Twenty Years After: Hierarchical Core-Stateless Fair Queueing

Zhuolong Yu, Jingfeng Wu, and Vladimir Braverman, Johns Hopkins University; Ion Stoica, UC Berkeley; Xin Jin, Peking University

https://www.usenix.org/conference/nsdi21/presentation/yu
Twenty Years After:
Hierarchical Core-Stateless Fair Queueing

Zhuolong Yu
Johns Hopkins University

Jingfeng Wu
Johns Hopkins University

Vladimir Braverman
Johns Hopkins University

Ion Stoica
UC Berkeley

Xin Jin
Peking University

Abstract
Core-Stateless Fair Queueing (CSFQ) is a scalable algorithm proposed more than two decades ago to achieve fair queueing without keeping per-flow state in the network. Unfortunately, CSFQ did not take off, in part because it required protocol changes (i.e., adding new fields to the packet header), and hardware support to process packets at line rate.

In this paper, we argue that two emerging trends are making CSFQ relevant again: (i) cloud computing which makes it feasible to change the protocol within the same datacenter or across datacenters owned by the same provider, and (ii) programmable switches which can implement sophisticated packet processing at line rate. To this end, we present the first realization of CSFQ using programmable switches. In addition, we generalize CSFQ to a multi-level hierarchy, which naturally captures the traffic in today’s datacenters, e.g., tenants at the first level and flows of each tenant at the second level of the hierarchy. We call this scheduler Hierarchical Core-Stateless Fair Queueing (HCSFQ), and show that it is able to accurately approximate hierarchical fair queueing. HCSFQ is highly scalable: it uses just a single FIFO queue, does not perform per-packet scheduling, and only needs to maintain state for the interior nodes of the hierarchy. We present analytical results to prove the lower bounds of HCSFQ. Our testbed experiments and large-scale simulations show that CSFQ and HCSFQ can provide fair bandwidth allocation and ensure isolation.

1 Introduction

Fair queueing is a canonical mechanism to provide fair bandwidth allocation to network traffic by ensuring that each flow gets its fair share irrespective of the other flows. This way, fair queueing enforces isolation between competing flows, which ensures that normal flows are protected from ill-behaving flows. There is a long history of research on fair queueing [1–12]. Many of the proposed solutions require to maintain per-flow state in the switch, and rely on complex data structures and scheduling algorithms to realize fair queueing.

Core-Stateless Fair Queueing (CSFQ) [13] is a scalable algorithm to realize fair queueing. Compared to the alternatives, CSFQ has the unique property that it does not maintain per-flow state in the network. With CSFQ, the sources or switches at the edge classify traffic into flows and estimate per-flow rate. In turn, the switches in the network estimate the fair rate, and use probabilistic dropping to regulate each flow to its fair rate without maintaining per-flow state.

While CSFQ was proposed more than twenty years ago, it has not taken off. This is primarily due to two reasons. First, it requires changes to the IP protocol (i.e., adding a field to the IP header) and coordination across all switches (routers) in the network. Second, CSFQ requires switches to estimate the fair rate, compute a drop probability, and update the header of each packet. To perform these operations at line rate we need hardware support. These challenges are exacerbated by the fact that routers belong to different, often competing, Internet Service Providers (ISPs), which would all need to cooperate to upgrade their infrastructures to support CSFQ.

However, two emerging technologies are making CSFQ relevant again: (i) the advent of cloud computing and (ii) the increased popularity of programmable switches. Cloud providers own large datacenters consisting of many thousands of servers. Since a datacenter is typically owned by a single administrative entity (cloud provider) that controls both the software and hardware, it is relatively easy for a cloud provider to upgrade all its switches and servers to support CSFQ. FairCloud [14] proposes to apply CSFQ for network isolation in datacenters, but it does not have a hardware implementation for CSFQ. The emergence of programmable switches makes it possible to implement sophisticated packet processing at line rate. In particular, as we will show in this paper, existing programmable switches are powerful enough to support CSFQ at line rate.

While datacenter deployment removes the adoption barriers for CSFQ, it also raises new challenges. In particular, while CSFQ has been designed for a flat hierarchy, the traffic in today’s datacenters is naturally structured in a multi-level hierarchy. For example, at the top level we typically have...
tenants and at the bottom level we have the flows of those tenants. The mechanism of choice to manage such traffic is hierarchical fair queueing \([9,10,15]\), where each non-leaf node distributes its excess bandwidth (i.e., the bandwidths unused by some of its children) across its children. This allocation policy is consistent with a per-tenant payment granularity, i.e., network resources are divided between tenants in proportion to their payments \([14]\). In this case, if a flow of a tenant stops sending data, that tenant would want to re-allocate the flow’s bandwidth to its other flows, and not to the flows of other tenants in the datacenter.

However, implementing hierarchical fair queueing is challenging. Existing solutions require per-flow state, and more importantly, require complex queue management and packet transfers in a hierarchy of queues \([9,10,15]\). Because of the implementation complexity, hierarchical fair queueing is not supported by today’s high-speed hardware switches.

To address this challenge, we propose Hierarchical Core-Stateless Fair Queueing (HCSFQ). CSFQ only provides fair queueing, not hierarchical fair queueing. Directly extending CSFQ to support hierarchical fair queueing would require a hierarchy of queues. HCSFQ is able to accurately approximate hierarchical fair queueing and it is highly scalable. The key difference of our approach is that HCSFQ requires only a single queue, not a hierarchy of queues. HCSFQ also requires no packet scheduling. HCSFQ recursively computes the fair rate of each node starting from the root, and then limits the rate of each flow to its fair share rate. To the best of our knowledge, HCSFQ is the first solution that enables hierarchical fair queueing on commodity hardware at line rate while requiring neither per-flow state nor hierarchical queue management.

An important distinction of HCSFQ from CSFQ is that HCSFQ keeps the state of the interior nodes of the hierarchy in the switch. The state of the interior nodes is necessary to support hierarchical fair queueing, as the fair share rates of distinct interior nodes are typically different. The excess bandwidth of a flow is only shared with its sibling flows. That is, if a flow changes its sending rate, it would impact the fair rate of the sibling flows, but not necessarily of other flows in the hierarchy. Note that similar to CSFQ, HCSFQ does not maintain per-flow state (i.e., the state of the leaf nodes). Fortunately, for today’s multi-tenant clouds, the number of tenants is orders of magnitude smaller than the number of flows, and commodity switches have sufficient on-chip memory to maintain the state for these interior nodes.

We exploit the capability of programmable switching to provide the first realization of CSFQ and HCSFQ on commodity hardware. While conceptually simple, implementing these schedulers on a programmable switch raises several technical challenges. First, they use a complex formula to estimate the rates, which includes several floating-point multiplication, divisions and exponentiation operations. Unfortunately, these operations are not supported by today’s programmable switches. To get around this challenge, we leverage high-precision timestamps and a window-based mechanism to estimate these rates. Second, these algorithms rely on probabilistic packet dropping to limit the flows to their fair rates. Unfortunately, probabilistic packet dropping cannot be directly implemented in these switches. We discretize the probability computation to approximate the dropping probability with bounded error. To discretize these probabilities we leverage the switch’s random number generator and take advantage of multiple stages. Third, computing the fair rate exhibits a circular dependency. Unfortunately, the switch data plane consists of a multi-stage processing pipeline, and the later stages cannot modify the state in the previous stages. To address it, we judiciously use packet recirculation, and periodically update the fair rate to minimize recirculation overhead.

