SQR: In-network Packet Loss Recovery from Link Failures for Highly Reliable Datacenter Networks

Ting Qu*†,Raj Joshi†, Mun Choon Chan†, Ben Leong†, Deke Guo*, Zhong Liu*
*National University of Defense Technology, †National University of Singapore

***Abstract*—In datacenter networks, flows need to complete as quickly as possible because the flow completion time (FCT) directly impacts user experience, and thus revenue. Link failures can have a significant impact on short latency-sensitive flows because they increase their FCTs by several fold. Existing link failure management techniques cannot keep the FCTs low under link failures because they cannot completely eliminate packet loss during such failures. We observe that to completely mask the effect of packet loss and the resulting long recovery latency, the network has to be responsible for packet loss recovery instead of relying on end-to-end recovery. To this end, we propose Shared Queue Ring (SQR), an on-switch mechanism that completely eliminates packet loss during link failures by diverting the affected flows seamlessly to alternative paths. We implemented SQR on a Barefoot Tofino switch using the P4 programming language. Our evaluation on a hardware testbed shows that SQR can completely mask link failures and reduce tail FCT by up to 4 orders of magnitude for latency-sensitive workloads.**

I. Introduction

Datacenter computing is dominated by user-facing services such as web search, e-commerce, recommendation systems and advertising [1]. These are *soft real-time* applications because they are latency-sensitive and the failure to meet the response deadline can adversely impact user experience and thus revenue [1]. Such application-level deadlines can be translated into flow completion time (FCT) targets for the network communication between the *worker* processes that work together to serve the user requests [2]. There have been many proposals to reduce the FCTs of latency-sensitive flows for user-facing soft real-time applications under normal network conditions [1]–[6]. In this paper, we study the problem of reducing FCTs in the presence of link failures.

Link failures are common in datacenter networks [7] and they have outsized impact on short latency-sensitive flows. Such flows typically operate with a small TCP congestion window so when there is packet loss, the TCP receivers cannot send enough duplicate ACKs within one RTT [8]. As a result, fast retransmission is rarely triggered and the lost packets are often recovered via retransmission timeouts (RTOs) [9]. Such timeout events result in much larger delays than the lifespan of the short flows and significantly increase FCT [10].

A large number of approaches have previously been proposed to reduce the impact of link failures, including fast re-routing [11]–[15], flowlet-based load balancing [16], [17] and re-configurable topologies [18], [19]. All these approaches inevitably rely on *link failure detection* which has a minimum delay. Among them, the state-of-the-art ShareBackup [19] takes as little as 730 μs to recover from a link failure and it relies on F10's link failure detection technique [18] which has a delay of about 30 μs. This total delay of 760 μs on a 10 Gbps link translates to about 950 KB, and some 600 1500-byte packets could be lost. Even if we can reroute immediately after detecting the link failure (30 μs) using a pre-computed backup path, some 25 1500-byte packets could still potentially be lost. This suggests that while existing proposals can achieve low FCTs under normal network conditions, they cannot maintain or keep these low FCTs *stable* under link failures, even when using state-of-the-art link failure recovery techniques. In other words, in the face of link failures, the datacenter network stack today is unable to provide any bounds or strong reliability guarantees (up to five or six 9's) on FCTs or the network latency.

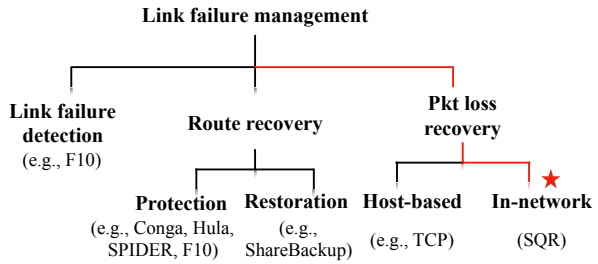


Fig. 1. Design space for link failure management.

We observe that to completely mask the effect of packet loss and the resulting long recovery latency, the network has to be responsible for packet loss recovery, instead of relying on end-to-end recovery. To this end, we propose **S**hared **Q**ueue **R**ing (SQR), an on-switch mechanism to recover packets that could be lost during the period from the detection of a link failure to the completion of the subsequent network reconfiguration. SQR is therefore complimentary to existing methods of link failure detection and route reconfiguration as shown in Fig. 1.

It is not possible to know in advance if a link will fail when a packet is sent, since link failures occur randomly and cannot be predicted [7]. We define the *route failure time* to be the time taken to detect a link failure and for the network to recover. We observe that by estimating the upper bound on the route failure time, a switch can cache a copy of recently sent packets for this duration. Then, in the event of a link failure, we can avoid packet loss by retransmitting the cached copy of these previously transmitted packets on the backup

978-1-7281-2700-2/19/$31.00 2019 © IEEE

Table I
ASIC PACKET BUFFER TRENDS

ASIC	Year	Packet Buffer
Trident+ [21]	2010	9 MB
Trident II [22]	2013	12 MB
Trident II+ [23]	2015	16 MB
Tomahawk+ [24]	2016	22 MB
Tomahawk II [25]	2017	42 MB

path. Naively, this can be implemented as a *delayed queue* that temporarily delays (stores) every packet passing through it for a configurable amount of time. When the link fails, we can retransmit the cached packets from this delayed queue. Unfortunately, no queuing engine today readily provides the queuing discipline required to realize such a delayed queue. Furthermore, existing queuing engines, including those in programmable ASICs, cannot be programmed to implement a custom packet scheduling algorithm that implements a delayed queue. To realize a delayed queue in any other way, the basic primitive required is packet storage. In a switch dataplane, even a programmable one, the packets can only be stored in the packet buffer of the queuing engine [20]. This packet buffer storage can only be utilized by placing packets into the default FIFO queues, which send out packets as fast as possible without introducing any delay.

In this paper, we describe a technique to emulate a delayed queue in the dataplane of a programmable switch. We do so by retaining a copy of a sent packet in a FIFO queue. If this packet reaches the head of the queue before being sufficiently delayed, we use egress processing to route this packet back into the FIFO queue. While this approach is, in principle, sufficient to emulate a delayed queue, it is challenging to ensure that no packet is missed out and the packets are retransmitted *in order*. Furthermore, there are two costs involved – the egress pipeline processing required to build and maintain the emulated delayed queue, and the additional packet buffer required for the packets in the delayed queue (i.e. the cached packets). A naive implementation could inflict additional egress processing delays on other flows going through the switch. SQR avoids this with a *Multi-Queue Ring* architecture that exploits unused egress processing capacity. The egress pipeline is provisioned to support all ports at full packet rate. In practice, most networks will almost always have spare egress processing capacity available [26]. We only use the idle ports so that other traffic passing through the switch is uninterrupted.

We implemented SQR on a Barefoot Tofino [27] switch[1]. We show using experiments on a hardware testbed using trace-driven workloads that:

- SQR can completely mask the effect of link failures from end-point transport by preventing packet loss;
- Coupled with current link failure detection (F10 [18]) and route reconfiguration schemes (ShareBackup [19]), SQR can reduce the tail FCT by 10 to 1000 times for web and data mining workloads in the presence of link failures; and

[1]A simple bmv2 version of SQR is available at: https://git.io/fjbnV

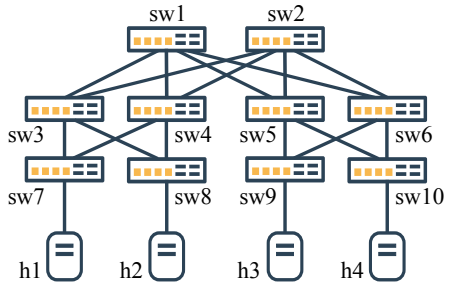


Fig. 2. Testbed.

- SQR's overhead in terms of packet buffer consumption, additional egress processing and ASIC hardware resources is low, thereby demonstrating the feasibility of our solution.

