ResQueue: A Smarter Datacenter Flow Scheduler

Hamed Rezaei
University of Illinois at Chicago
hrezae2@uic.edu

Balajee Vamanan
University of Illinois at Chicago
bvamanan@uic.edu

ABSTRACT

Datacenters host a mix of applications: foreground applications perform distributed lookups in order to service user queries and background applications perform batch processing tasks such as data reorganization, backup, and replication. While background flows produce the most load, foreground applications produce the most number of flows. Because packets from both types of applications compete at switches for network bandwidth, the performance of applications is sensitive to scheduling mechanisms. Existing schedulers use flow size to distinguish critical flows from non-critical flows. However, recent studies on datacenter workloads reveal that most flows are small (e.g., most flows consist of only a handful number of packets). In light of recent findings, we make the key observation that because most flows are small, flow size is not sufficient to distinguish critical flows from non-critical flows and therefore existing flow schedulers do not achieve the desired prioritization. In this paper, we introduce *ResQueue*, which uses a combination of *flow size and packet history* to calculate the priority of each flow. Our evaluation shows that ResQueue improves tail flow completion times of short flows by up to 60% over the state-of-the-art flow scheduling mechanisms.

CCS CONCEPTS

• **Networks** → **Transport protocols**; **Network protocol design**.

KEYWORDS

Datacenter Networks, Congestion Control, Flow Scheduling

ACM Reference Format:
Hamed Rezaei and Balajee Vamanan. 2020. *ResQueue*: A Smarter Datacenter Flow Scheduler. In *Proceedings of The Web Conference 2020 (WWW '20), April 20–24, 2020, Taipei, Taiwan.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3366423.3380012

WWW '20, April 20–24, 2020, Taipei, Taiwan

ACM ISBN 978-1-4503-7023-3/20/04.
https://doi.org/10.1145/3366423.3380012

1 INTRODUCTION

Modern datacenters host user-facing, foreground applications that query large amounts of Internet data (e.g., Web Search) and background applications that perform background tasks such as data reorganization, replication, and backup [7, 8]. Foreground applications such as Web Search access data that is distributed among hundreds of servers. Every Web Search query must wait for a response from most of the worker servers (e.g., 99% of servers) to achieve a good trade-off between query response time and result quality. Therefore, foreground applications' performance is sensitive to higher percentiles (i.e., 99^{th} to 99.9^{th}) of Flow Completion Times (FCT) [13]. While foreground applications generate mostly short flows (e.g., 1KB to 10KB), background applications transfer large amounts of data (e.g., 1MB to 100MB) across the network, which requires high throughput [2]. Thus, a well-designed datacenter network must provide low tail flow completion times for short flows and high throughput for long flows.

The problem is challenging because both short flows (from foreground applications) and large flows (from background applications) compete for network bandwidth at switches. It is likely for short flows to get stuck behind several long flows and suffer elongated tail FCTs. Similarly, long flows incur packet loss from competing bursts of short flow packets and lose throughput. Recent studies from Microsoft [5] and Facebook [28] show the extent of packet losses in their networks.

Load balancing approaches [3, 16, 18, 19], congestion control approaches [1, 4, 12, 15, 17, 23, 25, 30, 31, 33], and packet scheduling approaches [2, 6, 9, 14, 21, 22, 26] all address this problem, either directly or indirectly. However, when most flows are short, packet scheduling approaches tend to be most effective because there is little time for load balancing and congestion control to kick in. Indeed, recent studies show that the average flow size of typical datacenter workloads such as Web Search is less than one kilobyte [28, 32]. Motivated by these recent studies, we, therefore, focus our attention on flow scheduling.

One of the well-known solutions for flow scheduling is to use multi-level queues at switches that gradually demote flows from high to low priority (i.e., each flow starts at the highest priority queue and moves down in priority after sending some packets) [6]. However, recent studies on production datacenters show that flows in Web Search workload

are extremely small such that about 75% of the flows only send a single packet to the receiver [28, 32]. This study reveals that *flow size* is insufficient to effectively distinguish between flows and prioritize critical flows over other flows.

