ITSY: Initial Trigger-Based PFC Deadlock Detection in the Data Plane

Xinyu Crystal Wu
Department of Computer Science
Rice University

T. S. Eugene Ng
Department of Computer Science
Rice University

***Abstract*—Lossless networks are increasingly popular for high-performance applications in data centers and cloud environments. To realize a lossless network in Ethernet, the Priority-based Flow Control (PFC) protocol is adopted to guarantee zero packet loss. PFC, however, can induce in-network deadlocks and in severe cases cause the entire network to be blocked. Existing solutions have focused on deadlock avoidance strategies; unfortunately, they are not foolproof. Therefore, deadlock detection is a necessity. In this paper, we propose ITSY, a novel system that correctly detects and solves deadlocks entirely in the data plane. It does not require any assumptions on network topologies and routing algorithms. Unique to ITSY is the use of deadlock initial triggers, which contributes to efficient deadlock detection and deadlock recurrence prevention. We implement ITSY for programmable switches in the P4 language. Preliminary evaluations demonstrate that ITSY can detect deadlocks rapidly with minimal overheads and mitigate the recurrence of the same deadlocks effectively.**

***Index Terms*—PFC, Deadlock, RDMA, Programmable Switch**

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 [51], [52]. 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 [33] and Alibaba Cloud [2] 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 [14], TensorFlow [1], and CNTK [34] also exploit RDMA to enhance communication in public clouds.

Lossless Ethernet relies on hop-by-hop Priority-based Flow Control (PFC) to prevent buffer overflow [24]. With the PFC mechanism, packet loss can be avoided by pausing the immediate upstream switch. Once the queue length exceeds a pre-defined threshold, the switch sends a PFC pause frame to stop data transmission from the upstream switch. If the queue length decreases below another preset threshold, the switch sends a PFC resume frame to resume transmission. As long as sufficient headroom buffers are reserved for in-flight packets before pause frames take effect, no packet would be dropped.

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 [20], [38], [45]. Deadlocks could happen when routing rules form a loop [25], 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 [31], [46], [50], complex network updates [17], [25], port flaps [29], and misconfigurations [26], [52]. Furthermore, deadlocks do not recover automatically even after the problems (e.g. transient loop) that caused the deadlock formation have been fixed [22].

Approaches to combat deadlocks fall into two categories: avoidance and detection/recovery. Most recent research efforts focus on deadlock avoidance, but none of them is foolproof. For example, some approaches require special buffer management techniques to support multiple priority classes. However, commodity switches in modern data centers can only support two or three lossless priorities in practice [19], [23]. Some schemes rely on limiting the rate of data transmission and thus avoid reaching the threshold of generating pause frames [38]. However, precise and fine-grained control of the data rate may not always be guaranteed by switch implementations. Since no avoidance method can absolutely prevent deadlocks, an efficient and accurate deadlock detection method is a necessity.

Unfortunately, existing methods for detecting deadlocks in networks are insufficient to meet today's stringent performance requirements [3], [11], [41]. First, slow indirect inspection of switch ports is adopted to find suspected paused ports that might be deadlocked. Second, centralized controllers or switch software control planes are responsible for deadlock detection; however, both are inherently too slow to combat deadlocks effectively. Third, without finding the root cause of a deadlock, even if the current deadlock is resolved, there is no guarantee that the same deadlock will not immediately reoccur again.

In this paper, we propose ITSY—a novel deadlock detection mechanism entirely performed in the data plane. ITSY reacts quickly after the deadlock formation and provides the root cause information for resolving the deadlock, regardless of network topologies and routing protocols. Rather than continually monitor the throughput and queue occupancy of each switch port which incurs unnecessary overhead, ITSY only triggers the detection process when pause events happen. Instead of recording information for each switch in the traversed network path, ITSY only stores a small set of information,

guaranteeing the overhead is independent of the path length. ITSY provides a basis to analyze the initial trigger of the current deadlock, which helps to address deadlock recurrence. We have implemented ITSY for programmable switches in the P4 language [9]. Our preliminary results show that ITSY can detect deadlocks accurately with low latency and minimal switch memory consumption (less than 1KB per switch in a 10,000 switch network). Furthermore, resolving the initial trigger can effectively prevent the same deadlock from recurring.

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 [13], [40], [43], [49]. However, routing restrictions not only waste link bandwidth and reduce throughput [22], but also are incompatible with some topologies [11], [37] and routing protocols such as OSPF and BGP [44], [45]. Furthermore, when some links are down, rerouting could still create CBD and lead to deadlocks [29], [50].

