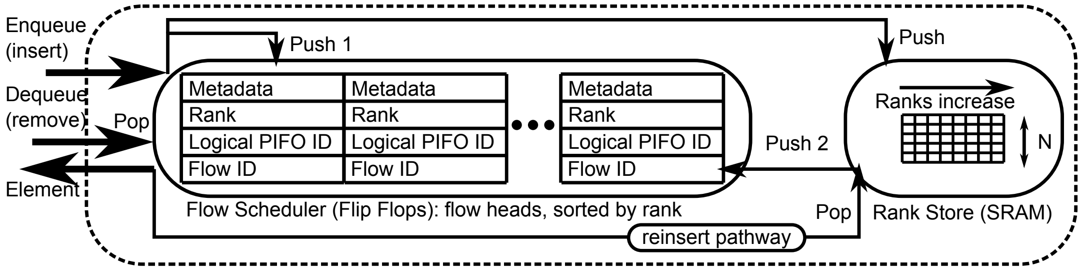


Figure 12: A single PIFO block with a flow scheduler and a rank store.

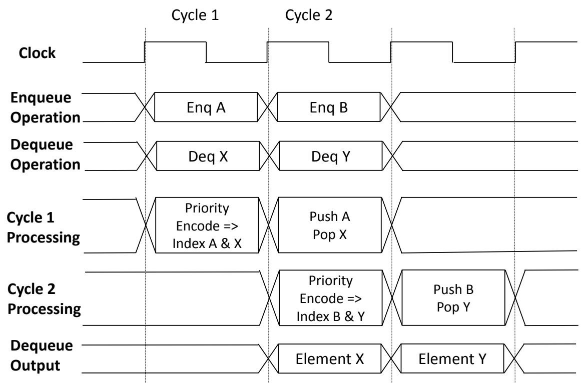


Figure 13: 2 stage pipeline for flow scheduler

2. Finding the first 0-1 transition in the bit mask, using a priority encoder circuit, to determine the index to push into.
3. Pushing the element into the appropriate 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 the flow scheduler for a single logical PIFO. To handle multiple logical PIFOs, we keep elements sorted by rank, regardless of which logical PIFO they belong to; hence, the push implementation doesn't change. To pop from a logical PIFO, we compare against all elements to determine elements with the same logical PIFO ID. Among these, we find the first using a priority encoder. We then remove this element by shifting the array.

The internal push and pop operations require 2 clock cycles each and hence need to be pipelined to support a throughput of 2 pushes and 1 pop every clock cycle. The first stage of this 2-stage pipeline for the push operation carries out the parallel comparison and priority encoder steps to determine an index; the second stage actually pushes the element into the array using the index. Similarly, for the pop operation, the first stage carries out 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. Figure 13 shows the 2-stage pipeline.

The pipelined implementation meets timing at 1 GHz and supports up to one enqueue/dequeue operation to any logical PIFO within a PIFO block every clock cycle. Because pops take 2 cycles, and a reinsert operation requires a pop followed by an access to the rank store for the next element, followed by a push, our implementation supports a dequeue from the same logical PIFO only once every 3 cycles (2 cycles for the pop and 1 cycle to access the rank store in SRAM). This restriction is inconsequential in practice. A dequeue every 3 cycles from a logical PIFO is sufficient to service the highest link speed in current switches, 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 Area overhead

We synthesized the design described above to a gate-level netlist in a 16-nm standard cell library to determine its area overhead. We first calculate area overheads for a baseline PIFO block that supports 1024 flows that can be shared across 256 logical PIFOs, and uses a 16-bit rank field and a 32-bit metadata field for each element in the PIFO. In addition, we assume our rank store supports 64K elements.

Table 1 calculates chip-area overheads when synthesized to a 16-nm standard-cell library. Overall, a 5-block PIFO mesh consumes about 7.35 mm^2 of chip area (including the area of the atom pipelines for rank computations). This is about 3.7% of the chip area of a typical switching chip today (using the minimum chip area estimate of 200 mm^2 provided by Gibb et al. [21]). Of course, a 5-block PIFO mesh is a significantly more capable packet scheduler compared to current switch schedulers.

Varying parameters from the baseline design.

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 feasibility because the flow scheduler compares against all flow entries in parallel. 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), finding that we can scale up to 2048 flows while still meeting timing at 1 GHz.

