Towards 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

Packet scheduling in switches is not programmable; operators only choose among a handful of scheduling algorithms implemented by the manufacturer. In contrast, other switch functions such as packet parsing and header processing are becoming programmable [10, 3, 6]. This paper presents a programmable packet scheduler that allows operators to program a variety of scheduling algorithms.

Our design exploits the insight that any scheduling algorithm can be deconstructed into two decisions: in what order packets depart and when they depart. The algorithms only differ in how the order and departure times are computed. We show how these decisions map to two well-understood abstractions: priority and calendar queues [11]. Priority and calendar queues can then be composed together to realize a broad range of sophisticated scheduling algorithms. Further, both abstractions can be realized using the same mechanism: a programmable push-in first-out queue (PIFO) that allows a packet to push itself into an arbitrary location in a queue by programming a packet field. A PIFO is feasible in hardware. Preliminary synthesis indicates that an unoptimized hardware design meets timing at 1 GHz on a 16 nm technology node and occupies only 5% additional die area relative to existing merchant-silicon switching chips.

CCS Concepts

•Networks → Programmable networks;

Keywords

Programmable scheduling; switch hardware;

1. INTRODUCTION

Historically, network switches have been fixed-function devices. They support a fixed set of protocols using a hardwired packet-processing pipeline. New features require hardware redesign, which can take several years. Recently, driven in part by Software-Defined Networking and rapidly changing requirements in datacenter networks, a new class of *programmable* switching architectures have emerged. Programmable parsing [17] allows a switch to handle new protocol formats. Switch architectures like Intel's FlexPipe [3], Cavium's Xpliant [6] and the Reconfigurable Match-Action Table architecture [10] provide a programmable switch pipeline, allowing an operator to flexibly specify packet transformations.

However, one switch function remains hardwired: *the packet scheduler*. The most common scheduling algorithm is a simple First-In First-Out (FIFO) queue. Additionally, some switches support strict priorities, deficit weighted round robin (DWRR) [29], and traffic shaping across a small number of queues. While the scheduler has configurable parameters (e.g., DWRR weights), its core algorithms cannot be changed. For instance, it is impossible to implement a new algorithm such as pFabric [7] on a switch today.

Programmable packet scheduling would be beneficial to network operators in datacenters, enterprises, and service-provider networks. Consider datacenter networks. Workloads here have diverse requirements. Some require low latency for short flows [25, 7], others require flexible bandwidth allocation across tenants or even flows [26, 21], while others minimize completion time of flow aggregates [16, 12]. With programmable packet scheduling, operators could deploy custom scheduling algorithms for specific application-level objectives.

This paper introduces a design for a programmable packet scheduler. The key innovation in our design is to reduce a complex computation involving multiple queues and packets to a simpler per-packet computation that can be run whenever a packet is enqueued. Our design uses the insight that every scheduling algorithm makes two decisions (§2): the order of packet departures from the switch and the time of packet departures. Any work-conserving scheduling algorithm (such as Weighted-Fair Queuing (WFQ) [15] or Shortest Remaining Processing Time (SRPT) [28]) can be translated into a particular departure order (e.g., for SRPT, this is the order determined by the remaining flow size). Similarly,

HotNets '15 November 16–17 2015, Philadelphia, PA USA
Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-4047-2/15/11 ...$15.00
DOI: http://dx.doi.org/10.1145/2834050.2834106 .

non-work-conserving algorithms such as traffic shaping can be realized by computing the departure time of each packet and releasing each packet at its departure time.

We observe that in many scheduling algorithms, the packet order or departure time can be determined at packet arrival (enqueue). For example, Start-Time Fair Queuing [19] is an implementation of WFQ that calculates a virtual start time for each incoming packet and dequeues packets in order of increasing virtual start time. Similarly, for traffic shaping, the shaping rate determines the time of departure for each packet. This naturally leads us to two basic abstractions (§2.1) that capture a variety of scheduling algorithms: a *priority queue* that determines the packet departure order, and a *calendar queue* [11] that decides the packet departure time. Further, calendar and priority queues can be composed (§3) to realize sophisticated *composite* scheduling algorithms such as hierarchical fair queuing, with and without shaping.

