Detecting and Resolving PFC Deadlocks with ITSY Entirely in the Data Plane

Xinyu Crystal Wu, T. S. Eugene Ng
Rice University

Abstract—**The Priority-based Flow Control (PFC) protocol is adopted to guarantee zero packet loss in many high-performance data centers. PFC, however, can induce deadlocks and in severe cases cause the entire network to be blocked. Existing solutions have focused on deadlock avoidance; unfortunately, they are not foolproof. Therefore, deadlock detection is a necessity. We propose ITSY, a novel system that correctly detects and resolves deadlocks entirely in the data plane. It works with any network topologies and routing algorithms. Unique to ITSY is the use of deadlock initial triggers, which contributes to efficient deadlock detection, mitigation, and recurrence prevention. ITSY provides three deadlock resolution mechanisms with different trade-off options. We implement ITSY for programmable switches in the P4 language. Experiments show that ITSY detects and resolves deadlocks rapidly with minimal overheads.**

I. Introduction

Driven by demand for ultra-low latency, high throughput network applications with low CPU overhead, lossless networks are widely deployed in modern data centers and cloud environments [44]. One typical implementation of such networks is lossless Ethernet, an attractive option to public cloud providers for supporting Remote Direct Memory Access (RDMA). For example, Microsoft Azure [29] and Alibaba Cloud [3] have adopted RDMA over Converged Ethernet on a large scale in their data centers to speed up the performance of processing large amounts of data and achieve minimal CPU overhead. Emerging distributed computing platforms and technologies such as FaRM [12], TensorFlow [1], and CNTK [30] also exploit RDMA.

Lossless Ethernet relies on hop-by-hop Priority-based Flow Control (PFC) to prevent buffer overflow [20]. The key idea is, when the queue length at a switch exceeds a pre-defined threshold, it sends a PFC pause frame to stop data transmission from the upstream switch; when the queue length falls below the threshold, it sends a PFC resume frame. Although effective at eliminating packet loss, PFC can induce a problem: deadlocks caused by cyclic buffer dependency (CBD), where no packets in the cycle can be propagated. Once deadlocks occur, PFC pause frames could spread to significant parts of the network fabric, causing a large percentage of flows to stop transmission. In the worst case, all ports along all paths could be paused and the whole network could be blocked.

Many large cloud providers have confirmed that deadlocks are common in practice [17], [33], [36]. Deadlocks could happen when routing rules form a loop [21], but it is not a unique product of routing loops—recent work has shown that even for tree-based topology with up-down loop-free routing, deadlocks could still occur due to link failures [28], [38], [42], complex network updates [21], port flaps [26], and misconfigurations [22]. Furthermore, deadlocks do not recover automatically even after the problems (e.g. transient loop) that caused the deadlock formation have been fixed [18].

Approaches to combat deadlocks fall into two categories: avoidance and detection/resolution. On one hand, most recent research efforts focus on deadlock avoidance [10], [11], [18], [19], [25], [33], [34], [40], [44], but none of them is foolproof (see §II for discussions). Since no avoidance method can absolutely prevent deadlocks, an efficient and accurate deadlock detection method is a necessity. On the other hand, existing methods for detecting and resolving deadlocks in networks [27], [35] are insufficient to meet today's stringent performance requirements as they operate in the control plane at an inherently much longer time-scale than the high-speed data plane (see §II for discussions).

In this paper, we propose ITSY—a novel, fast, and efficient deadlock detection and resolution mechanism entirely performed in the data plane. ITSY is compatible with any network topologies and routing protocols. Rather than continually monitoring the throughput and queue occupancy of each switch port which incurs high overhead, ITSY only triggers the detection process when pause events happen. Instead of recording information for each switch in the traversed network path, ITSY stores a small amount of information at switches independent of the path length. ITSY also provides a basis to analyze the initial trigger of a deadlock, which helps to address deadlock recurrence. Different deadlock resolution mechanisms are provided with different memory usage, data complexity, and traffic modulation trade-offs, and network operators could choose based on their own preferences. We have implemented ITSY for programmable switches in the P4 language [9]. We experimentally compare ITSY against existing solutions and find that ITSY can detect deadlocks accurately with many times shorter reaction time and lower overhead. We demonstrate the benefit of resolving the initial trigger for deadlock recurrence prevention. We further experimentally explore the quantitative trade-offs between the deadlock resolution mechanisms to guide network operators to choose a most suitable option.

II. Motivation

A. Deadlock Avoidance - Not Foolproof

Restricted routing. The most common solutions for deadlock avoidance are to restrict routing paths and avoid the formulation of CBD [11], [34]. However, routing restrictions not only waste link bandwidth and reduce throughput [18], but also are incompatible with some topologies [27] and routing protocols such as OSPF and BGP [36]. Furthermore, when some links are down, rerouting could still create CBD and lead to deadlocks [26], [42].

Buffer management. Another method is to assign packets different priorities hop-by-hop and put packets into different buffers accordingly [40]. The required number of priorities increases with the network scale. However, since the lossless nature of PFC requires sufficient buffer space for in-flight packets before pause message takes effect, only two to three lossless priorities can be used in practice even for switches with eight traffic classes, especially with the trend toward shallow buffers in modern data centers [16], [19], [43].

PFC pause frame restrictions. Recent proposed congestion control [10], [31], [44] can reduce the possibility of pause. Also, operators can limit pause frame propagation by assigning different PFC thresholds to switches based on their topological positions [18]. Although these methods can reduce the possibility of a deadlock, they cannot absolutely prevent deadlocks.

TTL-based mitigation. Under specific fat-tree topologies, it is possible to adopt a small TTL value that guarantees no packets can traverse a routing loop before getting dropped [18]. Although this method could eliminate packets traversing routing loops (one of the root causes for deadlocks), it only works for specific topologies and cannot prevent deadlocks formed by other root causes like link failures.

Bandwidth reservation. Tagger [19] requires to reserve the bandwidth for lossless packets forwarded through a list of pre-defined paths. However, the whole network is no longer lossless since even under normal network states, traffic not traveling on such paths could be dropped to ensure that the PFC would not be triggered. The involved manual configuration is also well-known to be error-prone [7], [8].

Rate limiting. Rate limiting is used to break the necessary condition—*hold and wait*—for the deadlock [33]. Nonetheless, manipulating the rate at fine granularity requires good control over the adjustment periods and queue variation, which cannot always be guaranteed to be precise enough.

Summary. Existing deadlock avoidance approaches address the problem to some extent, but they are not foolproof. Thus, deadlock detection is an important and necessary fail-safe.

B. Existing Detection Solutions Fall Short

Existing deadlock detection solutions rely on a centralized controller or switch local control planes [27], [35] to query port states and detect deadlocks. The programmability of control planes also enable flexible rerouting or draining strategies on switches for recovery. However, inherent delays between data planes and control planes together with the software delays of control plane applications make these solutions unable to response to deadlocks fast enough.

