P4air: Increasing Fairness among Competing Congestion Control Algorithms

Belma Turkovic and Fernando Kuipers
Delft University of Technology, the Netherlands
Email: {B.Turkovic-2, F.A.Kuipers}@tudelft.nl

Abstract—Congestion control algorithms are usually developed in isolation without thoroughly investigating their co-existence and interactions with other protocols and/or congestion control algorithms. As a result, flows using different algorithms and/or having different Round-Trip Times may over-power each other, resulting in unfair resource distribution, with a subset of the flows usually claiming most of the capacity.

To solve the aforementioned problem, we make use of programmable switches and the network programming language P4 to enforce fairness from within the network itself, instead of from the congestion control algorithms ran at the end-points. Our solution P4air continuously monitors the properties of all flows that pass through a switch and groups them based on the behavior of the congestion control algorithms used. Furthermore, for each group, it applies appropriate measures to suppress the aggressive flows and boost smaller flows. Our experiments, using modern programmable hardware (Barefoot Tofino switch), demonstrate significant performance gains for P4air in terms of fairness compared to state-of-the-art solutions.

I. INTRODUCTION

The field of congestion control – a key component of transport-layer protocols – continues to see innovation through many novel proposals, each claiming superiority for specific applications or scenarios [1], [2], [7], [13], [14], [20], [25], [29], [30], [41], [42], [44], [51], [55], [59], [63], [65]–[67]. Furthermore, new transport protocols, such as QUIC and MPQUIC, facilitate quick development of new transport features directly from the user space, enabling even more customization and more diverse network protocols in the future [12], [33], [62].

However, due to this abundance in new protocols and algorithms, it has become almost impossible to take their interactions with other protocols and algorithms into account. Consequently, even for algorithms designed with good fairness properties in mind, multiple fairness issues exist, especially when the bottleneck links are shared between flows using different congestion control algorithms or having different Round-Trip Times (RTTs) [5], [6], [8], [24], [36], [40], [53], [56], [60]. For example, classic TCP flows (using loss-based algorithms) fill the bottleneck queues (resulting in high queuing delay) and only react to the resulting packet loss. These algorithms over-power newer congestion control algorithms that also take delay measurements into account, nullifying their inherent advantages (e.g., low queuing delay) and making them unusable in a typical network, where the majority of flows still rely on loss-based algorithms. Furthermore, even when only flows using newer algorithms are present at the bottleneck, fairness can still be low, especially among flows having different RTTs.

Active queue management (AQM) solutions [3], [17], [22], [26], [45], [48], [49] have been proposed to improve fairness by deploying different dropping policies at the bottleneck. They detect congestion in the queue buildup phase, thereby improving the end-to-end latency and forcing the most aggressive flows to back off. However, by doing so, they only target one of the many metrics congestion control algorithms use to detect congestion (i.e., packet loss). In other words, they treat all flows as loss-based and are oblivious to the specific congestion control algorithms used. This has multiple disadvantages. First, algorithms that do not use loss as a metric (e.g., BBRv1) are never targeted by AQMs, potentially allowing them to overpower traditional loss-based algorithms (that would back-off upon detecting loss). Second, for algorithms that use delay as their primary metric, instead of targeting the more appropriate metric (increase in RTT) and avoiding the unnecessary retransmissions, AQMs trigger a more severe back-off mechanism by targeting packet loss. Third, AQMs usually react too late, i.e., when the buffer is already partially full and the back-off mechanism of the delay-based algorithm was already triggered (due to the increase in RTT). Additionally, most AQM solutions target the network edge and are not well suited for the network core that processes thousands of flows simultaneously. For example, state-of-the-art algorithms have problems when operating with a large number of flows and/or are too expensive to be implemented in devices, especially due to the high number of queues needed for ideal performance [10], [28], [54], [64], [68].

Contributions. In this paper, by taking advantage of the possibilities of switches with P4-programmable data-planes, we develop P4air, a P4 application, run entirely in the data-plane, that enforces fairness between all flows present on a switch. We show that P4air, in addition to improved fairness, can run on modern programmable hardware at line-rate (speeds reaching Tbps) without any loss of accuracy or performance.

First, we analyze the inter-, intra-, and RTT-fairness properties of congestion control algorithms in Sec. II. We use this as a base to classify the congestion control algorithms into four groups with high inter-fairness properties and similar behavior. Second, in Sec. III-A, we develop a “fingerprinting” solution...
that can classify, directly in the data-plane, the algorithms into the previously defined groups. Furthermore, for each group, we allocate several queues. To adapt to the current network state, we develop, in Sec. III-B, a queue reallocation algorithm (in the data-plane) that favors groups with most flows by assigning more queues to them. Next, in Sec. III-C, we complement our fingerprinting solution by developing an AQM-like solution, purely based on the data-plane) that favors groups with most flows by assigning more queues to them.