To motivate a hardware design for priority and calendar queues, we observe that both can be realized using a single building block (§4): a push-in first-out queue (PIFO) that supports insertions into a queue sorted according to a packet field. An operator programs different scheduling algorithms by changing how that number — the packet priority or departure time — is computed. The computation can occur either in an earlier stage in the switch pipeline, a different switch, or even at end hosts.

Conventional wisdom [29, 23] suggests that the sorting required to implement a PIFO in hardware is expensive; however, we find that modern transistor technology has reached a point where maintaining a sorted queue with 10,000s of entries is feasible and economical. We present a simple hardware design for a PIFO (§5) that meets timing at 1 GHz when synthesized on a 16 nm technology node and occupies only 5% additional die area relative to a merchant-silicon switching chip today. Overall, our results indicate that a programmable scheduler is well within reach, bringing to switch scheduling the same flexibility that the rest of the switch pipeline increasingly enjoys.

2. DECONSTRUCTING SCHEDULING

Our thesis is that any scheduling algorithm can be deconstructed into two decisions: in *what order* and *when* should packets leave the switch. Scheduling algorithms only differ in how the order and departure time are computed. Further, in many cases, the order or departure time can be determined when the packet is enqueued. To see why, we look at three popular packet-scheduling algorithms: pFabric [7], Weighted Fair Queuing (WFQ) [15] and traffic shaping [5].

pFabric

pFabric is a recent datacenter transport design that minimizes average flow completion times by scheduling packets according to their remaining flow size at each switch; i.e., it implements the SRPT scheduling algorithm.[1] For each packet, end hosts insert the remaining flow size as a packet field. At each switch, packets are dequeued in ascending order of remaining flow size.

Weighted Fair Queuing

Weighted-Fair Queuing (WFQ) [15] provides weighted max-min bandwidth allocation across flows sharing a link. Numerous implementations of WFQ exist, including Start-Time Fair Queuing [19] and Deficit Round Robin [29]. For concreteness, we consider Start-Time Fair Queuing (STFQ) .[2]

STFQ computes a *virtual start time* (`p.start`) for each packet using the algorithm below.

```
On enqueue of packet p of flow f:
--------------------------------------------
if f in T
  p.start = max(virtual_time, T[f].last_finish)
else
  p.start = virtual_time
T[f].last_finish = p.start + p.length / f.weight

On dequeue of packet p:
---------------------------------
virtual_time = p.start
```

Here, `last_finish` is a state variable maintained for each flow in table `T` that tracks the virtual finish time of its latest packet, while `virtual_time` is a queue-wide state variable updated on each dequeue. Packets are scheduled in order of their virtual start time (`p.start`).

Traffic shaping

Besides packet order, some scheduling algorithms determine the time at which packets depart from a queue. Traffic shaping is a canonical example and is used to limit flows to a desired rate. A shaper has two parameters: a shaping rate, r, and a burst allowance, B. The standard implementation uses a *token bucket* [5], which is incremented at rate r, subject to a cap of B tokens. A packet is transmitted immediately if the bucket has enough tokens when it is enqueued; otherwise, it has to wait until sufficient tokens accumulate. Transmitted packets decrement the token bucket by the packet size.

Alternatively, the transmission time of each packet can be calculated on enqueue as follows:

```
tokens = min(tokens + r * (now - last_arrival), B)
if p.length <= tokens
  p.send_time = now
else
  p.send_time = now + (p.length - token_count) / r
tokens = tokens - p.length
last_arrival = now
```

Here, `tokens` and `last_arrival` are two state variables, initialized to B and an initial time respectively. While a standard token bucket has only positive token counts, `tokens` can fall below zero in the algorithm above. It is easy to show that the transmission times calculated are still identical to those of the standard token bucket.

2.1 Basic Abstractions for Packet Scheduling

[1]There are two variants of pFabric [7], with and without starvation prevention. We consider the one without starvation prevention.

[2]The original WFQ implementation [15] is similar to STFQ, but uses a more complex virtual time calculation.

