Scalable Real-Time Bandwidth Fairness in Switches

Robert MacDavid, Xiaoqi Chen, Jennifer Rexford
Princeton University, Princeton, NJ, USA. {macdavid,xiaoqic,jrex}@cs.princeton.edu

***Abstract*—Network operators want to enforce fair bandwidth sharing between users without solely relying on congestion control running on end-user devices. However, in edge networks (e.g., 5G), the number of user devices sharing a bottleneck link far exceeds the number of queues supported by today's switch hardware; even accurately tracking per-user sending rates may become too resource-intensive. Meanwhile, traditional software-based queuing on CPUs struggles to meet the high throughput and low latency demanded by 5G users.**

We propose Approximate Hierarchical Allocation of Bandwidth (AHAB), a per-user bandwidth limit enforcer that runs fully in the data plane of commodity switches. AHAB tracks each user's approximate traffic rate and compares it against a bandwidth limit, which is iteratively updated via a real-time feedback loop to achieve max-min fairness across users. Using a novel sketch data structure, AHAB avoids storing per-user state, and therefore scales to thousands of slices and millions of users. Furthermore, AHAB supports network slicing, where each slice has a guaranteed share of the bandwidth that can be scavenged by other slices when under-utilized. Evaluation shows AHAB can achieve fair bandwidth allocation within 3.1ms, 13x faster than prior data-plane hierarchical schedulers.

***Index Terms*—P4, SDN, Network Slicing, Fair Queuing.**

I. Introduction

Fair bandwidth allocation between users is an important goal for network operators, since a minority of users demanding too much bandwidth should not negatively affect other users' quality of service. Yet, leaving bandwidth allocation entirely to congestion control running on end hosts may lead to unfair allocation between different congestion control algorithms. Fair bandwidth allocation is, therefore, a necessary function of the core network. As modern networks scale to higher speed and more users, implementing per-user fair bandwidth allocation becomes increasingly more challenging.

Network slicing is a network feature that allows an operator to divide its network resources into many virtualized networks. Slicing enables operators to rapidly create new service offerings for different markets, while achieving performance isolation and quality-of-service guarantees between different slices. To support slicing, the network needs to implement both *intra-slice* fairness where different users within the same slice gets a fair share of the slice's bandwidth, as well as *inter-slice* fairness where each slice gets its share of bandwidth proportional to its specified weight. Meanwhile, the idle capacity from underutilized slices must also be fairly distributed to other over-subscribed slices.

One real-life example of a sliced network is the mobile access network. As IoT and 5G becomes prevalent, we face scalability challenges in implementing fairness. A base station may serve 100-1000 user devices, which belong to different classes of services (IoT, smartphones, mobile broadband, first responders, etc.) and have different usage patterns. Each slice (class of service) gets its guaranteed share of bandwidth; when a slice has few active users, its unused bandwidth can be distributed to users in other slices. Meanwhile, we want different users within the same slice to fairly share the limited physical-layer bandwidth: every user in the same slice should be allocated the same maximum bandwidth limit, which should be increased or decreased in real time based on both the number of active users in the slice and the total bandwidth allocated to the slice.

The slice-based fairness paradigm also exists in other scenarios. A data-center network operator may slice its network capacity into multiple classes of service (free tier, spot instances, enterprise customers, etc.) and allocate bandwidth fairly between different tenants within the same slice. Likewise, a network-layer DDoS mitigation mechanism might slice the network to serve different websites, and fairly allocate the bandwidth between all (potentially malicious) clients visiting a particular website.

In all of these example use cases, the number of users within each network slice (from thousands to millions) far exceeds the number of hardware queues available on today's networking hardware, which commonly supports 8-32 queues per port. In today's mobile network, client rate-limiting and scheduling are sometimes implemented as a virtual network function running on server CPUs [9]. Such a setup supports versatile scheduling policies, yet it requires many CPU cores to serve high-speed traffic and often adds latency and jitter to the traffic. Meanwhile, maintaining ultra-low latency for latency-sensitive applications is one of the most important features in 5G and next-generation 6G networks, which already achieves sub-10ms end-to-end latency [15, 18].

The emergence of high-speed programmable network devices had enabled implementing Active Queue Management (AQM) algorithms directly in the switch data plane [13, 17, 20, 21]. Although recent works [10, 12] have offloaded many mobile core network functionalities onto programmable switches, traffic scheduling is a notable exception. To the best of our knowledge, no existing work has attempted to offload scalable slice-based fair bandwidth allocation to high-speed programmable switches. Cebinae [19] enforces long-term fair bandwidth allocation but takes seconds to converge. HCSFQ [21] supports slice-based fair bandwidth allocation but requires per-user memory to monitor each user's sending rate; this not only adds control-plane overhead for adding and removing users, but also leads to scalability challenges given the limited amount of memory in the data plane.

There are two main challenges for running fair bandwidth allocation directly within the data plane of high-speed programmable switches. Firstly, the available memory is insufficient for maintaining per-user state. We therefore need to use approximate data structures, whose memory footprint scales sub-linearly with the number of users (as discussed in § IV). Secondly, we can only perform a limited set of arithmetic operations in the data plane. We use lookup tables to implement approximated multiplication and division, which is then used for calculating linear interpolation. This enables us to implement real-time, closed-loop iterative update for the per-user bandwidth limit (as discussed in § V). Finally, without using separate queues for each user, we enforce per-user bandwidth limits via probabilistic packet dropping, achieving *approximate* fair bandwidth allocation.

In this paper, we present Approximate Hierarchical Allocation of Bandwidth (AHAB), a hierarchical per-user bandwidth limit enforcer directly implemented in the data plane of programmable switch hardware. AHAB dynamically adjusts the per-user bandwidth limit for each slice in real time, calculated using max-min fairness with the bandwidth demand of all users across all slices. The novelty of AHAB can be summarized as follows:

- **Scalability:** By using a novel approximate data structure, AHAB avoids maintaining per-user state in data-plane memory, thereby supporting millions of simultaneous users.
- **Fast Convergence:** When user traffic changes, AHAB's interpolation-based iterative bandwidth limit update converges to fair bandwidth allocation within 3.1ms, 13x faster than prior work [21].
- **Precise Enforcement:** We use probabilistic dropping to precisely enforce bandwidth limits. This allows users to steadily send at the fair rate observing the bandwidth limit, without requiring hardware queues to pace packets as needed by prior work.
- **One-stop Bandwidth Allocation:** AHAB supports an arbitrary number of hierarchy levels. Therefore, a single instance of AHAB in the core network can rate-limit traffic correctly to adhere to all downstream bandwidth bottlenecks. This is highly useful when downstream devices do not support sophisticated scheduling policies (e.g., legacy routers or thin Wi-Fi access points), or when the network operator is unable to arbitrarily adjust device configurations, possibly because the core and downstream networks are managed between different administrative entities (e.g., MVNOs and wireless carriers).

The rest of this paper is structured as follows. §II defines the hierarchical fair bandwidth allocation problem. §III presents an overview of AHAB's division of labor between control and data plane. §IV discusses how AHAB overcomes the scalability challenge by avoiding per-user memory using a customized approximate data structure, while §V describes how AHAB approximately calculates an interpolation-based bandwidth limit update given the arithmetic constraints in the data plane. Evaluation in §VI demonstrate that AHAB converges to a fair bandwidth allocation quickly within 5 ms, achieving both fairness and throughput stability. We discuss related work in §VII and conclude in §VIII.