### II. CLASSIFICATION

#### A. Groups of congestion control algorithms

To determine the patterns $P^4air$ will track, we start by examining the inter- and intra-fairness properties of all algorithms available in the Linux kernel, by generating two concurrent TCP flows over a single bottleneck link (Fig. 1a, using the same setup as [60]). For the fairness index, we choose Jain’s index that reflects the fairness in resource distribution among flows and which ranges from $1/n$ (worst) to 1 (best), where $n$ is the number of flows [27].

During the connection, end-hosts continuously probe for resources by increasing their sending rate until congestion is detected, e.g. in the form of packet loss or an increase in the observed RTT. Upon detection, each algorithm employs its back-off strategy, reducing the sending rate and starting the same process anew. Therefore, depending on the metric used, algorithms either back-off during the queue build-up phase (by detecting an increase in the observed RTT) or when the queue is full (by detecting packet loss). Depending on these metrics, the fairness index is (1) high if flows use algorithms that rely on the same metric, or (2) low if flows use algorithms that rely on different metrics (Fig. 1a).

Next, we use these observations to divide the algorithms into groups with good intra-fairness properties:

- **Purely loss-based algorithms.** The most commonly used congestion control algorithms, such as Reno [50] and Cubic [23] (default algorithm in the Linux kernel), fall into this group. They only rely on packet loss to detect congestion and are the most aggressive among all the analyzed groups. Queues are filled periodically and the sending rate is reduced only after doing loss.

- **Delay-based algorithms.** Algorithms from this group are proactive and among the least aggressive of the analyzed algorithms. They try to detect the point at which the queues start to fill and reduce their sending rate after detecting an increase in RTT (or eventually packet loss).

- **Loss-delay algorithms.** Some of the best-known algorithms from this group are TCP Compound (default algorithm for Windows Server until 2019 [43]) and TCP Illinois [37]. Since they incorporate delay measurements in the congestion window calculation, they are less aggressive than the purely loss-based group. However, they still mostly use loss as their primary metric and only reduce the sending rate upon detecting loss. Thus, queues are still filled (albeit at a slower pace).

- **Model-based hybrid algorithms.** Algorithms from this group try to build a model of the network, instead of using the standard AIMD (additive increase/multiplicative decrease) algorithm. The bottleneck bandwidth and round-trip time (BRR) algorithm [7], with its periodic pattern of increasing/decreasing the sending rate, is the best-known example of this group. However, unlike other protocols, it relies neither on packet loss nor increase in RTT to detect congestion.

Since model-based algorithms can employ very different
methods to estimate the available resources in the network, further sub-groups with their distinct patterns could be formed. Moreover, since the aggressiveness of the loss-delay and purely-loss algorithms can vary, they also can be further subdivided.

B. RTT fairness

Grouping flows based solely on the metric used to detect congestion does not guarantee good fairness (Fig. 1b). On the one hand, algorithms using the AIMD algorithm usually favor the flow with a lower RTT. This flow has a faster update loop and can, therefore, adjust its sending rate more often, claiming more resources. On the other hand, model-based algorithms, such as BBR, favor flows with higher RTTs, by allowing them more time to probe for resources and to dominate the queues in the process [39], [53], [61]. Consequently, when designing P4air we take these differences into account.

III. P4AIR

To detect the patterns and features of different algorithm groups, we decided to make use of switches with programmable data-planes. On the one hand, they offer the possibility to gather and export important packet meta-data (e.g., timestamps from different stages of processing, queue depth, etc.) directly from the data-plane [31]. On the other hand, they support stateful processing, which enables the switches to track the way flows react to certain events, such as loss or an increase in queue size. By leveraging these two features, we have designed an algorithm, called P4air, that enforces fairness among competing congestion control algorithms.

P4air consists of three modules split between the ingress and egress blocks (Fig. 2): (1) **Fingerprinting module** that groups flows based on their congestion control algorithm (Sec. III-A); (2) **Reallocation module** that, when necessary, redistributes the queues between groups (Sec. III-B); and (3) **Apply actions module** that enforces fairness among flows processed in the same queue by applying custom actions (Sec. III-C).

A. Fingerprinting module

The fingerprinting algorithm tracks the way the flows react to certain events, distinguishing between short-lived and long-lived flows, as well as different groups of congestion control algorithms (as discussed in Sec. II) used by long-lived flows.