Algorithm	Priority / departure time computation	Computed by	Abstraction
First-In First-Out	Wall clock time at packet enqueue	switch	priority queue
Strict Priorities	Packet's TOS field	end host	priority queue
pFabric [7] (Shortest Remaining Processing Time)	Remaining flow size for the packet's flow	end host	priority queue
Least Attained Service	Attained service for the packet's flow	end host	priority queue
Earliest Deadline First	Deadline for the packet's flow	end host	priority queue
Start-Time Fair Queuing [19]	Virtual Start Time for the packet's flow	switch	priority queue
Least Slack-Time First [22, 24]	Slack Time for each packet	Initialized at end hosts, decremented at switches	priority queue
Service-Curve Earliest Deadline-First scheduling [27]	Deadline based on service curve	switch	priority queue
Token-Bucket Shaping [5]	Time when there are enough tokens to transmit packet	switch	calendar queue
Stop and Go Queuing [18]	Start time of the next frame	switch	calendar queue

Table 1: Scheduling algorithms expressible using priority queues and calendar queues.

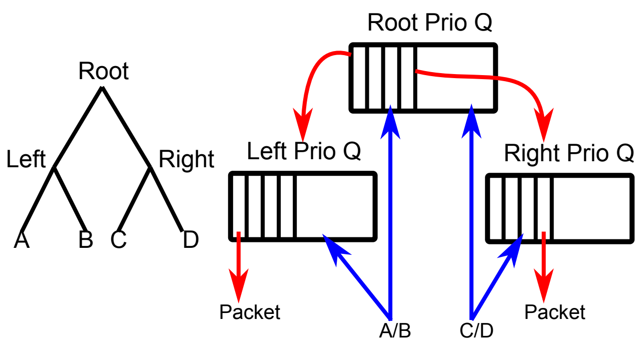

Figure 1: HPFQ using priority queues. Blue arrows show enqueue paths, red arrows show dequeue paths.

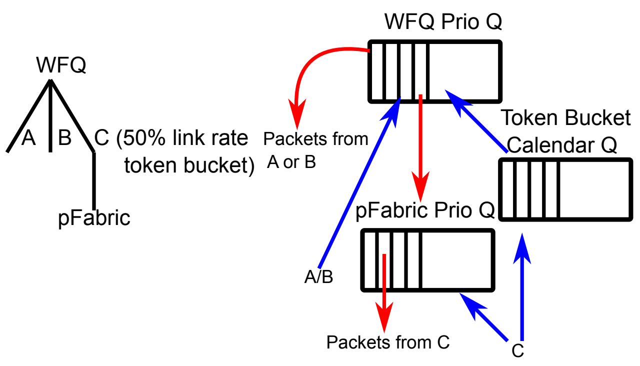

Figure 2: Combining hierarchies with shaping using priority and calendar queues. Blue arrows show enqueue paths, red arrows show dequeue paths.

The above algorithms determine the order or time of departure of a packet using a single number computed at enqueue time. For pFabric, this number is inserted by an end host. For WFQ, it is the virtual start time computed by the switch. For token bucket shaping, it is the wall-clock departure time. This motivates two natural abstractions: **priority queues** and **calendar queues**. Priority queues determine packet order for work-conserving algorithms like SRPT and WFQ, while calendar queues determine packet-departure times for non-work-conserving algorithms like shaping. These two abstractions with a programmable means of assigning an arriving packet a priority or departure time enable many scheduling algorithms (Table 1).

3. COMPOSITE SCHEDULING ALGORITHMS

While the previous section discusses individual scheduling algorithms, many practical scheduling algorithms compose multiple scheduling disciplines together. We consider two examples here, to show how such scheduling algorithms map to a composition of priority queues and calendar queues.

3.1 Hierarchical packet-fair queuing

Hierarchical packet-fair queuing (HPFQ) [8] acts on a class hierarchy represented as a tree (Figure 1). It first apportions link capacity fairly between the classes under the root of the tree. Then, within each class, it apportions link capacity fairly between sub-classes. It continues recursively until it hits the leaf nodes of the class hierarchy.