There is another issue that flow schedulers must address. In Web Search, gathering data from several servers synchronously causes *incast* congestion, which leads to performance degradation. Incast causes rapid queue buildup at the switch port connected to the receiver server (e.g., front-end server) [11]. A recent study reports that about 85% of incasts last for 50 microseconds or less, which is smaller than Round Trip Times (RTT) of most datacenter networks [32]. Moreover, the study shows that incast inter-arrival time is so short that retransmitted packets are likely to collide with another incast. Ideally, we would like to schedule flows such that minimum number of packets are dropped during incast, which would lead to shorter tail flow completion times.

We make the *key* observation that previous flow scheduling mechanisms are inefficient when short flows have only a handful number of packets to send and they do not distinguish between those flows that are already delayed and those that are not. Because most flows are short and their sizes are similar, the schedulers give the same priority to a large fraction of flows regardless of their *drop* history. For example, when two flows that have only one packet to send arrive at a switch while one of them being a retransmitted packet, existing schedulers (e.g., PIAS[6], UPS[22]) assign both packets to the same priority queue. Clearly, in this instance, it makes more sense to give a higher priority to the flow that has already suffered a retransmission. Moreover, recent studies [32] show that incast inter-arrival time is unpredictable and sometimes extremely short, which increases the chance of collision between a retransmitted packet and other packets. As such, we need a flow scheduling mechanism that ensures that retransmitted packets do not suffer repeated packet drops during incast. Therefore, we argue that modern flow schedulers should prioritize retransmitted packets, in addition to prioritizing shorter flows over the longer ones.

We propose *ResQueue*, a novel datacenter flow scheduling scheme, which is broadly applicable to many incast-heavy foreground applications, including Web Search, Data mining, and social networks. *ResQueue* uses a combination of packet drop history and flow size to determine each flow's priority. *ResQueue* tags packets at the endhost based on number of bytes sent; if the packet is a retransmitted packet, the endhost performs one additional step: it subtracts *one* from the calculated priority in the previous step (based on sent bytes) to give a higher priority to this packet. At switches, since short flows have only a handful number of packets to send, all their retransmitted packets will be scheduled in queue level-1, which is the highest priority queue. This queue is reserved for retransmitted packets only. Thus, *ResQueue* ensures that retransmitted packets are not dropped again if they collide with a burst of packets. By avoiding repeated packet drops, *ResQueue* improves both tail FCTs of short flows and throughput of long flows, as we show in our results section (Section 4).

In summary, we make the following contributions:

- *ResQueue* is the first scheduler to consider packet drop history in flow scheduling.
- *ResQueue* prioritizes retransmitted packets, in addition to short flows, which helps both flow completion times of short flows and throughput of long flows.
- *ResQueue* achieves up to 60% lower tail flow completion times compared to state-of-the-art flow schedulers without any loss of throughput with typical datacenter workloads.

2 MOTIVATION

New studies on datacenter networks reveal that Web search workload is not only bursty, but it is also dominated by extremely short flows [28, 32]. Their analysis shows that while about 75% of the Web search flows are nearly 1KB (figure 6.a in [28]), about 90% of them are smaller than 6KB. Given that state-of-the-art flow scheduling methods such as UPS [22] and PIAS [6] rely on flow size to prioritize some flows over the others, these methods give the same priority to about 80% of the flows, which creates inefficiencies in flow prioritization and leads to performance degradation in datacenter networks.

In Web search workloads, packets might be dropped more than once due to traffic burstiness. Most previous datacenter flow schedulers are designed to look at the flow size only to determine the flow priority, and, therefore, they fail to detect/prioritize the packets that are dropped more than once. We designed an experiment to see how the packet drop rate varies among different loads. Our at-scale ns-3 simulation shows that a packet could be dropped up to *seven* times at high loads. This is because when a packet is retransmitted, it might repeatedly collide with a burst of packets, which leads to excessive packet drops.