In summary, we make the following contributions.

- We extend CSFQ to HCSFQ, the first scalable, practical solution to implement hierarchical fair queueing on commodity hardware at line rate with no per-flow state and no hierarchical queue management.
- We exploit the capability of programmable switching ASICs to provide the first data plane design for CSFQ and HCSFQ.
- We implement a prototype of CSFQ and HCSFQ on a Barefoot Tofino Wedge 100BF-65X switch. Our experiments show that CSFQ and HCSFQ can provide fair bandwidth allocation and ensure isolation.

2 Background and Motivation

Our work is motivated by the need for network isolation in multi-tenant datacenters. CSFQ is a scalable solution for fair queueing. We review the background of CSFQ, and identify the opportunities for CSFQ in modern datacenters.

2.1 Core-Stateless Fair Queueing

Fair queueing provides max-min fairness for competing flows. A max-min fair bandwidth allocation is one that any increase of the allocation to some flows would necessarily decrease the allocation of some other flows. The basic way to realize fair queueing in a switch is to keep one queue for each flow and use a scheduling algorithm to pick which queue to dequeue a packet each time. There has been decades of research on fair queueing \([1–12]\). While we leave the extensive discussion to related work \((\S7)\), we emphasize that most solutions are not scalable because of the need to maintain per-flow state to classify flows and shape their rates with per-flow queues and complex queue management. As a result, commodity switches only support 10–20 queues.

CSFQ is a scalable algorithm to achieve fair queueing with a unique property that it does not maintain per-flow state in the network. Figure 1 shows the architecture of CSFQ. CSFQ divides the network into edge and core. The switches
or hosts at the edge, which do maintain per-flow state, use per-flow state to classify packets into flows and estimate per-flow arrival rate. Then the arrival rate of each flow is *carried in a custom packet header*. The switches in the core only estimate the *total* arrival rate of all flows, and then use it to estimate the *fair share rate* with an iterative algorithm. The switches compare the per-flow arrival rate in the packet header with the fair share rate to compute a drop probability, and drop packets to shape the rate of each flow to the fair share rate.

The key benefit of CSFQ is that the complexity (packet classification and flow rate estimation with per-flow state) is moved to the edge, making the core extremely simple. A core switch only maintains the state for *aggregate* variables (total arrival rate, total accepted rate and fair share rate), and only uses one queue for packet buffering. More importantly, the complexity of a core switch does not change with the number of flows, making the core scale-free.

### 2.2 Opportunities

CSFQ did not take off because it requires cooperation between ISPs to provide end-to-end isolation for Internet flows, and requires protocol and hardware changes. After twenty years, we believe the time for CSFQ has come because of two opportunities.

The first opportunity is from cloud computing. Cloud computing has become the fundamental infrastructure of today’s Internet. Datacenters power large-scale Internet services we use everyday such as search, social networking and e-commerce, and enterprises are increasingly moving their workloads to the cloud. Core bandwidth allocation and network isolation for datacenter networks is under a single administrative domain. Two-layer fair queueing for multi-tenant datacenters naturally eliminates the need of cooperation between different operators or ISPs, as a datacenter network is under a single administrative domain.

The second opportunity is from programmable switching ASICs. Traditional switching ASICs are fixed-function, and adding a new feature like CSFQ requires switch vendors to design a new ASIC. Emerging programmable switching ASICs, such as Barefoot Tofino [29], Broadcom Trident 4 [30] and Cavium XPliant [31], allow users to program the data plane and develop new features. Specifically, to implement CSFQ on a programmable switch, we can program the parser to parse the custom header of CSFQ (to carry per-flow rate), program the match-action tables to implement the CSFQ algorithm, and program the on-chip memory to store the aggregate state. Because a datacenter network is under a single administrative domain, it is easy for the operator to adopt the protocol and hardware changes with programmable switching ASICs.

### 3 Hierarchical Fair Queueing

A multi-tenant cloud has a natural two-layer hierarchy, with the tenants at the first layer and the flows of each tenant at the second layer. Network isolation for multi-tenant datacenters naturally requires hierarchical fair queueing. CSFQ only supports fair queueing, but not hierarchical fair queueing. Hierarchical fair queueing provides fair queueing in a hierarchical manner. Flows are grouped into flow aggregates in multiple layers. The root of the tree includes all the flows. Each node in the tree includes a subset of the flows, called a *flow aggregate*, and fairly allocates its bandwidth to its child nodes. This is done recursively until leaf nodes, each of which contains one flow. The flows are broadly defined, e.g., based on five-tuple or network management considerations. In the case of multi-tenant clouds, it is a two-layer bandwidth allocation. The bandwidth is first allocated to the tenants in the first layer, and then each tenant allocates its bandwidth to its own flows in the second layer.

Fair queueing allocates bandwidth fairly to competing flows, and is work conserving, i.e., unused bandwidth share of a flow can be allocated to other flows. The key benefit of hierarchical fair queueing is that it allows unused share of a flow to be allocated to other flows in the same flow aggregate, instead of being shared by all the flows. Fair queueing can be considered as a special case of hierarchical fair queueing that contains only one layer. Two-layer fair queueing for multi-tenant clouds is desirable because the payment is based on...
We first use a fluid model to formalize hierarchical fair queueing. There are four flows, i.e., \( f_1, f_2, f_3, \) and \( f_4 \). The arrival rates of the four flows are 1, 4, 5, and 5, respectively. The link capacity is 10. With only fair queueing, the unused share of \( f_1 \) is evenly allocated to all other three flows. As shown in Figure 3(a), the bandwidth allocation to the four flows is \( 1, 3, 3, 3 \). Suppose that \( f_1 \) and \( f_2 \) are in one flow aggregate \( (A_1) \), and \( f_3 \) and \( f_4 \) are in the other \( (A_2) \). With hierarchical fair queueing, the unused fair share of \( f_1 \) is only allocated to \( f_2 \), instead of also being shared by \( f_3 \) and \( f_4 \). Figure 3(b) shows the bandwidth allocation with two-layer hierarchical fair queueing, where the flows receive 1, 4, 2.5, and 2.5, respectively.

**Challenge.** Hierarchical fair queueing is known to be challenging to realize in switches at high speed. A traditional design to support hierarchical fair queueing is to leverage a hierarchy of queues, and each node in the hierarchy implements fair queueing for the queues of its child nodes. Figure 3(a) shows an example of such a design to support the two-layer hierarchy in Figure 3(b). This design has two major problems. First, the amount of state and the number of queues needed by this design is proportional to the number of nodes in the hierarchy. It needs to maintain per-flow state and the state of each interior node in the tree. Second, the design involves complex queue management with a hierarchy of queues, as packets need to be moved between queues in different layers. CSFQ does not require maintaining per-flow state, but naively extending CSFQ to support hierarchical fair queueing would still require a hierarchy of queues as shown in Figure 3(b). These two factors together make the design hard to scale to support a large number of flows. As a result, hierarchical fair queueing is not supported by today’s high-speed switches.

### 4 HCSFQ Design

We propose Hierarchical Core-Stateless Fair Queueing (HCSFQ), which generalizes CSFQ to support hierarchical fair queueing. HCSFQ is the first scalable solution that enables hierarchical fair queueing on commodity hardware at line rate without per-flow state and complex hierarchical queue management.