**Short-lived flows.** Whenever P4air receives a packet belonging to a new flow, it classifies it as a short-lived flow (Fig. 2). It distinguishes between two groups of short-lived flows: (1) ant flows, i.e., short, sparse flows that only transport a few, but usually critical, packets (e.g., ARP, DNS, DHCP) and (2) mice flows, i.e., TCP/QUIC flows still in the slow-start phase.

**End of the slow-start phase.** Mice flows typically transport only a small amount of data and, consequently, do not send enough to congest the switch on their own. However, if a long-lived flow would reside in this queue, it could significantly degrade their performance, causing unnecessary delays or even dropped packets. Thus, P4air implements a mechanism to detect the end of the slow-start phase, thereby distinguishing between long- and short-lived flows.

To do so, P4air first uses the timestamps of packets involved in the 3-way handshake, to estimate the flow’s RTT. To be precise, it subtracts the timestamp of the first SYN from the first packet sent after this SYN. Next, it estimates the bandwidth-delay product (BDP) as the product between the flow’s “fair” share – namely the output rate divided by the number of long-lived connections – and the estimated RTT of the flow. This value describes the number of packets (bytes) that should be sent per RTT for the TCP connection to fully utilize its share without filling the queues. Finally, P4air will reclassify a flow into one of the long-lived groups upon detecting either of the two following patterns: (1) a decrease in the number of processed packets (bytes) per RTT or (2) the number of processed packets (bytes) reaching the BDP within an RTT interval. Additionally, upon detecting the second pattern, P4air proactively drops a packet, forcing the flow to enter the congestion-avoidance phase. This way, the very aggressive slow-start phase is reduced to only the time needed to reach the bandwidth share the flow should ideally claim, avoiding the queue buildup for the mice queue.

**Long-lived flows.** Upon reclassifying a flow as a long-lived flow, P4air continuously executes two actions: (1) tracking of flow statistics, used to determine the group of the congestion control algorithm; and (2) recalculating the group, upon detecting specific patterns.

**Tracking of flow statistics.** For each processed packet P4air updates the following two statistics: (1) the number of processed packets (or bytes), and (2) the depth of the queue at the moment before the packet is placed in (enqueue queue length). In addition, as most congestion control algorithms change their behavior each RTT to react to events (or lack of them), we decided to aggregate these statistics per RTT. Furthermore, due to constraints of P4 programs (imposed to make sure that switches will run at line-rate), in particular, the lack of division and floating-point operations needed to track average values, we decided to store their maximum values.

Moreover, based on them, P4air tracks two additional metrics, called aggressiveness and BwEst counter. Aggressiveness tracks how fast a queue is being filled, differentiating between delay-, loss-delay, and purely loss-based algorithms. Every time the maximum enqueue length increases by 1%, aggressiveness is increased by 1. Otherwise, the aggressiveness is reset to 0. The BwEst counter tracks the number of patterns of increasing sending rate (by a factor of at least 1.125), which is typically used while probing for more bandwidth.

As illustrated in Fig. 2, the fingerprinting module is split between the ingress and egress blocks. To account for all packets belonging to a flow, including the ones dropped due to congestion, we decided to track the number of packets, as well as BwEst (calculated using (BwEst), in the ingress block. Similarly, since queuing statistics are not available in the ingress block (as the packet was not yet processed in the queue), the enqueue length, as well as aggressiveness, are
We decided to, initially, classify the group (or sub-group) of the flow the packet belongs to (Sec. III-A); (2) Reallocation module that processes the groups’ updates, as well as the allocation of queues between groups (Sec. III-B); and (3) Apply actions module that executes custom actions on packets to enforce fairness between flows being processed in the same queue (Sec. III-C).

Recalculating the group. We decided to, initially, classify all long-lived flows into the most conservative group (delay-based). Only upon detecting a more aggressive behavior, we reclassify them into more aggressive groups: at first loss-delay and, finally, the purely loss-based group. This way, for delay-based flows, a queue build-up is avoided, preventing them from sharing the queue with more aggressive flows (and triggering their back-off mechanism in the process). To do so, at the end of each RTT interval, P4air uses the flow’s statistics (as well as the statistics from the previous RTT interval) to tracks the following patterns:

- A continuous increase in the maximum enqueue depth for at least $m_{LD}$ RTT intervals, without any reduction in the sending rate, causes the newest flow assigned to the delay-based group to be reclassified as loss-delay.

- A continuous increase in the maximum depth of the queue for $m_{PL}$ ($m_{PL} > m_{LD}$) subsequent RTT intervals, causes the newest flow assigned to the loss-delay group to be reclassified as purely loss-based. This way, algorithms that use delay as their secondary metric (loss-delay) are differentiated from purely loss-based algorithms.