Most of previous solutions work when short flows are relatively large in size (i.e., 10KB-100KB). As an example, PIAS [6] demotes flows from higher priority queues to lower priority queues based on their *bytes sent*, but it cannot use the same approach if *most* flows are 1KB in size, as is the case in many datacenter workloads[28]. Clearly, in this instance, all packets get the same priority. Also, UPS [22] uses Least Slack Time First (LSTF) as its main mechanism for determining flow priority. However, due to the same reason, this method will not perform well if all flows are extremely small. Also,

there is a large body of deadline-based flow scheduling methods (i.e., [2], [30], [31], etc) that perform well when flow sizes are not very small. Further, it is not easy to get information on flow deadlines in many datacenters. A new study [29] reveals that flow characteristics such as flow deadline and flow size are often unknown prior to their transmission.

We introduce *ResQueue*, which does not require prior knowledge about the flows. It is designed to address the shortcomings of previous approaches. It prioritizes those packets that were dropped before and schedules them in higher priority queues in switches. *ResQueue* is extremely effective and easy to implement in real datacenter networks.

3 DESIGN

In this section, we discuss *ResQueue* that detects those packets that were dropped in the past, and prioritizes them over packets of that same size that did not incur loss. This is critically important because the performance of foreground applications (e.g., Web search) is highly sensitive to packet loss rate as TCP timeouts are multiple orders of magnitude larger than typical flow completion times. Our at-scale simulations show that while a large fraction of the Web application flows are delivered to the destination server in less than 100 microseconds, others that are dropped more than once could be delivered after 3-4 milliseconds (assuming RTO = 1ms). Bellow we discuss the mechanism of *ResQueue*, which is designed to rescue those packets that belong to short flows from being dropped again.

3.1 *ResQueue*

We introduce *ResQueue*, which is a novel flow scheduling method that is designed to give more accurate priority to flows based on *their size* and *their history of packet drops*. At its core, similar to PIAS [6], *ResQueue* implements a Multi Level Feedback Queue (MLFQ) at switches, in which a flow is demoted from higher-priority queues to lower-priority queues according to their number of sent bytes. The priority of each flow is calculated based on the number of bytes sent, and then, this priority is tagged on the packet by the sender server. In MLFQ mechanism, if there are N queues in the switch, all flows will be scheduled in the first queue when they start the data transmission (bytes sent = 0). Flow priority demotes when the flow has already sent some packets and exceeds a predefined threshold. Packets that are scheduled in the second queue will not be dequeued unless all packets in the first queue are dequeued. Due to space constraints, we do not provide the details of priority demotion here. The whole mechanism is explained in [6].

In *ResQueue*, we reserve the first queue for those *retransmitted* packets that belong to short flows (the size of this queue is 20KB in our experiments. We further discuss the size of reserved buffer in section 4). We do this reservation for only short flows because Web search workload is dominated by short flows, and, therefore, performance of the whole network will be determined by these short flows.

If the sender server is transmitting a normal packet, *ResQueue* acts like PIAS. However, if the sender server is *retransmitting* a packet (or a window of packets), it calculates the priority of the packet(s) in **two** rounds: first, it finds the appropriate queue level based on its number of sent bytes (as in [6]), and, second, it subtracts *one* from the calculated value because of its drop history. For example, if a flow has only one packet to send and this packet was dropped before, its bytes sent indicates that the packets should be enqueued in queue level-2; however, the server tags the retransmitted packet with priority 1 due to its drop history, and, therefore, all switches will enqueue this packet in queue level-1. We provision a reserved high-priority queue *only* for short flows that experienced packet loss. As in [32], Web flows are so small that they are often enqueued either in level-1 or level-2 queues, depending on their drop history.

Note that although *ResQueue* does not schedule large flows in high priority queue, it still improves throughput of these flows. While those large flows that experience one or more packet drops in the past will not collide with short flows, they get higher priority compared to other same-size flows that did not suffer from packet drops in the past. Imagine that a large flow's packets are scheduled in queue level-4, and, meanwhile, a window of its packets get dropped due to high congestion. When the window of packets is retransmitted, they will be scheduled in queue level-3 to avoid further delays to these packets. Although queue level-3 is not reserved for dropped packets, packets in this queue will drain faster than packets in queue level-4, which improves overall throughput to some extent.