Gill et al. observed that network redundancy is not entirely effective in reducing the impact of link failures [7]. Our work addresses this gap by enabling a *seamless hand-off* of packets from a failed route to an alternative route, thereby fully exploiting available multi-path redundancy. To the best of our knowledge, we show, for the very first time, that it is possible to handle link failures without a single packet being lost or reordered in a multi-gigabit datacenter network. Our proposed approach was not previously feasible because switches would not have enough packet buffer to cache packets for the route failure time. However, recent innovations have substantially reduced the route failure time (65 ms in Portland [28] to 760 μs in ShareBackup [19]) so that the number of packets to be cached is significantly reduced. On the other hand, on-chip shared packet buffer for switching ASICs has increased more than fourfold over the last 5-7 years (see Table I), making in-network seamless packet hand-off practical.

II. MOTIVATION

Link failures are dominated by connection problems such as cabling and carrier signaling/timing issues [29]. Gill et al. observed that link failures were more common than device failures, and some 136 link failures were observed daily at the 95$^{\text{th}}$ percentile [7]. Link failures usually last for a few minutes and exhibit a high variability in their rate of occurrence.

For any solution that tries to minimize the effects of link failures, there are two main delays involved: (i) *link failure detection delay*, the time it takes to detect that a link has failed, and (ii) *network reconfiguration delay*, the time required to reconfigure the network and restore route connectivity in response to the link failure. Together we refer to the sum of these delays as the *route failure time*. In this section, we show that although the route failure times have reduced from 65 ms in Portland [28] to 760 μs in ShareBackup [19], short latency-sensitive flows still suffer from high FCTs when there are link failures. To the best of our knowledge, ShareBackup currently has the lowest reported route reconfiguration time.

Setup. We do not have access to an optical switch, and so we *emulated* ShareBackup's behavior in our testbed by disabling a link and enabling it again after ShareBackup's route reconfiguration time. We refer to this simulation of ShareBackup as ShareBackup$'$ or SB$'$. Our testbed (Fig. 2) consists

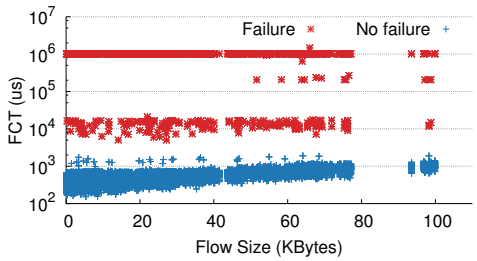


Fig. 3. FCTs of latency-sensitive web search flows [1] under link failures with ShareBackup as route recovery mechanism.

of a fat-tree topology built using a partitioned Barefoot Tofino switch (similar to [30]) and Intel Xeon servers equipped with Intel X710 NICs. All links are 10 Gbps and the network RTT between the hosts is about 100 μs. Each host runs Linux kernel 4.13.0 with TCP CUBIC. SACK is enabled and RTO_{min} is set to the smallest possible value of 4 ms. Host $h2$ sends short, latency-sensitive ($\leq$100 KBytes [16]) TCP flows to host $h4$ via the path $sw8 \rightarrow sw4 \rightarrow sw2 \rightarrow sw6 \rightarrow sw10 \rightarrow h4$. The flow sizes are drawn from the distribution of a web search workload [1]. The flows are sent one at a time with no other network traffic. Since the FCT of a small flow is less than 2 ms in normal case, we inject link failures between switches $sw6$ and $sw10$ every 20 ms to ensure that each flow experiences link failure at most once. To emulate ShareBackup's route reconfiguration, we use precise dataplane timer mechanisms to generate a link failure that lasts for exactly 760 μs. We use a *deflect-on-drop* switch dataplane mechanism to identify the flows affected by link failures.

FCTs under Link Failures. Fig. 3 shows the FCTs of the flows where failure-affected flows form three distinct horizontal clusters. The cluster of FCT values around 1 second is due to the SYN or SYN-ACK packet loss since the default retransmission timeout (RTO) for these packets is set to 1 second [31]. The middle two clusters of FCT values ($\sim$10 ms and $\sim$100 ms) are due to RTOs being triggered either due to *tail losses* in a cwnd or the complete loss of all packets in a cwnd. For failure affected flows, we did not observe any fast retransmissions. Overall, we see that when there are contiguous packet losses due to link failures, even with state-of-the-art fast recovery mechanisms like ShareBackup, the FCTs for short flows can increase by several orders of magnitude.

To further understand this result, we measured the TCP cwnd sizes for the above flows under no link failure (no loss) conditions. We found that, at 90th percentile, the cwnd size is about 10 MSS segments which is the default initial cwnd size on Linux [32]. The maximum observed cwnd size was 32 MSS segments which translates to 46,336 bytes with MSS being 1448 bytes. However, at 10 Gbps link speed, a route failure time of 760 μs translates to 950,000 bytes, i.e. 656 MSS segments after accounting for the Ethernet preamble, framing, and inter-frame gap. Therefore, the route is in the failed state for a much larger duration than the time it would take for a cwnd worth of packets to traverse a link in the network. This implies that it is very unlikely to have packet losses as "holes" *within* a cwnd so as to trigger fast transmissions. In our experiment with short flows, link failures always triggered expensive RTOs resulting in significantly longer FCTs.

The results presented above also hold true for other deployed TCP variants (DCTCP [1], TIMELY [6]) since they all employ the same mechanism for handling packet loss. In summary, our results (which concur with the results in [33]) show that the tail and whole window losses dominate in case of short flows, triggering RTOs and inflating FCTs under link failures. Therefore, to reduce tail FCTs under link failures, we need to avoid RTOs.

Discussion. The impact of RTOs can be alleviated to an extent by using microsecond-level RTO_{min} [34], which requires significant modifications to the end-host network stack [35]. A small RTO_{min} however risks reducing throughput due to spurious retransmissions [36] and leads to increased overall packet loss for incast-like scenarios [34]. Deciding on the right value for RTO_{min} is tricky and it is typically set at 5 ms in production datacenters [33], [37]. At this value, the majority of latency-sensitive flows are small enough to complete in one RTT [38] and therefore under link failures they would take at least twice as long to complete, irrespective of the value of RTO_{min}.

III. SQR Design

In §II, we argued that to eliminate high FCTs under link failures, we need to avoid RTOs. SQR therefore focuses on fast in-network recovery of packets lost during the route failure time, without involving the end hosts. Our key idea is to continuously *cache* a small number of recently transmitted packets on a switch and in the event of a link failure, retransmit them on the appropriate *backup* network path.

SQR runs entirely in the dataplane of an individual switch. Our design assumes the Portable Switch Architecture (PSA) [39] consisting of an ingress pipeline, a Buffer and Queuing Engine (BQE), and an egress pipeline. When an incoming packet enters the ingress pipeline, the primary egress port is determined by the network's routing scheme. Subsequently, the packets from the latency-sensitive applications will be marked if it belongs to a latency-sensitive flow[2] that needs to be protected by SQR. The packet passes through the BQE normally and when it arrives at the egress pipeline, it is subjected to SQR's processing if it is marked.

In the egress pipeline, SQR by default forwards a packet to the destination port and be proceeded normally. However, if a packet is *marked*, SQR performs the additional task of creating a copy of the packet and caches the copy for a time duration equal to the *link failure detection delay* (§II). By doing so, SQR ensures that packets are not lost if a link fails later. After this delay, SQR checks if the cached packet's primary egress port (on which the original packet was sent) is still operational. If the link is up, then it means that the original packet was transmitted successfully and the cached copy of

[2]Latency-sensitive applications can request SQR's protection by using a pre-defined set of TCP port numbers, IP Header TOS bits or VLAN IDs.

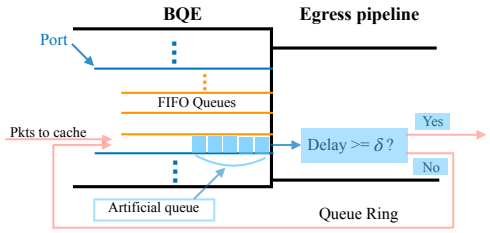


Fig. 4. Caching packets on switch using a FIFO queue.