Buffer management. Another method is to assign packets different priorities hop-by-hop and put packets into different buffers accordingly [27], [47]. The required number of priorities is determined by the longest path in the network, which increases with the network scale. However, today's commodity switches can only support two to three lossless priorities in practice [19], [23], which is insufficient.

PFC pause frame restrictions. Recent proposed congestion control protocols [10], [18], [28], [35], [51] can reduce the possibility of pause. Also, operators can limit pause frame propagation by assigning different PFC thresholds to ports and switches based on their positions in the topology [22]. 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 assign an appropriately small TTL value that allows packets to traverse the network while guaranteeing no packets can traverse a routing loop before getting dropped [22]. Although this method could eliminate packets traversing routing loops, one of the root causes for deadlocks, it only works for particular topologies and cannot prevent deadlocks formed by other root causes like link failures.

Pre-configured transmission. Deadlock avoidance can also rely on configuring the network in advance and adjusting the configuration dynamically on the fly. For example, Tagger [23] pre-defines expected lossless paths and configures pre-generated match-action rules to avoid deadlock. However, it involves human intervention and complex network configuration that is pointed out to be error-prone [7], [8], [16].

Rate limiting. Rate limiting is used to break the necessary condition—*hold and wait*—for the deadlock [38]. Nonetheless, in order to limit the port rate to specific values and to periodically adjust the port rate according to the queue length, highly precise control are required. However, precise and fine-grained control of the sending rate may not be always guaranteed by switch implementations.

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 [3], [11], [30], [32], [41] to query port states and detect deadlocks. After detecting deadlocks, the control plane computes and installs rerouting or draining strategies to switches. The programmability of central controllers or local control planes enables flexible detection and recovery policies. 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. A current trend in network data centers and cloud environments is that the networking hardware is becoming increasingly programmable [48]. Recent proposed programmable data plane designed with a set of new features provides a new opportunity for deadlock detection [4], [5], [21]. First, programmable parsers and deparsers enable us to customize packet headers and protocols. 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 reduces the high overhead introduced by interacting with the switch control plane. Finally, once compiled, data plane programs are guaranteed to run at line speed, which allows us to react quickly when the deadlock is formed.

Detecting deadlocks reactively. Rather than periodically check 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 once formed.

Preventing recurrence of the same deadlock. Breaking the deadlock by adjusting a switch on the CBD might be insufficient, as a chain of similar pause events could be generated along switches on the path again and cause the same deadlock. Experimental observations from existing work [38] show that frequent pause on upstream ports is the root cause of *hold and wait*, which then forms deadlocks. Therefore, figuring out the origin of frequent pause and mitigating the following pause events help to resolve deadlocks. The Ethernet node that initially sends the pause frame and leads to the deadlock can be regarded as the initial trigger. Identifying and dealing with the initial trigger can prevent the repeated formulations of the same deadlock.

III. System design

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 focuses on two main principles—1) spatial: a chain of pause events triggered hop by hop (causality-chain) forms a 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. A switch is identified as an initial trigger if it has not received any pause frame from the corresponding downstream when generating its own pause frame.

When a pause frame is triggered at a switch port, it might be caused by previously received pause frames at other ports. It is because some of the traffic from one switch port is destined to other ports that have already been paused. Therefore, 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. An initial trigger is identified when 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 then be applied to cover different scenarios, including the initial trigger on the deadlock loop or out of the deadlock loop. A port-based causality data structure is maintained in the data plane to determine how the metadata for deadlock detection is forwarded. The symbols used in the following are displayed in Table I.

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.

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 $I1$, making the corresponding position in the traffic mapping be set 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 pause 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 subsequent forwarding groups for the metadata used for deadlock detection. Notice that the direction of pause frames is from downstream to upstream, opposite to normal traffic flow. 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. To detect the causality-loop with minimal memory overhead, ITSY uses the switch suspecting a causality-loop could be formed, which we called a 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.

Given a generic initiator for the causality-loop detection, the following steps are followed:

- When the generic initiator sends out a checking message, it attaches its own $\{S_{gen-ini}, P_{gen-ini}, Seq_{id}\}$ to the packet header. The selection of the generic initiator

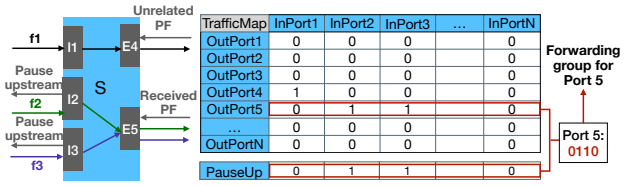


Fig. 1: Port-based causality data structure

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).