In addition, existing solutions detect deadlocks by proactively monitoring for blocked ports. Concretely, if the throughput of a port is zero while the corresponding queue length is non-zero, the port is regarded as a suspected port that can form deadlocks. However, the overhead of proactive monitoring is very high as it requires the periodic inspection of all ports of all switches in a network. In a normal network, most of time, the inspection will find nothing wrong.

Furthermore, current deadlock detection solutions are unable to eliminate the root cause of the detected deadlock. Therefore, even if the deadlock can be broken and the traffic flow can recover temporarily, none of them is able to prevent the same deadlock from reappearing again.

C. New Opportunity for Deadlock Detection

For each shortcoming of existing solutions, we propose a new alternative design strategy in ITSY.

Detecting deadlocks in the data plane. ITSY exploits the trend that switch data plane hardware in data centers is becoming increasingly programmable [5], [41]. Specifically, programmable parsers and deparsers enable us to customize packet headers. Metadata for deadlock detection can thus be piggybacked onto pause frames. Second, the provided stateful memory and ALUs make it possible to maintain state information directly in the data plane. A deadlock detection algorithm performed entirely in the data plane eliminates the high overhead introduced by interacting with the switch control plane. Finally, the data plane runs at line speed, which allows quick reaction when a deadlock occurs.

Detecting deadlocks reactively. Rather than periodically checking the status of switch ports, ITSY triggers the deadlock detection process only when an initial pause event happens. This has several advantages. First, the detecting process follows the direction of pause frame propagation, which greatly reduces the network overhead and switch memory consumption. Second, being triggered by the initial pause event, ITSY can detect deadlocks almost immediately.

Preventing recurrence of the same deadlock. Simply breaking the deadlock by adjusting a switch on the CBD cycle might be insufficient, as a chain of similar pause events could occur again and cause the same deadlock. Identifying the switch that instigates the pause events leading to a deadlock, termed the initial trigger, is therefore a crucial first step in diagnosing the root cause of the deadlock (e.g. a malfunctioning NIC or switch port) so as to prevent a recurrence. ITSY is designed to reveal the initial trigger to the network operator to facilitate such diagnosis.

III. Deadlock detection

ITSY leverages the programmable data plane to detect deadlocks and identify initial trigger nodes. Detection processes are based on different locations of the initial trigger - on the loop or out of the loop, respectively. A port-based data structure is used to keep track of causal relationships between pause events generated at different switches. ITSY attaches metadata for deadlock detection to pause frames or synthesized packets. Once a deadlock is detected, the initial trigger provides clues to mitigate potential pause events later and hence prevent the same deadlock from recurring. ITSY's correctness is based on two guarantees—1) spatial: a chain of pause events triggered hop by hop (causality-chain) forms a

Symbol	Meaning
S_{tri}	Node ID of the initial trigger
P_{tri}	Port ID of the initial trigger that sends pause frame
$S_{gen-ini}$	Node ID of the generic initiator
$P_{gen-ini}$	Port ID of the generic initiator that sends pause frame
Seq_{id}	Sequence number of checking message sent by $S_{gen-ini}$
S_{cur}	Switch ID of the current switch
ξ_p	Set of causal ports sending traffic to port p at current switch
δ_p	Set of ports that pause the upstream and be causal with port p
r_p	RESUME tag for port p of the current switch

TABLE I: Meaning of symbols used in this paper

loop (causality-loop) 2) temporal: all nodes on the causality-loop are paused simultaneously, indicating a real deadlock rather than just a CBD scenario. ITSY makes no assumptions about the topologies and routing algorithms in use.

A. Identifying the Initial Trigger

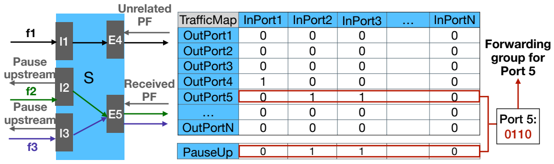
The initial trigger, which can be a server or a switch, is at the beginning position of the causality chain. A server generating a pause frame is immediately identified as an initial trigger since it is the destination of flows in the network. In the case of a switch, when an ingress port generates a pause frame due to the congestion at a corresponding egress port, it checks whether the egress port is paused or not. The switch is identified as an initial trigger if the egress port is not paused.

B. General Primitives for Deadlock Detection

Before discussing deadlock detection based on the location of initial triggers, we present primitives for solving two basic aspects of deadlock detection. Such primitives can be applied to cover different scenarios, including the initial trigger on the deadlock loop or out of the deadlock loop. The meanings of the symbols used in the following are displayed in Table I.

Port-based causality data structure. The port-based causality data structure maintains the causality relationship between different switch ports, as shown in Figure 1. For a switch with N ports, each port maintains a bit-map of size N to track all the relevant ports that send traffic to its egress queue, which we call a traffic mapping. Each bit indicates whether there are active packets transmitting from a certain ingress port to a certain egress port. In the above example, the egress queue of port $E4$ is occupied by packets of flow $f1$ from ingress port $I1$, setting the corresponding position in the traffic mapping to 1. It will be cleared to 0 when no active packet is in the egress queue. Similarly, for port $E5$, the bits for $I2$ and $I3$ are set to 1. Each port also maintains a bit that represents whether the corresponding port currently pauses the upstream switch. In the example, port $I2$ and $I3$ have already sent pause frames, the corresponding values are set to 1. When receiving a new pause frame, the switch would query the traffic mapping as well as the port state to determine the relevant ports for deadlock detection. As an example, if $E5$ receives a pause frame, the switch will traverse the bit-map for $E5$ and the port states of all ports. Since $I2$ and $I3$ both have causality with $E5$ and currently pause the upstream, the metadata for deadlock detection is forwarded to $I2$ and $I3$.

Causality-loop primitive. The causality-loop can be determined when the causality chain of pause frames has visited the same port of the same switch twice. The switch that suspects a causality-loop and initiates detection is called the generic initiator. Messages used for tracing the causality-chains are called checking messages. The packet header of a checking message is extended to record the unique ID $\{S_{gen-ini}, P_{gen-ini}\}$ of the generic initiator, as well as the Seq_{id} sent by the generic initiator. The Seq_{id} represents a unique episode of causality-loop detection. It is used to decompose different causality-chains from the same generic initiator when resume and pause frames alternate. The method works as follows.