We give a high-level overview of HCSFQ in Figure 3(c). In contrast to the traditional design in Figure 3(a), HCSFQ has two unique properties: (i) it does not maintain per-flow state, but only keeps the state of interior nodes; (ii) it does not require a hierarchy of queues, but only uses one queue. These two properties together dramatically simplify the design, making HCSFQ amenable to be implemented on high-speed switches under strict timing and resource constraints.

The major distinction between HCSFQ and CSFQ is that HCSFQ needs to maintain the state of interior nodes. This is necessary because HCSFQ aims to provide hierarchical fair bandwidth allocation for a flow hierarchy. Note that the naive design of extending CSFQ in Figure 3(b) also requires maintaining the state of interior nodes. In fair queueing, CSFQ only requires to keep one fair share rate, which is the same for all flows. But in hierarchical fair queueing, the fair share rates for different flows can be different if two flows are not siblings (i.e., do not have the same parent node). If a flow changes its rate, it would affect the fair share rate of its sibling flows, but not necessarily those of non-sibling flows. Figure 4 illustrates this with a concrete example. There is a two-layer hierarchy with four flows. At time \( T_1 \), the arrival rates for the four flows are 1, 4, 5, and 5 (the same as Figure 2). The fair share rate at \( L \) is 5, and those at \( A_1 \) and \( A_2 \) are 4 and 2.5. Then at time \( T_2 \), \( f_1 \) increases its arrival rate from 1 to 2. Then under fair bandwidth allocation, the new fair share rate for the subtree under \( A_1 \) becomes 3, so that \( f_1 \) receives 2 and \( f_2 \) receives 3. The rate change of \( f_1 \), however, does not affect the fair share rate for \( f_3 \) and \( f_4 \). This is because \( f_3 \) and \( f_4 \) are not sibling nodes of \( f_1 \).

CSFQ can be considered as a special case of HCSFQ which contains only one layer, and as such, it only carries the state for one interior node—the root.

#### 4.1 Fluid Model

We first use a fluid model to formalize hierarchical fair queueing. The fluid model considers a switch with output link capacity \( C \), and the flows are modeled as a continuous stream of bits. The flow hierarchy is represented as a directed graph \( G(V, E) \), where \( V \) is the set of nodes and \( E \) is the set of edges.
A node \( v \in V \) represents a flow aggregate (i.e., a set of flows), where \( r(v) \) is the arrival rate of the flow aggregate and \( c(v) \) is the capacity allocated to \( v \). A directed edge \( e(v, u) \in E \) represents that \( u \) is a child of \( v \).

Max-min fair bandwidth allocation ensures that the flows that are bottlenecked by a link receives the same output rate, which we call the fair share rate. Let \( \alpha(v) \) be the fair share rate that node \( v \) allocates to its children. If max-min fair bandwidth allocation is achieved, for a child node \( u \) of node \( v \), the flow aggregate at \( u \) receives a bandwidth allocation of \( c(u) = \min(r(u), \alpha(v)) \). The arrival rate of \( v \) is the sum of the arrival rates of its children, i.e., \( r(v) = \sum_{v(u) \in E} r(u) \). If \( r(v) > c(v) \), the arrival rate of \( v \) exceeds the capacity allocated to \( v \), and the fair rate \( \alpha(v) \) is the unique solution to

\[
 c(v) = \sum_{e(v,u) \in E} \min(\alpha(v),r(u)). \tag{1}
\]

If \( r(v) \leq c(v) \), the arrival rate of \( v \) is no more than the capacity allocated to \( v \), and all flows in \( v \) can be forwarded without dropping packets. In this case, by convention we have

\[
 \alpha(v) = \max_{e(v,u) \in E} r(u). \tag{2}
\]

The fair rate computation is done recursively from the root to the leaf nodes. When \( v \) is the root, we have \( c(v) = C \), where \( C \) is the link capacity. Then starting from the root, we can compute \( c(v) \) and \( \alpha(v) \) for each node in the tree.

Based on this fluid model, there is a simple algorithm to achieve max-min fair bandwidth allocation. In this algorithm, we first use the recursive computation to compute \( \alpha(v,\text{parent}) \) for each leaf node \( v \), which is the fair share rate allocated by \( v \)'s parent to \( v \). If \( r(v) \leq \alpha(v,\text{parent}) \), then no bits need to be dropped; otherwise, a fraction of \( (r(v) - \alpha(v,\text{parent}))/r(v) \) need to be dropped. Therefore, achieve max-min fair bandwidth allocation, each incoming bit of the flow in \( v \) is dropped by probability

\[
 \max(0, 1 - \frac{\alpha(v,\text{parent})}{r(v)}). \tag{3}
\]

4.2 HCSFQ Algorithm

The HCSFQ algorithm realizes the conceptual fluid algorithm in a real switch. Similar to CSFQ, HCSFQ does not maintain per-flow state, and only requires a single FIFO queue for packet buffering (Figure 3). The algorithm relies on two building blocks from CSFQ, which are arrival rate estimation and fair share rate estimation, and applies them recursively to compute the fair share rate for each leaf node.

Arrival rate estimation. The arrival rate estimation is used to estimate the arrival rate of a flow aggregate for a node in the hierarchy. Like CSFQ, it uses the canonical exponential averaging mechanism in networking for rate estimation. Let \( t_i \) and \( l_i \) be the arrival time and length of the \( i^{th} \) packet of the flow aggregate in node \( v \). We use \( r(v) \) to denote the estimated arrival rate of \( v \). It is updated each time a new packet of \( v \) arrives, based on the following equation,

\[
 r(v)_{\text{new}} = (1 - e^{-T/K}) \frac{l_i}{T_i} + e^{-T/K} r(v)_{\text{old}}, \tag{4}
\]

where \( T_i = t_i - t_{i-1} \) and \( K \) is a constant.

Fair share rate estimation. The fair share rate estimation is used to estimate the fair share rate that a node allocates to its children. The capacity of node \( v \) is \( c(v) \). Eq.(4) gives the arrival rate of the node \( r(v) \). If \( r(v) \leq c(v) \), then \( \alpha(v) \) is calculated using Eq.(2). Otherwise, \( \alpha(v) \) should be the unique solution to Eq.(1). We apply the iterative algorithm in CSFQ to approximately solve the equation. Specifically, for each node \( v \), we maintain the accepted rate estimation \( f(v) \), which is updated with Eq.(4) if the packet is not dropped. Then, \( \alpha(v) \) is approximately computed with the following formula,

\[
 \alpha(v)_{\text{new}} = \alpha(v)_{\text{old}} \frac{c(v)}{f(v)}. \tag{5}
\]

Note that the computation of \( \alpha(v) \) is iterative. It converges to the solution of Eq.(1) after several iterations, i.e., processing several packets. Similar to CSFQ, HCSFQ also uses a window of size \( K_c \) to account for inaccuracies introduced by exponential averaging in rate estimation. That is, \( \alpha(v) \) is updated only if the node is congested \( (r(v) > c(v)) \) or uncongested \( (r(v) \leq c(v)) \) for an interval of length \( K_c \).

Packet state. A packet \( \text{pkt} \) carries two pieces of state in the packet header, which are \( \text{pkt}\_r \) and \( \text{pkt}\_nodes \).

