Approximating Fair Queueing on Reconfigurable Switches

Naveen Kr. Sharma[*] *Ming Liu*[*] *Kishore Atreya*[†] *Arvind Krishnamurthy*[*]

Abstract

Congestion control today is predominantly achieved via end-to-end mechanisms with little support from the network. As a result, end-hosts must cooperate to achieve optimal throughput and fairness, leading to inefficiencies and poor performance isolation. While router mechanisms such as Fair Queuing guarantee fair bandwidth allocation to all participants and have proven to be optimal in some respects, they require complex flow classification, buffer allocation, and scheduling on a per-packet basis. These factors make them expensive to implement in high-speed switches.

In this paper, we use emerging reconfigurable switches to develop an approximate form of Fair Queueing that operates at line-rate. We leverage configurable per-packet processing and the ability to maintain mutable state inside switches to achieve fair bandwidth allocation across all traversing flows. Further, present our design for a new dequeuing scheduler, called Rotating Strict Priority scheduler that lets us transmit packets from multiple queues in approximate sorted order. Our hardware emulation and software simulations on a large leaf-spine topology show that our scheme closely approximates ideal Fair Queueing, improving the average flow completion times for short flows by 2-4x and 99th tail latency by 4-8x relative to TCP and DCTCP.

1 Introduction

Most current congestion control schemes rely on end-to-end mechanisms with little support from the network (e.g., ECN, RED). While this approach simplifies switches and lets them operate at very high speeds, it requires end-hosts to cooperate to achieve fair network sharing, thereby leading to inefficiencies and poor performance isolation. On the other hand, if the switches were capable of maintaining per-flow state, extracting rich telemetry from the network, and performing configurable per-packet processing, one can realize intelligent congestion control mechanisms that take advantage of dynamic network state directly inside the network and improve network performance.

One such mechanism is Fair Queueing, which has been studied extensively and shown to be optimal in several aspects. It provides the illusion that every flow (or participant) has its own queue and receives a fair share of the bandwidth under all circumstances, regardless of other network traffic. Having the network enforce fair bandwidth allocation offers several benefits. It simplifies congestion control at the end-hosts, removing the need to perform slow-start or complex congestion avoidance strategies. Further, flows can ramp up quickly without affecting other network traffic. It also provides strong isolation among competing flows, protects well-behaved flows from ill-behaving traffic, and enables bounded delay guarantees [34].

A fair bandwidth allocation scheme is potentially well suited to today's datacenter environment, where multiple applications with diverse network demands often co-exist. Some applications require low latency, while others need sustained throughput. Datacenter networks must also contend with challenging traffic patterns – such as large incasts or fan-in, micro-bursts, and synchronized flows, – which can all be managed effectively using a fair queueing mechanism. Fair queueing mechanisms can also provide bandwidth guarantees for multiple tenants of a shared cloud infrastructure [35].

Over the years, several algorithms for enforcing fair bandwidth allocation have been proposed [25, 27, 28, 33], but rarely deployed in practice, primarily due to their inherent complexities. These algorithms maintain state and perform operations on a per-flow basis, making them challenging to implement at data rates of 3-6 Tbps in hardware. However, recent advances in switching hardware allow flexible per-packet processing and the ability to maintain limited mutable state at switches without sacrificing performance [12, 6]. In this paper, we explore whether an efficient fair queueing implementation can be realized using these emerging reconfigurable switches.

We present *Approximate Fair Queueing (AFQ)*, a fair bandwidth allocation mechanism that approximates the various components of an ideal fair queueing scheme using features available in emerging programmable switches, such as the ability to maintain and mutate switch state on a per-packet basis, perform limited computation for each packet, and dynamically determine which egress queue to use for a given packet. We describe a variant of the packet-pair flow control protocol [24], designed to work with AFQ, that achieves close to optimal performance while maintaining short queues. We further prototype an AFQ implementation on a Cavium networking processor and study its feasibility on upcoming reconfigurable switches. Using a real hardware testbed and large-scale simulations, we demon-

[*]University of Washington
[†]Cavium Inc.

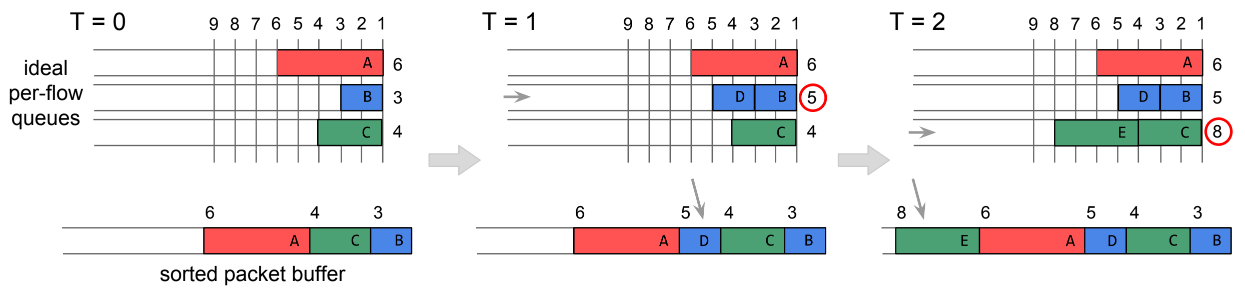


Figure 1: An example of the bit-by-bit round robin Fair Queueing algorithm. The algorithm buffers all packets in sorted order based on their departure round. When a blue packet D of size 2 arrives at T = 1, its departure round is calculated as 5 and is placed between packets A and C in the sorted buffer. Similarly, when a green packet of size 4 arrives at T = 2, its departure round is 8, and it is placed at the end of the departure queue.

strate AFQ's utility, showing it achieves fair bandwidth allocation for common datacenter workloads and traffic patterns, significantly improving performance over existing schemes. Specifically, AFQ reduces the average flow completion time of common workloads by 2-4x compared to TCP and DCTCP, and 99th percentile tail latency for short flows by up to 5-10x. We measure its overhead programmable switches by implementing AFQ in the P4 language and compiling it to a realistic hardware model, demonstrating that the resource overhead is modest.

2 Background

The idea of enforcing fair bandwidth allocation inside the network has been well studied and shown to offer several desirable properties. A straight-forward way of achieving such allocation is to have per-flow queues, as proposed by Nagle [31], serviced in a round robin manner. This is clearly impractical given today's network speeds and workload complexities. An efficient algorithm, called *bit-by-bit round robin (BR)*, proposed in [18], achieves ideal fair queueing behavior without requiring expensive per-flow queues. We describe this approach next since it forms the basis of our AFQ mechanism. We then provide background on the reconfigurable switch architecture.

2.1 Bit-by-Bit Round Robin (BR)

The *bit-by-bit round robin* algorithm achieves per-flow fair queueing using a round robin scheme wherein each active flow transmits a single bit of data every round. Then, the round ends, and the round number is incremented by one. Since it is impractical to build such a system, the BR algorithm "simulates" this scheme at packet granularity using the following steps.

- For every packet, the switch computes a *bid number* that estimates the time (round) when the packet would have departed.
- All packets are then buffered in a *sorted priority queue* based on their bid numbers, which allows dequeuing and transmission of the packet with the lowest bid number at any time.

Figure 1 shows a simple example of this approach. Although the BR algorithm achieves ideal fair queuing behavior, several factors make it challenging to implement given today's line-rate, 3-6 Tbps switches. First, to compute bid numbers for each packet, the switch must maintain the *finish round number* for each active flow. This is equal to the round when the flow's last byte will be transmitted and must be updated after each packet's arrival. Today's switches carry hundreds to thousands of concurrent flows [7, 37]. Their limited amounts of stateful memory makes it difficult to store and update per-flow bid numbers. Second, inserting packets into an ordered queue is an expensive $O(logN)$ operation, where N is the maximum buffer size in number of packets. Given the 12-20MB packet buffers available in today's switches, this operation is challenging to implement at a line-rate of billions of packets per second. Finally, switches need to store and update the current round number periodically using non-trivial computation involving: (1) time elapsed since last update, (2) number of active flows, and (3) link speed, as described in [25]. Today's line-rate switches lack the capability to perform such complex computations on a per-packet basis.

As noted, emerging reconfigurable switches allow flexible packet processing and the ability to maintain limited switch state, therefore we explore whether we can implement fair-queuing on these new line-rate switches. Further, recent work [38] has shown approximation to be a useful tool for implementing a broad class of in-network protocols for congestion control, load balancing, QoS and fairness.

2.2 Reconfigurable Switches

Reconfigurable switches provide a *match+action (M+A)* processing model: *match* on arbitrary packet header fields and then perform simple packet processing *actions*. In our work, we assume an abstract Reconfigurable Match Table (RMT) switch model, as described in [8, 9] and depicted in Figure 2. A reconfigurable switch begins packet processing by extracting relevant packet headers via a user-defined parse graph. The extracted header fields and packet metadata are passed onto a pipeline of

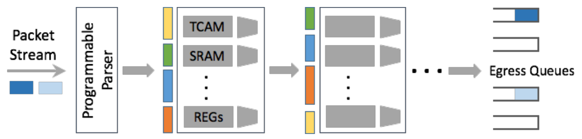


Figure 2: The architecture of a reconfigurable switch. Packets are processed by a pipeline of match+action stages with local state.

user-defined M+A tables. Each table matches on a subset of extracted headers and can apply simple processing primitives to any field. After traversing the pipeline stages, packets are deposited in one of multiple queues associated with the egress port for future transmission. The parser and the M+A pipeline can be configured using a high-level language, such as P4 [8] or PoF [43].