TrafficMap	InPort1	InPort2	InPort3	...	InPortN
OutPort1	0	0	0		0
OutPort2	0	0	0		0
OutPort3	0	0	0		0
OutPort4	1	0	0		0
OutPort5	0	1	1		0
...	0	0	0		0
OutPortN	0	0	0		0
PauseUp	0	1	1		0

Fig. 1: Port-based causality data structure

1) The generic initiator sends a checking message with its $\{S_{gen-ini}, P_{gen-ini}, Seq_{id}\}$ in the packet header. The selection of the generic initiator and when checking messages are sent are different for different use cases based on the location of the initial trigger (details in sections III-C and III-D).
2) When receiving a checking message, non-generic initiator switches parse and store the received $\{S_{gen-ini}, P_{gen-ini}, Seq_{id}\}$ in the data plane at the receive port, which are then used for generating the next checking message. The Seq_{id} is updated when receiving a newer checking message (higher Seq_{id}) from the generic initiator. The stored generic initiator info at a port is removed after receiving a resume frame.
3) When port p receives a checking message, non-generic initiator switches query the port-based causality data structure to obtain corresponding ports δ_p that currently pause the upstream and have causality with port p. If $\delta_p = \emptyset$, the switch will drop the current checking message and wait for the generation of the next checking message (see next bullet). Otherwise, the switch will forward the checking message to all ports in δ_p.
4) A non-generic initiator switch generates a new checking message when a pause frame is triggered by congestion on an egress port p. This new checking message will carry the corresponding $\{S_{gen-ini}, P_{gen-ini}, Seq_{id}\}$ stored at port p.
5) When a switch S_{cur} receives a checking message from port p whose $S_{gen-ini}$ is S_{cur}, Seq_{id} is the latest, and $P_{gen-ini}$ belongs to ξ_p, the causality-chain has passed the same port of the same switch again. A causality-loop is determined and a deadlock is potentially formed.

Lemma 1: The causality-loop is detected when the received checking message $\{S_{gen-ini}, P_{gen-ini}, Seq_{id}\}$ satisfies $S_{gen-ini} = S_{cur}$, $P_{gen-ini} \in \xi_p$ where p is the port receiving the checking message, and Seq_{id} equals the sequence number at $P_{gen-ini}$.

Proof: The checking message is propagated from a switch $S_{gen-ini}$ following the chain of pause events. When S_{cur} = $S_{gen-ini}$, the pause chain has returned to the generic initiator and thus a loop is formed. The next step is to make sure that the ports on the loop have correct causal relationships. The causality of other ports on the loop, except p at S_{cur} (e.g. $P1$ in Figure 2), is guaranteed by the characteristics of PFC pause frames due to rules 3 & 4 above. At S_{cur}, if $P_{gen-ini}$ belongs in ξ_p (e.g. $P2 \in \xi_{P1}$ in Figure 2), a causal relationship between p and $P_{gen-ini}$ is confirmed; and as the sequence number at $P_{gen-ini}$ equals Seq_{id}, a complete causality-loop belonging to the same detection episode is confirmed. □

Temporal consistency primitive. The paused ports on a detected causality-loop might be resumed during the process of causality-loop detection. To detect a true deadlock, the temporal consistency primitive checks if every port on the causality-loop is continuously paused since the initial pause event was triggered. Each episode is determined by the Seq_{id} of the generic initiator. Each port on a switch maintains a RESUME tag r_p representing whether the pause is resumed during the current detection episode. It is set to zero in each episode when the corresponding port is paused, and updated to 1 when receiving a resume frame. Upon detecting a causality-loop, the data plane of $S_{gen-ini}$ generates a temporal consistency check packet carrying $\{S_{gen-ini}, P_{gen-ini}, Seq_{id}\}$ and sends it through $P_{gen-ini}$. When the packet is received at port p of a switch, it is forwarded to ports that belong to the causal ports ξ_p and have sent PFC pause frames with the same $\{S_{gen-ini}, P_{gen-ini}\}$. Each switch port that receives the temporal consistency check packet would check the corresponding RESUME tag and the stored Seq_{id}. A deadlock is determined if all switches on the causality-loop maintain false RESUME tags and the same Seq_{id} as that during causality-loop detection.

Lemma 2: The temporal consistency is satisfied if $r_p = 0$ at every switch port p along the causality-loop and the ports all have the same sequence number as the Seq_{id} in the temporal consistency check packet.

Proof: 1) If $r_p = 1$ at some switch: port p is resumed without being paused again. At least one node on the causality-loop is resumed and not paused at the same time as others. 2) If $r_p = 0$ but Seq_{id} is not found at some switch, there must be a new pause frame after a resume frame, which induces a new independent episode of detection. $r_p = 1$ followed by $r_p = 0$ cannot happen in the same episode. Even if $r_p = 0$ holds for all switches on the loop, the temporal consistency is unsatisfied as multiple episodes are involved. 3) If $r_p = 0$ holds for all switches in the episode represented by the same Seq_{id}, no resume frame is transmitted during this episode. All ports maintain a paused state in the entire episode and the temporal consistency is confirmed. □

Theorem 1: A deadlock is determined by $S_{gen-ini}$ on the causality-loop, when $Seq_{id\_cau} = Seq_{id\_tem}$ where Seq_{id_cau} represents the episode when a causality-loop is detected and Seq_{id_tem} represents the episode when temporal consistency is confirmed.

Proof: Based on Lemma 1, the checking message with Seq_{id_cau} spreads along the causality-loop and comes back to the beginning position as $S_{gen-ini}$. Based on Lemma 2, all nodes on the causality-loop are guaranteed to maintain a paused state in the episode of Seq_{id_tem}. If these nodes also remain paused in the episode of Seq_{id_cau}, Seq_{id_cau} must equal to Seq_{id_tem}, representing the same episode. Therefore, the causality-loop is paused for the the whole detecting episode, and thus a deadlock is determined. □

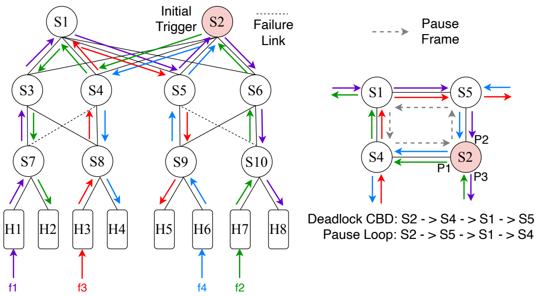


Fig. 2: Example of deadlock in the Clos network topology when initial trigger is on the loop

C. Initial Trigger on Loop

When the initial trigger switch is part of the deadlock CBD loop, the deadlock can be detected from the loop itself. Figure 2 shows an example of this case in the Clos network topology. The principle for deadlock detection is identical to the general primitives previously described in Section III-B.

Generic initiator selection. In this case, the initial trigger node is regarded as the generic initiator since it is at the beginning of the causality-chain and the causality-chain itself forms the CBD cycle.

Checking message propagation. Whether it is the initial trigger or non-initial trigger switch sending out a PFC pause frame, the current checking message $\{S_{tri}, P_{tri}, Seq_{id}\}$ is piggybacked onto the PFC pause frames. When a non-initial trigger switch detects that the next hop of the causality-chain has already been paused, no new pause frame can be generated. The checking message $\{S_{tri}, P_{tri}, Seq_{id}\}$ is then forwarded with a different priority. Notice that the checking message is in the opposite direction of the deadlock loop, and thus would not be blocked by the currently congested ports. Based on the causality-loop primitive, if a deadlock exists, the causality-loop must be detected by S_{tri}.