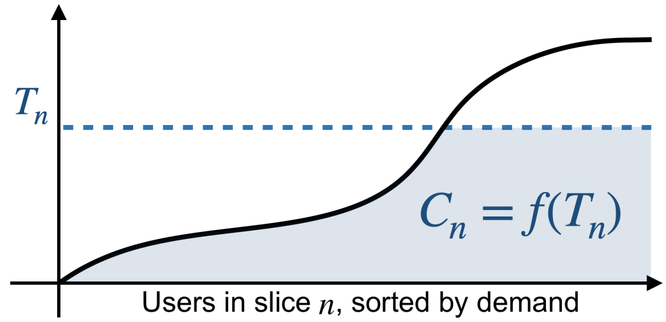

Fig. 1. We calculate and enforce per-user bandwidth limit T_n for all users in slice n, so that their total bandwidth consumed is equal to capacity C_n.

II. Hierarchical Fair Bandwidth Allocation

AHAB needs to allocate a network slice's available bandwidth fairly between all users in different slices based on max-min fairness. Here we reuse the same problem definition as in earlier works [21]. For simplicity of discussion, for now we assume all users and slices in our system have equal weight, although it is trivial to add weights and allocate bandwidth proportionally. Let us denote slice n's set of users as $child(n)$ (its "children" in the scheduling hierarchy). We also define each user m's bandwidth demand as R_m, sorted and plotted in Figure 1, and the total demand $\sum_{m \in child(n)} R_m$. When total demand exceeds the slice's capacity C_n, we can calculate a per-user bandwidth limit according to max-min fairness:

$$T_n = \arg\max_T \sum_{m \in child(n)} \min(T, R_m) \le C_n. \quad (1)$$

If we plot the limit T_n as a horizontal line in Figure 1, the shaded area under the intersection of T_n and the demand curve has area equal to the capacity C_n. Finding the right bandwidth limit T_n under max-min fairness is equivalent to finding the right "horizontal cut" across the demand curve.

Meanwhile, some slices may have idle capacity after demand from all users are satisfied. AHAB needs to re-allocate these unused bandwidth to other over-subscribed slices and adjust their capacity C_n upwards, similarly according to max-min fairness. This builds a two-level scheduling hierarchy. Although two scheduling levels are sufficient for many use cases (slices/users in mobile networks, tenants/VMs in data center networks, etc.), we can also define three or four levels. In the interest of space, we omit detailed examples and how to solve for the fair allocation; we refer interested readers to [3, 7, 14, 21] for a more comprehensive introduction.

III. AHAB System Overview

Figure 2 illustrates the basic design of AHAB. At a high level, we split the bandwidth allocation process into a fast-reacting data plane component and a more sophisticated control-plane component for hierarchical updates.

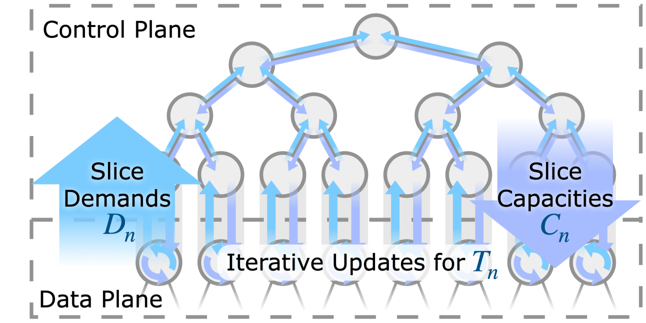

Fig. 2. The control plane maintains inter-slice fairness by periodically reading each slice bandwidth demands D_n and writing fair capacities C_n; the data plane keeps intra-slice fairness by iteratively updating bandwidth limits T_n.

Data Plane: Intra-slice Fairness. To quickly react to changes in individual user's traffic, AHAB calculates iterative updates for the per-user bandwidth limit T_n fully within the data plane, using approximated linear regression. This allows the intra-slice bandwidth allocation to converge within milliseconds after a user starts or stops sending, much faster than updating using the switch control plane.

Since it is impossible to perfectly predict the constantly changing per-user traffic demand, AHAB splits the traffic into very small time epochs (on the order of milliseconds) and uses the demand distribution in the past epoch as a prediction for the next epoch. At the end of each epoch, we use this demand distribution (as illustrated in Figure 1) to iteratively refine the per-user bandwidth limit T_n, such that the total bandwidth used by all users in the slice will be equal to the capacity C_n.

Control Plane: Inter-slice Fairness. The control plane periodically reads the per-slice total bandwidth demand D_n calculated by the data plane and writes the updated per-slice fair allocation capacity C_n to the data plane, on the order of once every 10-20ms. Surplus bandwidth from underutilized slices is reallocated to other slices using max-min fairness (§II).

Thanks to statistical multiplexing, the aggregated bandwidth demand of different slices changes on a longer timescale. Therefore, the slightly slower update of capacities has little impact on maintaining intra-slice fairness. Note that the allocated capacities only changes when some slices are underutilized.

IV. Scaling Beyond Memory Limits

In this section, we discuss how AHAB overcomes the scalability challenge imposed by hardware memory size limits. We first discuss how the bandwidth limit T_n is enforced on each user using their estimated sending rates. Subsequently, we show how AHAB avoids allocating per-user memory, using a novel approximate data structure that combines Count-Min Sketch with Low-Pass Filters to estimate per-user sending rates. Finally, we discuss how we share one approximate data structure across all slices using weight-based normalization.

A. Enforcing Bandwidth Limits

For the entire scheduling hierarchy to achieve bandwidth fairness, we must properly enforce bandwidth limit T_n on all users. Naively, we can allocate one queue per user and assign the bandwidth limit as the queue's drain rate. However, the number of users (thousands to millions) far exceeds the number of queues available in hardware switches (8-32 queues per port). Instead, we can enforce bandwidth limits using active queue management, or more specifically probabilistic dropping as discussed in [11, 16], as long as we know the user's sending rate. This approach does not require a traffic scheduler, and can be performed even if the switch has only a single queue.

For a user m in slice n with bandwidth limit T_n and sending rate R_m, we can enforce the bandwidth limit T_n by dropping its packets with probability $1 - \min\left(1, \frac{T_n}{R_m}\right)$ as described in [11, 16]. If a user uses less than the limit T_n, no packet will be dropped; otherwise, after probabilistic dropping the user's remaining packets will use bandwidth equal to T_n.

We also observe that TCP flows react poorly to traffic policing using probabilistic dropping, and we instead adapt the ECN-shaping technique proposed by Nimble [17] alongside approximate dropping to enforce bandwidth limits for TCP.

B. Avoiding Per-user Memory

Knowing a user's sending rate R_m is vital for correctly enforcing the bandwidth limit. As discussed in [16, 21], asking the sender of all traffic to attach their traffic rate to each packet is an easy yet unrealistic solution, as the sender might belong to a different administrative entity and may not honestly report the rate. Therefore, AHAB needs to measure each user's sending rate directly.

Recent works [17, 21] in queue scheduling within high-speed programmable switches rely on using the onboard memory to maintain per-user sending rate statistics, by allocating one traffic counter per user. However, programmable switches only have a limited amount of onboard memory in the data plane, limiting its scalability. At any given time, a core network switch may be servicing millions of users across thousands of base stations, making it infeasible to store any per-user state in memory, not to mention the hassle of keeping the memory allocation up-to-date when users constantly join or leave the network.