A reconfigurable switch provides several hardware features to support packet processing on the data path: (1) a limited amount of *stateful memory*, such as counters, meters, and registers, which can be accessed and updated to maintain state across packets, and (2) *computation primitives*, such as addition, bit-shifts, hashing, and max/min, which can perform a limited amount of processing on header fields and data retrieved from stateful memory. Further, switch metadata, such as queue lengths, congestion status, and bytes transmitted, can also be used in packet processing. Crucially, the pipeline stages can determine which transmit queue to use for a given packet based on packet header content and local state. Finally, a switch-local *control plane CPU* can also perform periodic bookkeeping tasks. Several such reconfigurable switches, – Cavium XPliant [12], Barefoot Tofino [6] and Intel Flexpipe [32] – are available today.

3 Approximate Fair Queueing

Any fair queuing router must perform per-flow management tasks to guarantee fair bandwidth allocation. These tasks include *packet classification* – which flow this packet belongs to, *buffer allocation* – whether this flow's packet should be enqueued or dropped, and *packet scheduling* – decide which flow's packet to transmit next. The key idea behind AFQ is to approximate the various components of a fair queueing scheme using features available in programmable switches.

Our design goals for AFQ include achieving per-flow max-min fairness [20], where a flow is defined as a unique 5-tuple. Our design should be implementable in high-speed routers running at line-rate. It must also be able to handle several thousand flows with varying packet sizes. We next provide an overview of our design.

3.1 Design Overview

Our design emulates the ideal BR algorithm described earlier. Like that algorithm, AFQ proceeds in a round robin manner, where every flow transmits a fixed number of bytes in each round. On arrival, each packet is assigned a departure round number based on how many bytes the flow has sent in the past, and packets are scheduled to be transmitted in increasing round numbers. Implementing this scheme requires AFQ to store the finish round number for every active flow at the switch and schedule buffered packets in a sorted order. It must also store and update the current round number periodically at the switch.

We approximate fair queueing using three key ideas. First, we store approximate flow bid numbers in sub-linear space using a variant of the count-min sketch, letting AFQ maintain state for a large number of flows with limited switch memory. This is made feasible by the availability of read-write registers on the datapath of reconfigurable switches. Second, AFQ uses coarser grain rounds that are incremented only after all active flows have transmitted a configurable number of bytes through an output port. Third, AFQ schedules packets to depart in an approximately sorted manner using multiple FIFO queues available at each port on these reconfigurable switches. Combining these techniques yields schedules that approximate those produced by a fair queueing switch. However, we show that AFQ provides performance that is comparable to fair queueing for today's datacenter workloads despite these approximations. Figure 3 shows the pseudocode describing AFQ's main components, which we explain in more detail in the next three sections.

3.2 Storing Approximate Bid Numbers

A flow's bid number in the BR algorithm is its finish-round number, which estimates when the flow's last enqueued byte will depart from the switch. The bid number of a flow's packet is a function of both the current active round number as well as the bid number associated with the flow's previous packet, and it is used to determine the packet's transmission order. AFQ stores each active flow's bid number in a count-min sketch-like datastructure to reduce the stateful memory footprint on the switch since such memory is a limited resource.

A count-min sketch is simply a 2D array of counters that supports two operations: (a) `inc(e,n)`, which increments the counter for element `e` by `n`, and (b) `read(e)`, which returns the counter for element `e`. For a sketch with `r` rows and `c` columns, `inc(e,n)` applies `r` independent hash functions to `e` to locate a cell in each row and increments the cell by `n`. The operation `read(e)` applies the same `r` hash functions to locate the same `r` cells and returns the minimum among them. The approximate counter value always exceeds or equals the exact value, letting us store flow bid numbers efficiently in sub-linear space. Theoretically, to get an ϵ approximation, – i.e., $error < \epsilon \times K$ with probability $1 - \delta$, where K is the number of increments to the sketch, – we need `c` = e/ϵ and `r` = $log(1/\delta)$ [17].

```
/* AFQ parameters */

S[][] : sketch for bid numbers
nH    : # of hashes in sketch
nB    : # of buckets in sketch
nQ    : # of FIFO queues
BpR   : bytes sent in each round

/* Count-min sketch functions */
func read_sketch(pkt):
   val = INT_MAX
   for i = 1 to nH:
      h = hash_i(pkt) % nB
      val = min(S[i][h], val)
   return val

func update_sketch(pkt, val):
   for i = 1 to nH:
      h = hash_i(pkt) % nB
      S[i][h] = max(S[i][h], val)
```

```
/* Enqueue Module */

R : Current round (shared w/ dequeue)

On packet arrival:
   bid = read_sketch(pkt)

   // If flow hasn't sent in a while,
   // bump it's round to current round.
   bid = max(bid, R * BpR)

   bid = bid + pkt.size
   pkt_round = bid / BpR

   // If round too far ahead, drop pkt.
   if (pkt_round - R) > nQ:
      drop(pkt)
   else:
      enqueue(pkt_round % nQ, pkt)
      update_sketch(pkt, bid)
```

```
/* Dequeue Module */

R : Current round number (shared)
i : Current queue being serviced

while True:
   // If no packets to send, spin.
   if buffer.empty()
      continue;

   // Drain i'th queue till empty.
   while !queue[i].empty():
      pkt = dequeue(i)
      send(pkt)

   // Move onto next queue,
   // increment round number.
   i = (i + 1) % nQ
   R = R + 1
```

Figure 3: Pseudocode for AFQ

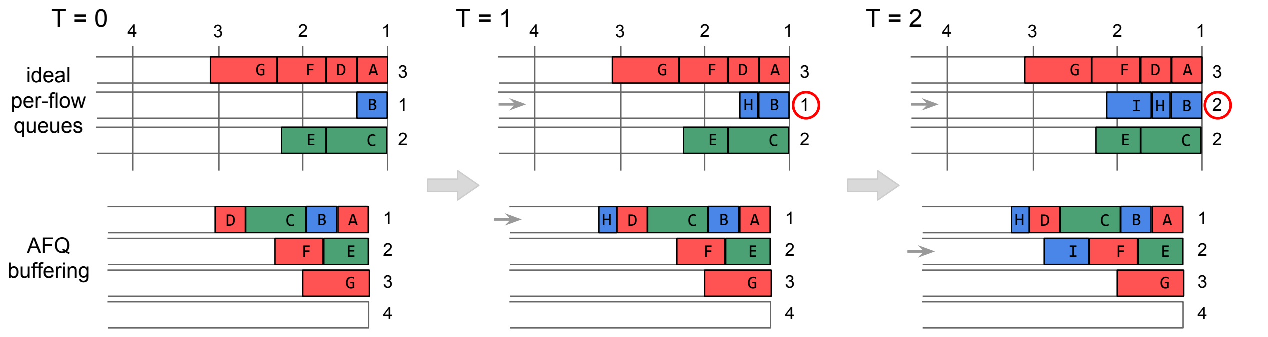


Figure 4: An example of the AFQ enqueue mechanism. As packets arrive, their bid numbers are estimated, and they are placed in an available FIFO queues. When a blue packet H arrives at T = 1, its bid number falls within round 1 and is placed in the first FIFO queue servicing round 1. When a subsequent blue packet I arrives at T = 2, its bid number falls in round 2; hence, it is placed is the second FIFO queue. For both packets H and I, we can see the approximation effects of using a large quantum of bytes per round and FIFO queues. An ideal FQ scheme using BR would transmit packet H before packets C and D, and packet I before E and F, as their last bytes are enqueued before the other packets in the per-flow queue. However, this reordering is upper-bounded by the number of active flows multiplied by the round quantum.

In hardware, a sketch is realized using a simple *increment by x* primitive and predicated read-write registers (as described in [40]), both of which are available in reconfigurable switches. On packet arrival, `r` hashes of the flow's 5-tuple are computed to index into the register arrays and estimate the flow's finish round number, which is used to determine the packet's transmission schedule. In practice, AFQ re-uses one of several hashes that are already computed by the switch for Link Aggregation and ECMP. Today's devices support up to 64K register entries per stage and 12-16 stages [22], which is sufficient for a reasonably sized sketch per port to achieve good approximation, as we show in Appendix E.

3.3 Buffering Packets in Approximate Sorted Order

The BR fair queuing algorithm ensures that the packet with the lowest bid number is transmitted next at any point of time using a sorted queue. Since maintaining such a sorted queue is expensive, AFQ instead leverages the multiple FIFO queues available per port to approximate ordered departure of buffered packets, similar to timer wheels [46].

Assume there are N FIFO queues available at each egress port of the switch. AFQ uses each queue to buffer packets scheduled to depart within the next N rounds, where in each round, every active flow can send a fixed number of bytes, i.e., `BpR` bytes (bytes per round). We next describe how packets are enqueued and dequeued in approximate sorted order using these multiple queues.