Temporal consistency guarantee. The temporal consistency primitive can take effect in this case by choosing the initial trigger as the start of the temporal consistency check. Deadlock is determined when the temporal consistency primitive holds.

D. Initial Trigger out of Loop

Some practical deadlock scenarios are affected by pause events sent from the initial trigger out of the deadlock loop. As shown in Figure 3, a misbehaving server $H8$ as the initial trigger node continually sends pause frames to the upstream, leading to a deadlock between $S2$, $S5$, $S1$ and $S4$. An indication of this case is that a middle switch receives multiple pause frames with the same $\{S_{tri}, P_{tri}\}$ from different ports. In Figure 3, this phenomenon is detected at $S2$ when it receives two pause frames with the same $\{S_{tri}, P_{tri}\}$ from $S6$ and $S4$.

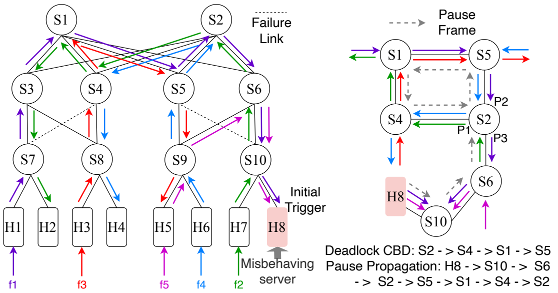


Fig. 3: Example of deadlock when initial trigger is out of loop

Lemma 3: If deadlock exists and S_{tri} is out of the causality-loop, one switch (called the middle switch) must receive at least two pause frames with the same $\{S_{tri},\ P_{tri}\}$ from different ports.

Proof: Once the causality-loop is formed and S_{tri} is out of the loop, there must be a switch at the junction between the inside and outside of the loop, such as $S2$ in Figure 3. This switch must have received pause frames from at least two directions. One is from the outside downstream switch (e.g., $S2\_P3$ receives PF from $S6$), the other one is from the switch on the causality-loop (e.g., $S2\_P1$ receives PF from $S4$). As both pause frames trace back to S_{tri}, the received $\{S_{tri},\ P_{tri}\}$ from different ports must be the same. □

Generic initiator selection. The middle switch receiving the same $\{S_{tri},\ P_{tri}\}$ from different ports is selected as the generic initiator to start a new process of causality-loop detection. Although unlike the on-loop case where the causality-loop is determined immediately upon being formed, the out-of-loop case can start the causality-loop detection no later than the formation of causality-loop.

Checking message propagation. As the middle switch is selected as the generic initiator, the $\{S_{gen-ini},\ P_{gen-ini}\}$ used for checking message in this case is the SwitchID and PortID of the middle switch, which we called S_{middle} and P_{middle}. When the causality-loop detection is triggered, the middle switch has received two potentially different Seq_{id} from two pause frames at different ports. The Seq_{id} from the second pause frame is used in the deadlock detection checking message. The middle switch generates a packet to carry the checking message $\{S_{middle}, P_{middle}, Seq_{id}\}$, which is forwarded along the causality-chain.

Temporal consistency guarantee. Temporal consistency primitive is invoked by choosing the middle switch as the start of the temporal consistency check.

IV. Deadlock resolution

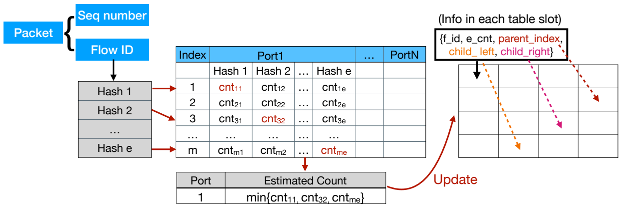
Upon detection of a deadlock, actions must be taken to resolve the deadlock. However, none of the existing solutions for deadlock resolution takes the data plane features into consideration. Also, even though temporary rerouting is a common method used for breaking a deadlock [27], it is not always viable if not all flows have multiple paths or if flows have routing policy restrictions. In addition, rerouting may create new congestion and deadlocks on other parts of the network. Therefore, the deadlock should be better resolved in the data plane without requiring route changes in the control plane. We present three deadlock resolution mechanisms that provide different trade-offs between memory usage, data complexity, and traffic modulation. All the mechanisms rely on detecting the deadlock with the initial trigger as the first step, however, they differ by the method of resolving the deadlock.

A. Count-min Based Drop

Although designed for lossless networks, protocols like RoCEv2 could still tolerate a small packet loss rate, i.e., 0.0001 [44]. Based on this property, a count-min based drop mechanism is proposed to break the deadlock. **Goal: Resolve the deadlock with an acceptable drop rate while minimizing affected traffic.** That is, the network becomes unblocked with only a little buffered traffic being dropped, and the actual performance impact is guaranteed to be tolerable.

Design 1: Identify the top-k largest flows for dropping packets. Since network traffic usually follow the power-law distribution [24], most packets that cause congestion and pause events are likely from the largest flows. Therefore, the count-min based drop mechanism starts to drop packets from the top-k elephant flows (k is a tunable parameter). We keep track of the top-k flows with a heap structure implemented with a consolidated table proposed by Qian et al. [32]. As shown in Figure 4b, each of the k table slots maintains a flow ID, estimated flow size, and the indices of the parent and children nodes in the heap. When a packet propagates through the switch, only one memory access is needed to retrieve the table slot for the root of the heap, which would be updated if it is smaller than the current flow size. Also, the indices of parent and children slots are updated only when necessary. As a result, lookup and update are both fast. Network operators can also provide their own preference, such as different k values, special lossless requirements for specific flows/paths, and different packet loss tolerance for different applications.

Design 2: Estimate the flow size and determine the loss tolerance with count-min sketch. Both top-k flow identification and loss rate calculation require the estimation of flow size which could be achieved by a count-min sketch. As shown in Figure 4, for each egress port, the switch maintains a count-min sketch with m rows and e cells per row based on the number of hash functions. For each packet, the switch will compute a series of hash values for the flow ID (e.g., 5-tuple) in order to update corresponding slots for that flow. Each slot provides an approximate count which would be updated when the sequence number of current packet is larger than that. To query the size of a flow, the minimum value of all mapping approximate counts would be returned. Notice that regular count-min sketch can only support update action, and the estimation accuracy decreases with the number of flows inserted. Also, only the active flows in the network is meaningful for resolving deadlock. To solve these concerns, we provide a reduction action that when detecting the completion of a flow (e.g., a FIN flag in the TCP header), the estimated count of such flow would be subtracted from the total traffic meter. Based on such flow size estimation with count-min sketch, the acceptable drop size could be obtained.