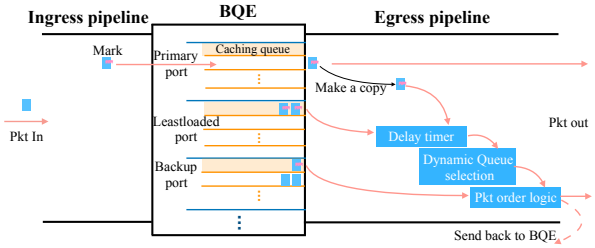


Fig. 5. Multi-Queue Ring architecture.

the packet is dropped. However, if the link is down, then the original packet was likely lost and a copy of the packet is cached again for time equal to the *network configuration delay* (§II). This additional delay allows the network to configure the backup path without losing the cached packets. After this second delay, the cached packets are sent on the backup path (port).

SQR's operation requires the following information in the switch dataplane: (i) The link status of the ports (*up* or *down*), (ii) the *backup* port (route) for each *primary* port (route) to a destination top-of-the-rack switch. SQR integrates with a link failure detection mechanism such as the one used in F10 [18] to update and maintain the status of the ports (albeit after a delay). It also integrates with a route recovery scheme (e.g. ShareBackup [19]) to determine the *backup* port for each *primary* port.

A. Caching Packets on the Switch.

Conceptually, the caching of packets can be achieved with a *delayed queue*, where each individual packet entering the queue is delayed for a fixed *minimum* amount of time (termed as *delay time*) before it leaves the queue. Unfortunately, there is no such primitive in the current switching ASICs. Furthermore, the queuing engines, including those in programmable ASICs do not support programming such custom scheduling to realize a delayed queue inside the queuing engine. In addition, packet storage, which is required to realize a delayed queue, is only available inside the queuing engine in the form of FIFO queues. Therefore, it is not straightforward to realize a delayed queue in existing switching ASICs.

SQR achieves the delayed queue functionality using a "Queue Ring" that combines the BQE's FIFO queue, egress pipeline processing and high-resolution timestamping. The high-level idea (shown in Fig. 4) is to place the packets to be cached inside a FIFO queue of a port on the switch. When the FIFO queue transmits the cached packet at a later time, high-resolution timestamping is used to check if the packet has been delayed for the required duration δ. If the packet is not sufficiently delayed ($delay < \delta$), the egress pipeline sends the cached packet back to the FIFO queue. Once a packet is sufficiently delayed ($delay >= \delta$) after passing through the FIFO queue one or more times, it exits the Queue Ring. This helps to build up an *artificial* queue of cached packets, since *effectively* no packet exits without being sufficiently delayed. In the steady state, where new packets enter the Queue Ring at a fixed rate R, the artificial queue build up remains fixed and equal to $R \times \delta$. Notice that each packet accumulates delays from two sources – (i) the queuing delay due to the artificial queue build-up, and (ii) the egress processing delay incurred in sending a packet back to the FIFO queue. Hereafter we will refer to the FIFO queue used to implement a Queue Ring as the *caching queue*.

B. Multi-Queue Ring Architecture

Our Queue Ring approach utilizes the egress processing of the port associated with the caching queue to emulate the delayed queue behavior. Unfortunately, the processing of these cached packets may affect the normal traffic passing through other queues of that port. Therefore, to minimize the impact on existing traffic, we do not use the same port for packet caching. Instead, SQR assigns one queue from the multiple queues [40] of each port as a *caching queue* and spreads the queued packets across a set of these caching queues. In particular, when a cached packet is to be sent back to a FIFO queue for additional delay, SQR dynamically chooses the caching queue that belongs to a port with the least utilization. We refer to this architecture that consists of multiple caching queues from the BQE that are connected to each other by the egress pipeline to form a *ring* as the Multi-Queue Ring (see Fig. 5). We exploit the fact that while the egress pipeline is provisioned to support all ports at full packet rate, there is almost always spare egress processing capacity available in the switch under typical network load conditions [26]. The spare capacity, however, is available on different ports at different times. Using a ring of multiple queues allows SQR to exploit the spare capacity by dynamically changing the set of low utilization ports.

An artifact of SQR's Multi-Queue Ring architecture is that when the cached packets exit after being buffered, they do not exit in the same order as they originally entered the Multi-Queue Ring. Therefore, in the event of a link failure, the exiting cached packets need to be ordered before they are sent to the backup port. To do so, SQR uses a counter-based packet sequencing mechanism. SQR's Multi-Queue Ring architecture is implemented with three components running in the egress pipeline (also shown in Fig. 5): (i) a delay timer (§III-C) to keep track of each cached packet's elapsed time, (ii) a queue selection algorithm (§III-D) to dynamically choose the next caching queue, and (iii) a packet order logic (§III-E) to order the cached packets before re-transmission.

C. Delay Timer

The delay timer first computes how long each packet has been buffered in the Multi-Queue Ring (called $\mathrm{ElapsedTime}$).

Algorithm 1: Delay Timer.

```
Initialization: ElapsedTime = 0, pkt.DelayEnough = 0;
foreach marked pkt in egress pipeline do
    diff = CurrentEgressTstamp − StartEgressTstamp;
    if diff > 0 then
        ElapsedTime = diff;
    else
        ElapsedTime = 2^n + diff;
    if ElapsedTime => δ then
        pkt.DelayEnough = 1;
end
```

Algorithm 2: Dynamic Queue Selection.

```
Input: PrimaryPort, LeastLoadedPort, BackupPort
foreach marked pkt in egress pipeline do
    if PrimaryPort == UP then
        if cached pkt then
            if pkt.DelayEnough ! = 1 then
                Send pkt to the LeastLoadedPort;
            else
                Drop cached pkt;
        else
            Make a copy and send the copy to LeastLoadedPort;
    else
        if cached pkt then
            if pkt.DelayEnough ! = 1 then
                Send pkt to the LeastLoadedPort;
            else
                Send pkt to BackupPort;
        else
            Send pkt to BackupPort;
end
```

To do so, when a copy of the original packet is created, the delay timer attaches the egress timestamp provided by the dataplane (called StartEgressTstamp) to the copied (cached) packet as metadata. As the cached packet passes through the Multi-Queue Ring, it enters the egress pipeline one or more times. Each time in the egress pipeline, the delay timer calculates the time elapsed so far (ElapsedTime) by taking the difference between the current egress timestamp (CurrentEgressTstamp) and the packet's StartEgressTstamp. The delay timer then compares the ElapsedTime with the required delay time (δ) to check if the packet has been buffered for at least the delay time. If so, the delay timer would set the DelayEnough field in the packet (later used by the queue selection algorithm in §III-D). The delay timer logic is summarized in Algorithm 1. Because of limited bit-width (n bits) clock register in the switch dataplane, the calculation needs to handle cases with value wrap around.

Delay Time (δ). This is the time for which each copied (cached) packet needs to be buffered on the switch. Since there is a delay in detecting link failures, δ is initially set equal to the upper bound of the link failure detection delay. When a link failure is detected, SQR dynamically increases δ by value equal to the network reconfiguration delay so as to hold the cached packets until the network reconfiguration is complete. Since the total packets being buffered on the switch is proportional to δ (c.f. §III), its value determines SQR's packet buffer requirement (§IV-D).

D. Dynamic Queue Selection

Recall from §III-B that SQR designates one queue on each port as the caching queue. In the Multi-Queue Ring, each time a cached packet is to be sent from the egress pipeline back to the BQE, the queue selection logic (Algorithm 2) decides to which caching queue to forward the packet. As the goal is to minimize the impact on other traffic, SQR selects the next caching queue from a port which has the least utilization *at the current moment* (called the LeastLoadedPort). A packet is sent to the LeastLoadedPort in the following cases: (i) if it is a freshly made copy of an original packet *and* the PrimaryPort is UP, or (ii) if it is an already cached packet that has not been sufficiently delayed. A sufficiently delayed cached packet (as indicated by the Delay Timer in §III-C) is dropped if the primary link is up. If the primary link is down and the incoming packet is an original packet or a sufficiently delayed cached packet, it is sent to the caching queue of the backup port for retransmission.