HPFQ cannot be implemented using a single priority queue, because — unlike the algorithms from §2 — the relative order of packet transmissions may change with future packet arrivals (see [8], Section 2.2, for an example). However, with a modification to the semantics of a priority queue to enqueue either a packet, or a reference to a different priority queue, HPFQ can be implemented as a hierarchy of priority queues.

In the example of Figure 1, an arriving packet is enqueued into either the Left or Right priority queue, depending on the packet's class. Next, a reference to the appropriate priority queue (Left or Right) is enqueued into the Root priority queue. To dequeue, we start at the Root priority queue and follow the reverse path, dequeuing references until we reach a packet. This effectively implements fair queuing at every level of the class hierarchy. The Root priority queue implements fair queuing across classes while the Left and Right priority queues implement fair queuing across the leaf nodes.

3.2 Fair-queuing hierarchies with shaping

The next example combines hierarchical queuing with traffic shaping. This algorithm is representative of scheduling algorithms provided by many switches [2]. We consider the algorithm described by the tree in Figure 2. Here, link capacity is divided across three classes A, B, and C in the ratio 4:1:2 using WFQ. We also require that C be token-bucket limited to 50% of the link capacity, regardless of the traffic offered by A and B. Further, when class C is serviced, we schedule its packets using the pFabric [7] algorithm.

This policy can be expressed using calendar queues and priority queues as follows. If there were no rate-limit on class C's packets, the implementation would be similar to HPFQ: a "WFQ priority queue" would implement WFQ across classes, and contain either packets from A or B, or references to the "pFabric priority queue". The pFabric priority queue would implement SRPT for class C. To rate-limit class C, we interpose a token-bucket calendar queue on the enqueue path from C to the WFQ priority queue. This ensures that the class C references are released to the WFQ priority queue at the shaped rate.

3.3 Other examples

Several non-hierarchical scheduling algorithms can also be realized by composing priority and calendar queues. One example is the class of Rate-Controlled Service Disciplines [34], which includes many non-work-conserving algorithms such as Jitter-EDD [31], Stop-and-Go [18], and Rate-Controlled Static Priority Queue [33].

3.4 Rules of composition

We outline rules of composition that we gleaned from this exercise. For future work, we plan to incorporate these rules into a language for programming scheduling algorithms.

1. A priority queue is *pulled* by other priority queues and external entities such as the link transmitter. A calendar queue *pushes* into other calendar queues and priority queues when the timestamp of the head packet arrives.
2. The root of the queuing system is always a priority queue that is pulled by the link transmitter.
3. Elements within priority and calendar queues are either packets or references to other priority queues.
4. There is only one copy of a packet present throughout the queuing system.
5. A priority queue is *recursively* dequeued: A dequeue from a priority queue that returns a priority queue reference, p, results in a dequeue from p recursively. This process must terminate in the transmission of a packet.
6. The programmer can only specify the sequence of enqueues into the system. The dequeue sequence is implicit: it is a recursive dequeue starting from the root priority queue.

4. THE PUSH-IN FIRST-OUT QUEUE

The previous sections show that packet scheduling can be broken down into two abstractions: priority queues that determine the order of packet departures and calendar queues that determine the wall-clock time of packet departures. We now show that both abstractions can be implemented using a common building block in hardware.

A common, modular hardware building block has considerable practical appeal: it can be implemented and optimized once as an IP core [4], and then be repeatedly instantiated, amortizing design effort. It allows the same piece of hardware to be re-purposed into a priority queue or a calendar queue, as required, without committing to a fixed number of each up front. Further, as described in §3, a basic building block with a clean interface allows us to programmatically compose multiple priority and calendar queues to express composite scheduling algorithms.

To design this block, we observe that internally both priority queue and calendar queue abstractions rely on the same mechanism: a sorted queue of packets, where a specific packet field is used to determine a packet's order in the queue. In the case of a priority queue, this is the packet's priority, while for a calendar queue, this is the packet's departure wall-clock time. In addition, for a calendar queue, we need a small amount of additional logic that releases packets based on the timestamp at the head of the queue. We call such a queue a push-in first-out queue (PIFO) [13] to denote the fact that packets can "push" themselves into any location, but depart only from the head.

