Scaling Switch-driven Flow Control with Aquarius

Wenxue Li
Chaoliang Zeng
iSing Lab, Hong Kong University of Science and Technology

Jinbin Hu
Kai Chen*
iSing Lab, Hong Kong University of Science and Technology

ABSTRACT

As datacenter networks support more diverse applications and faster link speeds, effective end-to-end congestion control becomes increasingly challenging due to the inherent feedback delay. To address this issue, switch-driven per-hop flow control (FC) has gained popularity due to its natural flow isolation, timely control loop, and ability to handle transient congestion. However, the ideal FC requires impractical hardware resources, and the state-of-the-art approximation approach still demands a large number of queues that exceeds common switch capabilities, limiting scalability in practice.

In this paper, we propose Aquarius, a scalable solution for per-hop FC that maintains satisfactory flow isolation with a practical number of queues. The key idea of Aquarius is to take independent control of different flows within the same queue, discarding the traditional practice of managing traffic collectively within the same queue. At its core, Aquarius applies a contribution-aware pausing mechanism on congested switches to enable individual control decisions for arriving flows, and uses an opportunistic re-assigning strategy on upstream switches to further isolate congested and victim flows. Experimental results demonstrate that Aquarius maintains comparable performance with 4× fewer queues, and achieves 5.5× lower flow completion times using the same number of queues, compared to existing solutions.

CCS CONCEPTS

• **Networks** → **Transport protocols**.

KEYWORDS

Datacenter Networks, Per-hop Flow Control

ACM Reference Format:
Wenxue Li, Chaoliang Zeng, Jinbin Hu, and Kai Chen. 2023. Scaling Switch-driven Flow Control with Aquarius. In *7th Asia-Pacific Workshop on Networking (APNET 2023), June 29–30, 2023, Hong Kong, China.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3600061.3600066

1 INTRODUCTION

Nowadays, datacenters are supporting an increasingly diverse range of applications, such as distributed computing, enterprise services, cloud storage, serverless computation, and large-scale data analysis [6, 7, 17, 18, 26]. These applications impose strict demands on network communications, requiring the underlying network fabric to simultaneously support their individual needs, such as high throughput, low latency, effective isolation between different services, and stable network quality delivery. Congestion control (CC) plays a critical role in supporting effective communications. Current datacenter networks (DCNs) mainly rely on complicated end-to-end CC mechanisms that utilize end-to-end congestion signals to adjust the sending rate, while using simple first-in-first-out (FIFO) queues at switches. Examples of end-to-end CC include DCTCP [3], DCQCN [32], Timely [20], Swift [15], and HPCC [19].

However, as DCN link speed continues to increase and application traffic becomes more bursty and harder to predict, end-to-end CC finds it challenging to remain effective (§2.1). First, it is difficult to control short flows in a timely manner. End-to-end congestion signals lose control of flows that finish within a single network round-trip time (RTT), and the proportion of these flows is growing with increasing link speed. Second, it is challenging to maintain stable throughput for large flows. Datacenter traffic bursts can cause non-negligible fluctuations even in sub-RTT timescales [1, 2]. End-to-end signals are stale by at least one RTT, and thus the endpoint reaction based on this stale information cannot handle the fast-emerging new network environment.

Because of the inherent drawbacks of end-to-end CC, a recent line of work tries to explore switch-driven control with a timely reaction to congestion [8, 27]. Among them, the switch-driven per-hop flow control (FC)[1] becomes attractive because of its natural flow isolation, timely control loop, and ability to handle transient congestion, which are promising features in orchestrating today's high-speed DCNs. However, the ideal switch-driven FC [16] requires the switch to allocate an exclusive queue for each flow, which is impractical.

BFC [8] is the state-of-the-art solution to approximate the ideal FC. It alleviates the requirement for switch resources by only controlling active flows that have queued packets in the switch buffer. Specifically, BFC assigns an exclusive queue to each active flow if possible, and multiple flows will share a queue if there are no free queues. However, BFC still requires a large number of physical queues that exceed common switch capability. As a result, BFC's performance will get a significant degradation when deployed in common switches with a limited number of queues (§2.2).

Based on the above observations, we ask: *can we push the switch-driven FC one step further towards practicality by making it feasible with a limited number of queues that is much less than the number of active flows?* It requires the solution must maintain satisfactory flow-level isolation even when multiple flows share the same queue.

*Kai Chen is the corresponding author.