Figure 1 depicts a comprehensive example of *ResQueue*'s mechanism. As we see in figure 1, the sender is retransmitting the red packet that was dropped in the previous round. Although the first round of priority calculation indicates that this packet should be scheduled in queue level-2 (based on sent bytes), dropped flag is set for this packet, which leads to priority escalation. Therefore, when the red packet arrives at the switches, this packet will be scheduled in an isolated high priority queue (queue level-1), which guarantees that this packet will not collide with a burst of packets again.

For the best performance, all servers across the datacenter and all switches in the path from source to destination need to support *ResQueue* to save the retransmitted packets. *ResQueue*'s implementation is similar to PIAS — *ResQueue* tags packets with priority information based on bytes sent at end-hosts; upon timeouts, the priority is decremented.

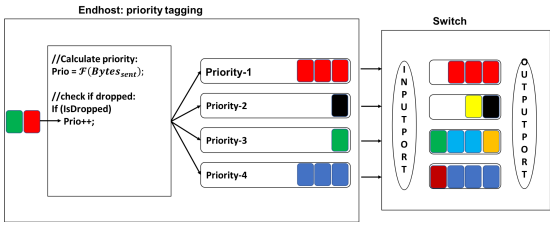

Figure 1: Retransmitted packet gets higher priority

3.2 Why *ResQueue* works?

Recent studies on datacenter networks show that incast inter-arrival time is so small that about 30% of the incasts arrive at most 50 microseconds after the previous one [32]. Considering this traffic pattern and the size of Web search flows in modern datacenter networks, retransmitted packets are likely to collide with a burst of packets, which causes excessive packet drops. Note that RTT in datacenter networks is usually higher than 50 microseconds, which further complicates the situation. Because flows are so small that they last only for a handful of RTTs (assuming drops), prioritizing retransmitted packets would save them from being dropped again after colliding with a burst of packets. Although there are not too many of retransmitted packets that collide with a burst of packets, it still negatively affects the higher percentiles (i.e., 99^{th} percentile) of flow completion times. Note that in datacenter networks we only care about tail (i.e., 99^{th} to 99.9^{th}) flow completion time of short flows. Therefore, *ResQueue* has the potential to play a crucial role in improving tail flow completion times of short flows.

3.3 Why not prioritize flows instead of packets?

If a flow already suffered from packet drops in the past, the performance would be higher if the flow remains prioritized until it finishes. However, there are some challenges that we need to address if the *whole* flow remains prioritized:

- If the whole flow is prioritized, large flows may remain prioritized for a long time, which causes performance degradation to short flows.
- We need a relatively large table at the endhost to track the flows that already suffered from packet drops. This mechanism requires a high number of unnecessary operations at the endhost.

There is at least one solution for the first challenge. For instance, the endhost can increase flow priority if larger number of packets are dropped, and decrease the priority if larger number of packets remained to be transferred. However, this solution is indeed complicated and needs to be further analyzed.

For the second challenge, our key insight is that because most short flows only consist of handful of packets (1-2 packets), we can consider each packet as a separate flow without adding complexity to the system. Therefore, prioritizing *retransmitted packets* would be efficient and easier to implement.

3.4 Packet reordering

In this section we investigate the effect of *ResQueue* on packet reordering at the receiver server. Packet reordering is an important issue as it directly affects the flow completion time of the flow. If some packets arrive out of order, the receiver server needs to spend considerable number of CPU cycles on reordering them, which further delays the flow completion time.

ResQueue only schedules *retransmitted* packets in the higher priority queue. These packets are already out of order and *ResQueue*'s prioritization mechanism guarantees that they will not arrive at the receiver later than the new window of packets. Thus, *ResQueue* does not worsen packet reordering. In fact, *ResQueue* somewhat alleviates packet reordering by making those late packets to arrive earlier.

4 EVALUATIONS

In this section, we discuss the details of our testbed and workloads, and we provide an exhaustive analysis of *ResQueue*'s performance in terms of flow completion time, throughput, and packet drop rate.