4.1 Programming the PIFO

The PIFO can be programmed by programming the computation of the field that determines a packet's location in the PIFO (Column 2 in Table 1 lists several computations). The logic of finding an appropriate location for the packet is unchanged across scheduling algorithms. From the switch's perspective, fields determining packet locations in a PIFO are of two types. They could be values available in the packet header such as the TOS fields used for strict priorities or the remaining flow size in pFabric [7]. Alternatively, they could be values computed from persistent state maintained in the switch such as the virtual start time for STFQ [19] and the wall-clock departure time for the token-bucket algorithm or Stop-And-Go Queuing [18]. In the first case, the end host computes/programs the priority and the switch simply reads it. In the second case, packet priorities can be computed using ALU operations on packet fields and persistent state.

Such ALU operations on packet fields and persistent state are possible in programmable switch architectures such as the Reconfigurable Match-Action Table (RMT) architecture [10], Intel's FlexPipe [3], and Cavium's XPliant [6]. For instance, stateful computations like the code snippets for STFQ and token-bucket shaping in section §2 can be implemented using a match-action table: the match field matching on a particular flow[3] identifier, and the action performing arithmetic to update the state. In practice, the complexity of field computations implemented on a switch will be limited because only a small number of operations can be performed on each packet at line rate. However, we expect the computational capabilities of programmable switch pipelines to improve with time. These improvements will also permit more involved field computations for programmable scheduling.

[3]A flow can be any group of packets based on some setting of header fields: e.g. a 5-tuple, or a source/destination address.

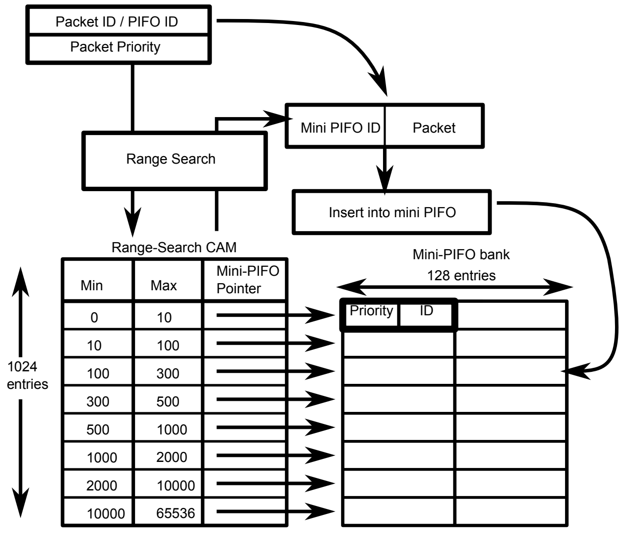

Figure 3: PIFO hardware implementation

5. HARDWARE FEASIBILITY

The core operation within a PIFO is insertion into a sorted queue. Hardware designs for sorted queues typically use heaps [9, 20] and are the basis of hierarchical schedulers in core routers. These heaps scale to large packet buffers exceeding 100 MBytes. However, these designs are hard to develop and verify because they require deep pipelines and intricate tree manipulations. As a result, they may not be feasible for datacenter switches which must support very high port densities and speeds at low cost.

Instead, we look at a simpler design that handles smaller buffers, sufficient for single-chip switches used in most datacenters. For instance, the Broadcom Trident [1] has a data buffer of 12 MBytes—an order of magnitude less than a core router. Targeting shallow-buffered chips dramatically simplifies our design.

5.1 A PIFO in hardware

We implement a PIFO using two data structures (Figure 3): a bank of "mini-PIFOs" (maintained in SRAM), and a content-associative memory (CAM) that maintains the minimum and maximum allowed priorities for each mini-PIFO. An arriving packet's priority is range searched, *in parallel*, against all priority ranges in the CAM to determine the mini-PIFO to enqueue into. The packet is then pushed into the chosen mini-PIFO in the correct position. This is analogous to the textbook bucket-sort algorithm [14].