APNET 2023, June 29–30, 2023, Hong Kong, China

ACM ISBN 979-8-4007-0782-7/23/06…$15.00
https://doi.org/10.1145/3600061.3600066

[1] In the rest of the paper, we refer to per-hop flow control as flow control for short.

In this paper, we provide a cautiously optimistic answer via Aquarius, a scalable solution of switch-driven per-hop FC that maintains satisfactory flow isolation with a practical number of queues (§3). At a high level, Aquarius departs from the traditional practice of collectively managing the overall traffic within the same queue and instead aims to take independent and precise control of different flows that are within the same queue. To achieve this, Aquarius primarily applies two mechanisms: *contribution-aware pausing* on the congested switch and *opportunistic re-assigning* on the corresponding upstream switches.

When a new active flow (*i.e.*, without any previously buffered packets) arrives at a switch, Aquarius dynamically allocates an empty queue if possible or the least-utilized queue for it. When the length of an egress queue exceeds a given threshold indicating possible congestion, Aquarius applies *contribution-aware pausing* to differentiate multiple flows within this queue and sends PAUSE messages only to the congestion-responsible flow's upstream queue. Aquarius determines responsible flows based on their buffer usages, total egress queue length, and expected fair occupation.

Uncongested flows continue to arrive at the congested switch and can cause a severe buffer overflow in extreme cases. To minimize this issue, we set two pausing thresholds: Q_{high} and Q_{low}. When the queue length exceeds Q_{low}, and there is a possibility of congestion, Aquarius only pauses the congestion-responsible flows. When the queue length exceeds the conservative threshold Q_{high}, all flows passing through the congested queue are paused to prevent severe buffer overflow.

When the upstream switch receives a PAUSE message, if the to-be-paused congested flow solely occupies a queue, Aquarius pauses that queue from outputting packets. Otherwise, if the congested flow shares a queue with other uncongested flows (*i.e.*, victim flows), Aquarius performs *opportunistic re-assigning* where Aquarius directs the subsequent packets of the congested flow to a dynamically reserved separating queue, which isolates congested and victim flows further. Meanwhile, Aquarius allows the original queue to continue transmitting, ensuring that the victim flows remain unaffected.

We have implemented Aquarius in the NS-3 simulator [22]. Our preliminary simulations demonstrate that Aquarius outperforms previous schemes significantly (§4). For instance, Aquarius ensures high throughput for uncongested and victim flows under a typical micro-benchmark. Besides, under the realistic Web Server traffic, Aquarius maintains comparable performance with 4× fewer queues and reduces the average FCT by 5.5× and 99^{th} percentile tail FCT by 5× using the same number of queues, compared with BFC.

2 BACKGROUND & MOTIVATION

2.1 Insufficiencies of End-to-End CC

Datacenter networks are experiencing several trends that make it increasingly challenging to attain satisfactory performance with end-to-end CC protocols.

① **Higher link speeds lead to an increase in the number of flows that can complete within a single RTT.** As datacenter link speeds grow rapidly from tens to hundreds of Gigabytes per second, more flows become "smaller" and can finish within a single RTT. To explicitly show this trend, we analyze four production datacenter workloads: *Web Search* [3], *Web Server* [24], *Facebook Hadoop* [24] and *Alibaba Storage* [19]. We measure three link speeds and calculate their bandwidth-delay products (BDPs), assuming a 12μs RTT, and then treat flows with sizes smaller than the BDP as capable of finishing in a single RTT. Fig. 1 shows the percentage of such flows. The results indicate that, as link speeds increase, more flows can finish within a single RTT, and up to 90% of flows can complete (in theory) under today's 100Gbps bandwidth.

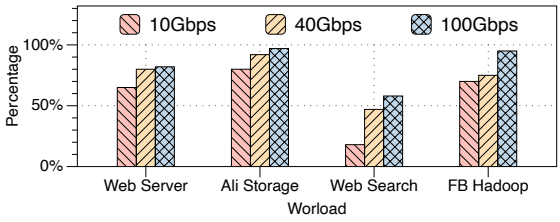


Figure 1: Percentage of flows that finish in a single RTT in four workloads and three link speeds.

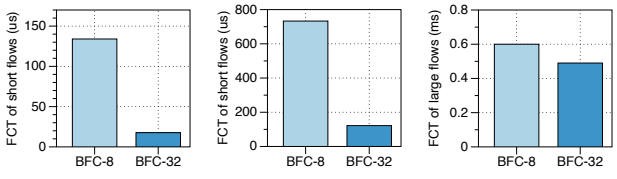
End-to-end CC protocols mainly rely on receiver-echoed signals (*e.g.*, ECN [3], RTT [20], loss [9], multi-bits INT [19], *etc.*) to adjust sending rates. Consequently, the sender requires at least one RTT to receive feedback and loses control of flows that can complete within the first RTT. The sender either blindly starts these flows at a high rate and risks congestion or starts them at a low rate and leads to network under-utilization [11].