The remaining parameters affect the area overhead of a flow scheduler, but have little effect on whether or not the

Component	Area in mm^2
Switching chip	200–400 [21]
Flow Scheduler	0.224 (from synthesis)
SRAM (1 Mbit)	0.145 [6]
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)
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: Flow scheduler area increases in proportion to the number of flows and meets timing at 1 GHz until 2048 flows.

flow scheduler circuit meets timing. For instance starting from the baseline design of the flow scheduler that takes up 0.224 mm^2 of area, increasing the rank width to 32 bits results in an area of 0.317 mm^2, increasing the number of logical PIFOs to 1024 increases the area to 0.233 mm^2, and increasing the metadata width to 64 bits increases the area to 0.317 mm^2. In all cases, the circuit continues to meet timing.

5.4 Interconnecting PIFO blocks

An interconnect between PIFO blocks is required for PIFO blocks to enqueue into and dequeue from other PIFO blocks. Because the number of PIFO blocks is small, we provide a full mesh between the PIFO blocks. Assuming a 5-block PIFO mesh, this requires 5*4 = 20 sets of wires between PIFO blocks. Each set of wires would need to carry all the inputs required for specifying an enqueue and dequeue operation on a PIFO block.

We calculate the number of wires in each set for our baseline design. For an enqueue operation, 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 operation, we need a logical PIFO ID (8 bits) and wires to store the dequeued element (32 bits). This totals up to 106 bits per set of wires, or 2120 bits across the entire mesh. This is a relatively small number of wires and can easily be supported on a chip. For example, RMT's match-action pipeline uses nearly 2× the number of wires between each pair of pipeline stages to move the packet header vector from one stage to the next [13].

6. DISCUSSION

6.1 Buffer management

Our design focuses on programmable scheduling and doesn't concern itself with how the switch's data buffers are allocated to flows within a PIFO. Buffer management can either use static thresholds for each flow or dynamic thresholds based on active queue management algorithms such as RED [18] and the occupancy of other ports [14]. In a shared-memory switch, buffer management is largely orthogonal to scheduling, and is implemented using counters that track the occupancies of various flows and ports. Before a packet is enqueued into the scheduler, if any of these counters exceeds a static or dynamic threshold, the packet is dropped. A similar design could be used to check thresholds before enqueueing into a PIFO block as well.

6.2 Priority Flow Control

Priority Flow Control (PFC) [5] 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 a particular set of flows. PFC can be integrated into our hardware design for PIFOs 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.

6.3 Multi-pipeline switches

The highest end switches today, such as the Broadcom Tomahawk [2], support aggregate capacities exceeding 3 Tbit/sec. At a minimum packet size of 64 bytes, this corresponds to an aggregate forwarding requirement of 6 billion packets/s. Because a single switch pipeline 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 such 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). This is 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 does facilitate a multi-pipeline implementation. A rank store supporting multiple pipelines is similar to what is required in 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).

7. RELATED WORK

The Push-in First-out Queue: PIFOs were first introduced by Chuang et al. [15] as a proof construct to prove that a combined input-output queued switch could emulate an output-queued switch running different scheduling algorithms. The same paper also shows how WFQ and strict priorities are specific instantiations of a PIFO. We demonstrate that PIFOs can be used as an abstraction for programming scheduling algorithms beyond WFQ and strict priorities, that they can be composed together to program hierarchical scheduling algorithms, and finally that they are feasible in today's transistor technology.

Packet scheduling algorithms: Many scheduling algorithms [34, 33, 23, 17, 10, 28, 22, 9] have been proposed in the literature. Yet, only a handful (DRR, traffic shaping, and strict priorities) are found in routers. Even programmable switches [13, 4, 8] treat packet scheduling as a black box. As shown in §3, PIFOs allow us to program these and other as-yet-unknown scheduling algorithms, without the power, area, and performance penalties of prior proposals [37] that require fully reconfigurable FPGAs.

Universal Packet Scheduling (UPS): UPS [30] uses the LSTF algorithm by appropriately initializing slack times at end hosts and proves that LSTF is universal under the strong assumption that packet departure times for a scheduling algorithm are known up front. LSTF is expressible using PIFOs, but the set of schemes practically expressible with LSTF is itself 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, because for such algorithms LSTF must know the departure time of each packet up-front, 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 that is updated when packets are dequeued. Without this variable, a new flow could have arbitrary start times, and be deprived of its fair share indefinitely.
4. Scheduling policies that aggregate flows from distinct endpoints into a single flow at the switch (Figure 14), because LSTF provides no ability to maintain and update switch state progammatically.