- \( \text{pkt}\_r \) is the arrival rate estimate of the flow the packet belongs to.
- \( \text{pkt}\_nodes \) is a list of node IDs that indicate the flow aggregates the packet belongs to in the flow hierarchy, excluding the leaf. For example, in Figure 2, if a packet \( \text{pkt} \) belongs to \( f_1 \) or \( f_2 \), then \( \text{pkt}\_nodes = [L,A_1] \).

CSFQ only carries \( \text{pkt}\_r \) in the packet header as there is no flow hierarchy. HCSFQ additionally carries \( \text{pkt}\_nodes \) to track the set of flow aggregates the packet belongs to in the hierarchy. Similar to CSFQ, both \( \text{pkt}\_r \) and \( \text{pkt}\_nodes \) are
Algorithm 1 HCSFQ(pkt)

1: `cur_α ← 0`
2: for `v ∈ pkt.nodes` do
3:     // estimate arrival rate
4:     `r[v] ← estimate_rate(pkt)`
5:     // calculate drop probability
6:     `prob ← max(0, 1 - cur_α[v])`  
7:     if `prob > rand(0, 1)` then
8:         `drop_flag ← TRUE`
9:     for `v ∈ pkt.nodes` do
10:         // allocate bandwidth
11:         if `v` is root then
12:             `c[v] ← link capacity`
13:         else
14:             `c[v] ← min(α[v, parent], r[v])`
15:         // update fair share rate
16:         if `r[v] > c[v]` then
17:             if `congestフラグ[v]` is `FALSE` then
18:                 `congestフラグ[v] ← TRUE`
19:             `start_time ← current_time`
20:             else if `current_time – start_time > K` then
22:             else
23:                 if `congestフラグ[v]` is `TRUE` then
24:                     `congestフラ格[v] ← FALSE`
25:                 `start_time ← current_time`
26:                 `tmp_α[v] ← 0`
27:             else if `current_time – start_time ≤ K` then
28:                 `child_r ← r[v] / r[v]` if `v.next`
29:                 `tmp_α[v] ← max(tmp_α[v], child_r)`
30:             else
31:                 `α[v] ← tmp_α[v]`
32:             `start_time ← current_time`
33:             `tmp_α[v] ← 0`
34:         `cur_α[v] ← α[v]`
35:         // drop or enqueue pkt
36:         if `dropフラグ` then
37:             `drop(pkt)`
38:         else
39:             `enqueue(pkt)`
40:         // update the packet rate
41:         `pkt_r ← min(cur_α, pkt_r)`

inserted at the edge. An edge switch (e.g., a software switch, a NIC or a ToR switch in datacenter networks) performs packet classification to get `pkt.nodes`, and uses Eq. (4) to estimate the flow rate `pkt_r`. Both `pkt_r` and `pkt.nodes` are transparent to end hosts and are removed by the switch at the last hop.

Hierarchical computation. The main difference between HCSFQ and CSFQ is that HCSFQ performs fair share rate estimation recursively in a hierarchical manner. In CSFQ, the arrival rate estimation for each flow is done at the edge, and a core switch only estimates the total arrival rate. In HCSFQ, because there is a hierarchy of flow aggregates, a core switch additionally estimates the arrival rate for each flow aggregate (i.e., the internal nodes in the tree). Similarly, in CSFQ, a core switch only calculates a fair share rate for the link, while in HCSFQ, a core switch additionally calculates a fair share rate for each flow aggregate. Importantly, the fair share rate estimation in HCSFQ is used to bridge the computation of different layers together. That is, for node `v`, the allocated bandwidth `c(v)` is used to estimate the fair share rate `α(v)`, which is then used to compute the allocated bandwidth of its children, i.e., `c(u)` for `u ∈ v.children`, in the next layer.

Algorithm 1 shows the pseudo code of the HCSFQ algorithm. When a packet `pkt` arrives at the switch, the switch updates the arrival rate estimate for each flow aggregate the packet belongs to using Eq. (4), and gets the fair share rate of the flow (line 1-4). Then the switch computes the dropping probability based on Eq. (3) and decides whether to drop the packet (line 5-7). After this, the switch recursively updates the fair share rate of each flow aggregate in the hierarchy (line 8-34). Based on whether the packet is dropped, the switch updates the accepted rate estimate for each flow aggregate (line 9-10). If node `v` is the root, then all flows are under this node, and its allocated capacity is the link capacity (line 11-12); otherwise, its allocated capacity is the max of the fair share rate allocated by its parent and its arrival rate (line 13-14). If the arrival rate of `v` is bigger than its allocated capacity, then the node is congested, and the fair share rate is updated based on Eq. (5) (15-21); otherwise, the fair share rate is the max arrival rate of its children, i.e., based on Eq. (2) (line 22-33).

Note that we use a window of length `K` for fair share update to account for inaccuracies in rate estimation. Based on the dropping decision, the switch drops or enqueues the packet (line 35-38). Finally, the arrival rate `pkt_r` is updated and will be used by the next-hop switch (line 39). Note that the loops (line 2-4 and line 8-34) are done in one pass and the fair share rate is updated based on `c[v, parent]` from the last round.

Figure 5 illustrates how the algorithm works to realize hierarchical fair queueing for the example in Figure 2(b). At the root, the total arrival rate of all flows `r(L)` is 15, and the capacity `c(L)` is the link capacity 10, which is below the arrival rate. The root fairly allocates the capacity to the two flow aggregates, `A_1` and `A_2`. The figure shows the stable state when the accepted rates and fair share rates of all the nodes have converged. After convergence, the accepted rate `f(L)` is 10, and the fair share rate `α(L)` is 5. At node `A_1`, the arrival rate `r(A_1)`, which is the sum of `r(f_1)` and `r(f_2)`, is 5, and the
weighted HCSFQ. The HCSFQ algorithm can be extended
to support flows and flow aggregates with weights. For node
v, we use w(v) to represent the weight of the flow or flow aggre-
grate of v. Under max-min fair bandwidth allocation, compet-
ing flows or flow aggregates at the bottlenecked link have
the same fair share rate r(v)/w(v). There are two changes
to the algorithm in order to incorporate the weight. The first
change is on the equation to compute the fair rate α(v) when
r(v) > c(v). Eq.(1) is changed to
\[ c(v) = \sum_{e \in \mathcal{E}, u \in e} w(u) \cdot \min(\alpha(v), \frac{r(u)}{w(u)}). \]
The second change is on the equation to compute the drop
probability. Eq.(3) is changed to
\[ \max(0, 1 - \alpha(v, \text{parent}) \cdot \frac{w(v)}{r(v)}). \]

### 4.3 Theoretical Guarantee

We have the following theorem to provide the theoretical guar-
antees for HCSFQ. The proof of the theorem is in Appendix.

**Theorem 1.** Consider a link with a hierarchical fair queueing
policy and a flow in the hierarchy. Let w_1, w_2, ..., w_n be the
weights of the nodes from the root to the flow. Let α_1, α_2, ..., α_n
be the constant normalized fair rate of the nodes from the
root to the flow. Let r_α = α_i w_i. If probabilistic dropping is
applied at the last layer, then the excess service received by
the flow that sends at a rate at no larger than R, is bounded
above by
\[ r_\alpha K (1 + \ln \frac{R}{r_\alpha}) + l_{\text{max}} \]
where l_{max} is the maximum packet length.