② **Bursty traffic leads to significant fluctuations in sub-RTT timescales.** As datacenter networks support more diverse applications, such as frontend query traffic and backend storage streams [14], the traffic becomes increasingly bursty due to the mixture of short and large flows along with the growing link speeds [1, 2]. Large flows can experience rapidly emerging and disappearing cross-traffic bursts, resulting in non-negligible network fluctuations even in sub-RTT timescales (*i.e.*, sub-RTT level fluctuations). BFC [8] demonstrates that when a large flow competes with cross-traffic on a single link, the fair-sharing rate of this large flow can experience up to 60% mean change in less than an RTT.

End-to-end CC is unable to handle these sub-RTT level fluctuations and thus fails to maintain stable throughput for large flows. This is because end-to-end signals rely on stale information by at least one RTT, while the network condition can change dramatically over a short period. Therefore, the endpoint reaction based on this stale information cannot adapt to the fast-emerging new network environment.

2.2 Existing Flow Control is Not Scalable

2.2.1 Ideal FC is Impractical. The ideal implementation of per-hop FC [16] assigns a dedicated physical queue and maintains states for every flow, including inactive flows that have no packets queued at the switch. The ideal FC has the following merits: (1) *Timely control*: a congested switch can directly regulate the upstream entity based on its buffer condition. Thus, it can reduce congestion or increase utilization within 1-Hop RTT (usually 1-2μs) instead of end-to-end RTT. Because of this, it can handle the transient congestion that



(a) Average **(b) 99^{th} percentile** **(c) Average**

Figure 2: FCT of BFC under Web Server distribution with 70% load and 5% 100-1 incast.

is caused by sudden bursts in sub-RTT timescales; (2) *Flow isolation*: individual flows are queued in separating queues and thus can be adjusted independently without any inter-flow interferences. Despite being promising, current switch capabilities cannot accommodate the physical queues demanded by ideal FC, rendering them impractical.

2.2.2 Scalability Issues Persist in BFC. BFC [8] is currently considered the leading approximation of the ideal FC that can be implemented in today's programmable switches. BFC assigns a dedicated queue to each active flow, with multiple flows sharing a queue when there are no available queues.

BFC requires more physical queues than the common switch can accommodate. Even though BFC removes the necessity for per-flow queues, it still requires dedicated physical queues to regulate each active flow. The original implementation of BFC uses 32-128 queues per port, which is considerably more than the available number of queues in most practical switches. Firstly, only state-of-the-art programmable switches [13] can support such a high number of queues per port, while typical switches are usually equipped with 8 queues or fewer. Secondly, physical queues are essential resources that are typically reserved for strong physical isolation between different tenants. Therefore, it is impractical to use all queues for intra-tenant traffic [29]. Consequently, an effective FC solution must consider this limitation and deliver satisfactory performance with a practical number of queues.

We conduct a simulation to demonstrate the performance of BFC in 32 (*i.e.*, BFC-32, as in the original paper) and 8 queues (*i.e.*, BFC-8) per port. We use the Web Server [24] workload with a 70% average load and 5% incast traffic and a fat-tree topology (details in §4). The results indicate that BFC-8 gets a significant degradation with about 7.6× higher average (Fig. 2a) and 6× higher tail (Fig. 2b) FCT for short flows, and has a 20% higher average FCT for large flows (Fig. 2c), compared with BFC-32. Note that the transmitting capacity per port is the same for both 8 and 32 queues, and thus the degradation in FCTs results from the severe interference between flows within the same queue.

3 DESIGN

3.1 Design Overview

Aquarius eliminates the traditional requirement for multiple separating queues by making an effort to control different flows independently that are mapped to the same queue. Consequently, Aquarius can maintain satisfactory flow isolation even with a minimal number of queues that is much less than the number of active flows. The overall architecture of Aquarius consists of three phases: (1) *dynamic flow mapping* on every switch passed by to allow uncongested flows to pass quickly, (2) *contribution-aware pausing* on the congested switch to precisely control flows responsible for congestion (*i.e.*, congested flows), and (3) *opportunistic re-assigning* on upstream switches to further isolate congested and victim flows. We will introduce each of them one by one.