Instead, we build a customized memory-efficient approximate data structure to track per-user sending rate, by combining two techniques: Low-Pass Filters (LPF) and Count-Min Sketch (CMS) [6]:

- The LPF is a self-decaying counter available in the switch hardware. If we add value x at time t to a LPF with previous value v_0 and last update time t_0, its new value becomes $v = x + v_0 e^{-(t-t_0)/\tau}$ where τ is its decay time constant. As discussed in [16], if we aggregate the packet sizes of a single user's traffic in a LPF, the LPF will report an exponentially-decayed moving sum of recent packet sizes, which is proportional to a good estimate of the user's instantaneous sending rate.
- The CMS [6] is an approximate data structure that answers frequency queries, using r rows of hash-indexed arrays each having c counters. Given an "insertion" with a size

and a user ID, we find one location per row by applying r different random hash functions over its ID, and increment the counters at those location by the size; when querying the total size of a particular ID, we find the same r locations and report the minimum of the r counters.

When used to estimate size of flows in traffic, CMS is good at reporting heavy flows, as it never underestimates flow sizes. However, a vanilla CMS can only track the total number of bytes sent by a user since the CMS is initialized, not the user's instantaneous sending rate. Although it is possible to run multiple instances of CMS in a round-robin fashion to query moving-window flow rates [5], such an arrangement adds complexity and requires 2x-4x more memory.

AHAB combines CMS with LPF by replacing individual counters in the CMS structure with LPF counters. When inserting a packet with its size and user ID, we apply the r hash functions over the user ID to locate one LPF counter per row, and add the packet size to these r LPF counters. When querying the instantaneous sending rate of the same user ID, we read the LPF counters in the same r locations, and use the minimum across their reported rate as the estimated sending rate of this user. This allows us to estimate per-user sending rate without the need to allocate per-user memory.

Note that CMS is a linear transformation in the ID dimension while LPF is also a linear transformation in the time dimension. Since they are commutative, CMS-LPF retains the additive-error guarantee from CMS:

Theorem 1. Let R_i be user i's sending rate reported by an ideal LPF counter, and $\sum_m R_m$ be the total sending rate across all users, again reported by ideal per-user LPF counters. When querying a CMS-LPF estimator of size $r \times c$, the estimated sending rate $\tilde{R}_i$ satisfies $\tilde{R}_i \geq R_i$ and $\Pr\left[\tilde{R}_i \leq R_i + \epsilon \sum_m R_m\right] \leq \delta$, with $\epsilon = e/r$ and $\delta = e^{-c}$.

In the interest of space we omit the full proof, which derives naturally from the proof of original CMS properties.

C. Sharing One Rate Estimator Across Slices

Naively, AHAB would allocate one CMS-LPF estimator for each slice. However, due to the natural skewness of traffic, not all slices will have lots of "heavy" users sending at high rates. Some slices may be underutilized and have no heavy user at all, and the memory dedicated for their estimators is wasted. Instead, we share a single CMS-LPF estimator across all slices. We can then exploit statistical multiplexing, as the heavy users and busy slices are now effectively using the unused memory sacrificed by the underutilized slices with no heavy user.

However, we note that CMS provides an additive error guarantee, meaning that the error of each user's estimated rate is of similar magnitude regardless of the true sending rate of the user. This is not a problem for intra-slice comparison, as we only care about enforcing bandwidth limits for heavy users and can safely ignore the underutilized users. Yet, different slices may have vastly different bandwidth allocations. If two slices of capacity 100Mbps and 10Gbps naively share the same CMS-LPF structure, the 10Gbps slice will dominate; "small" users of 200Mbps in the heavy slice will overwhelm the CMS while the "heavy" users of 30Mbps in the small slice become a rounding error.

To ensure the estimation error is scaled proportionally with the bandwidth of different slices, we perform *pre-update normalization*: we scale packet sizes inversely proportional to the weight of their parent slice before feeding them into CMS-LPF. Heavy users in smaller slices can now be accurately tracked as they are scaled up. Subsequently, estimated sending rates are also compared to scaled versions of bandwidth limit.

V. Approximate Arithmetic in the Data Plane

To achieve line-rate packet processing and low forwarding latency, high-speed programmable switches like Intel Tofino support a limited set of arithmetic operations, and we can only perform a constant number of computational steps per packet. Thus, it is infeasible to exactly track the bandwidth demands, precisely calculate the fair allocation, or accurately compute per-user drop probabilities.

Figure 3 shows an overview of the AHAB data plane, where we use *approximate* arithmetic heavily to implement probabilistic dropping and interpolation-based iterative update to the bandwidth limit. We also note that the approximate arithmetic techniques presented here are widely applicable to other applications running in programmable switches, beyond fair bandwidth allocation.

A. Approximated Probabilistic Dropping

For a given user, we obtain its estimated sending rate R_m from the CMS-LPF estimator and compare it against the per-user bandwidth limit T_n of its slice. For non-TCP traffic, we need to enforce the bandwidth limit by dropping the packet with probability $1 - \frac{T_n}{R_m}$.

Since we cannot calculate exact division in the data plane, we perform approximate arithmetic using a TCAM lookup table, similar to the approximate multiplication technique used in Nimble [17]. Here we truncate T_n and R_m's binary form to retain only the few most significant bits as i and j such that $T_n \approx i \times 2^k$ and $R_m = j \times 2^k$, and use (i, j) as index in the lookup table. To reduce the error bias of lookup table entries, we store $\frac{i+0.5}{j+0.5}$ instead of $\frac{i}{j}$ in the lookup table.

Subsequently, we can simply sample a number rnd uniformly at random between $[0, 1]$ using the random number generator, and compare it against $\frac{T_n}{R_m} \approx \frac{i+0.5}{j+0.5}$. If rnd is greater than $\frac{T_n}{R_m}$, we drop the packet.

B. Tracking the Bandwidth Demand

For slice n, it is infeasible for switches to track its entire bandwidth demand curve (shown in Figure 1) representing all user's sending rates, which we can neither store nor sort. However, we can track the actual bandwidth used by all users $f(T_n)$, which is a function of the currently-enforced bandwidth limit T_n and represented by the shaded area under the demand curve intersected with T_n. To get $f(T_n)$, we simply need to use a LPF to track the size of all packets that are not dropped.