Breaking up a large PIFO into a CAM and a bank of mini-PIFOs is key to scalability. The CAM serves as an index that points the packet to a small part of the PIFO kept in a mini-PIFO. Inserting in sorted order into a small mini-PIFO (around 100 entries) is then trivial: the arriving packet's priority is compared to all entries in the mini-PIFO in parallel to determine the exact location to insert into. This can be done with one read-modify-write operation on the mini-PIFO bank memory.

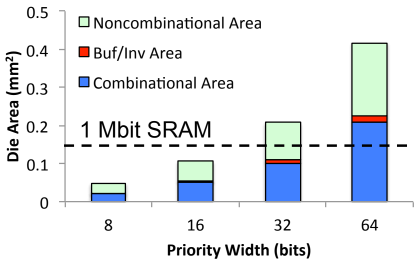

Figure 4: Range-search CAM area for different widths

What about the area cost of the range CAM? We synthesized an unoptimized 1024-entry range-search CAM with 16-bit priority numbers entirely in flip flops on a 16 nm technology node. The resulting circuit meets timing at 1 GHz and occupies only $0.105mm^2$ (CAM sizes for different bit-width priority numbers are given in Figure 4). For reference, 1 Mbit of SRAM consumes $0.149mm^2$ of area on a 16 nm technology node and the switch buffer has 96 Mbits of SRAM. A 1024-entry CAM, combined with 128 elements in each mini-PIFO can support 128K elements, which is more than the maximum buffer size of 60000 packets, assuming a 12 MByte buffer and a cell size of 200 bytes.[4]

Handling full mini-PIFOs: When a mini-PIFO becomes full, we split the mini-PIFO into two mini-PIFOs. This is required to avoid overflowing a mini-PIFO on a subsequent enqueue. When split, a full mini-PIFO with min and max allowed priority of m and M respectively becomes two mini-PIFOs: one with a range from m until the priority of the middle element (64th) in the mini-PIFO, and the second with a range from (middle element + 1) to M.

This split requires updating both the mini-PIFO banks and the CAM. To update the mini-PIFO bank, a naive implementation would require two write ports for the SRAM so that both the old and new mini-PIFOs can be written back after a split. We avoid this by internally organizing mini-PIFOs in two separate banks of physical memory. Each mini-PIFO either starts from the left or the right physical bank. During normal operation, if a mini-PIFO is read or written, we issue two parallel reads or writes to the left and right banks.

When we split a mini-PIFO, we write back one half of the mini-PIFO into the same row in the left bank (marking it left-aligned) and update the count of elements in that mini-PIFO to 64. We write back the other half of the mini-PIFO into a different, empty row, but in the right bank (marking it right-aligned). This reduces the memory requirement to exactly one read, modify, write operation on each physical memory bank, which is readily doable today.

Finally, we update the CAM by creating two new entries: one from m to the middle element and another from (middle element + 1) to M. For this, we endow the CAM with two

[4]A switch scheduler internally divides packets into smaller units called cells for fine-grained buffer allocation and cut-through switching. In the worst case, each packet contains only one cell.

Component	Area in SRAM-equivalent Mbits
Common state	
Data buffer	12 MB * 8 = 96 Mbits
Linked-list of free cells	60000 cells * 16 bits / cell = 0.96 Mbits
Baseline switch	
FIFO head/tail in flip-flops	1024 queues * (16-bit head + 16-bit tail) * 16 = 0.5 Mbits
FIFO linked-list pointers	60000 packets * 16 bits / packet = 0.96 Mbits
PIFO switch	
1024-entry CAM with 16 bit priorities	0.7 Mbits (from synthesis results)
mini-PIFO bank	128 pkts * 1024 mini-PIFOs * (16-bit ID + 16-bit priority) = 4 Mbits
Additional area of a PIFO switch	0.7 + 4.0 - 0.5 - 0.96 = 3.24 Mbits
Overhead relative to data buffer	3.24 / (96 + 0.96) = 3.3%
Overhead relative to switch chip	1.65%
Overhead for a 3-stage scheduler	5%

Table 2: Additional die area of a PIFO switch. We use a conversion factor of 16 for SRAM bits per bit of flip-flops.

write ports, which is straight forward because the CAM is built out of flip flops.