3.3.1 Enqueue Module

The enqueue module decides which FIFO queue to assign to each packet. On arrival, the module retrieves the bid number associated with the flow's previous packet from the sketch. If it is lower than the starting bid number for the current round, the bid is pushed up to match the current round. The packet's bid number is then obtained by adding the packet's size to the previous bid number, and the packet's departure round number is computed as the packet's bid number divided by `BpR`. If this departure round exceeds N rounds in the future, the packet is dropped, else it is enqueued in the queue corresponding to the computed round number. Note that the current round number is a shared variable that the dequeue mod-

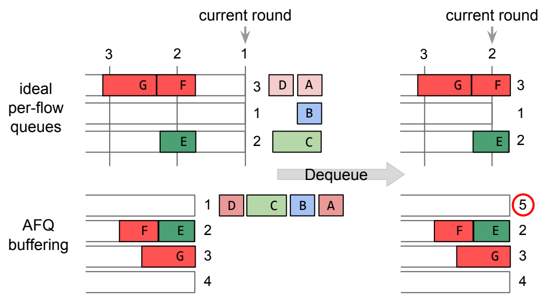


Figure 5: An example of the AFQ dequeue mechanism.

ule updates after it finishes draining a queue. Finally, the enqueue module updates the sketch to reflect the bid number computed for the current packet. Figure 4 shows an example of how AFQ works when various flows with variable packet sizes arrive at the same egress port.

Clearly, having more FIFO queues leads to finer ordering granularity and a better approximation of fair queuing. Switches available today support 24-32 queues per port [9, 12], which we show is sufficient for datacenter workloads. AFQ assumes that the total buffer assigned to each port can be dynamically assigned to any queue associated with that port. This lets AFQ to absorb a burst of new flow arrivals when several packets are scheduled for the same round number. Most switches already implement this functionality via dynamic buffer sharing [16].

3.3.2 Dequeue Module

The dequeue module transmits the packet with the smallest departure round number. Since the enqueue module already stores packets belonging to a given round number in a separate queue, AFQ must only drain the queue with the smallest round number. This is achieved by arranging all queues in strict priority, with the queue having the lowest round number assigned the highest priority. However, once empty, the queue must be bumped down to the lowest priority and the current round number incremented by 1. Note that this round number is shared with the enqueue module, which can then adjust its queueing behavior. The just-emptied queue is then used to store packets belonging to a future round number that is N higher than the current round number. Figure 5 shows the priority change and round assignment that occurs when a queue is drained to empty by a *Rotating Strict Priority* (RSP) scheduler. We describe in Section 4 how to implement this scheduler on reconfigurable switches.

An important implication of this design is that updating the current round number becomes trivial – increment by 1 whenever the current queue drains completely. Unlike the BR fair queuing algorithm, which must update the round number on every packet arrival, this coarse increment does not involve any complex computations or extra packet state, making it much more feasible to implement on reconfigurable switches.

3.4 Discussion

Several approximations govern how closely the AFQ design can emulate ideal fair queueing and present a fairness versus efficiency trade-off.

Impact of approximations: First, using a count-min sketch means that AFQ can over-estimate a packet's bid number in case of collisions. As the number of active flows grows beyond the size of the sketch, the probability of collisions increases, causing packets to be scheduled later than expected. However, as we show in the Appendix E, the sketch must be sufficiently large to store state only for active flows that have a packet enqueued at the switch, not all flows traversing the switch, including dormant ones that have not transmitted recently.

Second, unlike the BR fair queueing algorithm, which transmits one bit from each flow per round, AFQ lets active flows send multiple bytes per round. Since this departure round number is coarser than the bid number and AFQ buffers packets with the same round number in FIFO order, packets with higher bid numbers might be transmitted before packets with lower bid numbers if the switch received them earlier. This reordering can lead to unfairness within the round, but is bounded by number of active flows times `BpR` in the worst case.

`BpR` trade-off: Since AFQ buffers packets for the next N rounds only, the `BpR` must be chosen carefully to balance fairness and efficient use of the switch buffer. If `BpR` is too large, a single flow can occupy a large portion of the buffer, causing unfair packet delays and drops. If it is too small, AFQ will drop packets from a single flow burst despite having sufficient space to buffer them. The choice of `BpR` depends on network parameters, such as round trip times and link speeds, switch parameters, such as number of FIFO queues per port and total amount of packet buffer, as well as the endhost flow control protocol. We discuss how to set the `BpR` parameter after we describe the end-host transport protocol, which prescribe the rate adaptation mechanisms and determine the desired queue buildups on the switch.

4 Rotating Strict Priority (RSP) Scheduler

Given the Figure 3 pseudocode, implementing the dequeue module appears to be trivial. However, some hardware constraints make it more challenging than it seems. First, the two modules are generally implemented as separate blocks in hardware, which drives considerations regarding the sharing of state and synchronization issues between them. This is important since the decision of which queue to insert the packet into, or whether to drop the packet altogether, depends on the current round number. Second, the RSP scheduler, a custom mechanism, requires a queue's priority to be adjusted with respect to all other queues after it is completely drained by the dequeue module. This mechanism is currently not supported, so we explore multiple ways to implement the RSP scheduler on today's hardware.

Synchronizing the enqueue and dequeue modules. Our design requires the current round number to be shared and synchronized between the two modules. The RMT architecture outlined in [9] does not permit the sharing of state across pipelines stages or pipelines due to performance considerations, but other reconfigurable switches that support the disaggregated RMT model [15, 12] do not impose this constraint. However, a workaround on the RMT architecture is possible if we make the following modifications to the enqueue module. Instead of explicitly receiving a signal regarding round completion (through the increment of the round number), the enqueue model can maintain a local estimate of the round number and infer round completion by obtaining queue metadata regarding its occupancy.

An empty queue corresponding to a given round number implies that the queue has been completely drained, and the enqueue module then locally increments its estimate of the round number and tries adding the packet to the next queue. Eventually, the enqueue module will identify a queue that is either not empty or that corresponds to a round number that it has not previously assigned to any incoming packet; it then assigns the packet to this queue. Note that we have replaced explicit signaling by providing access to queue occupancy data, which is supported on reconfigurable switches such as Barefoot's Tofino and Cavium Xpliant, at least at a coarse-grain level (i.e., pipeline stages have access to a coarse-grain occupancy level for each queue, if not the exact number of bytes enqueued).

Emulating RSP using a generic DRR scheduler. Deficit Round Robin (DRR) is a scheduling algorithm that guarantees isolation and fairness across all queues serviced. It proceeds in rounds; in each round it scans all non-empty queues in sequence and transmits up to a configurable *quantum* of bytes from each queue. Any deficit carries over to the next round unless the queue is empty, in which case the deficit is set to zero. We note that RSP is simply a version of DRR with the quantum set to a large value that is an upper-bound on the number of bytes transmitted by flows in a round. With a very high quantum, a queue serviced in DRR is never serviced again until all other queues have been serviced. This is equivalent to demoting the currently serviced queue to the lowest priority. Crucially, the DRR emulation approach indicates that the hardware costs of realizing RSP should be minimal since we can emulate its functionality using a mechanism that has been implemented on switches. However, we note that many modern switches implement a more advanced version of DRR, called Shaped DWRR, a variant that performs round robin scheduling of packets from queues with non-zero deficit counters in order to avoid long delays and unfairness. Unfortunately, the RSP mechanism cannot be emulated directly using DWRR due to its use of round robin scheduling across active queues.

Emulating RSP using strict priority queues. We now consider another emulation strategy that uses periodic involvement of the switch-local control plane CPU to alter the priority levels of the available egress queues. When the priority level for a queue is changed, typically through a PCIe write operation, the switch hardware instantaneously uses the queue's new priority level to determine packet schedules. The challenge here is that the switch CPU cannot make repeated updates to the priority levels given its clock speed and the PCIe throughput. We therefore designed a mechanism that requires less frequent updates to the priority levels (e.g., two PCIe operations every 10us) using hierarchical schedulers.

Our emulation approach splits the FIFO queues into two strict priority groups and defines hierarchical priority over the two groups. All priority level updates are made by switching the upper-level priority of the two sets of queues; these updates are made only after the system processes a certain number of rounds. Suppose we have $2 \times n$ queues, split into two groups (G^1, G^2) of n queues each. In each group, all n queues are serviced using strict priority. Initially, G^1 has strict priority over G^2. Packets with round number $1 \rightarrow n$ are enqueued in $G^1_{1\rightarrow n}$, whereas packets with round $(n+1) \rightarrow 2n$ are enqueued in $G^2_{1\rightarrow n}$. Packets with a round number greater than that are dropped. After a period τ, or when all queues in G^1 are empty, we switch the priorities of G^1 and G^2, making all queues of G^2 higher priority than G^1. Queues in each group retain their strict priority ordering. After the switch, we allow packets to be enqueue on G^1's queues for rounds corresponding to $(2n+1) \rightarrow 3n$.

This approach is feasible using hierarchical schedulers available in most ToR switches today. It reduces the number of priority transitions the switch must make and is implementable with the help of the management/service CPU on the switch. The time period τ depends on the link-rate and number of queues. Our experiments with the Cavium Xpliant switch indicate that $\tau = 10\mu s$ is both sufficient and supportable using the switch CPU. The disadvantage of this emulation approach is that the number of active queues the system can use could drop from $2n$ to n at certain points in time. However, our evaluations show that AFQ can perform reasonably well even with a small number of queues (viz., 8 queues for 40 Gbps links).