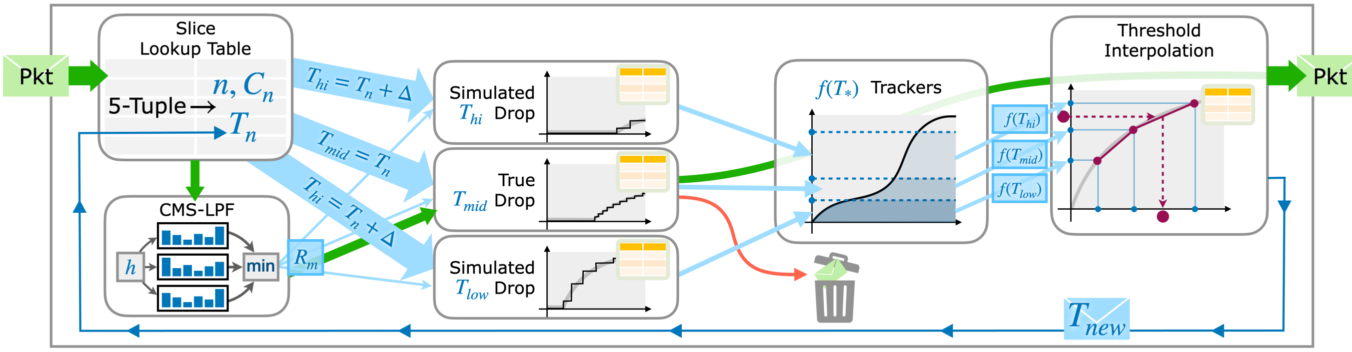

Fig. 3. When a packet arrives, AHAB first maps it to a slice n and estimates its user's sending rate R_m using the CMS-LPF estimator (§IV-B), then uses probabilistic dropping to enforce slice n's bandwidth limit T_n (§V-A). AHAB also maintains two bandwidth limit candidates T_{low}, T_{hi} and tracks the hypothetical total bandwidth usage $f(\cdot)$ (§V-B), which is used to derive a more accurate bandwidth limit T_{new} via approximated linear interpolation (§V-C).

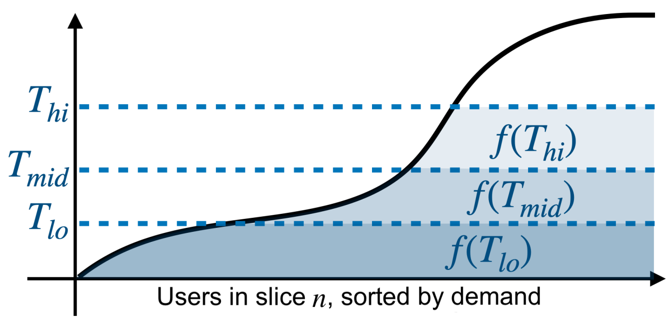

Fig. 4. Relationship between bandwidth limit candidates T_{low}, T_{mid}, T_{hi} and total bandwidth consumption $f(T_{low})$, $f(T_{mid})$, $f(T_{hi})$.

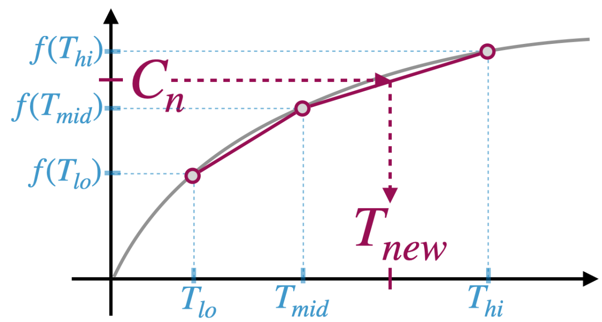

Fig. 5. Finding new bandwidth limit using interpolation. When we plot $f(T_*)$, x-axis in this figure refers to the bandwidth limit T_* (y-axis in Fig. 4), while y-axis in this figure refers to the area under line T_* in Fig. 4.

Still, comparing $f(T_n)$ with C_n only tells us whether we are over- or under-utilizing the capacity C_n, i.e., whether we should increase or decrease T_n. This does not say much about what is the ideal limit or how much should we change T_n.

To better analyze how to update T_n, we further specify two *candidate* bandwidth limits, a lower candidate $T_{low} = T_n - \Delta$ and a higher candidate $T_{hi} = T_n + \Delta$, where Δ is the maximum step-size we want to change T_n. For example, we may use $T_{low} \approx 0.5T_n$ and $T_{hi} \approx 1.5T_n$. From now on, we also refer to $T_{mid} = 1.0T_n$ as the middle candidate.

We now track two more hypothetical total transmitted bandwidth $f(T_{low})$ and $f(T_{hi})$, by generating two hypothetical probabilistic dropping decisions in addition to the real dropping decision. Using the same lookup table technique discussed in §V-A, we approximately calculate $\frac{T_{hi}}{R_m}$ and $\frac{T_{lo}}{R_m}$ and track the packets that are hypothetically not dropped under T_{low} or T_{hi} respectively. As illustrated in Figure 4, $f(T_{low})$, $f(T_{mid})$, and $f(T_{hi})$ are the shaded area under the demand curve intersecting with different horizontal lines.

Figure 5 plots the monotonically-increasing function f. The optimal bandwidth limit $\tilde{T}$ satisfies $f(\tilde{T}) = C_n$, thus we need to calculate a new limit T_{new} that is as close to $\tilde{T}$ as possible.

C. Update Bandwidth Limit via Interpolation

A naive policy for updating the bandwidth limit is simply comparing $f(T_{mid})$ with C_n: if $f(T_{mid}) > C_n$, i.e., the slice is sending too much traffic, we choose T_{low} as the new limit, otherwise we choose T_{hi}. This policy works well enough for very small step size Δ (e.g., 1%-5% of T_{mid}), however, such small steps converge too slowly; conversely, when using a larger Δ the limit never converges.

Instead, given the three candidate points on the f curve, we can produce a much more accurate estimate of the optimal bandwidth limit using linear interpolation. Let us first assume the target lies between the lower and higher candidate points, i.e., $f(T_{low}) < C_n < f(T_{hi})$. Without loss of generality, assume we need to adjust to a higher limit, i.e., $f(T_{mid}) < C_n < f(T_{hi})$. We calculate the new bandwidth limit as

$$T_{new} = T_{mid} + \frac{C_n - f(T_{mid})}{f(T_{hi}) - f(T_{mid})} \times (T_{hi} - T_{mid}). \quad (2)$$

As illustrated in Figure 5, the interpolated estimate T_{new} is a much better estimate of the ideal bandwidth limit, compared to naively choosing T_{hi}.

In Figure 6 we illustrate the process using an example. To calculate $\frac{C_n - f(T_{mid})}{f(T_{hi}) - f(T_{mid})} = \frac{1012}{1690}$, we first use the same approximate division lookup table technique discussed earlier in §V-A, except the division results are now stored as a (mantissa, exponent) pair. We first truncate the numerator and denominator to get most significant non-zero bits i =`01111`/j =`11010`, and retrieve the approximate division result $\frac{i+0.5}{j+0.5} = \frac{2396}{2^{12}}$.

After the approximate division, we need to multiply the result by Δ. To make this calculation easier, we choose Δ to be a power of 2, reducing the multiplication into a bit-

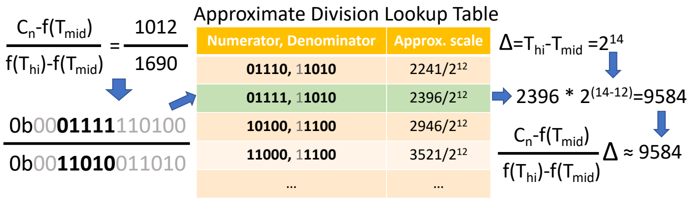

Fig. 6. We use a lookup table to implement approximate linear interpolation. We first match on the highest binary bits of numerator and denominator to get a scaled division result, then multiply Δ via bit shifting for the final result.

shift. In practice, we set $\Delta = 2^{\lfloor \log_2 (\frac{1}{2} T_{mid}) \rfloor}$, meaning $T_{low} = T_{mid} - \Delta \approx 0.5 T_{mid}$ and $T_{hi} = T_{mid} + \Delta \approx 1.5 T_{mid}$.