(a) Count-min sketch for ITSY (b) Top-k table for ITSY

Fig. 4: Update and query the count-min sketch for resolution

Design 3: Drop packets gradually within the tolerable packet loss rate. To break the deadlock, at least one packet worth of traffic should be dropped. This mechanism gradually drops more packets to guarantee the network could restore to a normal state. When a drop occurs, the queue of the paused port would be shortened and traffic begins to drain from the loop. If the queue lengthens again during the resolution before falling below the PFC threshold, twice the number of packets would be dropped than in the previous round. Based on the number of dropped packets and the estimated count from count-min sketch, the mechanism would stay within the tolerable packet loss rate requirement.

This mechanism is practical, no operation is needed for new arriving traffic, and no additional network bandwidth is consumed. However, it requires the network to accept certain packet losses and the data structures incur some memory overhead on the switch.

B. *Port-level Resolution*

Port-level resolution mechanism is proposed in case that network operators have stringent requirements on the lossless features. **Goal: Resolve the deadlock without packet loss while still not involving changes to the network routing.** It should provide feasible strategies to obtain extra available buffer as well as avoid the interference of external traffic.

Design 1: Dynamic buffer allocation to absorb queuing packets. This mechanism achieves dynamic adjustments to the headroom buffer region which is originally shared by all ports and used to absorb packets before pause frame takes effect. Once a deadlock is detected, the switch that declared the deadlock would shift extra buffer space from the headroom region to the queue on the deadlock loop. The declared switch could therefore un-pause the upstream switch and the traffic on the deadlock loop could then make progress.

Design 2: Completely block external traffic that might compete for the available buffer. Note that there could be some external traffic being forwarded to the on-loop switch port while not arriving from the port on the loop. Such traffic could interfere with deadlock resolution by competing for the buffer space against the traffic on the deadlock loop. To avoid such interference, the port-level resolution mechanism only resumes the traffic that arrives from the ports on the causality loop. External traffic from out-of-loop ports are not allowed to be transmitted until the queue decreases below the PFC threshold. Take the scenario in Figure 2 as an example. Flow f1 and f4 could resume transmission at the initial trigger switch S2, while flow f2 would still be suspended. Therefore, the traffic load of blocked ports on the causality loop could decrease, thereby reducing the queue occupancy and eliminating the deadlock.

Design 3: Determine the ports of external traffic interfering with deadlock resolution. To determine the ports that do not belong to the deadlock loop, corresponding port-loop relationships should be obtained. During the temporary consistency check, a list of switch ports on the causality loop is recorded. During the deadlock resolution, the resume frame is sent with the recorded port list. The upstream switch would recognize the next causal port for resume and then remove the corresponding field from the list. Therefore, the additional overhead of the packet would be very trivial.

This mechanism breaks the deadlock by achieving external traffic blocking at the port level without any loss of packets in the network. Also, it requires less buffer space compared with the existing buffer management for deadlock avoidance [40] since it is independent of the flow path length and even a single packet size adjustment is enough to break the deadlock.

C. *Flow-level Resolution*

Under port-level resolution, some of the blocked traffic from out-of-loop ports may actually be forwarded to the out-of-loop egress ports and would not interfere with the deadlock resolution. Therefore, flow-level resolution mechanism is proposed as an extension to port-level resolution. **Goal: Provide more fine-grained flow control during deadlock resolution.** Both buffer adjustment and port-loop relationship determination of this mechanism are the same as the port-level one, but flow-level selective blocking for external traffic is added.

Design 1: Only block external flows that enter the loop. This mechanism provides a flow-level pause/resume strategy, coexisting with the current port-level PFC strategy. During deadlock resolution, the ports that belong to the causality loop would trigger the port-level resume frame as before. In contrast, the out-of-loop causality ports would trigger the flow-level resume frame, resuming the flows that do not enter the loop. For new arriving traffic, the flow-level pause frame would only pause the flows forwarded to the paused ports as well as enter the deadlock loop. Once the queue of paused ports on the causality loop decreases below the PFC threshold, the deadlock resolution phase ends and a normal port-level resume frame would be triggered.

Design 2: Resume and pause flows based on ECMP group information. Rather than encoding every flow's 5-tuple in the flow-level PFC frames, we use ECMP group information to represent a group of flows transmitted to the paused outgoing ports on the loop. All traffic flows with the same ECMP group information are not allow to be transmitted, thus the messaging overhead could be significantly reduced. For example, in Figure 5, the ECMP group g_d1 at switch D is meant to be forwarded to the paused port on the deadlock loop. During deadlock resolution, the flow-level frame is sent with a corresponding list of ECMP groups g_d1, g_e1, g_g1. The irrelevant traffic (e.g., pink one) with a different ECMP group could be resumed while the interfering traffic (e.g., green one)

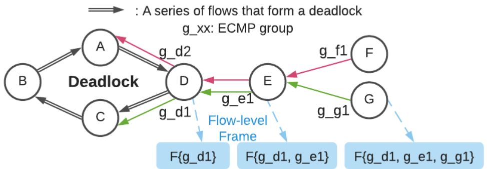


Fig. 5: Flow-level resolution based on ECMP groups

would remain in paused state. For new arriving flows, the switch connected to the host would pre-calculate the ECMP group for the next few hops and flows that belong to the known interfering ECMP groups would be blocked. This mechanism minimizes the impact on other traffic that do not interfere with deadlock resolution. However, the transmission of the messages carrying ECMP groups would somewhat increase bandwidth consumption.

V. Discussion

Initial trigger handling. In addition to breaking the deadlock, the recurrence of deadlock could be further prevented by handling the initial trigger. If the initial trigger is a switch, one crucial strategy is to identify the heavy hitters which send a large amount of traffic and thus cause the congestion. The count-min based drop mechanism can be used for heavy hitter detection, and once identified, further steps from recent proposals can be leveraged to limit the heavy hitters [39].

The initial trigger may also be a server when there is a flow control issue or a malfunctioning NIC. Due to limited memory resources in the NIC, a flow control issue may cause thousands of PFC pause frames to be sent per second from a server [16]. In addition, bugs in the receiving pipeline of the NIC can cause the server to be unable to handle the received packets and continually send pause frames [16], [44]. To handle such an initial trigger server, a micro-controller of the NIC could prevent it from generating pause frames or the connected switch would disable lossless mode since the NIC is not functioning at all.

Concurrent deadlocks. Multiple concurrent deadlocks might exist in the network. If they occur at completely different positions without any shared parts on the loop, both deadlock detection and resolution are independent. Otherwise, if they involve overlapping nodes or links, the deadlock detection could still be distinguished with different causality loops and episodes, while the deadlock resolution may need additional operations. The count-min based drop mechanism could work as normal, while the port-level and flow-level resolution could experience unexpected interference, as different overlapping deadlocks may conflict in their need to block external traffic. To solve this issue, a priority operation could be added to guarantee that the switch is not allowed to start a new deadlock resolution until the current one is completed.