5 End-host Flow Control Protocol

Although AFQ is solely a switch-based mechanism that can be deployed without modifying existing end-hosts to achieve significant performance improvement, a network-enforced fair queuing mechanism lets us optimize the end-host flow control protocol to extract even more gains. This section describes our approach, adapted from literature, for performing fast ramp ups and keeping queue sizes small at switches. If all network switches provided fair allocation of bandwidth, the bottleneck bandwidth could be measured using the packet-pair approach [26], which sends a pair of packets back-to-back and measures the inter-arrival gap.

Packet-pair flow control. We briefly describe the packet-pair flow control algorithm. At startup, two packets are sent back-to-back at line-rate, and the returning ACK separation is measured to get an initial estimate of the channel RTT and bottleneck bandwidth. Normal transmission begins by sending packet-pairs paced at a rate equal to the estimated bandwidth. For every packet-pair ACK received during normal transmission, the bandwidth estimate is updated and the packet sending rate adjusted accordingly. If the bandwidth estimate decreases, a few transmission cycles are skipped, proportional to the rate decrease, to avoid queue buildup. Similarly, when the bandwidth estimate increases, a few packets, again proportional to the rate increase, are injected immediately to maintain high link utilization as described in [24], which also studies the stability of such a control-theoretic flow control.

Although this approach works well for an ideal fair-queuing network, we need to make some modifications for it to be robust against approximations introduced by AFQ. The complete pseudocode of our flow control protocol is available in Appendix A.

Robust bandwidth estimation. Since AFQ transmits multiple bytes in a single round, the packet-pair approach can incorrectly estimate bottleneck bandwidth if two back-to-back packets are enqueued in the same round and transmitted one after the other. This is not an issue if the BpR is less than or equal to 1 MSS, where MSS is the maximum segment size of the packets in the pair, and it holds true for our testbed and simulations. However, if the BpR is greater than twice the MSS, we must ensure that the very first packet-pair associated with a flow maps onto different rounds to get a reasonable bandwidth estimate using the inter-arrival delay. We accomplished this by adding a delay of `BpR-MSS` bytes at line-rate in between the packet-pairs at the end-host. This careful spacing mechanism, described in [45] measures the cross-traffic observed in a short interval and extrapolates it to identify the number of flows traversing the switch at that juncture. The protocol records the packet-pair arrival gap at the receiver and piggybacks on the acknowledgment to avoid noise and congestion on the reverse path. To further reduce variance, the protocol keeps a running EWMA of bandwidth estimates in the last RTT and uses the average for pacing packet transmission.

Per-flow ECN marking. Unlike an ideal fair-queueing mechanism, where the packet with the largest round number is dropped on buffer overflow, AFQ never drops packets that have already been enqueued. As a result, AFQ must maintain short queues to absorb bursty arrival of new flows. To dissipate standing queues and keep them short, we rely on a DCTCP-like ECN marking mechanism. Each sender keeps track of the fraction of marked packets and instead of transmitting packets at the estimated rate, the protocol sends packets at estimated rate times $(1-\alpha/2)$. This optimization ensures that any standing queue is quickly dissipated. Further, unlike simple drop-tail queues, AFQ lets us perform per-flow ECN marking, which we exploit by marking packets when the enqueued bytes for a flow exceed a threshold round number. We set this number to 8 rounds in our simulations, which keeps per-flow queues very short without sacrificing throughput.

Bounding burstiness. Finally, since we have a fairly accurate estimate of the base RTT and the fair-share rate for each flow, we bound the number of inflight packets to a small multiple of the available bandwidth delay product (BDP) – similar to the rate based TCP BBR [10], – currently set to 1.5x the BDP in our implementation. This reduces network burstiness, especially when new flows arrive, by forcing older flows to stop transmitting due to their reduced BDP. This optimization keeps queues short, avoiding unnecessary queue buildup and packet drops.

We now perform a simple back of the envelope calculation to determine how to set the `BpR` parameter. As noted, we can use any end-host mechanism with AFQ, including standard ones such as TCP and DCTCP. Prior work has shown that DCTCP requires a queue of size roughly $1/6^{th}$ of the bandwidth delay product for efficient link utilization [3]. If the average round-trip latency of the datacenter network is d and the peak line rate is l, then we require $d \times l/6$ amount of buffering for a single flow to ensure maximum link utilization. Further, if we have `nQ` queues in the system, then we set `BpR` to $d \times l/(6 \times nQ)$. In practice, this is less than a MSS for a 40 Gbps link, 20 us RTT, and 10-20 queues. Further, the amount of buffering required by a single flow can be even lower by using an end-host protocol that leverages packet-pair measurements (such as that described above). Section 6.2.3 provide empirical data from our experiments to show that our end-host protocol does indeed maintain lower levels of per-flow buffer buildup than traditional protocols and that packet drops are rare.

6 Evaluation

We evaluated AFQ's overall performance, fairness guarantees and feasibility using: (1) a hardware prototype based on a Cavium network processor within a small cluster, (2) large-scale packet-level simulations, and (3) a programmable switch implementation in P4.

6.1 Hardware Prototype

Existing reconfigurable switches do not expose the programmability of internal queues, we therefore built a prototype of an AFQ switch using a programmable network processor. The Cavium OCTEON platform [14] has a multi-core MIPS64 processor with on-board memory and 4x10Gbps network I/O ports alongside several hardware-assisted network/application acceleration units, such as a traffic manager, packet buffer management units, and security co-processors. All of these components are connected via fast on-chip interconnects providing high performance, low latency, and programmability for network applications ranging from 100Mbps to 200Gbps.

6.1.1 AFQ Switch Implementation

We built a 4-port AFQ switch on top of the network processor using the Cavium Development Kit [13]. Figure 6 shows the high-level architecture, which includes 4 ingress pipelines, 4 egress pipelines, 32 FIFO packet queues, and a count-min sketch table containing 4 rows and 16K columns. The number of ports was fixed due to hardware limitations while all other individual components, such as ingress/egress pipelines, queue and table sizes were configured based on available resources.

Each ingress and egress pipeline instance runs on a dedicated core, sharing access to packet buffer queues and the count-min sketch stored on the on-board DRAM. The ingress pipeline implements most of the AFQ functionality. First, it parses the packet and computes multiple hashes using on-chip accelerators for indexing into the count-min sketch. Next, it estimates the current round number for the packet using the algorithm shown in Figure 3. Finally it updates the count-min sketch and enqueues the packet in the queue corresponding to the estimated round number. The egress simply dequeues packets from the queue corresponding to the current round being serviced, re-encapsulates the packets and transmits them to the specific port based on a pre-loaded MAC table.

Each packet queue maintains a shared lock to avoid race conditions arising from concurrent accesses of the ingress and egress cores. Other queue state updates and sketch table reads/writes use lock-free operations. We use the software reference counting technique to avoid TOCTOU race conditions.

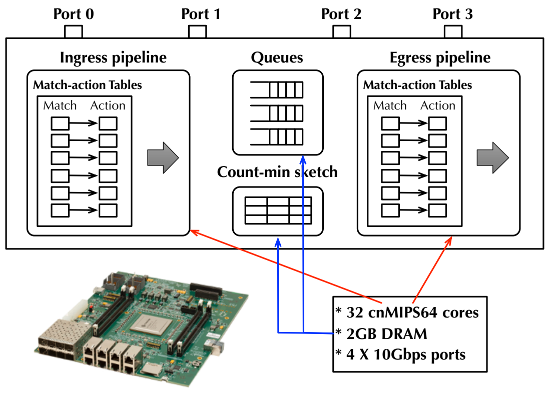


Figure 6: High-level architecture of the AFQ switch prototype.

6.1.2 End-host Protocol Implementation

We implemented the packet-pair flow control protocol (Section 5) in user-space on top of UDP and integrated it with our workload generator. The implementation uses hardware timestamps from the NIC to measure the spacing between packet-pairs to accurately obtain bandwidth estimate and RTT samples, similar to prior work [30]. The flow control re-implements standard TCP sequencing, fast retransmit, and recovery in user-space atop UDP.

6.1.3 Hardware Testbed and Workload

Our testbed includes 8 Supermicro servers, 2 Cavium XPliant switches and the prototype AFQ switch atop the network processor described above. All servers are equipped with 2x10Gbps port NICs. We created a 2-level topology using VLANs to divide the physical ports on the two switches. We integrated the prototyped AFQ switch into the aggregation switch which runs the AFQ mechanism at the second layer of the topology. The end-to-end latency is approximately 200μs, most of which is spent inside the network processor.

We set up 4 clients and 4 servers that generated traffic using the enterprise workload described in [1], such that all traffic traversed the AFQ switch in the aggregation layer. Each client opened 25 concurrent long-running connections to each server, and requested flows according to a Poisson process at a rate configured to achieve desired network load. We compared four schemes,

- Default Linux TCP CUBIC with droptail queues
- DCTCP [2] with ECN marking droptail queues
- DCTCP with our AFQ mechanism
- Our packet-pair flow control with AFQ mechanism

For DCTCP, we enabled the default kernel DCTCP module and set the ECN marking threshold to $K = 65$ packets. For a fair comparison, we relayed the TCP and DCTCP traffic through our emulated switch.