Memory fragmentation: The split causes memory fragmentation because we use more mini-PIFOs than absolutely required. However, we can bound the worst-case memory fragmentation. Since the split always divides the old mini-PIFO into two equal halves, the worst-case fragmentation is $\sim 2\times$. Hence, by designing the mini-PIFO bank to accommodate twice the maximum size of the switch buffer in packets, we can guarantee that there is always space in the PIFO for an arriving packet.

Sharing hardware between multiple logical PIFOs: Our discussion so far has assumed a single PIFO with a variable number of mini-PIFOs assigned to it as the PIFO grows and shrinks. In a shared-memory switch, PIFOs from different output ports can share the same underlying CAM and mini-PIFO structures. Our design accommodates this naturally, by adding a logical PIFO ID qualifier to the CAM structure. This adds little overhead: the PIFO ID only needs a few bits of exact match — much cheaper than the range-search logic.

5.2 Overhead Analysis

We now analyze the additional die area of a PIFO switch relative to a baseline switch with 12 MBytes of buffer, up to 60K cells (assuming a cell size of 200 bytes), and 1024 FIFO queues. For ease of comparison, we express all area numbers using an equivalent amount of SRAM. In the worst case, the data buffer can hold up to 60K single-cell packets. So, we assume a 16-bit packet/cell ID.[5] Table 2 summarizes our calculations: the additional overhead of a PIFO switch is about 3.3% relative to the baseline switch buffer and queuing structures. Buffering takes up 50% of the die area of a typical chip, implying that the area overhead of a PIFO switch over the baseline switch chip is about 1.65%.

As §3 shows, sophisticated scheduling algorithms require composing multiple logical PIFOs. Logical PIFOs that do not need to be simultaneously accessed, such as PIFOs at the same level of a hierarchical scheduling algorithm (e.g., the Left and Right PIFOs in Figure 1) can share the same underlying physical structures. However, logical PIFOs with concurrent accesses (e.g., the Root and Left/Right PIFOs in Figure 1) need separate structures. Our analysis shows that 3 PIFO stages would take about 5% (1.65×3) of the die area. This would provide a very powerful scheduler, supporting programmable 3-level hierarchies. By comparison, today's datacenter switches support a few simple scheduling algorithms in up to 2 levels, across 8–64 queues.

[5]The packet ID is the cell ID of its first cell.

6. RELATED WORK

Many scheduling algorithms [29, 28, 19, 15, 8, 22, 18, 7] have been proposed over the years. Yet, only a handful are deployed today. Even programmable switch architectures [10, 3, 6] currently treat packet scheduling as a black box provided by the switch manufacturer. Our work enables the practical realization of these and other as-yet-undiscovered scheduling algorithms. Further, we achieve programmability without the significant power, area, and performance penalties of prior proposals [30] that require fully reconfigurable FPGAs.

Priority and calendar queues are similar to rate regulators and schedulers used in Rate-Controlled Service Disciplines (RCSD) [34] to express a class of non-work-conserving scheduling algorithms. We demonstrate that these two abstractions can be composed (§3) more generally to realize a wide range of scheduling algorithms, and that both can be implemented using the PIFO building block (§4). We present a hardware design for PIFOs (§5) that requires a sorted queue of packets. Avoiding sorting in hardware was the impetus for work on constant time approximations to fair queuing such as DRR [29] and SFQ [23]. Our work shows that advances in transistor technology warrant revisiting the premise that sorting is impractical at line rate.

7. CONCLUSION

This paper shows that programmable packet scheduling is within reach. Using a PIFO, it is possible to realize programmable priority and calendar queues, which can be stitched together to realize a wide variety of scheduling algorithms. Preliminary synthesis (§5) indicates that a PIFO can meet timing at 1 GHz on current technology nodes (16 nm) with minimal area overhead. We are currently implementing PIFOs on the NetFPGA [32] platform to conduct end-to-end experiments using PIFOs.