Algorithm 1 Enqueueing and Contribution-aware Pausing

Inputs:
Q_{low}, Q_{high}: two queue length thresholds
T_{min}, T, T_{avg}: minimum, current, and averaged arriving intervals between adjacent packets
$Occu_{fair}$: size of fair flow occupation

```
1:  function ENQUEUE(packet)
2:      key = <packet.egressPort, hash(packet.FID)>
        // Assign queue for new active flow
3:      if flowTable[key].size == 0 then
4:          flowTable[key].mappedQ = min_{Q.length} {Q | Q ∈ packet.egressPort}
5:      packet.q = flowTable[key].mappedQ
6:      flowTable[key].size += 1
        // Make pausing decision
7:      if packet.q.length > Q_high then
8:          flowTable[key].pauseNum += 1
9:          packet.congested = true                    ▷ Congestion flag
10:     else if packet.q.length > Q_low & T_avg < T_min then
        // Contribution-aware pausing
11:         if flowTable[key].size > Occu_fair then
12:             flowTable[key].pauseNum += 1
13:             packet.congested = true
14:     T_avg = α * T + (1 − α) * T_avg                ▷ Update T_avg
15:     if flowTable[key].pauseNum == 1 then
        // Pause upstream entity
16:         send PAUSE(packet.upstreamQ, packet.FID)
```

Dynamic flow mapping. Aquarius maintains a FlowTable that records states, *e.g.*, the mapped queue (`mappedQ`), the number of queued packets (`size`), *etc.*, for active flows that have packets buffered on the switch. Aquarius uses the five tuples of source and destination address, port, and IP protocol as flow identifier (FID), and uses the combination of the flow's egress port and the hash of FID to index the table entry. Algorithm 1 illustrates the enqueueing logic. Aquarius regards a new active flow (*i.e.*, without previous packets buffered on the switch) as un-congested and dynamically allocates an empty queue if possible or assigns it the queue with the least length (Line 3). Aquarius maps the packets of existing flows to the same queue to ensure in-order delivery (Line 5). The underlying design principle is to enable uncongested flows to pass quickly.

3.2 Contribution-aware Pausing

We show the architecture comparison between ideal FC, BFC, and Aquarius in Fig. 3. There are four flows ($f_1 \sim f_4$) with different arriving rates passing the congested switch and f_4 sharing an upstream egress port with f_5. Ideal FC allocates an exclusive queue

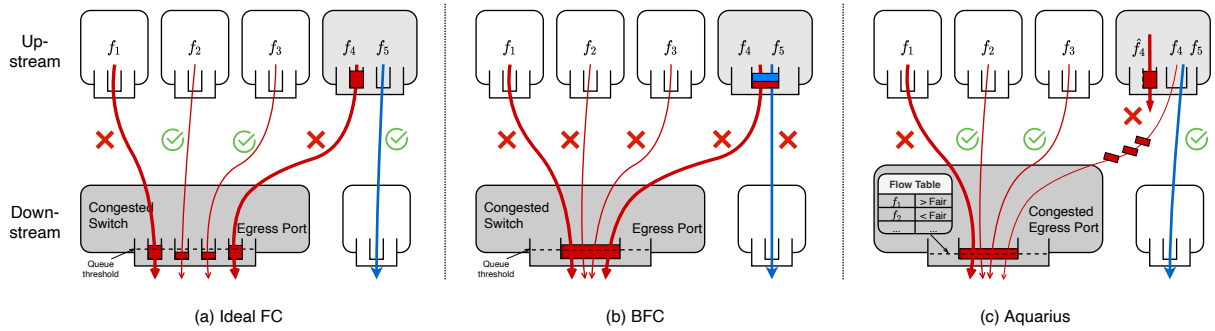


Figure 3: Architecture comparison between ideal FC, BFC, and Aquarius. BFC unnecessarily pauses uncongested and victim flows. Aquarius precisely pauses the real congestion-responsible flows while leaving other flows unaffected, consequently approximating the ideal FC's behavior.

for each flow, and thus $f_1 \sim f_5$ are controlled independently, *i.e.*, the congestion-responsible f_1 and f_4 are paused and f_2, f_3 and f_5 are unaffected, as shown in Fig. 3(a). We make a challenging setting for BFC and Aquarius by assuming there is only one available queue on the congested egress port[2]. Additionally, f_5 is occasionally mapped to the same queue as f_4 on the upstream switch, as shown in Fig. 3(b). As BFC controls traffic within a queue as a whole, all flows ($f_1 \sim f_5$) are paused, resulting in unfair degradation to f_2, f_3, and f_5.