Consider a parent and its children in the hierarchy. Let the
number of children be k. Let r_{\alpha'} be the weighted fair rate of
the parent, and r_{\alpha'}^{(j)} be the weighted fair rate of the j-th child.
Suppose the inter-arrival time of every packet is at least \( \tau \), and
\[ r_{\alpha'} \geq \frac{1}{1 - e^{-\tau/K}} \sum_{j=1}^{k} r_{\alpha'}^{(j)}. \]
The parent node does not drop packets.

**Remark.** The first conclusion bounds the excess service that
can be received by a flow. The second conclusion provides
the theoretical condition for only performing probabilistic
dropping at the leaf node.

### 5 Data Plane Design and Implementation

In this section, we describe a data plane design to imple-
ment CSFQ and HCSFQ on new-generation programmable
switches. Programmable switches enable users to program the
multi-stage match-action pipeline in the switch data plane to
implement custom features. Users can also access the on-chip
memory and implement stateful operations using the register
arrays provided by programmable switches. Programmable
switches also support a set of primitive actions (e.g., recircu-
late, bit shift, add and subtract) which make HCSFQ possible.
Based on the constructs of programmable switches, we show
how to design and implement the rate estimation, the fair rate
computation and the flow shaping logic (i.e., Algorithm 1) on
programmable switches. Our HCSFQ implementation con-
tains 1952 lines of code in P4 and is compiled to Barefoot
Tofino ASIC [29]. The code is open-source and available at
https://github.com/netx-repo/HCSFQ.

#### 5.1 Single Layer

We first describe how to implement CSFQ, i.e., single-layer
HCSFQ, which is used as a building block to implement multi-
layer HCSFQ. There are three challenges to implement single-
layer HCSFQ on programmable switches: rate estimation, proba-
bilistic drop, and fair rate update. We describe each
challenge and its solution as follows.

**Rate estimation.** The switch needs to estimate two rates: the
total arrival rate \( r \), and the accepted rate \( f \). Both rates are
estimated with Eq.(4). Because switches have strict timing
and resource requirements, an action in a match-action table
can only contain a small number of operations in a limited
operation set. The equation cannot be directly implemented in
the switch data plane due to two reasons. First, the equation
involves several multiplication, division and exponentiation
operations on floating points. These operations are quite com-
plex and require multiple clock cycles to compute. As such,
they are not typically supported by the switch data plane.
Second, a rate (\( r \) or \( f \)) is stored in a register of the on-chip
memory. To update the rate, the switch needs to read the rate
from the register, uses the equation to calculate the new rate,
and then updates the register. A register can only be accessed
by its own stage, but the equation includes multiple arithmetic
operations, which requires multiple stages to compute.

We leverage the high-precision timestamps available in the
data plane, and use a window-based mechanism to estimate
the rates. Programmable switches are able to provide high-
precision timestamps at the granularity of one nanosecond.
To estimate a rate, the switch maintains a pair of registers
(reg.byte and reg.start). One register (reg.byte) stores the
total bytes of packets the switch has received in the current
window. The other register (reg.start) stores the start times-
stamp of the current window. For each incoming packet, the
switch first checks the current timestamp and compares it
with \( \text{reg.start} \) to see if the packet belongs to the current window. If so, the switch adds the size of the packet to \( \text{reg.byte} \); otherwise, the switch clears \( \text{reg.byte} \) and sets \( \text{reg.start} \) to the current timestamp. The switch keeps another register \( \text{reg.rate} \) to store the current rate estimate. When a window is passed, the switch uses \( \text{reg.byte} \) to update \( \text{reg.rate} \), which can be done with either a direct assignment, or a moving average. Our experiments indicate that using a moving average (implemented with several bit shift and addition operations) works better with either a direct assignment, or a moving average. We convert the condition to discrete assignment and drops packets.

The key benefit of this window-based mechanism is that the switch can provide nanosecond-granularity timestamps, we can use a small window size to accurately estimate flow rate and capture sudden packet bursts. It is important to note that the rate estimation is local to the switch and only uses timestamps to divide time into windows. So there is no need for time synchronization between switches.

**Probabilistic drop.** Probabilistic drop is used to regulate the flows to the fair share rate. The switch uses the fair share rate \( \alpha \) and the flow arrival rate \( r \) to compute the probability to drop packets of the flow (Eq.3 and line 5 in Algorithm 1). Then the switch checks the condition \( \max(0, 1 - \alpha/r) > \text{rand}(0, 1) \) to decide whether to drop an incoming packet or not. Similar to rate estimation, the challenge is that switches do not support the division operation to compute the probability. One way to solve the problem is to use a similar window-based mechanism as rate estimation, i.e., divide time into windows with window size \( \delta \), and keep counters to allow up to \( \delta r \) packets to pass in each window and drop all remaining packets. The drawback of this approach is that it introduces bursty packet drops, which do not work well with congestion control. We want to mimic the behavior of CSFQ to have random packet drops that are uniformly distributed in the packet stream.

We discretize the probability computation to approximate the drop probability with bounded error. We leverage the random number generator provided by the data plane and use multiple stages to realize the discretized computation. Specifically, to check the condition \( \max(0, 1 - \alpha/r) > \text{rand}(0, 1) \), it is sufficient to check \( \text{rand}(0, 1) > \alpha/r \). We multiply \( r \) to both sides of the inequality, and transform the condition to

\[
\text{rand}(0, r) > \alpha.
\]

If the switch can generate a random number between 0 and \( r \), then we can simply compare the generated random number and \( \alpha \) to decide whether to drop a packet. However, some switches can only generate a random number in a range of a power of two, i.e., in \([0, 2^n - 1]\), where \( n \) is a given value at compilation time and cannot be a variable. One possible solution is to use a large value for \( n \) at compilation time and use \( \text{rand}(0, 2^n - 1)\%r \) to approximate \( \text{rand}(0, r) \). But the modulo operation on an arbitrary number may not be supported, and the generated numbers are not uniformly distributed in \([0, r]\). We solve this problem by discretizing the probability computation. We use an integer, instead of a floating point, for the probability. We convert the condition to

\[
\text{rand}(0, 2^n - 1)\cdot r > (2^n - 1)\cdot \alpha.
\]

While multiplication is not directly supported, we can convert a multiplication operation into several bit shift and addition operations. Since \( n \) is small and one stage can do multiple operations, a multiplication can be done in a few stages. This solution introduces errors because the random number is an integer in \([0, 2^n - 1]\), instead of a real number in \([0, 1]\). However, the error is bounded by \( 1/2^n \), which reduces exponentially with \( n \). When \( n \) is 7, the error introduced by the approximation is bounded by 1/128, which is smaller than 1%.

**Fair rate update.** When the link is congested, the fair share rate is the unique solution to Eq.(1). Because HCSFQ does not maintain per-flow state, it uses \( \alpha_{\text{new}} = \alpha_{\text{old}}c/f \) (Eq.5) to approximately compute the fair share rate, where \( c \) is the capacity and \( f \) is the accepted rate. Like rate estimation and probabilistic drop, Eq.(5) cannot be supported because it contains multiplication and division. What is more challenging is that the fair rate update introduces the following circular dependency to the packet processing.

\[
\text{read } \alpha \to \text{enqueue/drop} \to \text{update } f \to \text{update } \alpha
\]

Specifically, the switch needs to read \( \alpha \) to compute the drop probability. Then based on whether to enqueue or drop a packet, the switch updates the accepted rate \( f \), which is then used to update \( \alpha \). Because a register can only be accessed by its own stage, the new value of \( \alpha \) cannot be used to update the register that stores \( \alpha \) in a previous stage.