4.1 Experimental Methodology

We use ns-3 [27] simulator to simulate a leaf-spine datacenter topology, which is common in datacenters [2]. In our topology, the fabric interconnects 400 servers through 20 leaf switches that are connected to 10 spine switches (i.e., there is an over-subscription factor of 2). All links are 10 Gbps and the round trip time across the network is 80 microseconds. We use workload characteristics reported in recent studies [28, 32] to create a realistic traffic in our simulations. Hence, our short flows' sizes are in the range of 1KB to 6KB, while 80% of the flows are 1KB in size. Also, we use 1 MB flows as our large flows. All servers are spread across the network uniformly randomly and the switches use shallow buffers as suggested in prior work (e.g., [2]).

4.2 Flow Completion Times

Tail flow completion time is the key determinant of application performance in datacenters. Thus, we only evaluate *ResQueue*'s performance in lowering *tail* flow completion time of short flows. Note that median flow completion time

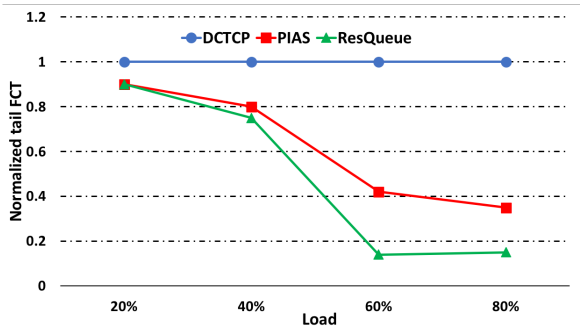

Figure 2: Normalized tail flow completion time of short flows

will be the same in both PIAS and *ResQueue*, as *ResQueue* targets dropped packets only, which affects higher percentiles of flow completion time.

Our experiments show that *ResQueue* improves performance of PIAS (in terms of tail flow completion time) by a factor of 1.6x for loads greater than 20%, on average. Figure 2 shows the result of our experiments when incast degree (i.e., number of concurrent senders) is 32. We discuss more about sensitivity of *ResQueue*'s performance to different incast degrees later in this section.

As we see in figure 2, although PIAS outperforms DCTCP in lowering tail flow completion times, *ResQueue* lowers tail flow completion times more by reducing the total number of packet drops. In particular, *ResQueue* achieves impressive gains of over 2x at higher loads ($\geq$ 60%) over PIAS.

4.3 Throughput

Although *ResQueue* is mainly designed for lowering tail flow completion times of short flows, it improves the throughput of large flows as well. *ResQueue* schedules dropped packets in a higher priority queue depending on their size. Therefore, when some large flows compete for bandwidth, those packets that were previously dropped will get higher priority. For example, if a packet should be scheduled in queue level-4 (based on flow's bytes sent), it will be scheduled in queue level-3 if and only if it is a retransmitted packet. This mechanism will expedite the dropped packets, which provides higher throughput for large flows.

Figure 3 shows throughput comparison between PIAS and *ResQueue*. While the absolute throughput decreases when load increases due to higher contention with other flows at switches, *ResQueue*'s throughput remains higher than PIAS even at high loads. Instead of repeatedly dropping packets from a small subset of flows, ResQueue distributes packet losses to more flows. As a result, more senders throttle their rates, which helps alleviate saturation. Figure 3 shows that *ResQueue* 's relative advantage over PIAS increases with load.

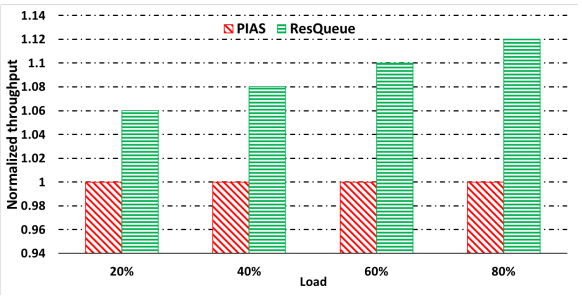

Figure 3: Normalized throughput of large flows

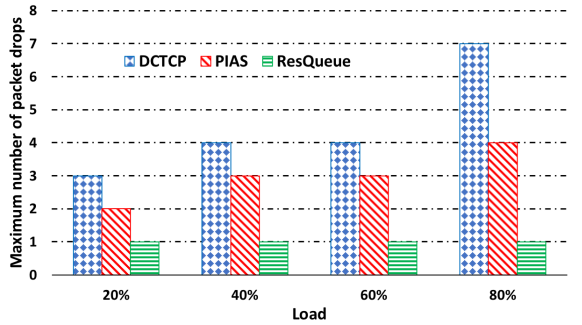

Figure 4: Maximum drops for any packet

Overall, *ResQueue* improves the throughput of large flows by a factor of 1.08x relative to competition, for loads greater than 20%, on average.