By contrast, Aquarius employs *contribution-aware pausing* to mimic the ideal FC's behavior by allowing independent and precise control of various flows. To achieve this, Aquarius records the size of each flow and checks if it exceeds the fair occupation ($Occu_{fair}$) of the egress queue (Line 11). When a flow exceeds this fair occupation, it is responsible for the congestion and should be paused. As depicted in Fig. 3(c), Aquarius pauses flows f_1 and f_4 while allowing flows f_2 and f_3 to continue uninterrupted. To calculate $Occu_{fair}$, Aquarius applies a hardware-friendly shifting operation:

$$Occu_{fair} = \lceil L >> \lceil \log_2 N \rceil \rceil, \tag{1}$$

where L is the egress queue length and N is the number of active flows on this queue. $Occu_{fair}$ denotes the equitable allocation of queue capacity among all active flows. In cases where the number of active flows is not a power of two, $Occu_{fair}$ is rounded up to prevent the under-regulation of flows.

Pause thresholds. Aquarius allows uncongested flows and a small portion of congested flow (detail in §3.3) to continue transmitting. Thus, two queue length thresholds, Q_{low} and Q_{high}, are set to prevent the downstream congested queue from becoming overwhelmed. When queue length L exceeds Q_{low} but is less than Q_{high}, and there is a possibility of congestion (Line 10), Aquarius will only pause the flow that is responsible for congestion. When L exceeds the conservatively large Q_{high}, Aquarius will pause all upstream flows to avoid severe buffer overflow (Line 7).

To determine whether congestion is possible, we use a time window T_{min}. If the current arriving interval T_{avg} is less than T_{min}, Aquarius determines that congestion will likely happen. We use the iteratively updated averaged T_{avg} instead of T to reduce the interferential noise (Line 14). T_{min} is given by $\frac{MTU*NQ_{active}}{\mu}$, where MTU is the maximum packet length, μ is the output capacity of the port, and NQ_{active} is the number of queues that are not paused in the port. We set Q_{low} and Q_{high} as $\frac{HRTT*\mu}{NQ_{active}}$ and $\frac{3*HRTT*\mu}{NQ_{active}}$, respectively, where $HRTT$ is the 1-Hop RTT. A pre-configured match-action table indexed with NQ_{active} and μ can be used to compute these values.

[2]This setting is common on congested switches because congestion is usually caused by the sudden appearance of a large number of flows, which results in more flows than available queues, and multiple flows are inevitably mapped to the same queue.

3.3 Opportunistic Re-assigning

As depicted in Figure 3, both flows f_4 and f_5 are initially mapped to the same upstream egress queue in BFC and Aquarius. However, BFC controls flows within the same queue as a whole, thus causing both f_4 and f_5 to be paused together, which results in unfair degradation to f_5. In contrast, Aquarius employs *opportunistic re-assigning* to further isolate f_4 and f_5, allowing f_5 to continue transmitting.

To be specific, the PAUSE message includes the QueueID and FID of the flow to be paused (f_{TBP}). Upon receiving the PAUSE message, the upstream switch first checks:

- If f_{TBP} solely occupies a queue: Aquarius pauses this queue.
- If f_{TBP} is sharing a queue with other un-congested flows: Aquarius directs the subsequent packets of f_{TBP} to a queue that is reserved to hold congested flows (`rsvQ`), and pauses `rsvQ`. Meanwhile, Aquarius allows the original queue to continue transmitting, thus leaving the un-congested flows unaffected.

Aquarius determines the queue's state using a QueueTable that records the number of flows mapped to each queue, FIDs of congested flows notified by downstream entities, *etc.*

The `rsvQ` can be dynamically allocated or statically configured. We adopt dynamic allocation in §4, where `rsvQ` is dynamically selected from empty queues when it is needed and has not been allocated. All congested flows share the same `rsvQ`. When `rsvQ` becomes empty, it is released to the initial *dynamic flow mapping*. The downstream switch notifies upstream switches with a RESUME message after its congestion is reduced, as shown in Algorithm 2. When all RESUME messages of the assigned congested flows are received, and the previously buffered packets are drained totally (to ensure in-order delivery), `rsvQ` is resumed.