VI. Evaluation

Setup. We have set up a simulation environment using the BMv2 software switches in the NS3 simulator. The experiments are performed in the CloudLab platform [13], each node has eight-core 2.0GHz CPU and 32GB of RAM. We have evaluated our system on the VL2 topology [14] and a large fat-tree topology [2] while mainly focusing on a region of 20 switches. Multiple deadlock scenarios are created with different initial trigger locations, different deadlock loop length, different involvement of edge, aggregation and core switches and so forth.

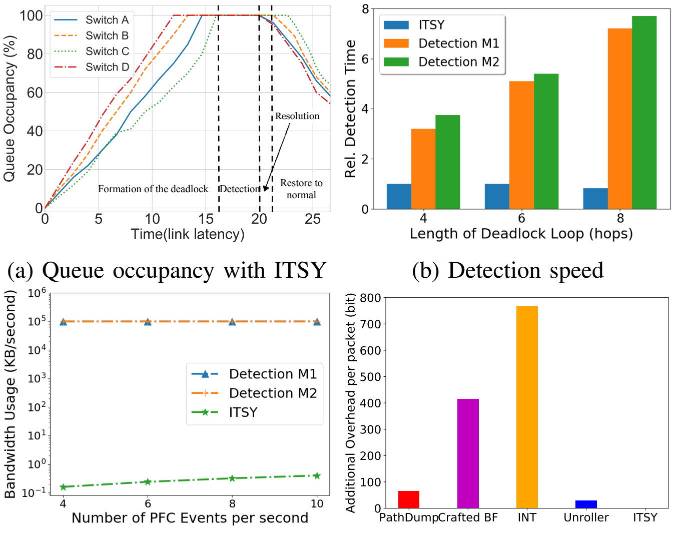
Workloads. We simulate empirical workloads from production networks for our evaluation. The flow size distribution is taken from a data center network supporting web search [4] with a diverse mix of small and large flows. 30% of the flows are 1–30MB, so that multiple large flows can be concurrently active from/to one switch port. The arrival time of different flows is based on a Poisson process and flow arrival rate is varied to obtain different load utilizations in the network. The source and destination for traffic are chosen uniformly at random and TCP transport is used. To guarantee the formation of deadlock, we inject several persistent non-congestion controlled flows to create congestion.

A. Demonstration of ITSY

We first demonstrate that ITSY could response quickly to the deadlock as well as successfully resolve the deadlock. The detection and resolution behaviors are evaluated by deadlocks created with two failed links which then lead to rerouting and finally a deadlock loop. Although the resulting congestion could spread to a large number of upstream switches, we mainly measure the queue occupancy of switch ports on the deadlock loop over time. Figure 6a displays the results of a deadlock with 4 switches involved in the loop and we use the one-hop link latency as the unit for time measurement since link latency determines how fast ITSY messages propagate. The queue occupancy increases first, then reaches the PFC pause threshold. The pause frames would be propagated and eventually form the deadlock at the time 16. The switch declares the deadlock after 4 link latency time units (needed for the temporal consistency check step). Then ITSY starts to resolve the deadlock and the hold-and-wait situation could be broken after 1 link latency. The queue occupancy would continue to decrease, and the network would return to a normal state. In all test cases, ITSY has no false positive and no false negative. In cases without causality-loop, no deadlock is declared by ITSY. In cases that a causality-loop is formed but then one of the ports is immediately resumed, the temporal consistency check could detect this situation without mistakenly declaring any deadlock.

B. Comparison with Control Plane Based Solutions

Existing control plane based solutions. We compare ITSY against two existing deadlock detection methods proposed by Shpiner *et al.* [35]. Both methods are control-plane based and need to proactively monitor all switch ports in the network. The detection message is sent in the direction of deadlocked traffic forwarding so that it requires the switches to support enough traffic priorities. 1) **Detection method 1 (M1)**: Each deadlock-suspected ports periodically send the detection packet with a randomized unique identifier. The next-hop switch would compare the received identifier to the one sent



(a) Queue occupancy with ITSY

(b) Detection speed

(c) Bandwidth overhead

(d) Per-packet overhead

Fig. 6: (a) Demonstration (b) (c) Compare with control plane based methods (d) Compare with data plane based methods

by the switch. If the received identifier is smaller, the switch would replace its original identifier with the received one in the next sending period. A deadlock is declared when the received identifier is equal to the one generated by the switch. Each detection packet has only a one-hop lifetime and each switch needs to send multiple rounds of packets in order to successfully detect the deadlock. 2) **Detection method 2 (M2)**: Similar to the previous method, the detection packets are sent through the deadlock-suspected ports periodically. However, the full path information, including the unique identifiers, a group of egress ports and a special index, would be maintained in the packet header. The switch would check whether its own information is found in the incoming packet to determine a deadlock. As these packets store the path information, their size grows linearly with the network diameter.

Detection speed. We first measure the detection speed for different solutions to show how fast they could react to the deadlock scenario. Figure 6b displays the normalized detection time under different length of the deadlock loop, with ITSY as the baseline. We ignore the software delays of the control plane for processing the monitored packets in M1 and M2; if such delays are counted, the actual detection time would be somewhat higher. It could be observed that the time required for detecting deadlocks with *M1* and *M2* is at least 3.2× and 3.75× higher than that with ITSY, respectively. The improvement of ITSY on detection speed becomes even more obvious with increasing loop length. This is because ITSY tightly follows the pause events propagation while the other two relies on periodic inspection. Also, the two existing methods would experience extra delays between the switches and the centralized controller.

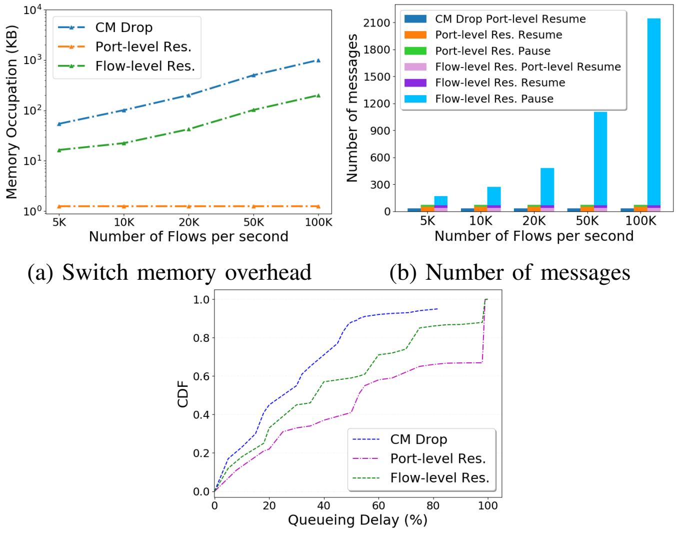
Additional bandwidth overhead. To compare the network overhead between these solutions, we measure the amount of extra bandwidth usage during detection under different number of pause events per second, as shown in Figure 6c. Since the checking message of ITSY is only sent when there is a pause event, the overall network bandwidth consumption is vanishingly small (less than 1 KB per second). In contrast, the other two deadlock detection methods must continually collect information from all switch ports in the network even if no pause frame is generated.