Acknowledgements

We thank Amy Ousterhout and the HotNets reviewers for their thoughtful feedback and suggestions that improved the presentation of the paper. We thank Ion Stoica for helpful discussions. This work was partly supported by NSF grant IIS-1065219. We thank the industrial partners of the MIT Center for Wireless Networks and Mobile Computing (Wireless@MIT) for their support.

8. REFERENCES

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

[2] IEEE 802.1: 802.1Qaz - Enhanced Transmission Selection. http://www.ieee802.org/1/pages/802.1az.html.

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

[4] Semiconductor Intellectual Property. https://en.wikipedia.org/wiki/Semiconductor_intellectual_property_core.

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

[6] XPliant^{TM}Ethernet Switch Product Family. http://www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html.

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

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

[9] R. Bhagwan and B. Lin. Fast and Scalable Priority Queue Architecture for High-Speed Network Switches. In *In Proceedings of Infocom 2000*. IEEE Communications Society, March 2000.

[10] 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.

[11] R. Brown. Calendar Queues: A Fast 0(1) Priority Queue Implementation for the Simulation Event Set Problem. *Commun. ACM*, 31(10):1220–1227, Oct. 1988.

[12] M. Chowdhury, Y. Zhong, and I. Stoica. Efficient Coflow Scheduling with Varys. In *SIGCOMM*, 2014.

[13] S. Chuang, A. Goel, N. McKeown, and B. Prabhakar. Matching Output Queueing with a Combined Input Output Queued Switch. Technical report, Stanford, CA, USA, 1998.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

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

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

[17] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown. Design Principles for Packet Parsers. In *Architectures for Networking and Communications Systems (ANCS), 2013 ACM/IEEE Symposium on*, pages 13–24, Oct 2013.

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

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

[20] A. Ioannou and M. Katevenis. Pipelined Heap (Priority Queue) Management for Advanced Scheduling in High-Speed Networks. *Networking, IEEE/ACM Transactions on*, 15(2):450–461, April 2007.

[21] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, A. Greenberg, and C. Kim. 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)*, pages 297–311, Lombard, IL, 2013. USENIX.

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

[23] P. McKenney. Stochastic fairness queuing. In *INFOCOM '90, Ninth Annual Joint Conference of the IEEE Computer and Communication Societies. The Multiple Facets of Integration. Proceedings, IEEE*, pages 733–740 vol.2, Jun 1990.

[24] R. Mittal, R. Agrawal, S. Ratnasamy, and S. Shenker. Universal Packet Scheduling. In *Proceedings of the Fourteenth ACM Workshop on Hot Topics in Networks*, 2015.

[25] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, D. Ongaro, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The Case for RAMCloud. *Commun. ACM*, 54(7):121–130, July 2011.

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

[27] H. Sariowan, R. L. Cruz, and G. C. Polyzos. SCED: A Generalized Scheduling Policy for Guaranteeing Quality-of-service. *IEEE/ACM Trans. Netw.*, 7(5):669–684, Oct. 1999.

[28] 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.

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

[30] A. Sivaraman, K. Winstein, S. Subramanian, and H. Balakrishnan. No Silver Bullet: Extending SDN to the Data Plane. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, 2013.

[31] D. C. Verma, H. Zhang, and D. Ferrari. Delay Jitter Control for Real-Time Communication in a Packet Switching Network. In *Communications Software, 1991,'Communications for Distributed Applications and Systems', Proceedings of TRICOMM'91., IEEE Conference on*, pages 35–43. IEEE, 1991.

[32] G. Watson, N. McKeown, and M. Casado. Netfpga: A Tool for Network Research and Education. In *2nd workshop on Architectural Research using FPGA Platforms (WARFP)*, volume 3, 2006.

[33] H. Zhang and D. Ferrari. Rate-Controlled Static-Priority Queueing. In *INFOCOM '93. Proceedings.Twelfth Annual Joint Conference of the IEEE Computer and Communications Societies. Networking: Foundation for the Future, IEEE*, pages 227–236 vol.1, 1993.

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