The divide-then-multiply calculation can be applied as a single bit shift. In the example in Figure 6, we have $\Delta = 2^{14}$ and need to calculate $\frac{2396}{2^{12}} \cdot 2^{14}$, which can be simplified into a left shift: $2396 << (14 - 12) = 9584$. Finally, we finish the last addition operation in the approximate linear interpolation, and obtain the new bandwidth limit $T_{new} = T_{mid} + 9584$.

Similarly, when adjusting towards a lower limit, we use

$$T_{new} = T_{mid} - \frac{f(T_{mid}) - C_n}{f(T_{mid}) - f(T_{low})} \times (T_{mid} - T_{low}). \quad (3)$$

Notice that we use subtraction from T_{mid} instead of adding up from T_{lo} to interpolate. This is because the approximate division has a constant relative error proportional to the result. By subtracting the result from T_{mid}, we can make more accurate fine-grained adjustments near T_{mid} to better converge towards the optimal bandwidth limit. Instead, if we use

$$T_{new} = T_{low} + \frac{C_n - f(T_{low})}{f(T_{mid}) - f(T_{low})} \times (T_{mid} - T_{low}), \quad (4)$$

the approximated interpolation is more accurate near T_{low} and has a larger error near T_{mid}.

When C_n falls out of the range $[f(T_{low}), f(T_{hi})]$, our estimate candidates are too far off from the ideal bandwidth limit, and we clip the update by choosing $f(T_{low})$ or $f(T_{hi})$ directly. Clipping prevents overshooting caused by using linear interpolation outside of the two candidate points.

We further note that although CMS-LPF will introduce over-estimation errors across the board for all estimated rates, our closed-loop bandwidth limit update process will naturally adapt to this error. When all rates are slightly over-estimated while the bandwidth limit T_n is not yet over-estimated, users will suffer from an unnecessarily high drop probability, leading to less than C_n total traffic; AHAB will then automatically raise T_n to account for such global over-estimation.

D. Iterative Update Using Worker Packets

To achieve fast convergence towards intra-slice fairness, AHAB updates the bandwidth limit T_n fully within data plane. At the end of every epoch, AHAB calculates a new bandwidth limit T_{new} for each slice using approximate interpolation, and use it as the new bandwidth limit for the next epoch.

However, T_n is stored in a register memory lookup table near the beginning of the switch's packet-processing pipeline while the new limit T_{new} is only available in later pipeline stages; the pipeline's memory access constraint does not allow us to write T_{new} back to the same register memory directly. Therefore, at the end of every epoch, we generate one worker packet per slice by packet cloning, and use packet recirculation to let the worker packet go through the pipeline a second time, carrying and writing the T_{new} value.

Although the update is slightly delayed due to packet recirculation (about $0.65\mu s$), only a very small fraction of packets near the beginning of the epoch are affected, therefore the actual difference in enforcement due to the delayed update is negligible. As we show in §VI, this closed-loop update process rapidly converges to the fair bandwidth allocation.

E. Supporting Weighted Allocation

A network operator sometimes needs to allocate bandwidth in proportion to a pre-assigned weight, for example when implementing differentiated services. AHAB supports weighted fair allocation at both the slice level and the user level.

To support weighted fair bandwidth allocation between users in the same slice, we scale each packet's length using the user's weight: if a user m has weight w_m, a packet with size x is scaled into $\frac{x}{w_m}$ before being used to calculate the user's scaled sending rate R_m in the CMS-LPF estimator. This way, we can directly compare different user sending rates R_m against the same per-user bandwidth limit T_n.

Meanwhile, the control plane is more flexible and trivially supports allocating bandwidth to different slices based on their weight. We simply divide each slice's demand and capacity by its weight before computing the max-min fairness allocation.

We also note that the weights assigned to slices / users can be easily updated at run time. To adjust the weight for a subset of users, we adjust the rules installed in the slice lookup table in the data plane; to adjust the weight of a slice, we modify it directly from the control plane.

VI. Evaluation

Using a prototype implementation running in a hardware testbed, we show that AHAB can quickly achieve fair and stable bandwidth allocation between users. Compared to the prior state-of-the-art, HCSFQ [21], AHAB not only converges to the target fair bandwidth allocation faster (in 3.1ms), but also achieves comparable or better fairness and throughput stability. Subsequently, we use real-world traffic traces in a simulation-based experiment to show that AHAB scales well to 5.9-23.9 million users with a reasonable memory footprint, and CMS-LPF has a minimal impact on scheduling fairness.

A. Testbed Experiment Setup

We evaluate AHAB's real-world scheduling fairness using a hardware testbed with two sender and receiver servers connected via an Intel Tofino Wedge32-X programmable switch, which runs a prototype implementation of AHAB written in 2,000 lines of P4 [1]. Both servers have a 20-core CPU and a Mellanox ConnectX-5 2x100Gbps NIC, and run Ubuntu 20.04. The sender sends TCP flows using `iperf3` with

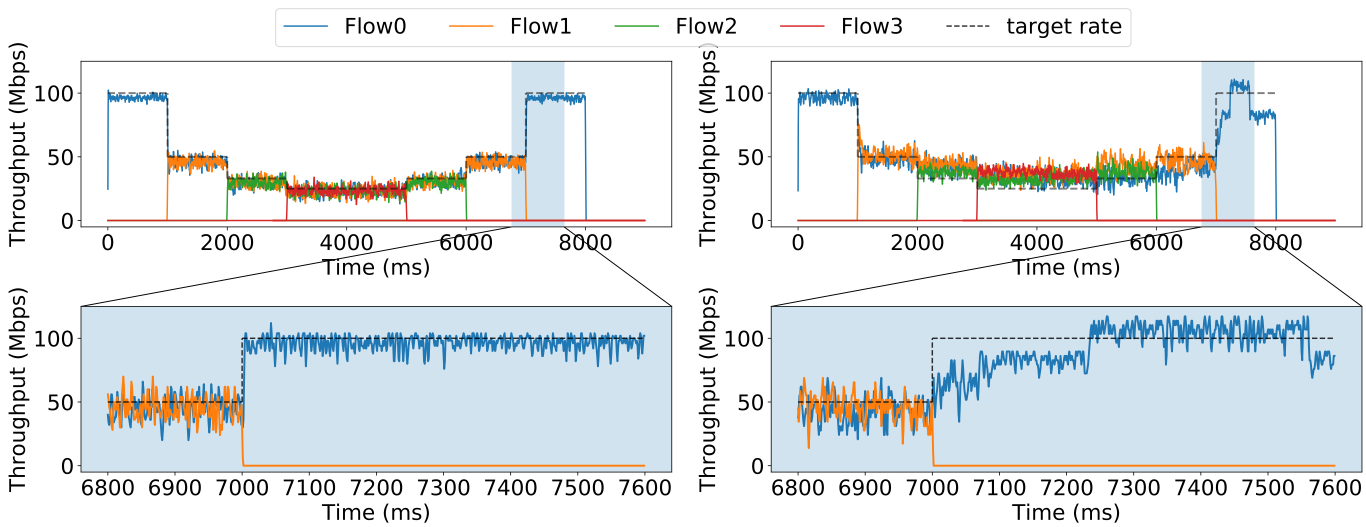

Fig. 7. AHAB (left) converges to fair bandwidth allocation within 3.1ms on average, while HCSFQ [21] (right) needs 42.3ms.