- 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 new checking message from the generic initiator. The stored generic initiator info at a port is removed after receiving a resume frame.
- 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.
- 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.
- 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: If deadlock exists and $S_{gen-ini}$ is on the CBD, the causality-loop must be detected by $S_{gen-ini}$, when the received $S_{gen-ini} = S_{cur}$, $P_{gen-ini} \in \xi_p$ where p is the port receiving the checking message, and Seq_{id} indicates the same episode.

Proof 1: For any given deadlock, a causality-loop must be implied. The checking message spreads with the causality-chain must also follow the causality-loop and return back to $S_{gen-ini}$. At this moment, $S_{cur} = dS_{gen-ini}$. The next step is to make sure that the checking message received at port p has causalities with ports in ξ_p connected to the upstream switches. If any port in ξ_p is equal to the received $P_{gen-ini}$, S_{cur} must have already sent out the checking message with the same $P_{gen-ini}$. It indicates there must be a repetitive ID and the causality-chain comes back to the beginning position of the causality-loop. If the recently sent $P_{gen-ini}$ has the same Seq_{id} as the received $P_{gen-ini}$, the causality-chain must have traversed one port of one switch twice in the same episode, which verifies the causality-loop.

Temporal consistency primitive: The temporal consistency primitive is triggered after the causality-loop of pause behaviors is detected by the above mechanism. Notice that the paused ports on the causality-chain might be resumed during the process of causality-loop detection. Therefore, even if a causality-loop is detected, deadlock may not actually exist. Without this primitive, a deadlock could be declared mistakenly.

We leverage a strategy that checks if every node of the causality-loop is still paused ever since the initial pause event has been triggered. Each episode is determined by the Seq_{id} of the generic initiator. Once a pause event occurs, it will hold until a resume frame is received. Each port on the switch maintains a RESUME tag r_p representing whether the pause is resumed during the current detection process. It is set to zero in each episode when the corresponding port is paused, and updated to 1 when receiving a resume frame. A synthesized temporary consistency check packet carrying $\{S_{gen-ini}, P_{gen-ini}, Seq_{id}\}$ is sent passing through the causality-loop, which is obtained by querying the traffic mapping data structure and comparing the corresponding generic initiator ID. The synthesized packet is forwarded to ports that belong to the causal ports ξ_p and have sent PFC packets 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: A deadlock is determined if and only if for each switch and each port p along the causality-loop, $r_p = 0$ and Seq_{id} is the same as the one used to determine the causality-loop.

Proof 2: 1) If $r_p = 0$ holds in the whole episode represented by the same Seq_{id}: no resume frame is transmitted during the current detecting episode. All ports of the causality-loop is paused through the entire duration and a deadlock is determined. 2) If $r_p = 1$ at some switch: port p is resumed and the status is not changed in the current episode. At least one of the nodes on the causality-loop is resumed so that pause events are unable to form a deadlock. 3) If Seq_{id} is changed at some switch, there must be a new pause frame following a resume frame. However, this must start a new independent episode of detection, which means $r_p = 1$ followed by $r_p = 0$ cannot happen in the same episode. Therefore, even if $r_p = 0$ holds for all switches on the loop, they are involved in different episodes so that the deadlock is not determined in the current episode. It is the job of the new detection episode to determine whether a deadlock is formed.

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.

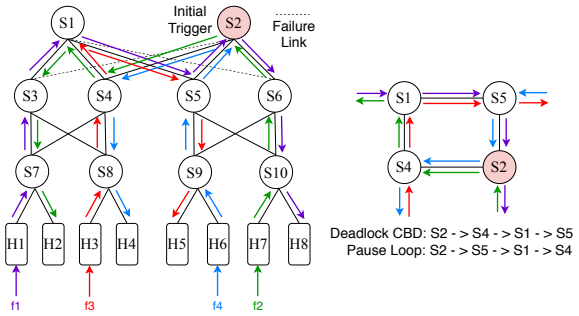


Fig. 2: Example of deadlock in the Clos network topology when initial trigger is on the loop