- A periodic pattern of increasing/decreasing the sending rate (tracked using the BwEst metric and parameter $m_{M}$), causes the flow to be classified as model-based (BBR), exploiting the fact that in each probe (drain) bandwidth phase, a BBR flow deliberately increases (decreases) the sending rate by 1.25 (0.75) times the measured bandwidth-delay product.

Parameters $m_{LD}$, $m_{PL}$, and $m_{M}$, are configurable and define the sensitivity of the Fingerprinting module. By lowering these values, we decrease the time needed to detect each group. However, the probability of misclassification might increase, especially for the more conservative groups. By increasing these values, more aggressive classes (e.g., the loss-based group) might never be detected, which reduces the accuracy.

Fig. 3 illustrates the fingerprinting process for the representatives of the four groups: Cubic for purely loss-based, Illinois for loss-delay, Vegas for delay-based, and BBR for model-based algorithms. First, P4air classifies all four flows as mice flows. After they reach their BDP, P4air drops a packet, forcing all four flows to enter the congestion-avoidance phase, and classifies them into the delay-based group. However, Cubic, Illinois, and BBR start filling the queues, without backing off and are reclassified into the loss-delay group. Next, due to Cubic’s very aggressive approach, the queues and the aggressiveness continue to increase, causing P4air to classify it into its correct group: purely loss-based. Similarly, after P4air detects the periodic increase in BBR’s sending rate, it reclassifies it into the model-based group. In this scenario, this occurs after this pattern is recognized twice.

Location vs. accuracy. As Fig. 3 illustrates, to correctly detect the more aggressive flows, P4air needs to be deployed on the bottleneck switch, i.e. a switch at which the queues are formed. Otherwise, due to no increase in the queuing delay, the Fingerprinting module would classify these algorithms as delay-based algorithms. However, when loss and loss-delay algorithms are not filling the queues, they would also not interfere with the present delay-based algorithms. In other words, if there would not be a bottleneck, there would also not be a problem that P4air needs to solve. In contrast, algorithms like BBR (that do not rely on the queuing metrics) can always be detected due to their periodic pattern.

B. Reallocation module

Every time the Fingerprinting module reclassifies a flow, the Reallocation module executes two actions: (1) it stores and updates the flow’s group, and (2) it runs the queue reallocation
algorithm, making sure that long-lived flows are distributed evenly across all available queues.

**Updating and storing of the group.** For most flows, group recalculations can only be done in the egress block, i.e., after the egress statistics (e.g., aggressiveness, enqueue depth) are known. However, to ensure that the packet is queued correctly, the flow’s group needs to be known in the ingress block. Consequently, the register (stateful memory array) storing the estimated group must be allocated in the ingress block and is, as such, not accessible from the egress block. To solve the abovementioned problem, *P4air* recirculates all packets that trigger a reclassification (Fig. 4a). For every recirculated packet, *P4air* updates this register to make sure that packets belonging to the same flow are processed in the same queue. *P4air* uses an additional register array to store the queue information. Furthermore, for every recirculated packet, *P4air* updates this register to make sure that flows belonging to the same group are processed together, by assigning a new queue using a sequential index (one per-group). As the next step, it updates the boundaries of each group (parameters $l_i$) according to Fig. 4a. For every other packet (not recirculated), *P4air* checks if the stored queue is outside of the corresponding $l_i$ values (that might have been updated) and, if so, assigns a new queue using the sequential index (Fig. 4b). This way, whenever reallocation occurs, *P4air* reassigns all flows processed in the queue that was assigned to a different group uniformly among the other remaining queues belonging to the group. Similarly, it assigns the first $fpq−1$ flows (and the flow that triggered the $l_i$ update) belonging to the group that gained a queue to the new queue.

**Queue reallocation algorithm.** To leverage standard scheduling mechanisms, such as Round Robin (RR) or FQ (Fair Queuing), we designed a new queue reallocation algorithm. Moreover, to be able to deploy this algorithm on the hardware switches running at line-rate, we avoided operations that would introduce significant computational overhead (e.g., loops and floating-point operations). Our algorithm makes sure that groups that have more flows are also assigned more queues, thereby distributing all the flows evenly across all the available queues. First, to ensure that packets belonging to the same flow are processed in the same queue, *P4air* uses an additional register array to store the queue information. Furthermore, for every recirculated packet, *P4air* updates this register to make sure that flows belonging to the same group are processed together, by assigning a new queue using a sequential index (one per-group). As the next step, it updates the boundaries of each group (parameters $l_i$) according to Fig. 4a. For every other packet (not recirculated), *P4air* checks if the stored queue is outside of the corresponding $l_i$ values (that might have been updated) and, if so, assigns a new queue using the sequential index (Fig. 4b). This way, whenever reallocation occurs, *P4air* reassigns all flows processed in the queue that was assigned to a different group uniformly among the other remaining queues belonging to the group. Similarly, it assigns the first $fpq−1$ flows (and the flow that triggered the $l_i$ update) belonging to the group that gained a queue to the new queue.