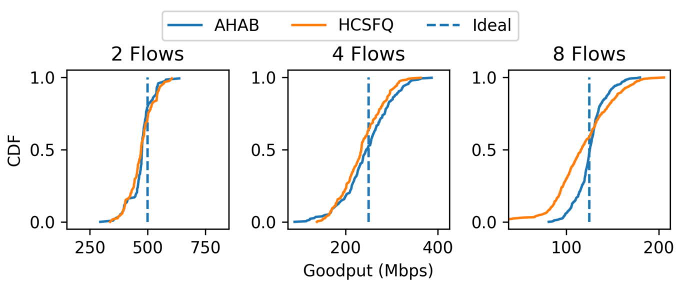

Fig. 8. Cumulative distribution of competing TCP flows's goodput when using AHAB versus HCSFQ.

Linux's default congestion control (cubic), and send UDP flows using either `iperf3` or a customized Go script that performs millisecond-level throughput measurement. We set the iterative update epoch time to 1ms and configure the LPF rate estimator's time constant to τ=4ms. Unless otherwise noted, we use a CMS-LPF estimator with size 3x2048.

In all experiments, we treat each flow as a unique user, using its 5-tuple (source and destination IP/port pairs) as the user ID. We note that real-world traffic may be grouped more coarsely; for example, one user in a mobile network may include all flows destined for the same device (same destination IP).

B. Fast Convergence

We now compare AHAB to the state-of-the-art of hierarchical fair queuing based on programmable switch: HCSFQ [21]. HCSFQ iteratively converges to the fair rate via Additive Increase Multiplicative Decrease, limiting its convergence speed when the number of users decreases and the fair rate increases.

To demonstrate the difference in convergence time, we program both AHAB and HCSFQ to enforce fairness between four UDP flows in a single slice, with a fixed 100Mbps capacity. All four flows have the same 100Mbps constant sending rate, but have different starting and ending time: they run between T=0-8s, T=1-7s, T=2-6s, and T=3-5s, respectively.

Figure 7 shows the actual bandwidth used by the four flows over time, after bandwidth limit enforcement done by AHAB (left) or HCSFQ (right). At a longer timescale (top), the two schedulers behaved similarly. However, if we zoom in to a smaller timescale and plot the millisecond-level per-flow throughput (bottom) immediately after T=7s (where flow 1 stopped), we can see AHAB converges much faster than HCSFQ to allow flow 0 to use the full 100Mbps bandwidth. We measured the time for flow throughput to converge to within 10% of ideal fair bandwidth limit; AHAB's interpolation-based update only needs around three iterations to converge, taking only 3.1ms on average (at most 5ms), more than 13x faster than HCSFQ (average 42.3ms, at most 234ms).

C. Fairness and Goodput Stability

We first demonstrate that AHAB can effectively enforce fair bandwidth allocation for TCP flows, by simultaneously running 2, 4, or 8 flows sharing a slice with fixed 1Gbps capacity. In Figure 8 we plot the cumulative distribution function (CDF) of the TCP goodput of all flows, reported by `iperf3` in 1-second intervals across 60 seconds. Ideally, all flows exhibit the same goodput across time, leading to a steeper CDF. The stability achieved by AHAB is comparable to that of HCSFQ: on average, the goodputs of flows enforced by AHAB are within 12.1% of ideal fair share, while those enforced by HCSFQ exhibits 15.5%.

Meanwhile, we also show approximate probabilistic dropping can effectively achieve fair bandwidth allocation for non-TCP traffic, even with very different sending rates. We let multiple UDP flows share the same slice with fixed 100Mbps capacity, and configured their sending rate to be 10Mbps, 20Mbps, 30Mbps, and so on. In Figure 9, we show the throughput achieved by these flows, when 4, 8, and 16 flows are sent simultaneously. In the latter two cases, the slice is over-utilized and approximate probabilistic dropping kicks in.

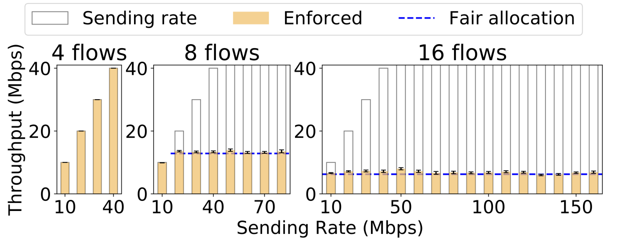

Fig. 9. Given flows with various sending rates, AHAB's approximate probabilistic dropping achieved fair bandwidth allocation within 6% error.

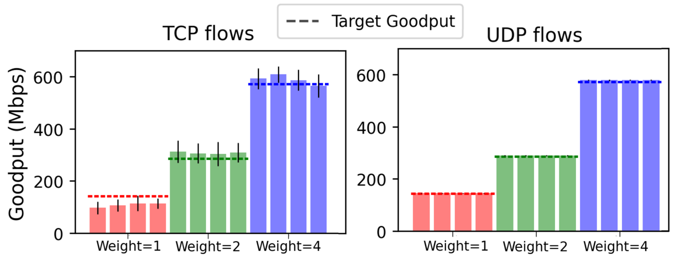

Fig. 10. Goodput of weighted TCP and UDP flows sharing one 1Gbps slice.

Although AHAB needs to apply vastly different dropping probabilities for the wide range of sending rates, the resulting allocation is quite fair. On average, the mean throughput achieved is within 4% and 6% of the fair bandwidth allocation target, for 8 and 16 flows respectively. This corresponds to the error of the approximated division using the lookup table.

Figure 10 demonstrates AHAB's support of weighted fairness. We start three groups of flows with weight 1x, 2x, and 4x respectively, with four flows per group, all sharing one slice with 1Gbps capacity. Flows with the same weight achieve the same throughput, proportional to their allocated weight, and their attained throughput averages within 15% and 1% of the weighted fair allocation for TCP and UDP, respectively.

D. Inter-slice Fairness

In Figure 11 we demonstrate that AHAB rapidly adjusts to the changing bandwidth demands of different slices. We set up an experiment where Slice 1 always has x users (TCP flows), and Slice 2 is initially idle with no user. At T=10s x users in Slice 2 that starts sending, lasting until T=50s. At T=20s another x users join Slice 2 and start sending until T=40s. In the ideal case, all bandwidth is fully allocated to Slice 1 between T=0-10s as well as T=50-60s, fairly shared between x users; the total bandwidth is split in half between Slice 1 and Slice 2 during T=10-50s. When more users are added to Slice 2 during T=20-40s, users in Slice 2 each get a lower share while users in Slice 1 are not affected.

Figure 11 shows three scenarios: the total bandwidth shared by the two slices are 100, 1000, and 4000 Mbps, respectively, and we also have x=2, 20, and 80 users proportionally. We plot and compare the average goodput attained by the users in each slice, which is also a good indicator of fairness between slices. The bandwidth allocation between slices always quickly converged to fairness (within a few RTTs). When users send UDP traffic instead, AHAB instantly achieves near-perfect fair allocation for all three cases; the result is omitted here.