To address these two problems, we first observe that the update equation \( \alpha_{\text{new}} = \alpha_{\text{old}}c/f \) in HCSFQ is already an approximation, and the correct \( \alpha \) is iteratively computed after several updates until \( f \) converges to \( c \). As such, we replace the update equation with an additive-increase multiplicative-decrease method, which increases or decreases \( \alpha \) each time if \( f \) is not equal to \( c \). This ensures that the value for \( \alpha \) converges to the correct value. Note that in the original CSFQ, \( \alpha \) is also computed iteratively to converge to the correct value.

To address the circular dependency, we leverage packet recirculation available in programmable switches, and let the recirculated packets carry the new value of \( \alpha \) to update the register for \( \alpha \) in a previous stage. Switches have limited band-width for recirculation. We judiciously use packet recirculation to minimize recirculation overhead. We follow the same scheme as CSFQ: update \( \alpha \) only when the node is congested or uncongested for a window length of \( K_c \). Given the window size \( K_c \), \( \alpha \) is updated by at most \( 1/K_c \) times per second. As a concrete example, let \( K_c \) be 10\,\mu s. Then \( \alpha \) is updated by at most 100K times per second, and the amount of recirculation traffic is only a tiny fraction of the switch capacity.
(b) Different weights.

Figure 6: Testbed experiments of fair queueing for UDP. Flow 1–24 send at 2Gbps and Flow 25–32 send at 8Gbps.

5.2 Multiple Layers

The single-layer design is used as a building block to support multiple layers. As shown in Algorithm 1 and Figure 5, the processing of HCSFQ on a packet is performed layer by layer, from the root to the leaf node. This well matches the multi-stage packet processing pipeline of programmable switches. The layers in HCSFQ can be mapped to the stages in the pipeline, which naturally processes packets sequentially stage by stage. The major difference between HCSFQ and CSFQ is that HCSFQ needs to store more states as it has multiple layers. CSFQ is a single-layer HCSFQ and only maintains the state for three variables, which are the total arrival rate \( r \), the accepted rate \( f \), and the fair share rate \( \alpha \). Each variable use multiple registers as described in §5.1. HCSFQ maintains the state for all interior nodes, each of which includes the three variables. Commodity switches have 10-100 MB on-chip memory [32], which is able to support a large number of interior nodes. For a two-layer HCSFQ for tenant-level and flow-level isolation in multi-tenant datacenters, a switch needs to maintain per-tenant state, but not per-flow state. With 10-100 MB memory, the switch can support millions of tenants. In terms of the number of layers, our prototype supports up to four layers on Barefoot Tofino. There is no theoretical limit on the number of layers given the scalable algorithm design. The constraints for practical implementations mainly come from the restricted hardware primitives to implement the algorithm as we describe in §5.1. These constraints are not fundamental. Newer programmable switches (e.g., Barefoot Tofino 2) have more stages and provide more hardware primitives to support more layers. Despite this, we expect HCSFQ with 2–4 layers should be sufficient to provide hierarchical isolation for many datacenter scenarios (e.g., multi-tenancy).

6 Evaluation

In this section, we provide experimental results to demonstrate the performance of HCSFQ. We first evaluate the performance of single-layer HCSFQ (i.e., CSFQ), and show that it can provide fair queueing (§6.1). We then evaluate the performance of two-layer HCSFQ, and show that it can provide hierarchical fair queueing to enforce tenant-level and flow-level isolation for multi-tenant datacenters (§6.2). Finally, we use simulations to evaluate HCSFQ in a large-scale datacenter environment and compare it with several alternatives (§6.3).

All testbed experiments are conducted on a hardware testbed with a Barefoot Tofino Wedge 100BF-65X switch. Each server is configured with an 8-core CPU (Intel Xeon E5-2620 @ 2.1GHz), 64GB memory and one 40G NIC (Intel XL710), and runs Ubuntu 16.04.6 LTS with Linux kernel 4.10.0-28-generic. Our switch implementation contains both the edge and core functionalities for HCSFQ. Therefore, our prototype provides hierarchical fair queueing without modifications to either the software or hardware of the end hosts. By default, we use TCP Cubic provided by the Linux kernel.

6.1 Fair Queueing Experiments

We first evaluate the capability of HCSFQ to provide fair queueing. Fair queueing requires one-layer HCSFQ. We cover both UDP and TCP traffic with equal or different weights. In the experiments, we use four servers as the senders and one server as the receiver. Each sender sends 8 flows (based on five-tuple), and a total of 32 flows are sent to a receiver. All servers are connected to the switch with 40Gbps links. The bottleneck link is the link between the switch and the receiver.

**UDP.** If all UDP flows have the same sending rate, they would get similar bandwidth under the tail-drop FIFO queue in the switch. To make the experiment more interesting, we assign different sending rates to the UDP flows. We let 24 flows (Flow 1–24) send at 2Gbps and 8 flows (Flow 25–32) send at 8Gbps. As shown in Figure 6(a), without HCSFQ, Flow 25–32 obtain higher bandwidth than Flow 1–24 because Flow 25–32 have larger sending rates. In comparison, HCSFQ is able to fairly allocate bandwidth to the flows.

Figure 7: Testbed experiments of fair queueing for UDP. Flow 1 is sending at a different rate every 2 seconds. Flow 2 is sending at 20Gbps.

**Figure 8: Testbed experiments of fair queueing for TCP.**
HCSFQ supports weighted fair queueing. We assign weight 1 to Flow 1–24 to and weight 2 to Flow 25–32. As shown in Figure 6(b), without HCSFQ, the result is the same as that with equal weights in Figure 6(a). On the other hand, HCSFQ is able to allocate the bandwidth based on the weights. Flow 25–32 achieve higher throughput than Flow 1–24.

We also evaluate HCSFQ when the UDP flows dynamically change their rates. We let Flow 1 send at a different rate every 2 seconds (10Gbps, 20Gbps, 30Gbps and 40Gbps, respectively) and let Flow 2 keep sending at 20Gbps. Without HCSFQ, when the link is congested (from 4s to 8s), each flow achieves a throughput in proportional to its sending rate. With HCSFQ, two flows get the fair share (20Gbps) when the link is congested.

TCP. Figure 8(a) shows the throughput of the flows with and without HCSFQ. Because TCP congestion control provides fair bandwidth allocation, the flows have similar throughput even without HCSFQ. Adding HCSFQ to the switch does not change the bandwidth allocation and thus has a similar result.

However, TCP cannot support weighted fair queueing. To show the benefits of HCSFQ, we let Flow 1–24 have weight 1 and Flow 25–32 have weight 2. Without HCSFQ, the result in Figure 8(b) is similar to that in Figure 8(a). With HCSFQ, the flows get bandwidth in proportional to their weights. The flows with higher weights (Flow 25–32) receive more bandwidth than those with lower weights (Flow 1–24).