Algorithm 2 Dequeueing Logics of Aquarius

```
1: function DEQUEUE(packet)
2:     key = <packet.egressPort, hash(packet.FID)>
3:     flowTable[key].size -= 1
4:     if packet.congested == true then
5:         flowTable[key].pauseNum -= 1
6:         if flowTable[key].pauseNum == 0 then
               // Resume upstream entity
7:             send RESUME(packet.q, packet.FID)
```

Further optimization. To prevent victim flows from being blocked, the original queue must continue transmitting. The already buffered packets of f_{TBP} (congested) are forced to arrive at the downstream switch because they are not allowed to re-enter the ingress pipeline in today's programmable switches. Therefore, a small portion of f_{TBP} is not paused, as shown in Figure 3(c). On switches that support multiple egress pipelines [30], we can direct the buffered packets to a separating next-level egress queue, achieving complete control over congested flows. This approach provides additional optimization for managing congested flows.

4 PRELIMINARY RESULT

We perform NS-3 simulations to evaluate the performance of Aquarius under both micro-benchmark and realistic traffic loads. We compare Aquarius with BFC [8].

Micro-benchmark. To validate the effectiveness of Aquarius, we simulate a typical micro-benchmark. The simulation setup is illustrated in Fig. 4, where flows $f_1 \sim f_3$ are directed to R_1, while f_4 goes to R_2. All links in the network have a capacity of 100Gbps and a propagation delay of 1μs. We limit the transmission rate of f_1 to 1/3 line rate (33Gbps) and allow f_2 to f_4 to transmit at line rate (100Gbps). Consequently, R_1 becomes the bottleneck because its input rate significantly exceeds the link capacity, causing a queue buildup on its connected L_1 port. We limit $f_1 \sim f_3$ to the same egress queue on L_1, and initially map f_3 and f_4 to the same egress queue on L_2, while all switches have 8 queues per port.

We measure the average throughput of flows f_1 to f_4 and compared Aquarius with BFC. BFC manages flows within the same queue as a whole and pauses f_1 and f_2 too frequently and for too long. Because f_3 and f_4 share the same queue, BFC inevitably pauses f_4. As a result, BFC causes throughput degradation to f_1, f_2, and f_4, as shown in Fig. 5(b). In contrast, Aquarius achieves a relatively fair partition of bottleneck link capacity between f_1 to f_3, despite their different transmission rates. This is because Aquarius determines that f_2 contributes the most to the congestion and proactively pauses it for a longer time than f_1 and f_3. Moreover, when the PAUSE message is propagated to L_2, Aquarius further isolates f_3 to a separating queue and only pauses f_3, ensuring the throughput of f_4 is not affected.

Realistic Web Server traffic. We simulate a 3-layer fat-tree topology comprising 48 switches and 128 servers, with a 1:1 oversubscription ratio. All links are equipped with 100Gbps capacity and have a propagation delay of 1μs. The switch applies Equal-Cost Multi-Path (ECMP) as load balancing and has a 12MB total buffer. We use a synthetic Web Server workload with a 70% average load, as

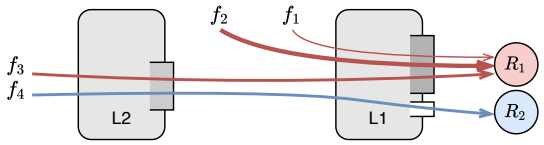
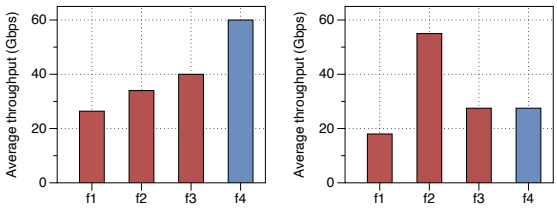


Figure 4: Micro-benchmark setting.



(a) Aquarius **(b) BFC**

Figure 5: Average throughput for flows $f_1 \sim f_4$.

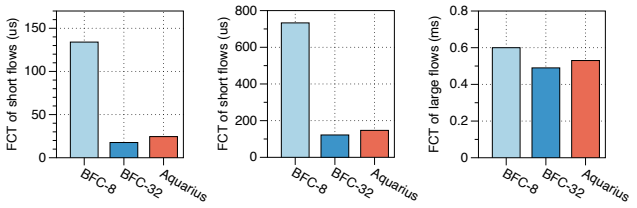
well as 5% load 100-to-1 incast traffic with sizes ranging from 50KB to 200KB. Aquarius is configured with 8 queues per port and compared with BFC-8 and BFC-32. Both 8- and 32-queue configurations have the same transmitting capacity per port.