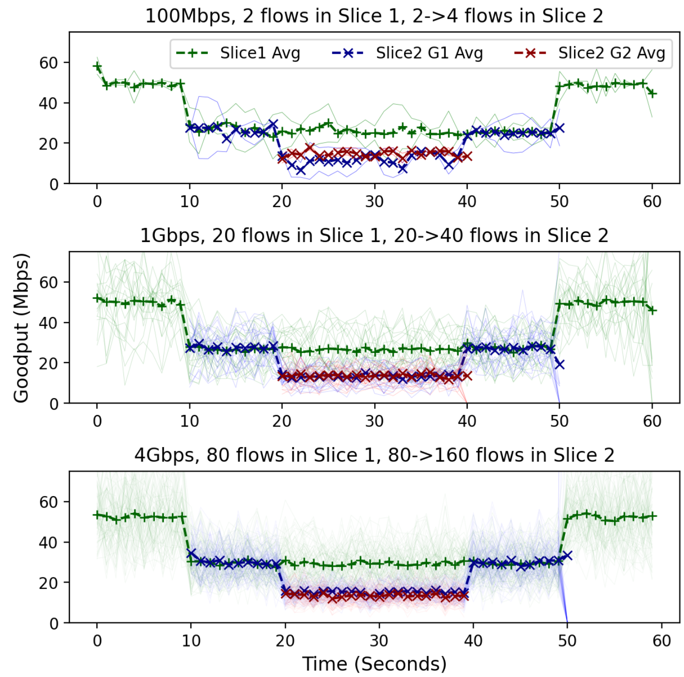

Fig. 11. Three experiments showing two slices sharing a common bottleneck. Slice 1 uses half the bandwidth even when Slice 2 has twice as many flows.

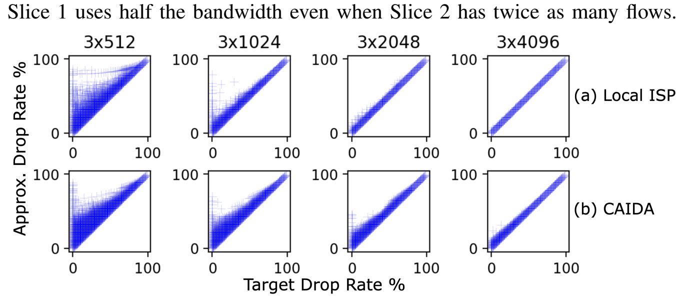

Fig. 12. 3x4096 CMS-LPF estimator is sufficient for serving millions of users, with negligible error in drop probability.

E. Scalability

To evaluate AHAB's performance at scale, we run trace-based simulation experiments to understand how much memory is needed to support a large number of users.

We collected a 15-minute anonymized traffic trace from the core network of a local Internet Service Provider and played the trace through a Python-based simulator. We treat each of the 5,980,000 unique source-destination IP pairs as an user, and let all users share a single slice with capacity C_n set to 0.84Gbps, equal to the average throughput of the trace. Due to natural fluctuations in traffic rate, the instantaneous bandwidth demand often exceeds C_n. The simulator calculates the fair per-user bandwidth limit T_n for each epoch, and then calculates the "target" probabilistic drop rate $1 - \min(\frac{T_n}{R_m})$ using the ground-truth per-user sending rate R_m.

Meanwhile, we also simulate the per-user estimated sending rate $\widetilde{R_m}$ reported by CMS-LPF estimators of different sizes, and use $\widetilde{R_m}$ to calculate the "approximated" drop probability $1 - \min(\frac{T_n}{\widetilde{R_m}})$. Shrinking the size of CMS-LPF estimator reduces the accuracy of rate estimation, which in turns leads to more error in the drop probability.

CMS-LPF Dimension	Hardware Memory Utilization	Supported # of Users
2048x3 (24KB)	1.56%	2,990,000
4096x3 (48KB)	2.60%	5,980,000
16384x3 (768KB)	8.85%	23,920,000 (est.)

TABLE I

MEMORY UTILIZATION AND SUPPORTED NUMBER OF USERS W.R.T. DIFFERENT SIZES OF THE CMS-LPF ESTIMATOR.

Resource	Instr. Words	Hash Units	TCAM
Utilization	28.6%	37.5%	5.2%

TABLE II

UTILIZATION OF OTHER SWITCH HARDWARE RESOURCES.

As shown by Figure 12(a), using a CMS-LPF estimator with size 3x512 led to a fraction of packets with drop rate higher than the target; although most errors lie in over-utilized users, some users with a target drop rate of 0% (under-utilized users) also experience significant packet drops. Meanwhile, CMS-LPF size 3x4096 is sufficient to reduce errors to negligible level. We conclude that a CMS-LPF estimator with size 3x4096 is sufficient for AHAB to accurately produce per-user rate estimate for the 5,980,000 unique users in our trace.

We also run the same simulation using five minutes of CAIDA Anonymized Internet Trace 2018 [4]. The trace has an average throughout of 3.5Gbps and has 7,300,000 unique flow 5-tuples. We obtain similar results, as shown in Figure 12(b).

Now we analyze the switch hardware resources used by AHAB, and specifically focus on the memory used by the CMS-LPF estimator. As shown in Table I, a small 2048x3 CMS-LPF estimator only costs a small fraction (1.56%) of all stateful memory available on the switch hardware, yet it already supports accurately enforcing bandwidth limit for 3 million users. We can fit a much larger sketch than what we used in the prototype: allocating a 16384x3 CMS-LPF estimator costs 8.85% of the available memory. Assuming similar traffic skewness as in our ISP trace, a single programmable switch can support 23.9 million devices across all slices, which is sufficient for many application scenarios. We also report other resource utilization in Table II.

As for the number of slices, our prototype program supports up to 16,000 slices. The Tofino switch supports 3.2Tbps aggregated throughput, which can be shared among 2,000 downstream base stations. It is possible to expand further by adding more entries to the slice lookup table and allocating more per-slice bandwidth demand trackers, as they only occupy a small fraction of the total data-plane memory usage (with the majority being the CMS-LPF rate estimator). The primary limiting factor on the number of slices is control-plane speed, as supporting N slices requires the control plane to read N demands and write N capacities per update.

VII. RELATED WORK

Fair Queuing using Estimated Rate: Core Stateless Fair Queuing [16] is a network architecture where edge nodes estimate the rate of incoming flows and attach the rate to packets, while core nodes in the network choose a fair per-flow rate and enforce it using probabilistic dropping. It requires maintaining per-flow state to estimate sending rates. Approximate Fairness through Differential Dropping [11] uses a shadow buffer that holds recent packets to approximately derive per-flow rates, and similarly performs probabilistic dropping. It is not straightforward to implement a large shadow buffer given the computational constraints present in today's high-speed programmable switches. Also, both works require one dedicated hardware queue per "slice" (group of flows).

Rank-based Scheduling: In Push-In, First-Out (PIFO) queues, each packet is pushed in with a certain rank, and packets with the highest rank are transmitted first. Admit-In, First-Out [20] and SP-PIFO [2] both approximate the behavior of a PIFO queue on commodity programmable switches. AIFO uses a sample of recently admitted packets to estimate the rank distribution of packets in the queue, which is used to decide a threshold and reject low-ranked packets from being admitted. Meanwhile, SP-PIFO uses an array of strict-priority queues and dynamically adjust the mapping from ranks to queues using estimated quantile distribution of ranks. These works both assume an oracle which assigns ranks to packets.