**C. Apply actions module**

Traditional AQMs can only probabilistically drop packets or use ECN marking to trigger the senders to back off. However, as mentioned earlier, for some congestion control algorithms, less severe actions, such as delaying a packet, might be more appropriate. Furthermore, newer congestion control algorithms might not react to these indicators and require new actions to be designed. Hence, to target each group’s specifics, *P4air* uses the custom actions listed below.

**Dropping a packet.** This action targets the algorithms that use loss as the primary metric (purely loss-based and loss-delay algorithms), similar to standard AQMs.

**Delaying a packet.** This action targets delay-based algorithms. By delaying (instead of dropping) a packet, they will back off, reducing the need for retransmissions and improving the average end-to-end delay in the process. Just delaying a packet is not possible in P4, thus, we implemented this action by recirculating a packet back to the ingress.

**Changing the receiver window.** A flow’s sending rate is determined by the minimum of the receiver and congestion windows. Thus, by reducing the receiver window, the sender is
TABLE I: Memory consumption on a 24-port switch. \( \beta \) is the memory used to track the current group and queue and \( \alpha \) the memory used to track the RTT and flow statistics.

<table>
<thead>
<tr>
<th>( n_{flows} )</th>
<th>Total Memory (KB)</th>
<th>( \alpha ) [%]</th>
<th>( \beta ) [%]</th>
<th>Reallocn</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 2^{10} )</td>
<td>17.53</td>
<td>86.89</td>
<td>5.84</td>
<td>7.27</td>
</tr>
<tr>
<td>( 2^{11} )</td>
<td>33.92</td>
<td>89.81</td>
<td>6.04</td>
<td>4.16</td>
</tr>
<tr>
<td>( 2^{12} )</td>
<td>66.57</td>
<td>91.53</td>
<td>6.15</td>
<td>2.32</td>
</tr>
<tr>
<td>( 2^{13} )</td>
<td>131.73</td>
<td>92.51</td>
<td>6.22</td>
<td>1.28</td>
</tr>
</tbody>
</table>

Fig. 4: Reallocation algorithm.

Forced to back off. However, to do so, the window in the ACK packets, sent from receiver to sender, needs to be modified. For this action to work, packet transfers in both directions have to cross the same bottleneck.

Sensitivity. The sensitivity of the Apply actions module, i.e., how often \( P4air \) applies actions to the flows, determines the link utilization, as well as the distribution of resources among flows. If the sensitivity is set too high, back-off mechanisms are triggered before the flows even started congesting the network, leading to low utilization. In contrast, if configured too low, they are triggered too late (or at all), allowing aggressive flows to claim more resources and leading to unfairness. Consequently, to keep the utilization high, while still targeting aggressive flows, we decided to execute this module only when the flow the packet belongs to is sending above its BDP.

D. Overhead & Limitations.

Memory overhead. Contrary to standard AQM solutions, such as Codel, the memory overhead of \( P4air \) scales linearly on both the number of flows it wishes to track (127b per flow), as well as the number of output ports \(((5 + n_{queues}) \log_2(n_{flows}) + 3 \log_2(n_{queues}) + 8 \log_2(n_{flows}/n_{queues})\) bits per output port). However, as
We have used the topologies shown in Fig. 5. To perform experiments using (1) the Mininet emulation environment with the topologies suffice for our purposes. We performed the experiments using (1) the Mininet emulation environment with the P4 software switch (bmv2 [47], Fig. 5a) and (2) a testbed with a Barefoot Tofino switch [58] (Fig. 5b). To perform measurements, we relied on tcpdump, iperf3, socket statistics, and P4 statistics exported directly from the switch.

B. Tuning of the fingerprinting algorithm

P4air has multiple tunable parameters ($m_{LD}$, $m_{PL}$, and $m_M$) that provide a trade-off between detection accuracy and detection delay. Fig. 6a - 6f show the impact of changing the $m_{LD}$, $m_{LD}$, and $m_{LD}$ on the detection of each group.