C. Comparison with Data Plane Based Solutions

Existing data plane based solutions. We further compare ITSY against state-of-the-art data plane based mechanisms for detecting forwarding loop. Although these methods are designed for forwarding loop detection and not for PFC deadlock detection, we found that they are quite general and could be applied to detect the causality-loop of deadlock, one of the crucial steps in ITSY. 1) **PathDump** [37]: tracking the paths and detecting routing loops with a special VLAN tag. It requires certain restrictions on the network topology. 2) **Crafted BF** [23]: a method that adds a probabilistic bloom filter into packets to store switch information. 3) **INT** [15]: each switch records their own ID in the incoming packet. It detects the loop at the cost of significant packet header space which grows linearly with the network diameter. 4) **Unroller** [23]: encoding a varying fixed-size subset of the traversed path based on the minimal identifier of switch to detect the routing loop.

Per-packet overhead. We measure the overhead required for each packet with the fat-tree topology involving 20 nodes, as shown in Figure 6d. PathDump adds a fixed overhead on each packet. The per-packet overhead of Crafted BF and Unroller is the minimum value that can guarantee no false positives. The overhead of INT is based on tracking a path with 8 hops, it could be even larger with an increasing number of traversed hops. In contrast, ITSY does not add information in every data packet, it reactively detects the causality loop and piggybacks information only on pause frames. Therefore, there is no per-packet overhead for ITSY.

Number of hops required for detection. Among all the above alternative approaches, Unroller is the latest and requires much fewer bits than the others. Therefore, we further compare ITSY against Unroller based on the number of hops needed for detection in the worst cases. Assume L is the number of switches in the loop and B is the number of hops before the loop. For the *on loop* scenarios, ITSY relies on the initial trigger, and the causality loop is detected within L hops when the initial trigger has passed the same port of the same switch twice. However, Unroller relies on the minimal identifier encountered, and needs at most $2L-1$ hops to detect the loop: 1) the detection packet needs up to $L-1$ hops to reach the switch with the smallest identifier; 2) another L hops is needed thereafter for the detection packet to reach the smallest-identifier switch again and report the loop. For the *out-of-loop* scenarios, ITSY needs $L+B$ hops to find the generic initiator, namely, the middle switch on the loop; then it needs L more hops to detect the loop. In contrast, in these *out-of-loop* scenarios, Unroller could fail to detect the causality-loop with the single minimal identifier. To get around



(a) Switch memory overhead

(b) Number of messages

(c) CDF of queueing delay

Fig. 7: Trade-off between proposed resolution mechanisms

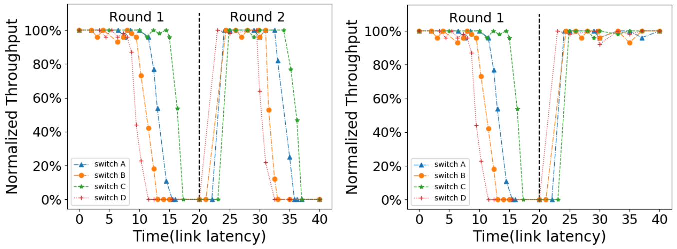
this, Unroller has to gradually reset the identifier, which would induce multiple extra passes over the loop. Consequently, Unroller would require at most $4.67 * (B + L)$ hops [23].

D. *Trade-off between Three Proposed Resolution Mechanisms*

We finally evaluate the three proposed resolution mechanisms of ITSY and discuss the trade-off using different metrics including switch resource overhead, network overhead and the impact on other traffic. Network operators may choose the preferred mechanisms according to their requirements.

Switch resource overhead. The switch resource usage of three resolution mechanisms is evaluated under different number of flows per second, as shown in Figure 7a. The memory overhead of port-level resolution and flow-level resolution mainly comes from the port-based causality data structure. We assume that ITSY could be deployed in a large-scale network with $10,000$ 64-port switches, so 14-bit SwitchID and 6-bit PortID are required to support the uniqueness requirement. The memory overhead of port-level mechanism is quite small and is independent of the number of flows per second. The flow-level mechanism needs to maintain the hash-based ECMP group information as well as the paused flows. Therefore, its memory overhead would increase with the number of flows. The count-min based drop mechanism would require additional memory for the count-min sketch and the heavy hitter data structure. However, all these mechanisms could fit comfortably in today's programmable data planes that usually have tens of MB of memory [6].

Data complexity. To analyze the data complexity of different proposed resolution mechanisms, we summarize the number of messages sent during deadlock resolution based on different message types, varying the number of flows per second. As shown in Figure 7b, count-min based drop mechanism only needs to resume the deadlock ports as normal resume frame does, so that it incurs a very small number of messages. Port-level resolution mechanism would send additional pause frame to block the ports that contain traffic which may enter the loop. The data complexity of these two mechanisms do not increase with the number of flows. In contrast, flow-level resolution mechanism requires dynamically sending the resume and the pause frame in order to achieve more fine-grained flow-level pause on external traffic, thus leading to more messages transmitted in the network.



(a) Not resolving initial trigger

(b) Resolving initial trigger

Fig. 8: Benefits of resolving initial trigger out of the loop

Impact on traffic. We evaluate the impact of different resolution mechanisms on traffic based on the queuing delay, which is normalized by the maximum queuing delay of a single switch. As shown in Figure 7c, most packets under count-min based drop experience a delay less than 50%; the CDF does not reach 100 percent since delay is undefined for dropped packets. Port-level resolution blocks all the traffic from ports outside the causality loop, so packets may experience the maximum queuing delay but no loss. Flow-level resolution affects a smaller number of packets since flows that do not enter the causality loop are not blocked.

E. *Benefits of Resolving the Initial Trigger*

To demonstrate the benefits of resolving initial triggers, we simulate a misbehaving server that continually pauses the connected edge switch that leads to a deadlock. A baseline solution is to break the deadlock without resolving the initial trigger. We measure the normalized throughput at different switch ports and results are shown in Figure 8. Although the baseline can recover from the deadlock for a while, if the traffic pattern does not change, the deadlock will reappear. ITSY breaks the deadlock, identifies the initial trigger, which can be removed to prevent the recurrence of the deadlock.

VII. Conclusion

In this paper, we propose ITSY to detect and resolve deadlocks in PFC networks. We identify the initial trigger to mitigate the recurrence of the same deadlock. The deadlock scenarios are analyzed and detected based on the location of the initial trigger. Upon detecting the deadlock, multiple deadlock resolution solutions are provided for network operators to choose based on their requirements. ITSY can be implemented entirely in the data plane, which achieves low overhead and reacts quickly to deadlocks.

Acknowledgment

We thank the anonymous reviewers for their feedback. This research is sponsored by the NSF under CNS-1718980, CNS-1801884, and CNS-1815525.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM computer communication*, 38(4):63–74, 2008.