Tracking Port Utilization. SQR tracks the egress utilization of all the ports by maintaining a moving window of the number of bytes transmitted on each port. The size of the window is the time interval over which the number of transmitted bytes are accumulated. We discuss window sizing in §III-F. SQR maintains a LeastLoadedPort and the corresponding LeastUtilization. When an original packet arrives at the egress pipeline, the utilization of its egress port is updated. If this utilization is lower than the LeastUtilization, SQR will update the LeastUtilization to the current utilization and the LeastLoadedPort to the current egress port. When an original packet is transmitted on the LeastLoadedPort, SQR will also update the value of LeastUtilization.

Algorithm 3: Packet Order Logic.

```
Input: NextPktTag, PrimaryPort, BackupPort
foreach marked pkt in egress pipeline do
    if PrimaryPort == UP then
        if pkt.DelayEnough == 1 then
            NextPktTag = PktTag + 1;
    else
        if PktTag == NextPktTag then
            NextPktTag + = 1;
        else
            if PktTag > NextPktTag then
                Send pkt to BackupPort;
end
```

E. Packet Order Logic

When a link failure happens, the delay timer (§III-C) and the dynamic queue selection (§III-D) would send the cached copies of recently transmitted packets to the backup port (path). However, since cached packets are circulated through a *ring* of queues, the order in which they are sent to the backup port may not be the same as the original arrival sequence. To ensure that packet ordering is preserved, the packet order logic (Algorithm 3) first needs to know the original ordering of the packets. To achieve this, the packet order logic consists of a monotonically increasing packet counter in the egress pipeline. When an original packet to be protected by SQR enters the egress pipeline, the counter value (PktTag) is added to the packet as metadata and gets copied to the corresponding cached packet. The packet order logic also maintains an expected next counter number (NextPktTag). Both the PktTag and the NextPktTag are used to ensure correct packet ordering as following: (i) if the cached packet's

PktTag is equal to the expected NextPktTag, SQR just sends the packet and updates the NextPktTag (lines 6-7); (ii) if the cached packet's PktTag is larger than the NextPktTag, it will send this packet back to the backup port's caching queue and wait for the packet with the correct PktTag (line 9-10) to be sent first. When a cached packet with a PktTag leaves the switch due to either being dropped after sufficient buffering or sent on the backup path, SQR updates the NextPktTag (lines 4, 7). Since the cached packets are ordered before being sent, this may add extra delay on recovery time (§IV-B).

F. Implementation

We implemented SQR on a Barefoot Tofino switch [27] in about 1,100 lines of P4 code. A common action performed by SQR is to send a packet from the egress pipeline back to the BQE. This action is achieved with two primitives, egress-to-egress cloning (also called mirroring) and packet drop. For each cached packet, the SQR metadata is added when the cached packet is first created and is removed before the packet is sent out of the switch. The SQR metadata contains three fields: (i) PktTag: used by packet order logic for reordering (§III-E); (ii) StartEgressTstamp: used by delay timer to record when the cached packet was created (§III-C); (iii) PrimaryPort: used by queue selection logic to track the cached packet's primary port (§III-D).

The delay timer, queue selection logic and the packet order logic are implemented using a series of exact match-action tables and stateful ALUs. The *delay time* is stored in a dataplane register and can be dynamically configured based on the link failure detection mechanism being used. For computing link utilization (§III-D), we set the moving window size larger than the network RTT to avoid sensitivity to transient sub-RTT traffic bursts [16]. At the same time, we also avoid setting the window so large that it would aggregate the bytes of entire short flows and make SQR sluggish to react to the flow churn. Since the network RTT in our testbed is about 100 μs and the minimum FCT in our evaluation workloads is about 157 μs, we used a window size of 150 μs in our prototype. The LeastLoadedPort and LeastUtilization are also maintained using dataplane registers. We note that SQR's implementation requires standard primitives such as egress mirroring, encap/decap (for SQR metadata), registers and match-action tables which are specified in the PSA [39] and also available in fixed-function ASICs. Therefore, SQR can be implemented on any programmable ASIC based on the PSA [39] or it could be baked into fixed-function ASICs.

IV. Performance Evaluation

We evaluate our SQR prototype by answering three questions: (1) How effective is SQR in masking link failures from end-point TCP stack, such that RTOs will not be triggered? (2) When SQR is integrated with other network reconfiguration systems (e.g. ShareBackup), how much is the reduction in FCTs under link failures for latency-sensitive workloads? (3) What is the cost (overhead) of SQR in terms of effect on other traffic and consumption of resources in the switch hardware? We perform the evaluation on the same hardware testbed as described in §II unless otherwise mentioned.

A. Experimental setup

Workloads. We consider two empirical workloads with short flows taken from production datacenters: a web search workload [1] and a data mining workload [38]. The CDF of flow sizes for these two workloads is shown in Fig. 6. For both the workloads, we consider flow sizes up to 100 KB since these represent latency-sensitive flows [16]. We use a server-client model in which a server sends TCP flows of sizes drawn from these two distributions to a client. Specifically, in our testbed (Fig. 2), host $h2$ sends TCP flows to host $h4$.

Background Traffic. We run the Spark TPC-H decision support benchmark to generate background traffic. It contains a suite of database queries running against a 12 GB database on each worker. The master node is $h4$ (see Fig. 2) which communicates with the slave nodes $h1$ and $h2$ via the paths $sw10 \rightarrow sw6 \rightarrow sw2 \rightarrow sw4 \rightarrow sw7$ and $sw10 \rightarrow sw6 \rightarrow sw2 \rightarrow sw4 \rightarrow sw8$, respectively. The query job is submitted to the master node and multiple tasks run on the three nodes.

Baseline Schemes. Recall that SQR integrates with a link failure detection and a network reconfiguration scheme (§III). We consider the link failure detection method suggested in F10 [18] (detection delay = 30 μs) and two different network reconfiguration methods: ShareBackup [19] (SB$'$) and local rerouting (LRR), in the following configurations:

1) **SB$'$:** As explained in §II, SB$'$ is our emulated version of ShareBackup that takes an additional 730 μs to restore network connectivity via backup switches after a link failure is detected.
2) **LRR: L**ocal **R**e**R**outing runs a path probing protocol [16], [17] to proactively-determine a backup port for each primary port. When the link on a primary port is detected to be down, the traffic is immediately re-routed to the backup port thus incurring no network reconfiguration delay.
3) **SB$'$+SQR:** SQR integrated with SB$'$ involves setting the backup port to be the primary port itself since ShareBackup uses optical switching to restore connectivity on the same port. The initial delay time is 30 μs and is increased to 760 μs on link failure detection (§III-C).
4) **LRR+SQR:** SQR integrated with LRR involves setting the backup ports to the ones determined proactively. The delay time is 30 μs at all times.

Link Failure Model. SQR helps with link failures where multiple paths are available. Therefore, we inject a link failure every 20 ms between $sw6$ and $sw10$ while $h2$ is sending traffic to $h4$ (Fig. 2). Similar to §II, for SB$'$ we restore the failed link after the route failure time (760 μs).

B. Masking Link Failures from TCP

First, we evaluate SQR's effectiveness in masking link failures from the end-point transport protocol (TCP). We compare TCP's behavior under link failure when running SB$'$ alone to that when running SB$'$ along with SQR. $h2$ starts an iperf client to send TCP traffic to an iperf server running

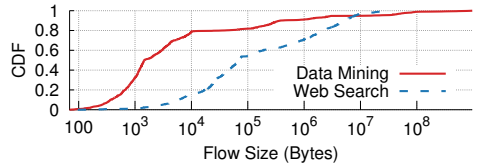
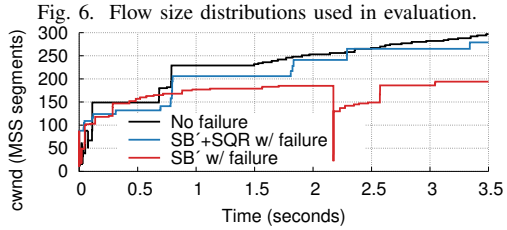
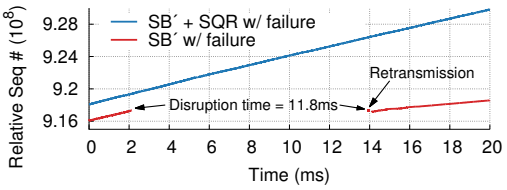


Fig. 6. Flow size distributions used in evaluation.