Different TCP algorithms. There are many TCP congestion control algorithms. Without in-network enforcement, the flows using aggressive congestion control algorithms would get more bandwidth. In this experiment, we let Flow 1–24 use TCP Cubic (provided by default in Linux) and Flow 25–32 use TCP BBR. As shown in Figure 9(a), without HCSFQ, because TCP BBR is more aggressive than TCP Cubic, the flows with TCP BBR get almost all the bandwidth. On the other hand, HCSFQ is able to provide fair queueing, regardless of the TCP algorithms they use. We have also tried TCP Reno, which performs similar to TCP Cubic.

Different RTTs. In this experiment, we increase the RTT of Flow 25–32 by 0.4 ms using Linux Traffic Control (Linux tc). The default RTT measured by ping in the testbed, i.e., the RTT of Flow 1–24, is 0.3 ms (mostly host overhead). The TCP throughput is inverse proportional to RTT [33]. In our case, the flows with 0.3 ms RTT (Flow 1–24) should have 0.7/0.3 ≈ 2× higher bandwidth than the flows with 0.7 ms RTT (Flow 25–32), which is close to what we see in Figure 9(b). On the other hand, HCSFQ is able to provide fair queueing even when the flows have different RTTs.

Convergence. We let four flows from different clients join and leave a link every 16 seconds to evaluate convergence. Figure 10 shows the UDP result. Flow 1 and 2 send at 40Gbps and 3 and 4 send at 20Gbps. Without HCSFQ, the result is the same as that in Figure 9(a). On the other hand, HCSFQ is able to provide fair queueing, regardless of the TCP algorithms they use. The flows get the fair share (20Gbps) when the link is congested.

Mixed UDP and TCP traffic. We evaluate HCSFQ under a mixed workload with both UDP and TCP traffic, and consider the impact of ill-behaved UDP flows on TCP flows. In the experiment, Flow 1–24 are TCP flows, and Flow 25–32 are UDP flows that send at 3.2Gbps. As shown in Figure 12(a), without HCSFQ, because UDP flows are not affected by TCP
### 6.2 Hierarchical Fair Queueing Experiments

We now evaluate the capability of HCSFQ to provide hierarchical fair queueing. We show that two-layer HCSFQ can provide tenant-level and flow-level isolation for multi-tenant datacenters. Similar to the previous experiments, we use 4 servers to send a total of 32 flows to a receiver. To evaluate hierarchical fair queueing, we let tenant A contain 24 flows (Flow 1–24) and tenant B contain 8 flows (Flow 25–32).

**UDP.** We set the sending rates of all 32 UDP flows to 8 Gbps. As shown in Figure 13(a), without HCSFQ, the flows have similar throughput. Because tenant A has three times as many flows as tenant B, the total throughput of A is three times as that of B. With HCSFQ, two tenants get the same total throughput. Because A has more flows, each flow in A has lower throughput than that in B.

To evaluate weighted hierarchical fair queueing, we assign different weights to tenant A’s flows. We let Flow 1–8 have weight 2 and Flow 9–24 have weight 1. We assign the same weight to tenant A and B. As shown in Figure 13(b), the result without HCSFQ is the same as it in Figure 13(a). All flows receive the same bandwidth, regardless of tenants and weights. With HCSFQ, because the two tenants have the same weight, the bandwidth allocation to the two tenants stays the same. In tenant A, a flow with weight 2 has double throughput as a flow with weight 1. In tenant B, all flows have the same weight, and thus they have the same throughput.

TCP. TCP congestion control does not recognize tenants. Figure 14(a) shows the throughput of 32 TCP flows. Similar to the UDP experiment, without HCSFQ, every flow receives the same amount of bandwidth, and tenant A has higher total throughput. With HCSFQ, the bandwidth is allocated equally to the two tenants, and each flow in A has lower throughput than each flow in B. We also assign weights to the TCP flows as the UDP experiment, and the result is in Figure 14(b). Similarly, with HCSFQ, Flow 1–8 in tenant A have lower throughput than Flow 9–24, because Flow 1–8 lower higher weight. The flows in tenant B have the same throughput because we do not change their weights.

---

**Figure 13:** Testbed experiments of hierarchical fair queueing for UDP. Two tenants should have the same total throughput.

**Figure 14:** Testbed experiments of hierarchical fair queueing for TCP. Two tenants should have the same total throughput.

**Figure 15:** Simulation result under the web search workload.

**Figure 16:** Simulation result under the web search workload with injected UDP traffic.
to use a small practice with DCTCP, and can provide significant improvement over a representative datacenter topology and workload as the smaller flows can finish faster with a fair share rate.

Web search workload with injected UDP traffic. To evaluate performance isolation, we inject additional ill-behaved UDP flows to the web server workload. The UDP flows are evenly distributed in the topology and occupy about half of the total bandwidth of the network. Figure 16 shows that TCP and DCTCP perform significantly worse than others, because they do not have performance isolation between TCP and UDP flows. HCSFQ performs better than AFQ and SP-PIFO, because AFQ and SP-PIFO map different flows to a small number of queues and aggressive UDP traffic overloads the queues shared by multiple TCP and UDP flows, while HCSFQ drops excessive UDP packets before they enter the queues.

Incast. This experiment evaluates HCSFQ in an incase scenario where a receiver requests for a 4.5MB file distributed over \( N (=30–180) \) sender nodes. We follow the common practice to use a small \( RTO_{\text{min}} \) (200\( \mu \text{s} \)) for all schemes [37]. As shown in Figure 17(a), when the number of flows grows, HCSFQ achieves a lower request completion time compared with SP-PIFO, TCP and DCTCP, and is close to AFQ. SP-PIFO does not handle the incast traffic pattern well, because there are many packets arriving at the same time with similar ranks saturating some queues and getting dropped. Figure 17(b) shows that HCSFQ achieves low average completion times for individual flows as the number of flows changes.

Web search workload with two tenants. This experiment evaluates hierarchical fair queueing with two tenants. Tenant 1 sends five times as many flows as tenant 2, and should have higher FCT than tenant 2.

Scalability with many tenants. We show the scalability of HCSFQ on supporting many tenants and flows. When there are many tenants and flows, the share of each tenant/flow is small and the bias from rate estimation and rate update in each step will accumulate. In this experiment, we examine 50 tenants. Half of the tenants (tenant 1-25) have one VM in each server, and the other half (tenant 26-50) have two VMs in each server. Each VM has a long-lasting TCP flow with another VM of the same tenant in another rack. We set the bandwidth of access links and leaf-spine links to 10Gbps and 40Gbps respectively in order to accommodate more tenants and flows than previous experiments. Figure 19 shows that TCP, DCTCP, AFQ and SP-PIFO do not provide tenant-level fairness, and the tenants with more flows have higher total throughput. In comparison, HCSFQ provides fair bandwidth allocation between tenants, regardless of the number of flows each tenant has.
7 Related Work

Fair queueing. There is a long history of work on fair queueing. The original proposal from Nagle [1] introduces the idea of using separate FIFO queues for flows to achieve fair bandwidth allocation. The bit-by-bit round robin (BR) algorithm [2,3] computes a bid number to estimate the departure time for each packet, and transmits the packet with the lowest bid number with a priority queue. To avoid expensive priority queues, several algorithms, such as SFQ [4] and DRR [5], propose to map flows to a small number of FIFO queues, which do not work well when the number of flows are far larger than the number of queues. Another approach is probabilistic packet dropping, which maintains per-flow state to estimate drop probability, such as FRED [6], RED-PD [7] and AFD [8]. CSFQ [13] is distinct from these algorithms in that it does not require per-flow state, per-flow queues or an expensive priority queue. Hierarchical fair queueing adds a hierarchy to fair queueing, which require not only per-flow state, but also a hierarchy of queues [9,10,15]. HCSFQ eliminates both requirements, making hierarchical fair queueing feasible to be implemented in high-speed hardware switches.