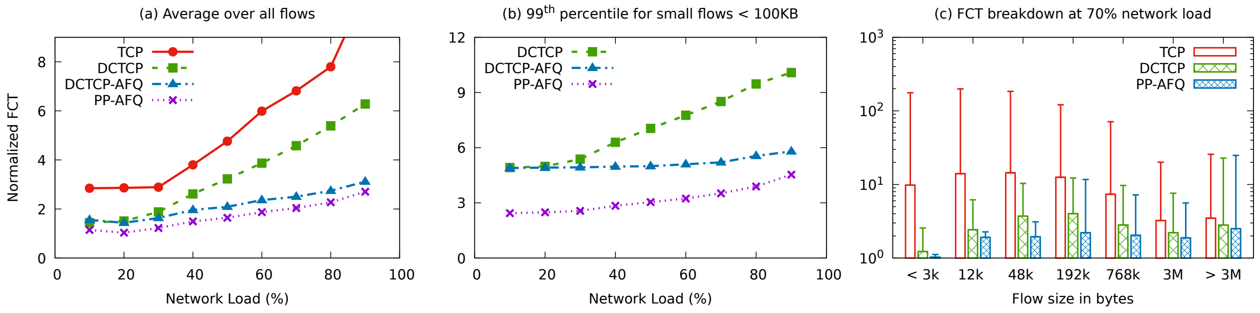


Figure 7: FCT summary for the enterprise workload on our hardware testbed. (a) average FCT for all flows, (b) tail latency for short flows, and (c) average and 99th percentile (using error bar) for various flow sizes. Note, TCP does not appear in (b) as its performance is outside the plotted range.

6.1.4 Overall Performance

We use flow completion time (FCT) as the evaluation metric and report the average and 99th percentile latency over a period of 60 seconds. Figure 7 shows FCT statistics for various flow sizes as we increase the network load; data points are normalized to the average FCT achieved in an idle network. AFQ improves DCTCP performance by 2x and TCP performance by 10x for both average and tail flow completion times. The benefits of AFQ are more visible at high network loads when there is substantial cross-traffic with high churn. In such a scenario, TCP and DCTCP take multiple RTTs to achieve fair bandwidth allocation, and suffer long queueing delays behind bursty traffic; whereas AFQ lets new flows achieve their fair share immediately and isolates them from other concurrent flows, leading to significantly more predictable performance.

Figure 7(b) also shows the improvement from our packet-pair end-host flow control over DCTCP, as the packet-pair approach avoids slow-start and begins transmitting at fair bandwidth allocation immediately after the first ACK. This fast ramp-up along with fair allocation at AFQ switches translates to significant FCT improvement, especially for short flows, as shown in Figure 7(c).

6.2 Software Simulation

We also studied AFQ's performance in a large-scale cluster deployment using an event-driven, packet-level simulator. We extended the mptcp-htsim simulator [36] to implement AFQ and several other comparison schemes.

6.2.1 Simulation Topology and Workload

We simulated a cluster of 288 servers connected in a leaf-spine topology, with 9 leaf and 4 spine switches. Each leaf switch is connected to 32 servers using 10Gbps links; and each spine switch is connected to each leaf using 40Gbps links. All leaf and spine switches have a fixed-sized buffer of 512KB and 1MB per port respectively. The end-to-end round-trip latency across the spine (4 hops) is $\approx 10\mu s$. All flows are ECMP load balanced across all spine switches. We use a small value of min-RTO = $200\mu s$ for all schemes, as suggested in [4].

We used both synthetic and empirical workloads derived from traffic patterns observed in productions datacenters. The synthetic workload generates Pareto distributed ($\alpha = 1.1$) flows with mean flow size 30KB. The empirical workload is based on an enterprise cluster reported in [1]. Flows arrive according to a Poisson process at randomly and independently chosen source-destination server pairs from all servers. The arrival rate is chosen to achieve a desired level of utilization in the spine links. Both workloads are heavy-tailed with majority bytes coming from a small fraction of large flows; both also have a diverse mix of short and long flows, with the enterprise workload having more short flows.

6.2.2 Comparison Schemes

- **TCP**: Standard TCP-Reno with fast re-transmit and recovery, but without SACKs, running on switches with traditional drop-tail queues
- **DCTCP**: The DCTCP [2] congestion control algorithm with drop-tail queues supporting ECN marking on all switches; marking threshold set to 20 packets for 10Gbps links and 80 packets for 40Gbps links
- **SFQ**: Same TCP-Reno as above with Stochastic Fair Queueing [29] using DRR [39] on all switch ports; with 32 FIFO queues available at each switch port
- **AFQ**: Our packet-pair flow control with AFQ switches using 32 FIFO queues per port, a count-min sketch of size 2x16384, and a `BpR` of 1 MSS
- **Ideal-FQ**: An ideal fair queueing router that implements the BR algorithm (described in [18]) and uses our packet-pair flow control at the end-host

6.2.3 Overall Performance

We compared the overall performance of various schemes in the simulated topology by measuring the FCT of all flows that finished over a period of 10 seconds in the simulation. Figures 8 and 9 show the normalized FCT (normalized to the average FCT achieved in an idle network) for all flows, short flows (<100KB) and flows bucketed across different sizes at varying network loads and workloads.

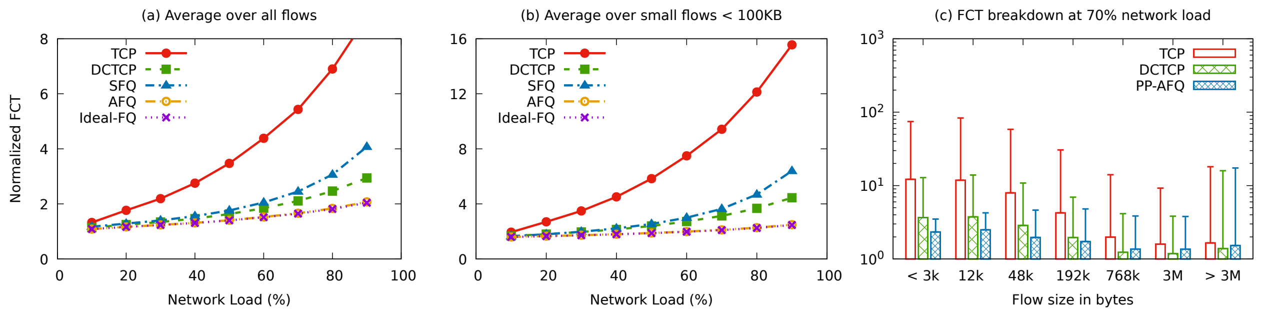


Figure 8: Flow completion times for synthetic workload in the cluster. (a) average FCT for all flows, (b) average FCT for flows shorter than 100KB, and (c) average and 99th percentile (using error bar) for various flow size buckets at 70% network load.

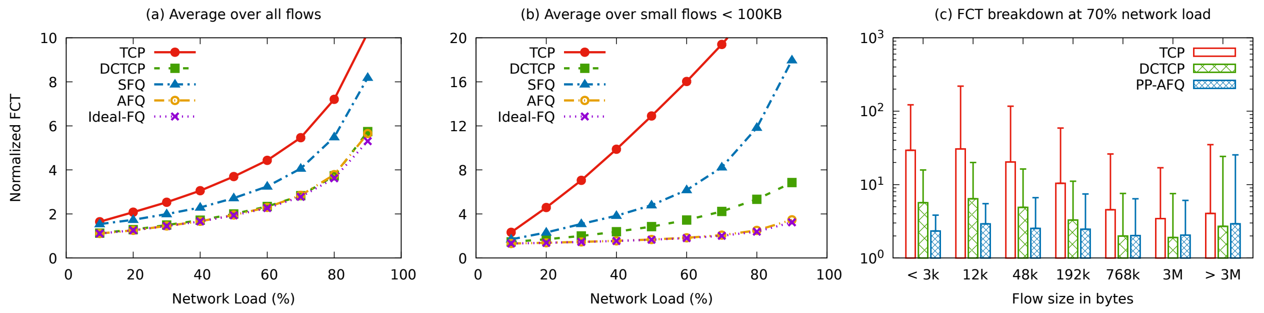


Figure 9: Flow completion times for enterprise workload, with each graph showing the same metrics as in Figure 8.

Our simulation results match previous emulated observations. As expected, most schemes perform close to optimal at low network load, but quickly diverge as network traffic increases. Traditional TCP keeps switch buffers full, leading to long queueing delays, especially for shorter flows. DCTCP improves the performance of short flows significantly since it maintains shorter queues, but is still a factor of 2-4x away from ideal fair-queuing behavior. SFQ works very well at low network loads when the number of active flows is comparable to number of queues, however as network traffic increases, collisions within a single queue become more frequent leading to poor performance. AFQ achieves close to ideal fair queuing performance for all network load, which is 3-5x better that TCP and DCTCP for tail latency of short flows: irrespective of other network traffic, all flows immediately get their fair share of the network without waiting behind other packets. This leads to significant performance benefit for shorter flows, which do not have to wait behind bursty traffic.

To further understand the performance gains, we measured several other metrics, such as packet drops, retransmissions, average queue lengths, and buffer occupancy distribution during the experiment. Figure 10(a) shows the average bytes dropped per flow for each scheme. As expected, standard TCP drops on average one packet per flow, and DCTCP has negligible drops at low network load. However, at higher loads, drops are more frequent, leading to occasional re-transmission and performance penalty. This is also reflected in the average queue length shown in Figure 10(b). Both DCTCP and packet pair with AFQ are able to maintain very short queues, but with an interesting difference in the buffer occupancy distribution as shown in Figure 10(c). We took periodic snapshots on the queue every $100\mu s$, to count how many packets belong to each flow in the buffer and plotted the CCDF of number of packets per flow across all snapshots. AFQ with packet-pair flow control rarely has more than 5 packets enqueued per flow at the core links, whereas DCTCP and TCP have many more packets buffered per flow. This can lead to unfairness when bursty traffic arrives, such as during an incast, which we discuss next. In summary, AFQ achieves similar performance to DCTCP for all flows, and 2x better performance for short flows while maintaining shorter queues and suffering fewer drops by ensuring fair allocation of bandwidth and buffers.