(a) TCP congestion window



(b) TCP sequence number (zoomed view after 2 seconds)

Fig. 7. TCP sender's cwnd and seq number progression for SB′ with and without SQR. Link failure occurs after about 2 seconds.

on $h4$ (see Fig. 2). To properly observe the TCP sequence numbers from captured traces, we set TSO off (only for this experiment). We use n2disk [41] to capture the packet traces and the tcp_probe kernel module to capture the TCP sender's connection statistics. About 2 seconds after starting the flow, we inject a link failure on the link between the switches $sw6$ and $sw10$. Fig. 7 shows one instance of the result. Results are similar when link failure is introduced at a different location in the network.

Fig. 7a shows the evolution of the TCP sender's cwnd. We see that with SB′ alone, the TCP sender reduces its cwnd size drastically when there is a packet loss due to link failure. However, when SB′ is enhanced with SQR, the link failure has no impact on the TCP sender and the cwnd grows like the no-failure case. In Fig. 7b, we plot the TCP stream's sequence number of packets as sent by the sender. With SB′ alone, when the link fails, the TCP sender stops sending due to absence of ACKs and times out leading to a disruption time of about 12 ms. By the time the TCP sender recovers from the timeout, SB′ has already restored the connectivity and the sender resumes by first retransmitting the lost packets. However, when SB′ is coupled with SQR, the TCP sender is not affected by the link failure and the TCP sequence number grows smoothly.

Recovery Time. While Fig. 7 shows the TCP sender's perspective, the perspective from a TCP receiver is different. Upon link failure, while the route is being reconfigured, SQR *holds* the packet transmission thereby introducing a time small gap. This small time gap, called the *recovery time*, is an unavoidable effect seen by a TCP receiver. Fig. 8 shows the CDF of the recovery time for over 30,000 TCP flows where the link failure is masked in a SB′+SQR configuration. The

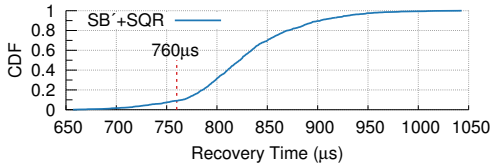


Fig. 8. Recovery time.

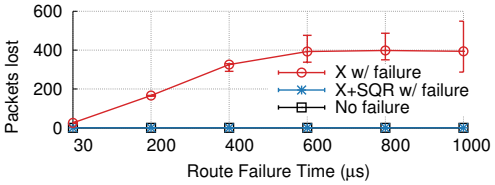


Fig. 9. Number of packets lost for different route failure time.

recovery time is larger than SB′ route failure time (760 μs) in about 90% instances for two reasons. First, SQR needs to reorder the packets before retransmission which adds some additional recovery delay. Second, the underlying delayed queue causes each individual packet to be delayed for a time *at least* equal to the *worst-case* route failure time. The packets that are delayed for longer than the *actual* route failure time are those that are not lost and would be delivered to the receiver again. Retransmitting these *extra packets* also contributes to the additional recovery time. Note that these extra packets do not affect the TCP receiver's state and the resultant FCT for short flows [42]. In about 10% instances, the recovery time was lower than 760 μs. We believe that in these instances, due to "natural" gaps in the packet transmission, the packets arrived after a link failed and before the route was successfully recovered, thereby getting buffered for less than 760 μs.

Packet Loss. The number of packets lost during a link failure depends on the recovery scheme's route failure time. A scheme with a higher route failure time would stress SQR. Fig. 9 shows the number of packets lost for a generic route recovery scheme X, whose route failure time varies from 30 μs (LRR) to 1000 μs (F10 [18]). Beyond the route failure time of 600 μs, the number of lost packets does not increase as TCP loses almost the whole cwnd and the transmission is stalled. When X is coupled with SQR, the packet loss remains zero even when the route failure time increases.

C. Latency-sensitive Workloads

Next, we evaluate how effective is SQR at keeping FCTs low for latency-sensitive workloads under link failures. We use 1,000 different flow sizes from the web and data mining workloads (§IV-A) and send 30 flows of each flow size yielding a total of 30,000 flows. The flows are sent from $h2$ to $h4$ while the link between $sw6$ and $sw10$ is brought down every 20 ms (see Fig. 2). The total route failure time is 30 μs for LRR and 760 μs for SB′.

We first focus on the FCTs of flows which faced link failures i.e. we ignore the flows that were not affected by a link failure. We showed in §II that even with SB′, the FCTs can increase by several orders of magnitude when there are link failures (see Fig. 3). Fig. 10 shows that when SB′ is coupled with SQR, the FCTs for the failure-hit flows are only slightly

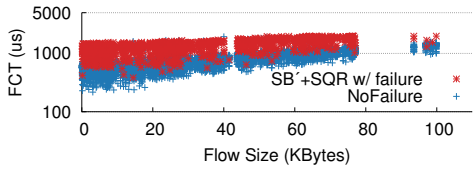


Fig. 10. FCTs of latency-sensitive web search flows [1] under link failures with SB′+SQR as route recovery mechanism.

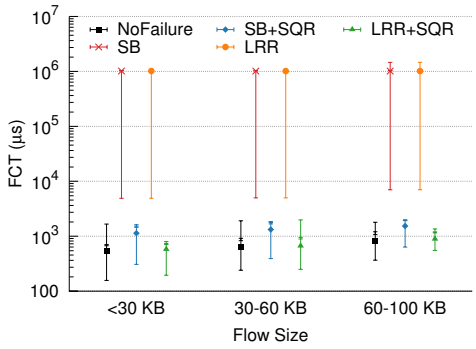


Fig. 11. FCTs of failure-hit web search flows [1] compared to no failure.

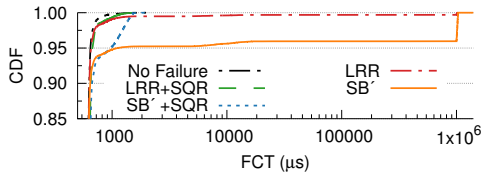
higher than the FCTs of no failure flows. Fig. 11 shows the FCTs for failure-hit web search flows when running SB′ and LRR schemes with and without SQR. We show the results for three different ranges of flow sizes. The vertical bars show the minimum, median, 95th percentile, 99th percentile and the maximum values of FCT. We observe that when coupled with SQR, the tail FCTs of failure-hit flows for both SB′ and LRR are reduced by about 3 to 4 orders of magnitude. If the packets of a flow arrive after the link has failed and before the route is reconfigured, the *recovery time* (see §IV-B) of these packets will be less than the route failure time. Therefore, even though SB′ has a 760 μs route failure time, the minimum and median values of FCT for SB′+SQR are only about 200 μs higher than the no failure or LRR+SQR scenarios.

Fig. 12 shows the FCT distribution for all the 30,000 flows involved in an experiment run, including those not affected by link failures. For both the data mining and web search workloads, the tail FCT of SB′ is slightly worse than LRR. This is because SB′ has a longer route failure time compared to LRR. While SQR helps in cutting down the overall tail FCT for both SB′ and LRR, its reduction in FCT for LRR is slightly more than that for SB′. This is because although SQR prevents packet loss, it inflicts a *recovery time* delay (see §IV-B) which is higher for SB′ than that for LRR.