Choosing $m_{LD} = 4$. As all flows are, by default, assigned into the delay-based group, the choice of $m_{LD}$ should ensure that only delay-based flows remain, while all the others are reclassified into the loss-delay group. By analyzing the detection time and accuracy for different values of $m_{LD}$ (Fig. 6a - 6b), we find that for $m_{LD} = 4$, the probability of false positives for the delay-based algorithms is low enough.

Choosing $m_{PL} = 12$. By increasing $m_{PL}$, the probability of misclassifying a loss-delay algorithm decreases, even for the aggressive algorithms from this group (e.g., Illinois, Fig. 6c). However, in addition to the increase in detection delay (Fig. 6d), the accuracy of the fingerprinting module for the least aggressive purely loss-based algorithms, such as New Reno, decreases as well (Fig. 6c). Thus, we choose $m_{PL} = 12$, as the best compromise between accuracy and detection delay.

Choosing $m_M = 4$. Finally, we evaluated the influence of $m_M$ on the detection accuracy of the model-based algorithms. All algorithms probe for bandwidth in their slow-start phase and, depending on their classification into the delay-based group (when the tracking for aggressiveness and BWEst starts), they can cross the threshold $m_M$. However, only BBR does so periodically, every 10 seconds, and will always be correctly identified if $m_M$ is set high enough.

C. P4air performance

Resource utilization. Our Tofino implementation, when tracking a maximum of $2^{16}$ flows, used less than 14% of the available header and metadata memory, less than 18% of the total register memory and less than 5% of hash generators available on the switch (shared between forwarding and P4air).

The RTT-estimation algorithm. First, we evaluated the accuracy of the RTT estimation by varying the link delay and external traffic. In all the scenarios without external traffic, the difference between the configured and estimated RTT was less than 0.52ms ($\leq 1.5\%$ of $RTT_{conf}$, Fig. 7a). As this value was nearly constant in our experiments, we conclude that this overhead is the processing delay on both the servers and the switch. Furthermore, as Fig. 7b illustrates, by processing the new flows in a separate queue (as in P4air), the effect of long-lived connections on the RTT accuracy is not significant.

Sensitivity. If the sensitivity of the Apply actions module ($s$) is set too high, all flows are punished too aggressively and the overall utilization drops significantly, reaching as little as 50%. In contrast, if $s$ is set too low, aggressive flows are never punished and the fairness will remain low (Fig. 7c).

Different actions. The action to change the receiver window offered the biggest performance boost (Fig. 7e, Fig. 7d).

D. Inter- and Intra-Fairness: P4air vs. existing solutions

Effect of fingerprinting. Distributing flows to queues based on their congestion control group significantly improves fairness (Fig. 7f), especially when the number of flows increases and their interactions become more detrimental. As Fig. 7j illustrates, P4air (and Idle P4air) leverage the good intra-fairness properties of most algorithms and flows, consequently, rarely overpower each other, i.e. their throughput is clustered around the ideal throughput. In contrast, by queuing flows without taking into account their group (Different Queues), two distinct clusters are present: (1) overpowered flows at 6 − 7% of the ideal throughput and (2) aggressive flows at multiples of the ideal throughput.

Effect of the Apply actions module. Fingerprinting flows improves fairness, but does not prevent queue buildup. However, the Apply actions module targets the aggressive flows, forcing them to back off and, consequently, lowers the increase in RTT due to queuing (Fig. 7h). Furthermore, when the number of flows per queue increases, the Apply actions module makes sure that they remain fair to each other (Fig. 7e, 7f).
Idle $P4air$ vs. $P4air$. When a small number of flows compete per queue ($N_{flows} < 128$), the Fingerprinting and the Reallocation modules are enough to achieve good fairness properties (Fig. 7f). However, to reduce the queuing delay and to target a higher number of flows, the Apply actions module is needed.

Utilization. As Fig. 7f illustrates, both versions of $P4air$ were able to maintain a high link utilization ($\geq 91\%$), similarly to the results achieved by the comparison baselines (Different Queues and NoAQM).

Inter-Fairness. Both $P4air$ versions were able to significantly outperform the comparison baselines and realize a fair distribution of resources (Fig. 7f). Therefore, our results show that by using more information about the flow, such as the group of the congestion control algorithm, the switch can target the flow’s specifics, thereby enabling a fairer distribution of resources.

E. RTT fairness: $P4air$ vs. existing solutions

Effect of fingerprinting. While distributing flows to different queues, Idle $P4air$ does not take into account the flows’ RTT (but only the group), causing the flows with different RTTs to compete inside the same queue. However, in comparison to Different Queues, the Reallocation module makes sure that flows are more uniformly distributed between the queues. Consequently, the fairness index is higher (Fig. 7g, 7k).