4.4 Packet drop rate

In this section, we analyze packet losses in DCTCP, PIAS, and *ResQueue*. Figure 4 shows the maximum number of packet drops for *any* packet in DCTCP, PIAS, and *ResQueue* for different loads. We clearly see that DCTCP and PIAS suffer higher packet loss than *ResQueue* and their packet drops worsen with load. In contrast, a packet does not get dropped more than once with *ResQueue*, even at high loads.

Note that our highest priority level is reserved for retransmitted packets only and small flow packets start from the second level and get promoted to the highest priority level upon loss. Because retransmissions constitute a small fraction of packets, we do not observe losses in the highest priority queue (i.e., maximum number of packet drops for any packet in Figure 4 for *ResQueue* is 1 across all loads).

4.5 Sensitivity Analysis

4.5.1 Sensitivity to incast degree. In this section, we evaluate *ResQueue*'s sensitivity to incast degree. Figure 5 shows the normalized tail flow completion times of *ResQueue* compared to PIAS (PIAS' tail FCT for each incast degree is normalized

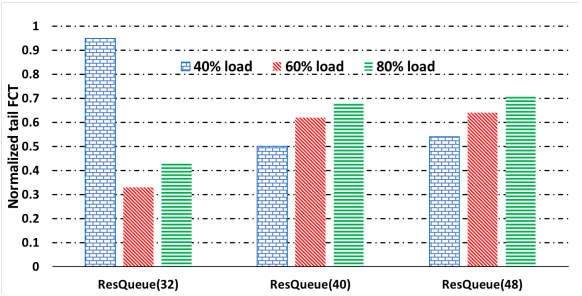

Figure 5: Sensitivity of *ResQueue* to incast degree (normalized FCT is 1 for PIAS)

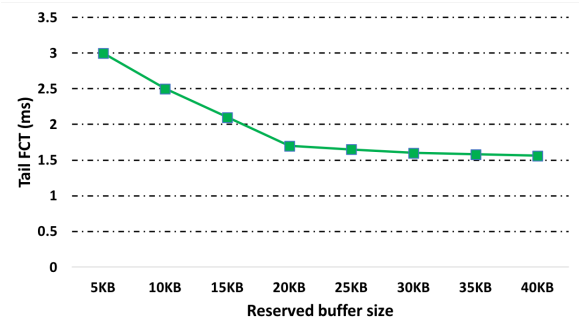

Figure 6: Sensitivity of *ResQueue*'s performance to size of the reserved buffer

to 1). We vary the incast degree as 32 (our default incast degree), 40, and 48 along X-axis, for different loads from 40% to 80% (typical operating point of datacenter networks). As expected, the tail flow completion times increase when incast degree and load increase. Overall, *ResQueue* achieves an average reduction in tail flow completion times by a factor of 1.6x, for loads greater than 40%. Thus, *ResQueue* achieves improvement over PIAS (i.e., normalized FCT less than 1) irrespective of incast degree and load.

4.5.2 Sensitivity to size of reserved buffer. *ResQueue*'s main mechanism is to escalate priority of dropped packets by storing them in a higher priority queue. Since we reserve the queue level-1 for those retransmitted packets that belong to short flows, we need to measure the buffer size that we need to reserve to achieve the highest performance. Because most flows are short, they are highly likely to use this buffer during congestion.