Fair Queuing in the Data Plane: Approximate Fair Queuing [13] implements scalable per-flow fair queuing by splitting traffic into calendar epochs. This design requires rapidly rotating the priority between multiple queues to serve different future epochs, and does not support a multi-layer scheduling hierarchy. Gearbox [8] proposed a new hardware design specifically supporting multi-level calendar queuing. Meanwhile, Hierarchical Core-Stateless Fair Queuing [21] extends CSFQ [16] and uses Addictive Increase, Multiplicative Decrease (AIMD) to iteratively find a fair per-user (per-tenant) sending rate limit using queue congestion status feedback. Although it can support multiple layers of scheduling hierarchy, its dependency on per-user memory for estimating per-user sending rates hurts scalability. The AIMD process also takes a relatively long time to adapt when the fair rate increases. Cebinae [19] uses leaky-bucket filters to estimate per-flow rate and enforce fairness by "taxing" the heavy flows, however it takes several seconds to converge to fair allocation. Nimble [17] implements precise TCP flow rate limiting by simulating queue draining in the data plane. However, it only supports fixed rates set by the control plane and requires per-flow memory. Instead, our work automatically adjusts and enforces fair per-user bandwidth limit within milliseconds timescale for millions of users.

VIII. CONCLUSION

We present AHAB, a data-plane hierarchical fair bandwidth limit enforcer. Using a novel approximate data structure, AHAB scales to millions of users across thousands of network slices. AHAB exploits approximate arithmetic to implement interpolation-based bandwidth limit update fully within the data plane, leading to fast convergence. Evaluation shows that AHAB converges to a fair allocation within 3.1ms, 13x faster than prior work, without sacrificing fairness or stability. The authors have provided public access to their code at https://github.com/Princeton-Cabernet/AHAB.

Acknowledgment

This work was supported by DARPA grant HR001120C0107. We sincerely thank Zhuolong Yu for his assistance in evaluation experiments. We also thank Henry Birge-Lee, Yufei Zheng, and reviewers of INFOCOM for their helpful comments and feedback.

References

[1] (2021) P4 - Programming Protocol-independent Packet Processors. https://p4.org.

[2] A. G. Alcoz, A. Dietmüller, and L. Vanbever, "SP-PIFO: Approximating Push-In First-Out behaviors using Strict-Priority queues," in *Proc. of 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*. USENIX Association, Feb. 2020, pp. 59–76.

[3] J. C. R. Bennett and H. Zhang, "Hierarchical packet fair queueing algorithms," in *Proc. of ACM SIGCOMM*, vol. 26, no. 4. New York, NY, USA: ACM, August 1996, p. 143–156.

[4] CAIDA, "The CAIDA UCSD Anonymized Internet Traces 2018 - July 19th," 2018, https://www.caida.org/data/passive/passive_dataset.xml.

[5] X. Chen, S. L. Feibish, Y. Koral, J. Rexford, O. Rottenstreich, S. A. Monetti, and T.-Y. Wang, "Fine-grained queue measurement in the data plane," in *Proc. of 15th International Conference on Emerging Networking Experiments And Technologies (CoNEXT'19)*. New York, NY, USA: ACM, Dec. 2019, p. 15–29.

[6] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, p. 58–75, Apr. 2005.

[7] S. Floyd and V. Jacobson, "Link-sharing and resource management models for packet networks," *IEEE/ACM Transactions on Networking*, vol. 3, no. 4, pp. 365–386, Aug. 1995.

[8] P. Gao, A. Dalleggio, Y. Xu, and H. J. Chao, "Gearbox: A hierarchical packet scheduler for approximate weighted fair queuing," in *Proc. of 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI'22)*, Apr. 2022, pp. 551–565.

[9] S. Hasan, A. Padmanabhan, B. Davie, J. Rexford, U. Kozat, H. Gatewood, S. Sanadhya, N. Yurchenko, T. Al-Khasib, O. Batalla, M. Bremner, A. Lee, E. Makeev, S. Moeller, A. Rodriguez, P. Shelar, K. Subraveti, S. Kandi, A. Xoconostle, P. K. Ramakrishnan, X. Tian, and A. Tomar, "Building flexible, low-cost wireless access networks with Magma," in *Proc. of 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI'23)*. USENIX Association, Apr. 2023.

[10] R. MacDavid, C. Cascone, P. Lin, B. Padmanabhan, A. Thakur, L. Peterson, J. Rexford, and O. Sunay, "A P4-based 5G user plane function," in *Proc. of ACM Symposium on SDN Research (SOSR'21)*. ACM, Oct. 2021, p. 162–168.

[11] R. Pan, L. Breslau, B. Prabhakar, and S. Shenker, "Approximate fairness through differential dropping," *ACM SIGCOMM Computer Communications Review*, vol. 33, no. 2, p. 23–39, April 2003.

[12] R. Shah, V. Kumar, M. Vutukuru, and P. Kulkarni, "TurboEPC: Leveraging dataplane programmability to accelerate the mobile packet core," in *Proc. of ACM Symposium on SDN Research (SOSR'20)*, Mar. 2020, p. 83–95.

[13] N. K. Sharma, M. Liu, K. Atreya, and A. Krishnamurthy, "Approximating fair queueing on reconfigurable switches," in *Proc. of 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*. Renton, WA: USENIX Association, Apr. 2018, pp. 1–16.

[14] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown, "Programmable packet scheduling at line rate," in *Proc. of ACM SIGCOMM*. New York, NY, USA: ACM, Aug. 2016, p. 44–57.

[15] G. Soós, D. Ficzere, P. Varga, and Z. Szalay, "Practical 5G KPI measurement results on a non-standalone architecture," in *Proc. of IEEE/IFIP Network Operations and Management Symposium*, Apr. 2020, pp. 1–5.

[16] I. Stoica, S. Shenker, and H. Zhang, "Core-stateless fair queueing: Achieving approximately fair bandwidth allocations in high speed networks," in *Proc. of ACM SIGCOMM*. ACM, Oct. 1998, p. 118–130.

[17] V. S. Thapeta, K. Shinde, M. Malekpourshahraki, D. Grassi, B. Vamanan, and B. E. Stephens, "Nimble: Scalable TCP-friendly programmable in-network rate-limiting," in *Proc. of ACM SIGCOMM Symposium on SDN Research (SOSR'21)*, Oct. 2021, pp. 27–40.

[18] D. Xu, A. Zhou, X. Zhang, G. Wang, X. Liu, C. An, Y. Shi, L. Liu, and H. Ma, "Understanding operational 5G: A first measurement study on its coverage, performance and energy consumption," in *Proc. of ACM SIGCOMM*. New York, NY, USA: ACM, 2020, p. 479–494.

[19] L. Yu, J. Sonchack, and V. Liu, "Cebinae: Scalable in-network fairness augmentation," in *Proc. of ACM SIGCOMM*, Aug. 2022.

[20] Z. Yu, C. Hu, J. Wu, X. Sun, V. Braverman, M. Chowdhury, Z. Liu, and X. Jin, "Programmable packet scheduling with a single queue," in *Proc. of ACM SIGCOMM*. ACM, Aug. 2021, p. 179–193.

[21] Z. Yu, J. Wu, V. Braverman, I. Stoica, and X. Jin, "Twenty years after: Hierarchical Core-Stateless fair queueing," in *Proc. of 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI'21)*. USENIX Association, Apr. 2021, pp. 29–45.