[3] Alibaba. Alibaba cloud - super computing cluster. https://www.alibabacloud.com/product/scc.

[4] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *ACM SIGCOMM*, 2010.

[5] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker. Snap: Stateful network-wide abstractions for packet processing. In *Proceedings of the 2016 ACM SIGCOMM*, pages 29–43, 2016.

[6] Barefoot. Tofino: World's fastest p4-programmable ethernet switch asics. https://www.barefootnetworks.com/products/brief-tofino/.

[7] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 155–168, 2017.

[8] R. Birkner, D. Drachsler-Cohen, L. Vanbever, and M. Vechev. Config2spec: Mining network specifications from network configurations. In *Proceedings of 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*, 2020.

[9] M. Budiu and C. Dodd. The p416 programming language. *ACM SIGOPS Operating Systems*, 51(1):5–14, 2017.

[10] W. Cheng, K. Qian, W. Jiang, T. Zhang, and F. Ren. Re-architecting congestion management in lossless ethernet. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 19–36, 2020.

[11] J. Domke, T. Hoefler, and W. E. Nagel. Deadlock-free oblivious routing for arbitrary topologies. In *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 616–627. IEEE, 2011.

[12] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. Farm: Fast remote memory. In *11th {USENIX} Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, 2014.

[13] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra. The design and operation of cloudlab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, 2019.

[14] 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. *Communications of the ACM*, 54(3):95–104, 2011.

[15] P. A. W. Group. In-band network telemetry (int) dataplane specification. https://github.com/p4lang/p4-applications/blob/master/telemetry/specs/INT.mdk, 2018.

[16] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn. Rdma over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 202–215, 2016.

[17] D. Halperin, S. Kandula, J. Padhye, P. Bahl, and D. Wetherall. Augmenting data center networks with multi-gigabit wireless links. In *Proceedings of the ACM SIGCOMM 2011*, pages 38–49, 2011.

[18] S. Hu, Y. Zhu, P. Cheng, C. Guo, K. Tan, J. Padhye, and K. Chen. Deadlocks in datacenter networks: why do they form, and how to avoid them. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pages 92–98, 2016.

[19] S. Hu, Y. Zhu, P. Cheng, C. Guo, K. Tan, J. Padhye, and K. Chen. Tagger: Practical pfc deadlock prevention in data center networks. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, pages 451–463, 2017.

[20] IEEE. Ieee 802.1 qbb - priority-based flow control. https://1.ieee802.org/dcb/802-1qbb/, 2010.

[21] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer. Dynamic scheduling of network updates. *ACM SIGCOMM Computer Communication*, 44(4):539–550, 2014.

[22] S. K. R. Kakarla, A. Tang, R. Beckett, K. Jayaraman, T. Millstein, Y. Tamir, and G. Varghese. Finding network misconfigurations by automatic template inference. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 999–1013, 2020.

[23] J. Kučera, R. B. Basat, M. Kuka, G. Antichi, M. Yu, and M. Mitzenmacher. Detecting routing loops in the data plane. In *Proceedings of the 16th International Conference on emerging Networking Experiments and Technologies*, pages 466–473, 2020.

[24] X. Li and C. Qian. Low-complexity multi-resource packet scheduling for network function virtualization. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 1400–1408. IEEE, 2015.

[25] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh, et al. Hpcc: high precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 44–58. 2019.

[26] V. Liu, D. Halperin, A. Krishnamurthy, and T. Anderson. F10: A fault-tolerant engineered network. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 399–412, 2013.

[27] P. Lopez, J. M. Martínez, and J. Duato. A very efficient distributed deadlock detection mechanism for wormhole networks. In *Proceedings 1998 Fourth International Symposium on High-Performance Computer Architecture*, pages 57–66. IEEE, 1998.

[28] C. Lou, P. Huang, and S. Smith. Understanding, detecting and localizing partial failures in large system software. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 559–574, 2020.

[29] Microsoft. Availability of linux rdma on microsoft azure. https://azure.microsoft.com/en-us/blog/azure-linux-rdma-hpc-available/.

[30] Microsoft. Cntk-the microsoft cognitive toolkit. https://docs.microsoft.com/en-us/cognitive-toolkit/.

[31] R. Mittal, V. T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats. Timely: Rtt-based congestion control for the datacenter. *ACM SIGCOMM Computer Communication*, 45(4):537–550, 2015.

[32] C. Qian, S. Shi, X. Shi, and M. Wang. Don't work on individual data plane algorithms. put them together! In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, pages 60–66, 2020.

[33] K. Qian, W. Cheng, T. Zhang, and F. Ren. Gentle flow control: avoiding deadlock in lossless networks. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 75–89. 2019.

[34] J. C. Sancho, A. Robles, and J. Duato. An effective methodology to improve the performance of the up*/down* routing algorithm. *IEEE Transactions on Parallel and Distributed Systems*, 15(8):740–754, 2004.

[35] A. Shpiner, E. Zahavi, V. Zdornov, T. Anker, and M. Kadosh. Unlocking credit loop deadlocks. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pages 85–91, 2016.

[36] B. Stephens, A. L. Cox, A. Singla, J. Carter, C. Dixon, and W. Felter. Practical dcb for improved data center networks. In *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*, pages 1824–1832. IEEE, 2014.

[37] P. Tammana, R. Agarwal, and M. Lee. Simplifying datacenter network debugging with pathdump. In *12th {USENIX} Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 233–248, 2016.

[38] C. Tan, Z. Jin, C. Guo, T. Zhang, H. Wu, K. Deng, D. Bi, and D. Xiang. Netbouncer: active device and link failure localization in data center networks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 599–614, 2019.

[39] F. Tang, H. Zhang, L. T. Yang, and L. Chen. Elephant flow detection and differentiated scheduling with efficient sampling and classification. *IEEE Transactions on Cloud Computing*, 2019.

[40] K. D. Underwood and E. Borch. A unified algorithm for both randomized deterministic and adaptive routing in torus networks. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 723–732. IEEE, 2011.

[41] T. Wang, H. Zhu, F. Ruffy, X. Jin, A. Sivaraman, D. R. Ports, and A. Panda. Multitenancy for fast and programmable networks in the cloud. In *12th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020.

[42] X. Wu, D. Turner, C.-C. Chen, D. A. Maltz, X. Yang, L. Yuan, and M. Zhang. Netpilot: automating datacenter network failure mitigation. In *Proceedings of the ACM SIGCOMM 2012*, pages 419–430, 2012.

[43] Z. Yu, C. Hu, J. Wu, X. Sun, V. Braverman, M. Chowdhury, Z. Liu, and X. Jin. Programmable packet scheduling with a single queue. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 179–193, 2021.

[44] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. Congestion control for large-scale rdma deployments. *ACM SIGCOMM Computer Communication*, 45(4):523–536, 2015.