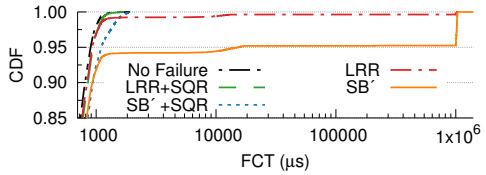
D. Overhead

Finally, we investigate the overheads incurred by SQR by measuring: (i) the packet buffer consumption, (ii) the reduction in switch throughput; (iii) the additional hop latency on the switch; and (iv) the hardware resources required when implemented on a programmable switch.

Packet Buffer Consumption. SQR uses the switch packet buffer to cache packets for the *delay time* (see §III-C). Since SQR uses a ring of queues, the packet buffer consumption at any time is equal to the total number of cached packets across the different caching queues. To measure the packet



(a) Data mining workload



(b) Web search workload

Fig. 12. CDF of FCTs for two workloads under link failures.

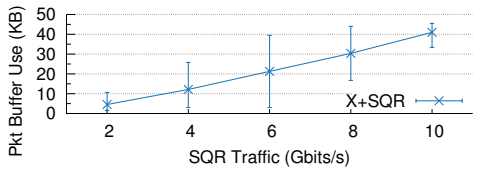


Fig. 13. Steady-state packet buffer consumption (per-port).

buffer consumption, we configured SQR's Multi-Queue Ring to use only a single queue (just for measurement). Then using the *queue depth* provided by the programmable dataplane, we measured the depth of the queue to obtain the packet buffer consumption. During steady-state (no link failure), SQR only caches packets for the link failure detection delay. Therefore, its steady-state packet buffer consumption depends only on the link failure detection mechanism. For a generic route recovery scheme (which we denote with X), Fig. 13 shows how the steady-state packet buffer consumption (per-port) increases with SQR traffic volume while using F10's link failure detection mechanism (detection delay = 30 μs). Clearly, the packet buffer consumption increases with increase in SQR traffic. For a 10 Gbps link, SQR will only need to handle up to 10 Gbps traffic in the worst case, even when there is an *incast* (>10 Gbps) of incoming latency-sensitive traffic. This is because SQR protects traffic on the egress link whose rate is constrained by the link speed. Therefore, the worst case packet buffer consumption per SQR-enabled port is given by,

$$\text{Worst Case Pkt Buffer} = \text{Link Speed} \times \text{Delay Time} \quad (1)$$

From equation 1, we would expect the worst-case steady-state buffer consumption for a 10 Gbps port with a 30 μs failure detection delay to be 37.5 KB. This matches our experimental results in Figure 13. However, when a link failure is detected, the delay time is increased to 760 μs in case of SB′+SQR. In this instance, according to equation 1, the buffer consumption for SB′+SQR would be 950 KB in the worst case. Fortunately, the failed-state is very short-lived (and will last only until the route is reconfigured), after which SQR returns to steady-state caching.

Impact of SQR on Normal Traffic. SQR incurs some additional egress pipeline processing to send insufficiently delayed cached packets back to the BQE (§III-A). To measure

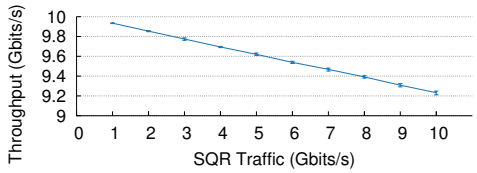


Fig. 14. Impact of SQR processing on normal line-rate traffic.

the impact of SQR's processing (maintaining a delayed queue) on the normal traffic, we configure SQR's Multi-Queue Ring to contain only a single caching queue on a port, say p_1. We then start line rate TCP (10 Gbps) background traffic whose egress port on the switch is also p_1. The background traffic uses a queue on port p_1 that is different from the SQR's caching queue, but has the same scheduling priority. A SQR-enabled flow (SQR traffic) is then started on another port p_2. All packets from this flow are cached using p_1's caching queue.

Fig. 14 shows the throughput of the line-rate background traffic for different rates of SQR traffic. We see that even at 10 Gbps, SQR traffic will occupy only about 750 Mbps of egress processing. This means that as long as the normal traffic is less than 9.25 Gbps, it will not be impacted by the processing overhead of 10 Gbps SQR traffic. In other words, a single 10 Gbps port can simultaneously support 9.25 Gbps of normal traffic and egress processing of 10 Gbps SQR traffic. Given that SQR uses dynamic queue selection (§III-D) to utilize only the $\mathrm{LeastLoadedPort}$ each time the next caching queue in the Multi-Queue Ring is chosen, the likelihood of negatively impacting the normal traffic is very low.

Switch Processing Latency. SQR is mostly non-intrusive to the SQR-protected original traffic, but incurs some additional dataplane processing. To measure the latency added by this additional processing, we send traffic from $h1$ to $h2$ along $sw7 \rightarrow sw4 \rightarrow sw8$ (Fig. 2). When a packet arrives at the ingress pipeline of $sw7$ or $sw4$, we add the ingress timestamp (*IngressTs*) to it. The difference between the two *IngressTs* of adjacent switches is the *hop latency*. We found that, on average SQR adds a negligible 4.3 ns of *additional* hop latency compared to a P4 program that does only L3 forwarding.

Hardware Resources Requirements. In Table II, we compare the hardware resources required by SQR to that required by *switch.p4*, which is a close-source production P4 program that implements all the network features of a typical datacenter ToR switch. SQR uses a relatively larger proportion of stateful ALUs for operations such as calculating the $\mathrm{ElapsedTime}$, determining the $\mathrm{LeastLoadedPort}$, and comparing the PktTag with the $\mathrm{NextPktTag}$. SQR's logic is achieved using exact match-action tables which require SRAM. However, SQR's overall resource consumption remains low. Also, since the combined usage of all resources by switch.p4 and SQR is less than 100%, switch.p4 can easily be enhanced by incorporating SQR.

V. Related Work

Two broad categories of related work relevant to SQR are route recovery and packet loss recovery.

Table II
Resource consumption of SQR compared to switch.p4

Resource	switch.p4	SQR	switch.p4 + SQR
Match Crossbar	51.56%	10.22%	61.59%
Hash Bits	32.79%	13.28%	44.75%
SRAM	29.58%	15.31%	41.35%
TCAM	32.29%	0.00%	32.29%
VLIW Actions	36.98%	6.77%	43.23%
Stateful ALUs	18.75%	15.63%	33.33%

Route Recovery. Among existing route recovery schemes, many attempt to achieve *fast re-routing* for multi-path datacenter topologies. Failure carrying packets [43] are proposed to avoid route convergence delay by carrying failed link(s) information inside data packets to notify other nodes. Fast Reroute (FRR) [12] used in MPLS networks can provide recovery in less than 50 ms during a link/node failure. Packet Recycling [44] takes advantage of cycle in the network topology where routers implement a cyclic routing table. SPIDER [11] and Blink [45] maintain a pre-computed backup next hop in the switch. Sedar et al. [13] implement the fast reroute primitive based on known port status in programmable data planes and in Data-Driven Connectivity [46] dataplane packets are used to ensure routing connectivity. Flowlet switching [47] based load balancing schemes such as CONGA [16] and HULA [17] are an implicit form of fast re-routing schemes since they avoid a failed path for routing subsequent flowlets. Another group of route recovery schemes consist of multi-path network architectures that allow fault-tolerance [38], [48]–[52]. Notably, F10 [18] designs an AB fat-tree and a centralized rerouting protocol to support downlink recovery. ShareBackup [19] uses a shared pool of backup switches for on-demand failure recovery which is facilitated by circuit switches. SQR is complementary to existing route recovery schemes as it helps them to avoid packet loss during their route recovery time and link failure detection time.

Packet Loss Recovery. Traditionally, packet loss recovery is left to end-point transport. However, for short latency-sensitive flows, end-host recovery incurs FCT penalty due to packet loss and timeout before recovering the lost packets (c.f. §II). Alternatively, end-to-end redundancy approaches can be used [33], [53], where the sender sends duplicate un-ACKed packets on separate paths. However, duplicating packets on the entire path increases the required network bandwidth. Since datacenter networks are often oversubscribed [54], this approach may increase network congestion. Instead taking up network bandwidth, SQR opportunistically utilizes free packet buffer on the switch to store the duplicate packets. In addition, the end-to-end redundancy methods require changes to the end-host TCP stack. To the best of our knowledge, SQR is the first attempt at in-network packet loss recovery and requires no changes to the end hosts.