Our analysis shows although a retransmitted packet is likely to collide with a burst of packets, the total number of retransmitted packets is not high. Figure 6 shows the sensitivity of *ResQueue* to size of the reserved buffer. As we see in the figure, there is no difference between performance of *ResQueue* and PIAS, if the size of reserved buffer is smaller than 5KB. Likewise, sizes larger than 20KB do not improve the performance of *ResQueue* as the buffer goes unused. Therefore, we have a *sweet spot* between 15KB and 20KB that minimizes packet drops for short flows and achieves high throughput for large flows. Clearly, 20KB of dedicated buffer is not a big amount of buffer and can be easily provided by switch vendors. Modern datacenter switches are equipped with a large (4MB to 50MB) shared buffer that is shared among all ports. Reserving 20KB of a 4MB buffer contributes to only 0.5% of the total buffer, which is almost negligible. Thus, *ResQueue*'s overhead is minimal, and our performance is robust for a range of loads and workloads.

5 RELATED WORK

There is a large body of work on flow scheduling in datacenters. Most of these schemes either require extensive hardware modifications or they rely on prior knowledge about the flows (e.g., deadline). Flow scheduling methods can be divided into two groups: *information agnostic* flow schedulers and *information aware* flow schedulers. Earliest Deadline First (EDF) [20] is the earliest information-aware packet scheduling approach and has been proven to be optimal for minimizing deadline misses. D3 [31] and PDQ [17] proactively assign flow rates based on deadlines, whereas D^2TCP [30] and MCP [10] reactively adjust sending windows based on deadlines. pFabric [2], PASE [24], and UPS [22] prioritize packets based on flow sizes/deadlines. All these approaches require explicit flow sizes or deadlines, which is known to be hard to obtain in practice [29].

Among information agnostic flow schedulers, PIAS [6] and Slytherin [26] infer the priority of flows based on in-network mechanisms such as MLFQ and ECN marks. While information-agnostic approaches are easier to deploy than information-aware approaches, both these approaches are not effective in mitigating repeated packet drops, which occurs when most flows are small.

6 CONCLUSION

We presented *ResQueue*, which identifies delayed packets and prioritizes them in switches. Unlike prior approaches that rely only on flow size to determine priority, *ResQueue* uses a combination of flow size and packet drop history to infer priority. *ResQueue* improves tail flow completion times, which is the key metric for a broad class of user-facing datacenter applications. Further, *ResQueue* does not require hardware changes at switches. The current trend of increasing traffic burstiness in datacenters combined with the emphasize on low tail flow completion times necessitate schemes such as *ResQueue* that consider packet history in flow scheduling.

REFERENCES

[1] Mohammad Alizadeh et al. 2010. Data center TCP (DCTCP) *(SIGCOMM '10)*.

[2] Mohammad Alizadeh et al. 2013. pFabric: Minimal Near-optimal Datacenter Transport *(SIGCOMM '13)*. ACM.

[3] Mohammad Alizadeh et al. 2014. CONGA: Distributed Congestion-aware Load Balancing for Datacenters *(SIGCOMM '14)*.

[4] Hamidrezae Almasi, Hamed Rezaei, Muhammad Usama Chaudhry, and Balajee Vamanan. 2018. Pulser: Fast Congestion Response using Explicit Incast Notifications for Datacenter Networks. *arXiv preprint arXiv:1809.09751* (2018).

[5] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 2018. 007: Democratically finding the cause of packet drops. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 419–435.

[6] Wei Bai et al. 2015. Information-Agnostic Flow Scheduling for Commodity Data Centers.. In *NSDI*.

[7] Luiz André Barroso, Jeffrey Dean, and Urs Hölzle. 2003. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro* 23, 2 (March 2003), 22–28.

[8] Luiz Andre Barroso and Urs Hoelzle. 2009. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines* (1st ed.). Morgan and Claypool Publishers.

[9] Li Chen, Kai Chen, Wei Bai, and Mohammad Alizadeh. 2016. Scheduling Mix-flows in Commodity Datacenters with Karuna *(SIGCOMM '16)*.

[10] Li Chen, Shuihai Hu, Kai Chen, Haitao Wu, and Danny HK Tsang. 2013. Towards minimal-delay deadline-driven data center TCP. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*. ACM, 21.