6.2.4 Incast Patterns

Incast patterns, common in datacenters, often suffer performance degradation due to poor isolation. In this setup, we started a client on every end-host which requests a chunk of data distributed over N other servers. Each sender replies back with $1/N$ of the data at the same time. We report the total transfer time of the chunk of data with a varying number of senders averaged over multiple runs. Simultaneous multiple senders can cause unfair packet drops for flow arriving later, causing timeouts that delay some flows and increase overall completion time. An ideal fair-queuing scheme would allocate

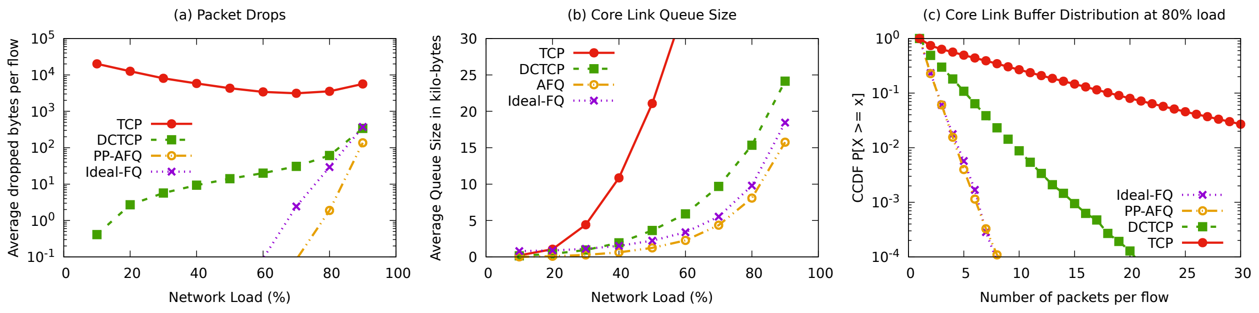


Figure 10: Packet drops, queue lengths and buffer occupancy distribution for enterprise workload in the cluster.

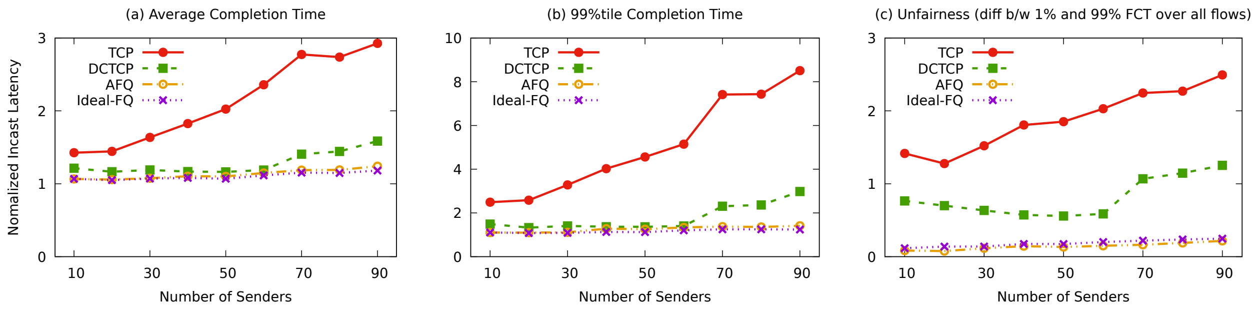


Figure 11: Completion time summary for an incast request of size 1.5MB from a varying number of senders.

equal bandwidth and buffer to each sender, hence finishing all transfers at roughly the same time.

Figure 11 shows the total completion time of various schemes for a total chunk size 1.5MB with varying number of senders. The receiver link has approximately 300KB of buffer, roughly around 200 packets. Most schemes perform well with few senders, but degrade when the number of senders overwhelms the receiver link buffer. This leads to packet drops for flows arriving later in a traditional droptail queue, sending them into timeouts. AFQ achieves close to optimal request completion time, even with large senders because it ensures each flow gets fair buffer allocation regardless of when it arrives. As a result packet drop are minimal, leading to fewer re-transmissions and lower completion time. Figure 12 shows the number of packet drops observed during the incast, packet re-transmissions, and buffer occupancy, which confirm the preceding observation. As expected, TCP drops several packets throughout the incast experiment, causing several re-transmissions. DCTCP performs much better and suffers zero packet drops until the number of senders exceeds the link buffer capacity. AFQ has even fewer drops than DCTCP because it distributes the available buffer space in a fair manner among all flows, as shown in Figure 12(c).

6.3 P4 Implementation

To evaluate the overhead of implementing AFQ on an actual reconfigurable switch, we expressed AFQ in the P4 programming language and compiled it to a production switch target. The P4 code ran on top of a baseline switch implementation [41] that provides common functionality of today's datacenter switches, such as basic L2 switching (flooding, learning, and STP), basic L3 routing (IPv4, IPv6, and VRF), link aggregation groups (LAGs), ECMP routing, VXLAN, NVGRE, Geneve and GRE tunneling, and basic statistics collection. The compiler implements the functionality proposed in [23] and compiles to the hardware model described in Section 2.2. It reports the hardware usage of various resources for the entire implementation.

Resource	Baseline	+AFQ		+AFQ-Large	
Pkt Header Vector	187	191	+2%	191	+2%
Pipeline Stages	9	12	+33%	12	+33%
Match Crossbar	462	465	+1%	465	+1%
Hash Bits	1050	1082	+3%	1092	+4%
SRAM	165	178	+8%	190	+15%
TCAM	43	44	+2%	44	+2%
ALU Instruction	83	90	+8%	90	+8%

Table 1: Summary of resource usage for AFQ.

Table 1 shows the additional overhead of implementing two variants of AFQ as reported by the compiler. AFQ uses a count-min sketch of size 2x2048, while AFQ-Large uses a sketch of size 3x16384. We can see the extra overhead is small for most resources. We need more pipeline stages to traverse the count-min sketch and keep a running minimum, and more SRAM to store all the flow counters. We also use extra ALU units to perform per-packet increments and bit-shifts to divide by `BpR`.

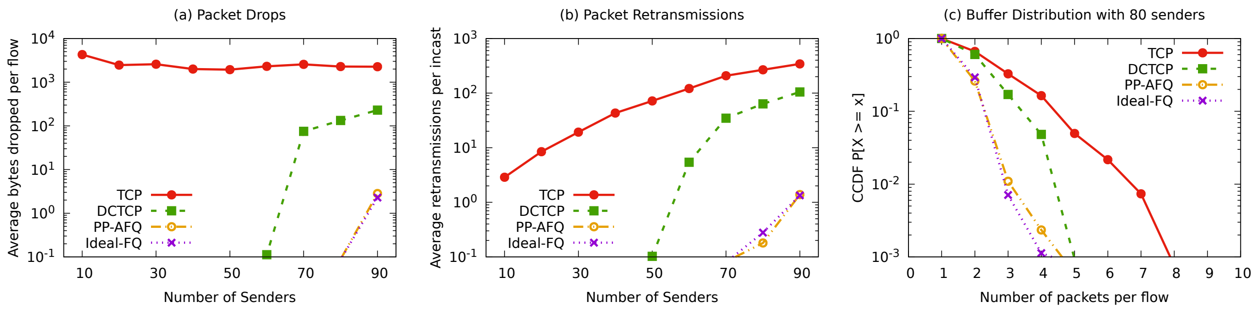


Figure 12: Packet drops, re-transmissions and buffer distribution across flows during incast traffic.

7 Related Work

Starting from Nagle's proposal [31] for providing fairness by using separate queues, several algorithms have been designed to implement a fair queueing mechanism inside the network. In [18], an efficient bit-by-bit round robin (BR) algorithm was developed to provide ideal fair queuing without per-flow queue support; [25] describes an efficient implementation of the BR algorithm. However, its inherent complexities make it hard to implement on today's high-speed routers.

Many algorithms were later proposed to reduce the complexity and cost of implementing fair queuing mechanisms. Most either use stochastic approaches or avoid complex per-flow management by using simpler heuristics. Stochastic Fair Queuing (SFQ) [29] hashes flows onto a reduced set of FIFO queues and perturbs the hashing function periodically to minimize unfairness. An efficient realization of SFQ using Deficit Round Robin (DRR) was proposed in [39]. However, its fairness guarantees are closely tied to the number of queues, and performance degrades significantly when the number of active flow exceeds the number of queues.

Several schemes [33, 28, 27] enforce fairness by dropping packets of flows sending faster than their fair share. They estimate flow rate by tracking recent history of packet arrivals or the current buffer occupancy. A variant, called Stochastic Fair Blue [19], uses an array of bloom filters to store packet counts and drop probabilities, which is similar to how AFQ stores round numbers. Core-Stateless Fair Queueing (CSFQ) [44] enforces fair allocation by splitting the mechanism between the edge and the core network. All complexity of rate estimation/labeling is at the edge, and the core performs simple packet forwarding based on labels. It achieves fairness by dropping packets with probability proportional to the rate above the estimated fair rate.