Hardware designs for priority queues: Hardware designs for a priority queue have been proposed in the past [11, 24]. These designs typically employ a heap and scale to a large number of entries. They are the basis for hierarchical schedulers in many deep-buffered core routers. However, they occupy significant area—enough to warrant a dedicated chip for the scheduler alone. They are unlikely to be feasible on merchant-silicon shared-memory switching chips where

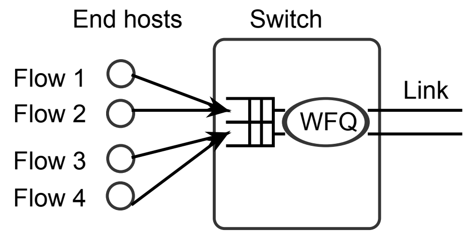


Figure 14: A switch's scheduling algorithm, such as WFQ, might aggregate flows from different end hosts into a single flow at the switch for the purpose of scheduling.

chip area is at a premium. In contrast, our design for the PIFO exploits two observations. First, there is considerable structure in the arrival stream of ranks: ranks within a flow strictly increase with time. Second, the buffering requirements for shared-memory switches today are much lesser than the buffering requirements of a core router. This permits a simpler design relative to heaps.

8. CONCLUSION

Until recently, it was widely assumed that the fastest switch chips would be fixed-function; a programmable device could not have the same performance. Recent research into programmable parsers [21], fast programmable protocol-independent switch chips [13], and languages to program them [12, 38], coupled with a recent 3.2 Tbit/s programmable commercial switch chip [8] 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 right 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.

PIFO appears to be a very promising abstraction: it includes a wide variety of existing algorithms, and allows us to express new ones. We have found it possible to implement PIFOs at line-rate with modest chip area overhead. We believe the most exciting consequence will be the creation of many new schedulers, invented by those who build and operate networks, 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. For those who need a new algorithm, they could create their own, or might even download one from an open-source repository or a reproducible SIGCOMM paper. To get there, we will need real switch chips with PIFO schedulers we can program. The good news is that we see no reason why a future generation of switch chips can not include a programmable PIFO scheduler.

9. REFERENCES

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

[2] High-density 25/100 gigabit ethernet StrataXGS tomahawk ethernet switch series. http://www.broadcom.com/products/Switching/Data-Center/BCM56960-Series.

[3] "in-band network telemetry". https://github.com/p4lang/p4factory/tree/master/apps/int.

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

[5] "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.

[6] Sram - arm. https://www.arm.com/products/physical-ip/embedded-memory-ip/sram.php.

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

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

[9] ALIZADEH, M., YANG, S., SHARIF, M., KATTI, S., MCKEOWN, N., PRABHAKAR, B., AND SHENKER, S. pFabric: Minimal Near-optimal Datacenter Transport. In *SIGCOMM* (2013).

[10] BENNETT, J. C. R., AND ZHANG, H. Hierarchical Packet Fair Queueing Algorithms. In *SIGCOMM* (1996).

[11] BHAGWAN, R., AND LIN, B. Fast and Scalable Priority Queue Architecture for High-Speed Network Switches. In *In Proceedings of Infocom 2000* (March 2000), IEEE Communications Society.

[12] 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. *SIGCOMM Comput. Commun. Rev. 44*, 3 (July 2014), 87–95.

[13] 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 *SIGCOMM* (2013).

[14] CHOUDHURY, A. K., AND HAHNE, E. L. Dynamic queue length thresholds for shared-memory packet switches. *IEEE/ACM Trans. Netw. 6*, 2 (Apr. 1998), 130–140.

[15] CHUANG, S.-T., GOEL, A., MCKEOWN, N., AND PRABHAKAR, B. Matching output queueing with a combined input/output-queued switch. *Selected Areas in Communications, IEEE Journal on 17*, 6 (Jun 1999), 1030–1039.

[16] CLARK, D. D., SHENKER, S., AND ZHANG, L. Supporting real-time applications in an integrated services packet network: Architecture and mechanism. In *Conference Proceedings on Communications Architectures &Amp; Protocols* (New York, NY, USA, 1992), SIGCOMM '92, ACM, pp. 14–26.

[17] DEMERS, A., KESHAV, S., AND SHENKER, S. Analysis and Simulation of a Fair Queueing Algorithm. In *SIGCOMM* (1989).

[18] FLOYD, S., AND JACOBSON, V. Random early detection gateways for congestion avoidance. *IEEE/ACM Trans. Netw. 1*, 4 (Aug. 1993), 397–413.

[19] FLOYD, S., AND JACOBSON, V. Link-sharing and resource management models for packet networks. *IEEE/ACM Trans. Netw. 3*, 4 (Aug. 1995), 365–386.