We measure the average and 99^{th} percentile tail FCT of short flows (< 3KB), and average FCT of large flows (> 1MB). The results depicted in Fig.6 demonstrate that Aquarius significantly reduces FCT compared to BFC-8 and is comparable to BFC-32. For instance, compared to BFC-8, Aquarius achieves approximately 5.5× lower average (Fig. 6a) and 5× lower 99^{th} percentile tail (Fig. 6b) FCT for short flows. Additionally, Aquarius attains a 15% faster completion time for large flows (Fig. 6c). Compared to BFC-32, Aquarius achieves comparable FCT for both short and large flows, with 4× fewer queues. This performance advantage of Aquarius stems from the ability to independently and accurately control the congested flows, reducing the interference to uncongested flows.

5 DISCUSSION & FUTURE WORK

Efficient flow table structure. Aquarius maintains flow states (*i.e.* FlowTable) using an array without collision resolution, similar to BFC. While using an array has the advantage of simple operations, the size of the FlowTable must be set to several hundred times the number of flows to limit the hash collision probability. This results in inefficient utilization of FlowTable entries, leading to nontrivial state overhead. To overcome this limitation, we plan to explore more hash table structures with effective collision resolution and constant computation overhead as part of our future work.

Further analysis and hardware feasibility. We plan to conduct additional experiments to gain a deeper understanding of Aquarius. This includes investigating its sensitivity to parameters, evaluating the performance impact of using the dynamically- *vs.* statically-allocated reserved queue, and determining the probability of an upstream switch having available queues. Furthermore, we will implement Aquarius on commodity programmable switches and measure the hardware resource overhead of Aquarius to estimate its hardware feasibility and packet processing capability.



(a) Average **(b) 99^{th} percentile** **(c) Average**

Figure 6: Average and 99^{th} percentile tail FCT for short flows, and average FCT for large flows, under Web Server distribution with 70% load and 5% 100-1 incast.

6 RELATED WORK

Priority-based flow control (PFC). PFC [23] is an Ethernet network protocol used to ensure lossless data transmission. However, PFC is known to have several drawbacks, including head-of-line (HOL) blocking, unfairness, congestion spreading, and deadlock. There are numerous works have attempted to eliminate PFC's drawbacks [12, 28, 31, 32]. While PFC enables different traffic to be buffered in separating queues, the mapping between traffic and queues is mostly statically configured using constant packet tags. In contrast, Aquarius dynamically maps flows to available queues and opportunistically isolates flows as necessary, providing a more flexible and efficient solution.

Queue scheduling and switch buffer management. Some works propose various queue scheduling methods to address the negative impact of large flows on the latency of short flows. Examples include fair queuing approximation [25] and priority-based scheduling [4]. Additionally, switch buffer management schemes [2] utilize various switch buffer characteristics to determine the admission threshold for each queue. These methods alone cannot reduce buffer occupancy and are orthogonal to Aquarius.

Proactive congestion control. A famous line of works [5, 10, 11, 21] has adopted a proactive congestion control scheme where receivers allocate credits to senders in advance of data transmission. However, receivers can only perceive congestion at the network's edges and cannot feel and proactively control congestion within the network. In contrast, Aquarius utilizes switches, which have a full understanding of both in-network and edge congestion, to make accurate and timely control decisions.

7 CONCLUSION

In this paper, we introduce Aquarius, a scalable solution for enhancing switch-driven flow control and making it more practical. Aquarius takes independent control of different flows that are mapped to the same queue, thereby achieving satisfactory flow-level isolation with a number of queues that is much less than the number of flows. Experimental results demonstrate that Aquarius effectively achieves fair partitioning in bottleneck flows and significantly reduces the tail FCT for short flows under realistic workloads.

ACKNOWLEDGMENTS

We thank our anonymous reviewers for their insightful comments. This work is supported in part by the Key-Area Research and Development Program of Guangdong Province (2021B0101400001), the Hong Kong RGC TRS T41-603/20R, and the GRF 16213621.