Other schemes – PIAS [5] and FDPA [11] also leverage multiple priority queues available in commodity switches to emulate shortest-job-next scheduling or achieve approximate fair bandwidth allocation using an array of rate estimators to assign flows to different priority queues. Although similar, AFQ uses multiple queues to emulate an ideal fair-queueing algorithm. A more recent approach PIFO [42] proposes a programmable scheduler that can implement variants of priority scheduling and ideal fair queuing at line rate by efficiently implementing $O(logN)$ sorted insertion complexity in hardware. However, like fixed-function schedulers, the number of distinct flows that can be scheduled is bound in hardware and can support up to 2048 flows. In addition, we discuss how to store approximate per-flow bid numbers in limited switch memory, and increment current round number efficiently, which has not been explored in prior work.

8 Conclusion

In this paper, we proposed a fair bandwidth allocation mechanism called Approximate Fair Queueing (AFQ), designed to run on emerging reconfigurable switches. We approximate the various mechanisms of a fair queueing scheduler using features available on reconfigurable switches. Specifically, we approximate the per-flow state regarding the number and timing of its previously transmitted packets using mutable switch state; we perform limited computation for each packet to compute its position in the output schedule; we dynamically determine which egress queue to use for a given packet; and we design a new dequeuing approach, called the Rotating Strict Priority scheduler, to transmit packets in approximate sorted order. Using a networking-processor-based prototype in a real hardware testbed and large scale simulations, we showed that AFQ approximates ideal queuing behavior accurately, improving performance significantly over existing schemes. We also showed that the overhead of implementing AFQ on top of programmable switches is fairly minimal.

Acknowledgments

We would like to thank the anonymous NSDI reviewers and our shepherd Mohammad Alizadeh for their valuable feedback. We also thank Antoine Kaufmann for many insightful discussions. This research was partially supported by the National Science Foundation under Grants CNS-1518702, CNS-1616774, and CNS-1714508.

References

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

[2] ALIZADEH, M., GREENBERG, A., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., AND SRIDHARAN, M. Data Center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM Conference* (2010).

[3] ALIZADEH, M., JAVANMARD, A., AND PRABHAKAR, B. Analysis of DCTCP: Stability, convergence, and fairness. In *Proceedings of the ACM SIGMETRICS Conference* (2011).

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

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

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

[7] BENSON, T., AKELLA, A., AND MALTZ, D. A. Network traffic characteristics of data centers in the wild. In *Proceedings of the ACM SIGCOMM Conference on Internet Measurement* (2010).

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

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

[10] CARDWELL, N., CHENG, Y., GUNN, C. S., YEGANEH, S. H., AND JACOBSON, V. BBR: Congestion-based congestion control. *Queue 14*, 5 (Oct. 2016), 50:20–50:53.

[11] CASCONE, C., BONELLI, N., BIANCHI, L., CAPONE, A., AND SANSÒ, B. Towards approximate fair bandwidth sharing via dynamic priority queuing. In *Local and Metropolitan Area Networks (LANMAN)* (2017), IEEE.

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

[13] CAVIUM. OCTEON Development Kits, 2016. `http://www.cavium.com/octeon_software_develop_kit.html`.

[14] CAVIUM. Cavium OCTEON SoC Development Board, 2017. `http://www.cavium.com/OCTEON_MIPS64.html`.

[15] CHOLE, S., FINGERHUT, A., MA, S., SIVARAMAN, A., VARGAFTIK, S., BERGER, A., MENDELSON, G., ALIZADEH, M., CHUANG, S.-T., KESLASSY, I., ORDA, A., AND EDSALL, T. dRMT: Disaggregated programmable switching. In *Proceedings of the ACM SIGCOMM Conference* (2017).

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

[17] CORMODE, G., AND MUTHUKRISHNAN, S. An Improved Data Stream Summary: The Count-Min Sketch and its Applications. *Journal of Algorithms 55*, 1 (2005), 58–75.

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

[19] FENG, W.-C., KANDLUR, D. D., SAHA, D., AND SHIN, K. G. Stochastic Fair Blue: A queue management algorithm for enforcing fairness. In *IEEE INFOCOM* (2001).

[20] JAFFE, J. Bottleneck flow control. *IEEE Transactions on Communications 29*, 7 (1981), 954–962.

[21] JAIN, R., CHIU, D., AND HAWE, W. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. *CoRR cs.NI/9809099* (1998).

[22] JIN, X., LI, X., ZHANG, H., SOULÉ, R., LEE, J., FOSTER, N., KIM, C., AND STOICA, I. NetCache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017).

[23] JOSE, L., YAN, L., VARGHESE, G., AND MCKEOWN, N. Compiling packet programs to reconfigurable switches. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (2015).

[24] KESHAV, S. A control-theoretic approach to flow control. In *Proceedings of the ACM SIGCOMM Conference* (1991).

[25] KESHAV, S. On the efficient implementation of fair queueing. *Journal of Internetworking: Research and Experience 2* (1991), 157–173.

[26] KESHAV, S. The packet pair flow control protocol. Tech. Rep. 91-028, ICSI Berkeley, 1991.

[27] LIN, D., AND MORRIS, R. Dynamics of random early detection. In *ACM SIGCOMM Computer Communication Review* (1997).

[28] Mahajan, R., Floyd, S., and Wetherall, D. Controlling high-bandwidth flows at the congested router. In *Network Protocols, 2001. Ninth International Conference on* (2001), IEEE, pp. 192–201.

[29] McKenney, P. E. Stochastic fairness queueing. In *INFOCOM'90, IEEE Computer and Communication Societies. The Multiple Facets of Integration. Proceedings, IEEE* (1990).

[30] Mittal, R., Lam, V. T., Dukkipati, N., Blem, E., Wassel, H., Ghobadi, M., Vahdat, A., Wang, Y., Wetherall, D., and Zats, D. TIMELY: RTT-based congestion control for the datacenter. In *Proceedings of the ACM SIGCOMM Conference* (2015).

[31] Nagle, J. B. On packet switches with infinite storage. In *Innovations in Internetworking*. Artech House, Inc., 1988, pp. 136–139.

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

[33] Pan, R., Breslau, L., Prabhakar, B., and Shenker, S. Approximate fairness through differential dropping. *ACM SIGCOMM Computer Communication Review 33*, 2 (Apr. 2003).

[34] Parekh, A. K., and Gallager, R. G. A generalized processor sharing approach to flow control in integrated services networks: The single-node case. *IEEE/ACM Transactions on Networking 1*, 3 (1993), 344–357.

[35] Popa, L., Kumar, G., Chowdhury, M., Krishnamurthy, A., Ratnasamy, S., and Stoica, I. Fair-Cloud: Sharing the network in cloud computing. In *Proceedings of the ACM SIGCOMM Conference* (2012).

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

[37] Roy, A., Zeng, H., Bagga, J., Porter, G., and Snoeren, A. C. Inside the social network's (datacenter) network. In *Proceedings of the ACM SIGCOMM Conference* (2015).

[38] Sharma, N. K., Kaufmann, A., Anderson, T., Krishnamurthy, A., Nelson, J., and Peter, S. Evaluating the power of flexible packet processing for network resource allocation. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, 2017), pp. 67–82.

[39] Shreedhar, M., and Varghese, G. Efficient fair queueing using deficit round robin. In *Proceedings on the ACM SIGCOMM Conference* (1995).

[40] Sivaraman, A., Cheung, A., Budiu, M., Kim, C., Alizadeh, M., Balakrishnan, H., Varghese, G., McKeown, N., and Licking, S. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the ACM SIGCOMM Conference* (2016).

[41] Sivaraman, A., Kim, C., Krishnamoorthy, R., Dixit, A., and Budiu, M. DC.P4: Programming the forwarding plane of a data-center switch. In *Proceedings of the ACM SIGCOMM Symposium on Software Defined Networking Research* (2015).

[42] Sivaraman, A., Subramanian, S., Alizadeh, M., Chole, S., Chuang, S.-T., Agrawal, A., Balakrishnan, H., Edsall, T., Katti, S., and McKeown, N. Programmable packet scheduling at line rate. In *Proceedings of the ACM SIGCOMM Conference* (2016).

[43] Song, H. Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking* (2013).

[44] Stoica, I., Shenker, S., and Zhang, H. Core-Stateless Fair Queueing: Achieving approximately fair bandwidth allocations in high speed networks. In *Proceedings on the ACM SIGCOMM Conference* (1998).

[45] Strauss, J., Katabi, D., and Kaashoek, F. A measurement study of available bandwidth estimation tools. In *Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement* (2003).

[46] Varghese, G., and Lauck, A. Hashed and hierarchical timing wheels: Efficient data structures for implementing a timer facility. *IEEE/ACM Transactions on Networking 5*, 6 (Dec. 1997), 824–834.

A End-host Pseudocode

Figure 13 shows our adapted packet-pair flow control at the end-host. Each flow begins by transmitting a pair of back-to-back packets and waits for the acks to return. The receiver measures the packet inter-arrival gap and returns it back to the sender piggybacked on the ack. After receiving the first ack, it starts normal transmission at the estimated rate in packet-pairs. For every ack received during normal transmission, the rate estimate is updated based on the packet gap and ECN marks.