Network isolation in multi-tenant cloud. Prior work has proposed techniques to provide performance guarantees and share bandwidth between multiple tenants [14,16–28,38,39]. However, existing works either can only enforce hierarchical fairness at end hosts, or can not be efficiently implemented in today’s hardware. For example, BwE [39] is a WAN bandwidth allocation mechanism which enforces hierarchical fair allocation at end hosts. FairCloud [14] proposes to apply CSFQ for network isolation in datacenters, but it does not have a hardware implementation for CSFQ and does not support hierarchical fair queueing. HCSFQ is to the best of our knowledge, the first solution to provide hierarchical fair queueing on commodity switches with small switch memory footprint and a single FIFO queue.

Programmable switches. Programmable switches have triggered many innovations in recent years [32,40–60]. Programmable packet scheduling is the most relevant to HCSFQ.

UPS [61] shows that Least Slack Time First (LSTF) provides a good approximation for many scheduling algorithms in practice. PIFO [10] provides a hardware design to realize the abstraction of a push-in first-out (PIFO) queue. It relies on a tree of PIFO queues to implement hierarchical fair queueing. AFQ [11] approximates fair queueing by using a few queues to emulate many queues. It stores per-flow counters in a count-min sketch, and does not support hierarchical fair queueing. SP-PIFO [12] uses several strict priority queues to emulate a PIFO queue, which can support fair queueing, not hierarchical fair queueing. Compared to them, we show how to leverage programmable switches to support fair queueing without per-flow state based on CSFQ, and present a new algorithm HCSFQ to support hierarchical fair queueing.

8 Conclusion

We present HCSFQ, a scalable algorithm for hierarchical fair queueing. Hierarchical fair queueing is a long standing problem in networking. Instead of relying on a hierarchy of queues with complex queue management, HCSFQ only keeps the state for the interior nodes and uses only one queue to achieve hierarchical fair queueing. This dramatically simplifies the design, and makes the design possible to be implemented in high-speed switches. Indeed, we have built a prototype for HCSFQ on programmable switches. Our prototype shows that HCSFQ works well with both UDP and TCP without any changes to either the hardware (e.g., NICs) or software (e.g., TCP/IP stack) of the end hosts.

To the best of our knowledge, HCSFQ is the first solution that has been demonstrated to provide hierarchical fair queueing on hardware switches at line rate. HCSFQ is not only theoretically interesting, but also has important practical implications. Network isolation is critical to multi-tenant clouds, which have a natural two-layer hierarchy. This hierarchy naturally requires the datacenter network to first allocate the bandwidth to the tenants, and then allocate each tenant’s bandwidth between the tenant’s flows. HCSFQ provides the first solution to enable this two-layer isolation in datacenter networks. Our prototype shows that this can be done without any changes to either the hardware (e.g., NICs) or software (e.g., TCP/IP stack) of the end hosts, and it works well with both UDP and TCP. We believe HCSFQ is a promising solution for network isolation in multi-tenant datacenters.

Acknowledgments

We sincerely thank our shepherd Dongsu Han and the anonymous reviewers for their valuable feedback on earlier versions of this paper. Xin Jin (xinjinpk@pku.edu.cn) is the corresponding author. This work is supported in part by NSF grants CCF-1652257, CNS-1813487 and CCF-1918757, and a Google Faculty Research Award.
References


A Proof of Theorem 1

Proof. The first conclusion is directly derived from the guarantee of CSFQ [13].

For the second conclusion, we consider a model with a parent and $k$ children. We add a script $i$ to represent the notations related to the parent, e.g., $r'_i$ is the estimated arrival rate of the $i$-th packets at the parent. We add a script $(j)$ to represent the notations related to the $j$-th child, e.g., $r^{(j)}_i$ is the estimated arrival rate of the $i$-th packets at the $j$-th child. Suppose the time episode is universal for all children. Suppose that $r'_0 = r'_0 = 0$ for $j = 1, \ldots, k$.

Suppose the inter-arrival time $T_i = \tau$ for all $i$. Suppose

$$r_{\alpha} \geq \frac{1}{1 - e^{-\tau/K}} \sum_{j=1}^{k} r^{(j)}_{\alpha}.$$  

Then we will show that the parent node $r_{\alpha}$ does not drop packets. To this end, we only need to prove that

$$r'_i = r_{\alpha}, \quad \forall i. \quad (9)$$

After the first drop, the package length is $h_i = h^{(1)}_i + \cdots + h^{(k)}_i$, where

$$h_i^{(j)} = \begin{cases} \ell_i^{(j)} - r^{(j)}_i, & r^{(j)}_i \leq r^{(j)}_{\alpha}, \\ \ell_i^{(j)} - r^{(j)}_i, & r^{(j)}_i > r^{(j)}_{\alpha} \end{cases}.$$ And by definition,

$$r'_i = (1 - e^{-T_i/K}) h_i + e^{-T_i/K} r'_{i-1}, \quad 1 \leq i \leq n.$$ We now recursively prove Eq. (9).

(i) First let $i = 1$.

We will use the following inequality to prove Eq. (9):

$$(1 - e^{-T_i/K}) \frac{h_i^{(j)}}{T_i} = \frac{r^{(j)}_{\alpha}}{r^{(j)}_i} \leq r^{(j)}_{\alpha}, \quad \forall j. \quad (10)$$

On the one hand, if Eq. (10) is true, we have

$$r'_1 = (1 - e^{-T_1/K}) \sum_{j=1}^{k} h_i^{(j)} \leq \sum_{j=1}^{k} r^{(j)}_i \leq r_{\alpha},$$

which implies Eq. (9) for $i = 1$.

On the other hand, recall $r^{(j)}_1 = (1 - e^{-T_1/K}) \frac{\ell_i^{(j)}}{T_i}$, we then prove Eq. (10) as following:

1. If $r^{(j)}_1 \leq r^{(j)}_{\alpha}$, then $h^{(j)}_1 = \ell^{(j)}_i \frac{r^{(j)}_{\alpha}}{r^{(j)}_1}$, thus

$$1 - e^{-T_1/K} \frac{h_1^{(j)}}{T_1} = (1 - e^{-T_1/K}) \frac{\ell_i^{(j)}}{T_i} \leq \frac{r^{(j)}_i}{r^{(j)}_{\alpha}} = r^{(j)}_{\alpha}.$$ Thus Eq. (10) holds.

2. If $r^{(j)}_1 \geq r^{(j)}_{\alpha}$, then $h^{(j)}_1 = \ell^{(j)}_i \frac{r^{(j)}_{\alpha}}{r^{(j)}_1}$, thus

$$1 - e^{-T_1/K} \frac{h_1^{(j)}}{T_1} = (1 - e^{-T_1/K}) \frac{\ell_i^{(j)}}{T_i} \frac{r^{(j)}_i}{r^{(j)}_{\alpha}} = r^{(j)}_{\alpha}.$$ Thus Eq. (10) holds. By (i) and (ii) and mathematical induction our proof is finished.