Overall, all existing route recovery and packet loss recovery schemes cannot seamlessly divert traffic from a failed path to an alternative path. The main reason is that they do not take into account the inevitable delay and the corresponding packet loss arising from link failure detection and route reconfiguration. Furthermore, since majority of the flows in datacenter

networks are small [55], competing approaches of reducing route failure time or flowlet-level switching to alternative paths are not able to mitigate the impact of link failures on short flows. This is precisely the gap that SQR addresses.

VI. Discussion

Hardware-assisted Link Failure Detection. High-speed network cable connectors such as QSFP+ and QSFP28 "squelch" their data input/output lanes on detecting loss of input/output signal levels [56]. Modern switching ASICs are able to detect such data lane squelching and provide primitives for fast failover [27]. We investigated such hardware-assisted link failure detection in our testbed using a Barefoot Tofino switch and an Intel XXV4DACBL1M (QSFP28 to 4xSFP28) cable. We found the worst-case detection delay to be around 2.755 μs. This implies that, with hardware support, link failure detection delays are even lower, and SQR's steady-state packet buffer consumption can be further reduced.

Alternatives to on-chip Packet Buffer. An alternative way to store cached packets could be to leverage the relatively large ($\sim$4 GB) DRAM available on the switch CPU. However, the switch CPU's limited bandwidth on its interface to the ASIC (PCIe 3.0 x4 [57]) and its limited processing capacity make this approach infeasible for implementing SQR. This limitation is common for all switches including fixed-function [23] or partially programmable [57]. In highly congested networks where the on-chip packet buffer is a scarce resource, using expandable packet buffers implemented via DRAM and connected directly [58], [59] or indirectly [60] to the ASIC is a better approach, since a CPU is not required to access the DRAM. Note that SQR's overall architecture still remains the same even when implemented with expandable packet buffer.

Handling Traffic Surges. SQR exploits the availability of spare buffer and egress processing from the least loaded ports dynamically. A prior measurement study has shown that high utilization and thus congestion happens on a small number of ports and not on all the ports of a switch at the same time [26]. Nevertheless, there remains a small possibility that when a switch is saturated on all ports, SQR could make the congestion worse by partially occupying the packet buffer. To address this, SQR implements a backstop mechanism that can dynamically pause packet caching (within nanoseconds) when we detect high buffer consumption, and resume only when spare buffer becomes available. With increasing adoption of delay-based congestion control protocols in datacenters [6], we expect such high buffer pressure events that can overwhelm an entire switch's packet buffer to be rare.

Deployability and Fault Tolerance. SQR runs independently on a singleton switch and thus SQR-enabled switches can be deployed incrementally in a network. A SQR-enabled switch adds link failure tolerance for each port, i.e. it can handle failures on multiple links emanating from it. Since link failures tend to be uncorrelated [7], a partial deployment of SQR-enabled switches can effectively bring down the impact of link failures. SQR will also be effective against failures such as line-card or switch failures that cause link failure detection schemes to report corresponding link failures. One limitation is that SQR will not be able to help in the event of link failures between the end hosts and the ToR switches due to the lack of alternative paths. Also, it is not designed to handle packet corruption losses. For datacenter networks, since most switches have higher availability than the links and concurrent traffic bursts on multiple switch ports [26] and concurrent link failures are rare [7], the probability of packets being lost due to simultaneous link and switch failures will be low.

Higher Link Speeds. SQR can scale to higher link speeds (25/50/100 Gbps) with an increase in buffer consumption (see equation 1). For a 100 Gbps port with a 30 μs link failure detection time, the worst-case steady-state buffer consumption is expected to be 375 KB. However, on average, latency-sensitive short-flows only comprise about 20% of the total bytes in typical datacenter networks [1]. Therefore, even at 100% link utilization on a 100 Gbps link, we expect SQR to handle about 20 Gbps of latency-sensitive traffic. For this average case, the worst-case steady-state buffer consumption is about 75 KB per port. When the link fails, the average case requirement of SB$'$+SQR spikes momentarily to 1.9 MB per port. Switching ASICs supporting 100 Gbps switches currently have around 42 MB ($>$ 1.9 MB) of packet buffer [61]. Also, the on-chip packet buffer size for ASICs increases with supported link speeds [62]. Therefore, SQR's consumption of packet buffer can be supported comfortably by modern ASICs.

VII. Conclusion

Achieving low and bounded FCTs under link failures is a step towards providing SLA guarantees on network latency in datacenter networks. We show that existing link failure management techniques fail to keep the FCTs low, as they cannot completely eliminate packet loss during link failures. By enabling caching of small number of recently transmitted packets, SQR completely masks packet loss during link failures from end-hosts. Our experiments show that SQR can reduce the tail FCT by up to 4 orders of magnitude for latency-sensitive workloads. While caching packets on the switch is an obvious idea, it is not straightforward to achieve and was not feasible until now. The significant reduction in route recovery times and increase in packet buffer sizes have made it feasible, while our design, implementation and evaluation of SQR demonstrates that it is both effective and practical. Our work suggests that on-switch packet caching would be a useful primitive for future switch ASICs.

Acknowledgements

This work was carried out at the SeSaMe Centre, supported by Singapore NRF under the IRC@SG Funding Initiative. It was also partly supported by the National Natural Science Foundation of China under Grant No.61772544 and the Singapore Ministry of Education tier 1 grant R-252-000-693-114. Ting Qu was supported by the China Scholarship Council ([2017]3109) to visit the National University of Singapore.

REFERENCES

[1] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data Center TCP (DCTCP)," in *Proceedings of SIGCOMM*, 2010.

[2] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, "Better Never than Late: Meeting Deadlines in Datacenter Networks," in *Proceedings of SIGCOMM*, 2011.

[3] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pfabric: Minimal near-optimal datacenter transport," in *Proceedings of SIGCOMM*, 2013.

[4] C.-Y. Hong, M. Caesar, and P. Godfrey, "Finishing flows quickly with preemptive scheduling," in *Proceedings of SIGCOMM*, 2012.

[5] A. Munir, G. Baig, S. M. Irteza, I. A. Qazi, A. X. Liu, and F. R. Dogar, "Friends, not foes: synthesizing existing transport strategies for data center networks," in *Proceedings of SIGCOMM*, 2015.

[6] R. Mittal, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, D. Zats *et al.*, "TIMELY: RTT-based Congestion Control for the Datacenter," in *Proceedings of SIGCOMM*, 2015.

[7] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: measurement, analysis, and implications," in *Proceedings of SIGCOMM*, 2011.

[8] P. Cheng, F. Ren, R. Shu, and C. Lin, "Catch the whole lot in an action: Rapid precise packet loss notification in data center," in *Procedings of NSDI*, 2014.

[9] K. Yedugundla, P. Hurtig, and A. Brunstrom, "Probe or Wait: Handling tail losses using Multipath TCP," in *Proceddings of IFIP Networking*, 2017.

[10] D. Zats, A. P. Iyer, R. H. Katz, I. Stoica, and A. Vahdat, "Fastlane: An agile congestion signaling mechanism for improving datacenter performance," in *Proceedings of SoCC*, 2013.

[11] C. Cascone, D. Sanvito, L. Pollini, A. Capone, and B. Sansò, "Fast failure detection and recovery in SDN with stateful data plane," *International Journal of Network Management*, vol. 27, no. 2, p. e1957, 2017.

[12] P. Pan, G. Swallow, and A. Atlas, "Fast reroute extensions to RSVP-TE for LSP tunnels," IETF RFC 4090, 2005, https://tools.ietf.org/html/rfc4090.