SENDER PROTOCOL

```
Startup():
  state = STARTUP
  SendPacketPair()

On AckReceive(pktpair, rtt):
  newGap = pktpair.gap

  if (rtt < minRTT):
    minRTT = rtt

  if state == STARTUP:
    /* Start normal packet transmission. */
    state = NORMAL
    gap = newGap
    SendPacketPair()
  else:
    /* Update rate estimate. */
    gap = (1 - GAIN) * gap + GAIN * newGap
    linkRate = MSS / gap
    bdp = linkRate * minRTT

    /* Throttle rate based on ECN marks. */
    rate = linkRate * (1 - alpha / 2)

SendPacketPair():
  /* Bound inflight bytes to roughly bdp. */
  if (inflight > CWND_FACTOR * bdp):
    /* Wait for ack or retransmission timeout. */
    return

  packet1 = nextPacket()
  packet1.first = true
  send(packet1)

  /* Add delay if necessary. */

  packet2 = nextPacket()
  send(packet2)

  if (state == STARTUP):
    Wait for AckReceive()
  else:
    nextSendTime = now() + 2 * MSS / rate
    scheduleTimer(SendPacketPair, nextSendTime)
```

RECEIVER PROTOCOL

```
OnPacketReceive (packet):
  if (packet.first == true):
    first_pktpair_time = now()
    pktpair_ts = packet.sendTime
  else:
    gap = now - first_pktpair_time
    ack = nextAck()
    ack.sendTime = pktpair_ts
    ack.gap = gap
    send(ack)
```

Figure 13: Pseudocode for endhost flow control protocol

B Convergence and Fairness

To demonstrate that AFQ does indeed assign each flow its fair share rapidly, we connected two hosts via a 10Gbps, 10μs RTT link and sequentially started-stopped flows at 1-second intervals. We used standard TCP end-hosts, and change the queueing mechanism from droptail to AFQ. The time series in Figure 14 shows the throughput achieved by each flow as they enter and exit the link. AFQ assigns each flow its fair share immediately, while a droptail queue exhibits high variance in throughput.

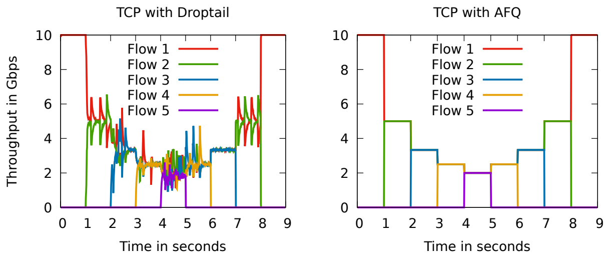


Figure 14: Convergence test

Next, we plot the FCT versus flow size from our cluster simulations in Figure 15 to demonstrate how fair each scheme is with respect to flow size. An ideal fair queuing scheme would be a straight line from the origin. All schemes achieve fairness over a period of time for long flows, but are significantly unfair to short flows either due to slow-start or queueing behind other flows in the network. AFQ lets all flows, regardless of size to achieve their fair share within an RTT, leading to better fairness.

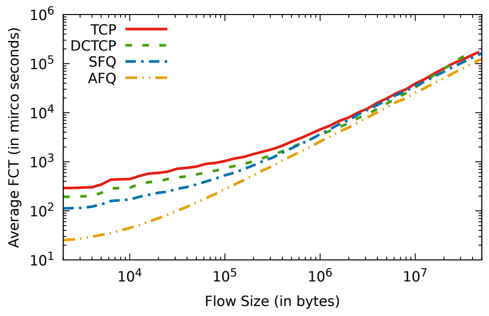
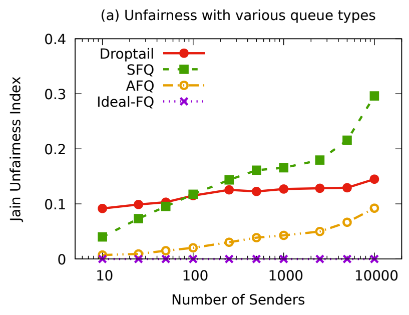
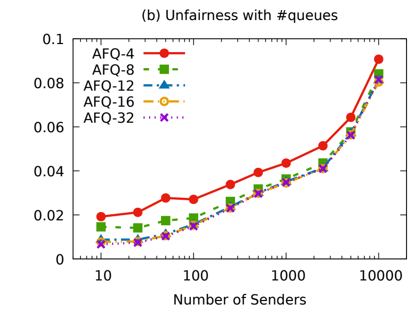


Figure 15: FCT vs flow size at 70% network load.

To further study AFQ's fairness guarantees, we simulated a 10Gbps, 25μs RTT link and increased the number of on-off senders transmitting concurrent flows on the link. We measured the Jain Unfairness index (1−Jain Fairness [21]) across all flows. Figure 16(a) shows the unfairness across different queueing schemes. AFQ has better fairness than other schemes, until the number of concurrent flows exceeds the sketch size. Figures 16(b) and (c), plot the same metric while varying the sketch-size and number of FIFO queues available to AFQ.





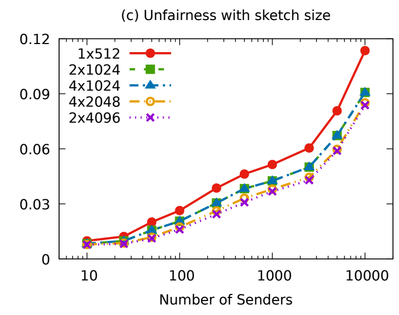
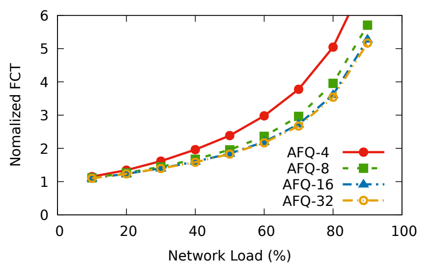
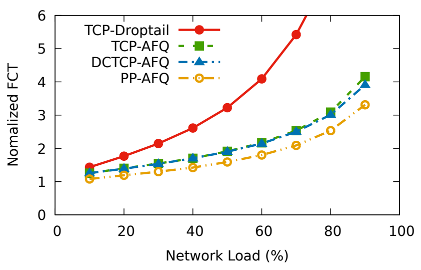


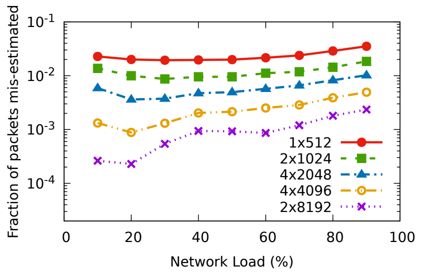
Figure 16: Micro-benchmarks showing deviation of various queuing mechanism compared to ideal fair queuing using a DCTCP end-host.



(a) FCT vs number of FIFO queues



(b) AFQ performance with end-host protocols



(c) Round Estimation vs Sketch Size

Figure 17: Benchmarks showing the impact of various AFQ parameters.

C Impact of Number of Queues on FCT

AFQ uses multiple FIFO queues to store packets in an approximate sorted order. To understand how many FIFO queues are required per-port to get accurate fair-queueing behavior, we ran the same enterprise workload while varying the number of FIFO queues available to the AFQ implementation and keeping `BpR` fixed at 1 MSS. Figure 17a shows the impact on average FCT of all flows as we varied the number of queues from 4 to 32. When fewer queues are available, AFQ buffers packets for very few rounds at any given time. This causes unnecessary packet drops during bursty arrivals, and also leads to poor bandwidth estimation at the endhost. Once there are sufficient queues to absorb packet bursts and accurately estimate bottleneck bandwidth, AFQ achieves near ideal fair queueing behavior, which occured around 16-20 queues. This is not surprising, given the analysis from [2], a queue of size roughly $1/6^{th}$ of the bandwidth delay product is required for efficient link utilization. For our testbed with 40 Gbps links and 20μs RTT, this value is ≈20KB, translating to about 15 queues.

D AFQ with Other End-host Protocols

As an in-network switch mechanism, AFQ can be deployed without modifying the end-host to achieve significant performance gains. To quantify the benefits, we simulate the same enterprise workload using TCP, DCTCP end-hosts with all switches implementing the AFQ mechanism. Figure 17b shows the significant improvement in average FCT when switching from droptail to AFQ behavior inside the network. Moving to DCTCP gives another small improvements due to shorter queues; finally, using packet-pair flow control eliminates slow-start behaviors, further reducing FCT. This matches our observations from the hardware prototype emulation.

E Impact of Sketch Size

AFQ stores bid numbers in a count-min sketch, trading off space for accuracy. To determine how large a sketch is required to achieve sufficient accuracy without affecting performance, we re-ran the enterprise workload in the leaf-spine topology while tracking exact bid numbers and those returned by the count-min sketch. During the 10 second simulation run, we count how many times a packet bid number was mis-estimated and enqueued in a later-than-expected queue. Figure 17c shows the mis-estimation rate as we change the sketch size. This is less than 1% using a relatively small sketch of 2x1024. This is not surprising since the collision probability is proportional to number of active flows that have packets enqueued at the switch, which is generally a few tens to hundreds. It is not affected by the total number of on-going flows, which could be several thousands. Such a low rate of mis-estimation does not significantly impact the flow-level performance, because a bad estimate delays the packet by only a small amount of time. Further, increasing the number of cells in each row has a more significant impact on the accuracy than increasing the number of rows.