[20] FLOYD, S., AND SPEER, M. F. Experimental results for class-based queueing.

[21] GIBB, G., VARGHESE, G., HOROWITZ, M., AND MCKEOWN, N. Design principles for packet parsers. In *Architectures for Networking and Communications Systems (ANCS), 2013 ACM/IEEE Symposium on* (Oct 2013), pp. 13–24.

[22] GOLESTANI, S. J. A stop-and-go queueing framework for congestion management. In *Proceedings of the ACM Symposium on Communications Architectures &Amp; Protocols* (New York, NY, USA, 1990), SIGCOMM '90, ACM, pp. 8–18.

[23] GOYAL, P., VIN, H. M., AND CHEN, H. Start-time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. In *SIGCOMM* (1996).

[24] IOANNOU, A., AND KATEVENIS, M. Pipelined Heap (Priority Queue) Management for Advanced Scheduling in High-Speed Networks. *Networking, IEEE/ACM Transactions on 15*, 2 (April 2007), 450–461.

[25] JEYAKUMAR, V., ALIZADEH, M., GENG, Y., KIM, C., AND MAZIÈRES, D. Millions of Little Minions: Using Packets for Low Latency Network Programming and Visibility. In *SIGCOMM* (2014).

[26] JEYAKUMAR, V., ALIZADEH, M., MAZIÈRES, D., PRABHAKAR, B., GREENBERG, A., AND KIM, C. EyeQ: Practical Network Performance Isolation at the Edge. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, 2013), USENIX, pp. 297–311.

[27] KALMANEK, C. R., KANAKIA, H., AND KESHAV, S. Rate controlled servers for very high-speed networks. In *Global Telecommunications Conference, 1990, and Exhibition.'Communications: Connecting the Future', GLOBECOM'90., IEEE* (1990), IEEE, pp. 12–20.

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

[29] MCKENNEY, P. Stochastic fairness queuing. In *INFOCOM '90, Ninth Annual Joint Conference of the IEEE Computer and Communication Societies. The*

Multiple Facets of Integration. Proceedings, IEEE (Jun 1990), pp. 733–740 vol.2.

[30] MITTAL, R., AGRAWAL, R., RATNASAMY, S., AND SHENKER, S. Universal Packet Scheduling. In *Proceedings of the Fourteenth ACM Workshop on Hot Topics in Networks* (2015).

[31] POPA, L., KUMAR, G., CHOWDHURY, M., KRISHNAMURTHY, A., RATNASAMY, S., AND STOICA, I. FairCloud: Sharing the Network in Cloud Computing. In *SIGCOMM* (2012).

[32] SARIOWAN, H., CRUZ, R. L., AND POLYZOS, G. C. SCED: A Generalized Scheduling Policy for Guaranteeing Quality-of-service. *IEEE/ACM Trans. Netw. 7*, 5 (Oct. 1999), 669–684.

[33] SCHRAGE, L. E., AND MILLER, L. W. The Queue M/G/1 with the Shortest Remaining Processing Time Discipline. *Operations Research 14*, 4 (1966), 670–684.

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

[35] SIVARAMAN, A., BUDIU, M., CHEUNG, A., KIM, C., LICKING, S., VARGHESE, G., BALAKRISHNAN, H., ALIZADEH, M., AND MCKEOWN, N. Packet transactions: A programming model for data-plane algorithms at hardware speed. *CoRR abs/1512.05023* (2015).

[36] SIVARAMAN, A., SUBRAMANIAN, S., AGRAWAL, A., CHOLE, S., CHUANG, S.-T., EDSALL, T., ALIZADEH, M., KATTI, S., MCKEOWN, N., AND BALAKRISHNAN, H. Towards programmable packet scheduling. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks* (New York, NY, USA, 2015), HotNets-XIV, ACM, pp. 23:1–23:7.

[37] SIVARAMAN, A., WINSTEIN, K., SUBRAMANIAN, S., AND BALAKRISHNAN, H. No Silver Bullet: Extending SDN to the Data Plane. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks* (2013).

[38] SONG, H. Protocol-oblivious Forwarding: Unleash the Power of SDN Through a Future-proof Forwarding Plane. In *HotSDN* (2013).

[39] VERMA, D., ZHANG, H., AND FERRARI, D. Guaranteeing delay jitter bounds in packet switching networks. In *Proceedings of TRICOMM* (1991), vol. 91, pp. 35–46.

[40] ZHANG, H., AND FERRARI, D. Rate-Controlled Service Disciplines. *J. High Speed Networks 3*, 4 (1994), 389–412.