REFERENCES

[1] Sepehr Abdous, Erfan Sharafzadeh, and Soudeh Ghorbani. 2021. Burst-tolerant datacenter networks with vertigo. In *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*. 1–15.
[2] Vamsi Addanki, Maria Apostolaki, Manya Ghobadi, Stefan Schmid, and Laurent Vanbever. 2022. ABM: active buffer management in datacenters. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 36–52.
[3] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*. 63–74.
[4] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pfabric: Minimal near-optimal datacenter transport. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 435–446.
[5] Inho Cho, Keon Jang, and Dongsu Han. 2017. Credit-scheduled delay-bounded congestion control for datacenters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 239–252.
[6] James Davidson, Benjamin Liebald, Junning Liu, Palash Nandy, Taylor Van Vleet, Ullas Gargi, Sujoy Gupta, Yu He, Mike Lambert, Blake Livingston, et al. 2010. The YouTube video recommendation system. In *Proceedings of the fourth ACM conference on Recommender systems*. 293–296.
[7] Google. 2023. Google Cloud Platform. https://cloud.google.com
[8] Prateesh Goyal, Preey Shah, Kevin Zhao, Georgios Nikolaidis, Mohammad Alizadeh, and Thomas E. Anderson. 2022. Backpressure Flow Control. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 779–805. https://www.usenix.org/conference/nsdi22/presentation/goyal
[9] Sangtae Ha, Injong Rhee, and Lisong Xu. 2008. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS operating systems review* 42, 5 (2008), 64–74.
[10] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. 2017. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 29–42.
[11] Shuihai Hu, Wei Bai, Gaoxiong Zeng, Zilong Wang, Baochen Qiao, Kai Chen, Kun Tan, and Yi Wang. 2020. Aeolus: A building block for proactive transport in datacenters. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 422–434.
[12] Shuihai Hu, Yibo Zhu, Peng Cheng, Chuanxiong Guo, Kun Tan, Jitendra Padhye, and Kai Chen. 2017. Tagger: Practical PFC deadlock prevention in data center networks. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*. 451–463.
[13] Intel. 2023. Intel Tofino 2. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html
[14] Lin Jin, Shuai Hao, Haining Wang, and Chase Cotton. 2019. Unveil the hidden presence: Characterizing the backend interface of content delivery networks. In *2019 IEEE 27th International Conference on Network Protocols (ICNP)*. IEEE, 1–11.
[15] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, et al. 2020. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 514–528.
[16] NT Kung and Robert Morris. 1995. Credit-based flow control for ATM networks. *IEEE network* 9, 2 (1995), 40–48.
[17] Mu Li, David G Andersen, Alexander J Smola, and Kai Yu. 2014. Communication efficient distributed machine learning with the parameter server. *Advances in Neural Information Processing Systems* 27 (2014).
[18] Songze Li, Mohammad Ali Maddah-Ali, and A Salman Avestimehr. 2015. Coded mapreduce. In *2015 53rd Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE, 964–971.
[19] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. 2019. HPCC: High precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*. 44–58.
[20] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-based congestion control for the datacenter. *ACM SIGCOMM*

Computer Communication Review 45, 4 (2015), 537–550.

[21] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. 2018. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 221–235.

[22] NS-3. 2023. A discrete-event network simulator for internet systems. https://www.nsnam.org/

[23] IEEE 802.1 Qbb. 2011. Priority-based Flow Control. https://1.ieee802.org/dcb/802-1qbb/

[24] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. 2015. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. 123–137.

[25] Naveen Kr Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. 2018. Approximating fair queueing on reconfigurable switches. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 1–16.

[26] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. 2021. Atoll: A scalable low-latency serverless platform. In *Proceedings of the ACM Symposium on Cloud Computing*. 138–152.

[27] Parvin Taheri, Danushka Menikkumbura, Erico Vanini, Sonia Fahmy, Patrick Eugster, and Tom Edsall. 2020. RoCC: robust congestion control for RDMA. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*. 17–30.

[28] Chen Tian, Bo Li, Liulan Qin, Jiaqi Zheng, Jie Yang, Wei Wang, Guihai Chen, and Wanchun Dou. 2020. P-PFC: Reducing tail latency with predictive PFC in lossless data center networks. *IEEE Transactions on Parallel and Distributed Systems* 31, 6 (2020), 1447–1459.

[29] Zhuolong Yu, Chuheng Hu, Jingfeng Wu, Xiao Sun, Vladimir Braverman, Mosharaf Chowdhury, Zhenhua Liu, and Xin Jin. 2021. Programmable packet scheduling with a single queue. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 179–193.

[30] Zhuolong Yu, Jingfeng Wu, Vladimir Braverman, Ion Stoica, and Xin Jin. 2021. Twenty Years After: Hierarchical Core-Stateless Fair Queueing.. In *NSDI*. 29–45.

[31] Yiran Zhang, Yifan Liu, Qingkai Meng, and Fengyuan Ren. 2021. Congestion detection in lossless networks. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 370–383.

[32] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion control for large-scale RDMA deployments. *ACM SIGCOMM Computer Communication Review* 45, 4 (2015), 523–536.