[11] Yanpei Chen et al. 2009. Understanding TCP incast throughput collapse in datacenter networks. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*. ACM.

[12] Inho Cho, Keon Jang, and Dongsu Han. 2017. Credit-Scheduled Delay-Bounded Congestion Control for Datacenters. In *Proceedings of SIGCOMM*. 239–252.

[13] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56, 2 (Feb. 2013).

[14] Peter X. Gao et al. 2015. pHost: Distributed Near-optimal Datacenter Transport over Commodity Network Fabric. In *Proceedings of CoNEXT*. 1:1–1:12.

[15] Mark Handley et al. 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 (SIGCOMM '17)*. ACM.

[16] Keqiang He et al. 2015. Presto: Edge-based Load Balancing for Fast Datacenter Networks *(SIGCOMM '15)*.

[17] Chi-Yao Hong, Matthew Caesar, and P. Brighten Godfrey. 2012. Finishing flows quickly with preemptive scheduling. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication (SIGCOMM '12)*. ACM, 127–138. https://doi.org/10.1145/2342356.2342389

[18] Abdul Kabbani, Balajee Vamanan, Jahangir Hasan, and Fabien Duchene. 2014. FlowBender: Flow-level Adaptive Routing for Improved Latency and Throughput in Datacenter Networks. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies (CoNEXT '14)*. ACM, New York, NY, USA, 149–160.

[19] Naga Katta et al. 2016. HULA: Scalable Load Balancing Using Programmable Data Planes *(SOSR '16)*. ACM.

[20] Chung Laung Liu and James W Layland. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)* 20, 1 (1973), 46–61.

[21] Mojtaba Malekpourshahraki, Brent Stephens, and Balajee Vamanan. 2019. Ether: Providing both Interactive Service and Fairness in Multi-Tenant Datacenters. In *Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019*. ACM, 50–56.

[22] Radhika Mittal, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Universal Packet Scheduling. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation (NSDI'16)*. 501–521.

[23] 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 *(SIGCOMM '15)*. ACM.

[24] Ali Munir, Ghufran Baig, Syed M. Irteza, Ihsan A. Qazi, Alex X. Liu, and Fahad R. Dogar. 2014. Friends, Not Foes: Synthesizing Existing Transport Strategies for Data Center Networks. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14)*. ACM, New York, NY, USA, 491–502. https://doi.org/10.1145/2619239.2626305

[25] Hamed Rezaei, Muhammad Usama Chaudhry, Hamidreza Almasi, and Balajee Vamanan. 2019. ICON: Incast Congestion Control using Packet Pacing in Datacenter Networks. In *2019 11th International Conference on Communication Systems & Networks (COMSNETS)*. IEEE, 125–132.

[26] Hamed Rezaei, Mojtaba Malekpourshahraki, and Balajee Vamanan. 2018. Slytherin: Dynamic, network-assisted prioritization of tail packets in datacenter networks *(ICCCN'18)*. IEEE.

[27] George F Riley and Thomas R Henderson. 2010. The ns-3 network simulator. In *Modeling and tools for network simulation*. Springer, 15–34.

[28] Arjun Roy et al. 2015. Inside the social network's (datacenter) network. In *ACM SIGCOMM Computer Communication Review*. ACM.

[29] Vojislav Đukić, Sangeetha Abdu Jyothi, Bojan Karlaš, Muhsen Owaida, Ce Zhang, and Ankit Singla. 2019. Is advance knowledge of flow sizes a plausible assumption?. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*. 565–580.

[30] Balajee Vamanan, Jahangir Hasan, and T.N. Vijaykumar. 2012. Deadline-aware Datacenter TCP (D2TCP). In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '12)*.

[31] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. 2011. Better never than late: meeting deadlines in datacenter networks. In *Proceedings of the ACM SIGCOMM 2011 conference (SIGCOMM '11)*. ACM, New York, NY, USA, 50–61. https://doi.org/10.1145/2018436.2018443

[32] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. 2017. High-resolution measurement of data center microbursts *(IMC'17)*. ACM.

[33] Yibo Zhu et al. 2015. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of SIGCOMM*. 523–536.