Effect of the Apply actions module. To increase the fairness, the Apply actions module is needed, especially when the RTT differences between the flows increase. As Fig. 7k illustrates, $P4air$ merges the two groups, the very aggressive flows at the right side and the overpowered flows at the left side, into one.

Utilization. As a consequence of the Apply actions module, i.e. the actions applied to the most aggressive flows, $P4air$ had the lowest utilization compared to all the other solutions (although still $\geq 90\%$).

Large $\Delta$RTT. $P4air$ was able to maintain a high fairness index, especially for lower $\Delta$RTT values. However, as we increased $\Delta$RTT, the fairness index reduced, although it was still higher than the comparison baselines.

V. Deployment considerations

While our evaluation demonstrates performance gains, especially in terms of fairness, a more extensive evaluation in more complex scenarios (involving multiple switches) is recommended. Furthermore, several limitations of the current implementation, listed below, should be considered.

Weighted queuing algorithms. If an algorithm that supports dynamic weights per queue is supported by the switch, flows belonging to the same group can be assigned into one queue with a weight set to the number of flows present, ensuring that all groups get their fair share of resources, simplifying $P4air$ by making the queue reallocation algorithm obsolete.

RTT-estimation algorithm. In our current implementation, to ensure an accurate RTT estimate, packets involved in the
(a) Average error in the RTT estimation for different RTTs. Each scenario is run 10 times.

![Graph showing RTT estimation error](image)

(b) Average error in the RTT estimation for different levels of congestion. Each scenario is run 10 times.

![Graph showing RTT estimation error](image)

(c) Sensitivity. Cubic is shown in red, Illinois in orange, and the value $s$ inside the circles. Each scenario is run 4 times.

![Graph showing sensitivity](image)

(d) RTT fairness for 256 different flows running the same congestion control algorithm for two different values of $\Delta RTT$. Each scenario is run 4 times.

![Graph showing RTT fairness](image)

(e) Average inter- and intra-fairness for 128 flows. The share of flows per group is varied between 0% and 100% with a step of 25%. Each ratio is run 4 times.

![Graph showing inter- and intra-fairness](image)

(f) Average Inter- and intra-fairness. The share of flows per group was varied between 0% and 100% with a step of 25% (as in Fig. 7e). All flows had the same RTT. For each $n_{flows}$, each combination (35 different) is run 4 times (140 in total). Theoretical minimum is shown as a dashed black line.

![Graph showing inter- and intra-fairness](image)

(g) Average RTT Fairness for 256 different flows. Each link delay was configured to a multiple of $\Delta RTT$. All flows used the same congestion control algorithm (one of the four groups, as in Fig. 7d). For each $\Delta RTT$, each group (4 different) is run 4 times (16 in total).

![Graph showing RTT fairness](image)

(h) Average delay vs. average utilization for scenarios shown in Fig. 7e. Ideal operating point (0, 100%).

![Graph showing delay vs. utilization](image)

(i) Average delay vs. average utilization for scenarios shown in Fig. 7d. Ideal operating point is (0, 100%).

![Graph showing delay vs. utilization](image)

(j) Average RTT vs. throughput per flow for a scenario with 128 flows, with each group having 25% of the flows (zoomed-in version of one of the scenarios shown in Fig. 7e). Ideal operating point is (1, 0).

![Graph showing RTT vs. throughput](image)

(k) Average RTT vs. throughput per flow for a scenario with 256 BBR flows with $\Delta RTT = 1\mu$s (zoomed-in version of one of the scenarios shown in Fig. 7d). Ideal operating point is (1, 0).

![Graph showing RTT vs. throughput](image)

Fig. 7: Evaluation of the P4Air algorithm on a Barefoot Tofino switch.
3-way handshake should not be delayed at any other switch in the network, nor should the connection’s RTT change. An inaccurate RTT estimate has the biggest effect on the sensitivity of the Apply actions module, as actions on flows with an overestimated RTT are applied later, allowing them to claim more bandwidth. Consequently, P4air’s fairness properties might decrease, but should remain as high as those of Idle P4air. There are three possible solutions to this problem: (1) the RTT-estimation algorithm can be extended to make use of the other switches (or at least all the bottlenecks) in the path to periodically summarize the total queuing delay (similar to [31]); (2) the RTT algorithm can be replaced with the one presented in [9]; (3) the Apply actions module could be modified to avoid the metrics that depend on the RTT estimate (e.g., queuing delay instead of BDP).