[13] R. Sedar, M. Borokhovich, M. Chiesa, G. Antichi, and S. Schmid, "Supporting Emerging Applications With Low-Latency Failover in P4," in *Proceedings of NEAT*, 2018.

[14] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "OpenFlow: Meeting carrier-grade recovery requirements," *Computer Communications*, vol. 36, no. 6, pp. 656–665, 2013.

[15] N. L. Van Adrichem, B. J. Van Asten, and F. A. Kuipers, "Fast recovery in software-defined networks," in *Proceedings of EWSDN*, 2014.

[16] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, F. Matus, R. Pan, N. Yadav, G. Varghese *et al.*, "CONGA: Distributed Congestion-aware Load Balancing for Datacenters," in *Proceedings of SIGCOMM*, 2014.

[17] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "Hula: Scalable load balancing using programmable data planes," in *Proceedings of SOSR*, 2016.

[18] V. Liu, D. Halperin, A. Krishnamurthy, and T. E. Anderson, "F10: A fault-tolerant engineered network." in *Procedings of NSDI*, 2013.

[19] D. Wu, Y. Xia, X. S. Sun, X. S. Huang, S. Dzinamarira, and T. E. Ng, "Masking failures from application performance in data center networks with shareable backup," in *Procedings of SIGCOMM*, 2018.

[20] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN," in *Proceedings of SIGCOMM*, 2013.

[21] Broadcom. (2010) Trident+ Buffer Size. [Online]. Available: https://goo.gl/9LUHWa

[22] Broadcom. (2013) Trident II Buffer Size. [Online]. Available: https://goo.gl/3eWY7T

[23] EdgeCore Networks. (2018) AS5812-54X Spec. [Online]. Available: https://goo.gl/ZKqF6F

[24] Broadcom. (2016) Tomahawk+ buffer size. [Online]. Available: https://goo.gl/3eWY7T

[25] Broadcom. (2017) Tomahawk II Buffer Size. [Online]. Available: https://goo.gl/3eWY7T

[26] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy, "High-resolution measurement of data center microbursts," in *Proceedings of IMC*, 2017.

[27] Barefoot Networks. (2018) Tofino. [Online]. Available: https://goo.gl/cdEK1E

[28] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, "Portland: a scalable fault-tolerant layer 2 data center network fabric," in *Procedings of SIGCOMM*, 2009.

[29] M. Foschiano, "Cisco Systems UniDirectional Link Detection (UDLD) Protocol," IETF RFC 5171, 2008, https://tools.ietf.org/html/rfc5171.

[30] P. G. Kannan, R. Joshi, and M. C. Chan, "Precise Time-synchronization in the Data-Plane using Programmable Switching ASICs," in *Proceedings of SOSR*, 2019.

[31] M. Sargent, M. Allman, and V. Paxson, "Computing TCP's Retransmission Timer," IETF RFC 6298, 2011, https://tools.ietf.org/html/rfc6298.

[32] J. Chu, N. Dukkipati, Y. Cheng, and M. Mathis, "Increasing TCP's Initial Window," IETF RFC 6928, 2013, https://tools.ietf.org/html/rfc6928.

[33] G. Chen, Y. Lu, Y. Meng, B. Li, K. Tan, D. Pei, P. Cheng, L. Luo, Y. Xiong, X. Wang *et al.*, "Fast and Cautious: Leveraging Multi-path Diversity for Transport Loss Recovery in Data Centers," in *Proceddings of ATC*, 2016.

[34] G. Judd, "Attaining the Promise and Avoiding the Pitfalls of TCP in the Datacenter," in *Proceedings of NSDI*, 2015.

[35] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller, "Safe and effective fine-grained TCP retransmissions for datacenter communication," in *Proceedings of SIGCOMM*, 2009.

[36] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan, "Measurement and Analysis of TCP Throughput Collapse in Cluster-based Storage Systems," in *Proceedings of FAST*, 2008.

[37] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph, "Understanding TCP incast throughput collapse in datacenter networks," in *Proceedings of WREN*, 2009.

[38] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: a scalable and flexible data center network," in *Proceding of SIGCOMM*, 2009.

[39] P. L. Consortium. (2018) Portable switch architecture. [Online]. Available: https://p4.org/p4-spec/docs/PSA.html

[40] Cisco. (2019) Configuring QoS - Catalyst 3850. [Online]. Available: https://bit.ly/2W6jOTo

[41] Ntop. (2018) n2disk. [Online]. Available: https://goo.gl/7DFkSp

[42] D. Borman, B. Braden, V. Jacobson, and R. Scheffenegger, "Protection against wrapped sequences," IETF RFC 7323, 2014, https://tools.ietf.org/html/rfc7323\#section-5.

[43] K. Lakshminarayanan, M. Caesar, M. Rangan, T. Anderson, S. Shenker, and I. Stoica, "Achieving convergence-free routing using failure-carrying packets," in *Proceedings of SIGCOMM*, 2007.

[44] S. S. Lor, R. Landa, and M. Rio, "Packet Re-cycling: Eliminating Packet Losses due to Network Failures," in *Proceedings of HotNets*, 2010.

[45] T. Holterbach, E. Costa Molero, M. Apostolaki, A. Dainotti, S. Vissicchio, and L. Vanbever, "Blink: Fast connectivity recovery entirely in the data plane," in *Proceedings of NSDI*, 2019.

[46] J. Liu, A. Panda, A. Singla, B. Godfrey, M. Schapira, and S. Shenker, "Ensuring Connectivity via Data Plane Mechanisms," in *Proceedings of NSDI*, 2013.

[47] S. Kandula, D. Katabi, S. Sinha, and A. Berger, "Dynamic Load Balancing Without Packet Reordering," *SIGCOMM CCR*, vol. 37, no. 2, pp. 51–62, 2007.

[48] J. H. Ahn, N. Binkert, A. Davis, M. McLaren, and R. S. Schreiber, "HyperX: topology, routing, and packaging of efficient large-scale networks," in *Proceedings of SC*, 2009.

[49] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Procedings of SIGCOMM*, 2008.

[50] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, "BCube: a High Performance, Server-centric Network Architecture for Modular Data Centers," in *Proceedings of SIGCOMM*, 2009.

[51] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu, "DCell: a Scalable and Fault-tolerant Network structure for Data Centers," in *Procedings of SIGCOMM*, 2008.

[52] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey, "Jellyfish: Networking data centers randomly," in *Proceedings of NSDI*, 2012.

[53] A. Vulimiri, O. Michel, P. Godfrey, and S. Shenker, "More is Less: Reducing Latency via Redundancy," in *Proceedings of HotNets*, 2012.

[54] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano *et al.*, "Jupiter Rising: A decade of Clos Topologies and Centralized Control in Google's Datacenter Network," in *Proceedings of SIGCOMM*, 2015.

[55] T. Benson, A. Akella, and D. A. Maltz, "Network Traffic Characteristics of Data Centers in the Wild," in *Proceedings of IMC*, 2010.

[56] SNIA. (2018) SFF-8679: QSFP+ 4X Hardware and Electrical Specification. [Online]. Available: https://members.snia.org/document/dl/25969

[57] Broadcom. (2018) Trident 3 Ethernet Switch Series. [Online]. Available: https://bit.ly/2HBgKut

[58] Broadcom. (2016) StrataDNX Qumran-AX Ethernet Switch Series. [Online]. Available: https://bit.ly/2K1GYYR

[59] EdgeCore Networks. (2016) AS5900-54X Spec. [Online]. Available: https://bit.ly/2VQ1RZb

[60] D. Kim, Y. Zhu, C. Kim, J. Lee, and S. Seshan, "Generic external memory for switch data planes," in *Proceedings of HotNets*, 2018.

[61] EdgeCore Networks. (2019) AS5816-64X Spec. [Online]. Available: https://www.edge-core.com/productsInfo.php?cls=1&cls2=5&cls3=166&id=309

[62] J. Warner. (2019) Packet Buffers. [Online]. Available: https://people.ucsc.edu/~warner/buffer.html