**Queue-assignment imbalance.** In the current implementation, we assume that flows complete uniformly across all queues. Otherwise, an imbalance in the queue assignment might occur, leading to more flows competing inside the same queue and receiving a lower share of the resources. To remove this assumption, P4air can be modified to keep track of the number of flows processed by each queue, as well as the identifiers of the queues (per group) having \( \leq fpq \) flows. This way, by sacrificing more memory \((n_{queues}\log_2(n_{flows}) + 4\log_2(n_{queues}))\), we can make sure that all queues process a similar amount of flows by enqueuing new flows into the saved queue (with \( \leq fpq \) flows). Note that such an imbalance is also possible with all AQMs, which usually assign flows to queues based on the hash of a flow identifier.

**Traffic shaping mechanisms & Multiple bottlenecks.** Traffic shaping mechanisms (e.g., other AQMs, P4air) deployed at other switches on the path and/or the presence of multiple bottlenecks might impact the patterns tracked by P4air and lead to misclassification. As (some) flows are shaped already and thus perform fair, this may not be an issue (or they could form a separate “marked” group). Nonetheless, a more extensive evaluation of the impact of different shaping mechanisms and multiple bottlenecks on the fingerprinting accuracy is needed.

**P4air placement in complex topologies.** With only a few strategically placed P4air switches, overall network behavior might benefit greatly. Developing a placement algorithm to determine the locations and amount of P4air switches in complex topologies has been beyond the scope of this work, but is important to consider when implementing P4air.

### VI. Related work

Many AQM algorithms (RED [22], ARED [21], SRED [46], FRED [35], REM [3], CHOKe [49], BLUE [17], AVQ [32], AN-AQM [57], DC-AQM [52], CoDel [45], PIE [48], SFB [17], SBQ) [34], SFQRED [16], FQ_CODEL [26]) were designed to detect and overcome static queues, reducing the queuing delay in the process. Classic AQMs, like RED, probabilistically drop packets based on the average number of packets inside a queue. However, studies have shown that their optimal configurations vary depending on parameters, such as capacity and number of flows, which causes network instabilities and traffic disruptions [18], [19], [35], [38].

Newer AQMs, like CoDel and PIE, were designed to overcome these issues and are, consequently, easier to manage and configure [45]. Moreover, when combined with scheduling algorithms providing isolation, such as FQ, they ensure high fairness at a wide range of bandwidths and flows. However, as a reaction to queue build-up, they can only drop a packet. Hence, they (1) cannot target newer congestion control algorithms that do not use loss as a metric, (2) lead to (potentially unnecessary) retransmissions, (3) usually require many queues, which are scare resources in hardware switches, and (4) do not take advantage of the inherently good intra-fairness properties that most congestion control algorithms have.

In contrast, solutions such as Virtualized Congestion Control create a translation layer in a hypervisor, enabling an easy upgrade of legacy algorithms and offering a data-center operator the ability to implement a single fair algorithm [11]. However, multiple issues might occur: (1) tenants might expect more isolation, (2) it is unusable in cases when flows are originating from multiple data-centers (using different “fair” algorithms), and (3) the solution can only implement certain TCP flavors by violating the TCP end-to-end semantics (i.e., acknowledging the packets not yet received).

### VII. Conclusion & Future Work

In this paper, we first developed a fingerprinting algorithm that harnesses the power of programmable data-planes to detect the congestion control algorithms used by flows. By instructing the bottleneck switch to track very simple metrics, such as queuing delay and sending rate, our algorithm is able to track the way the flows react to specific events (e.g., queue buildup), allowing it to classify the flows into one of the delay-, loss-, delay-loss, and model-based groups. Second, we used this knowledge, and the fact that most algorithms have very good intra-fairness properties, to enqueue flows using similar algorithms into the same queue. Third, for each of these groups, we developed custom actions, able to target their specifics, which we used to punish the most aggressive flows. P4air incorporates the aforementioned three elements, thereby enabling a switch to target each flow specifically and ensure a fair distribution of resources among all flows in the process.

**Future directions.** As future work, we plan to investigate (1) if the queue reallocation algorithm can be extended to take into account the flows’ RTTs, to improve the RTT fairness for very large \( \Delta RTT \), (2) the placement of multiple P4air switches in a network, (3) the influence of other AQMs or traffic shaping mechanisms on the Fingerprinting module, and (4) the possibilities of real-time inference for the Fingerprinting module (e.g., [4]). Furthermore, we plan to add support for newer congestion control protocols (e.g., PCC [13], PCVivace [14], Remy [65], Tao [55], and Sprout [66]), as well as Coupled, a.k.a. multipath, algorithms (e.g., LIA [51], OLIA [30], BALIA [63], and WVegas [67]).