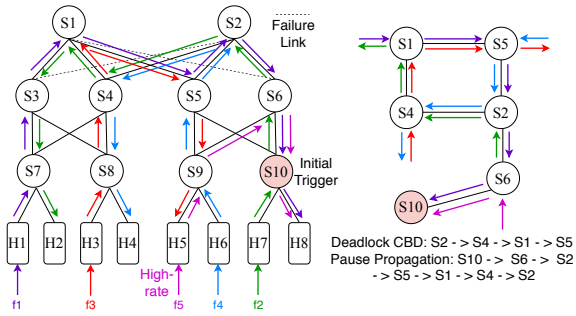


Fig. 3: Example of deadlock when initial trigger is out of loop

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. 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 switches out of the loop. As shown in Figure 3, malicious flow $f5$ with constant high sending rate causes a pause frame initially to be sent from $S10$, leading to the final deadlock loop between $S2$, $S5$, $S1$ and $S4$. Even if the deadlock can be broken from one of the switches on the loop, without solving continuous pause events from the initial trigger switch $S10$, $S6$ and $S2$ can be paused again and finally lead to a repeated formation of the same deadlock.

We observe that a signal of this case is that a middle switch receives multiple pause frames with the same $\{S_{tri}, P_{tri}\}$ from different ports. The Seq_{id} does not need to be the same as the initial trigger may alternately send pause and resume frames, which update the Seq_{id} received from outside the loop. However, the detection of this case only cares about the same Seq_{id} on the causality-loop. Therefore, even if the two pause frames with the same $\{S_{tri}, P_{tri}\}$ received from different ports have different Seq_{id}, the detection process can still be triggered. In the example of Figure 3, this phenomenon is detected at $S2$ when it receives two pause frames with the same $\{S_{tri}, P_{tri}\}$ from $S6$ and $S4$.

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 3: 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. This switch has received pause frames from at least two directions. One is from the outside downstream switch, the other one is from the switch on the causality-loop. As both pause frames can be traced back to the S_{tri}, the received $\{S_{tri}, P_{tri}\}$ from different ports must be the same. Notice that since pause and resume frames are possibly alternating continuously, the same $\{S_{tri}, P_{tri}\}$ received from the same port are not considered.

This condition could also be met even if there is no deadlock, as shown in Figure 4. $S7$ receives two pause frames with the same ID of $S1$ respectively from $S5$ and $S6$. However, no deadlock is formed. ITSY can deal with the non-deadlock cases normally as well as the deadlock cases. The correctness of deadlock detection is guaranteed by the general primitives in Section III-B.

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 deadlock detection. This process works in parallel with the previous process using the initial trigger as the generic initiator.

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 Seq_{id} from different ports. The one received from the port that has not stored the Seq_{id} is selected for the subsequent deadlock detection. The middle switch generates a synthesized 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.

Compared with the scenario that the initial trigger is on the loop, this case needs at most one additional round to detect the deadlock, which adds a small overhead.

IV. Handling Deadlock Based on Initial Triggers

Upon detection of a deadlock, actions must be taken to break the deadlock and prevent future recurrence of the dead-

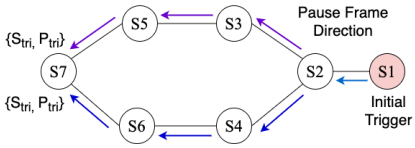


Fig. 4: A switch receives two pause frames with the same S_{tri} from different ports, but no deadlock exists

lock. Below we discuss several research directions. Details are left to future work.

Breaking the deadlock: Packet dropping or temporary rerouting are common methods used to break a deadlock [30], [32], [41]. However, rerouting is not always viable if not all flows have multiple paths. In addition, rerouting may create other deadlocks. Breaking a deadlock by dropping packets is more practical and several fast synchronous draining approaches have been proposed [39]. The packet loss rate that can be tolerated by protocols like RoCEv2 is about 0.0001 [51]. To break a deadlock, only a part of the buffered traffic needs to be dropped, thus the actual performance impact may be tolerable.

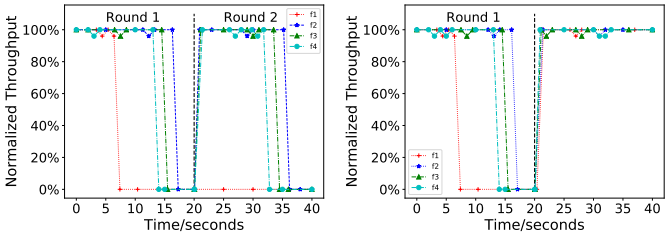
Initial trigger handling: In addition to breaking the deadlock, further operations are needed to handle the initial trigger in order to prevent the deadlock from forming again. If the initial trigger is a switch, a crucial step is to identify the heavy hitters which send a large amount of traffic and thus cause the congestion. Recent proposals for heavy hitter detection can be leveraged [12], [42]. Once identified, further steps can be taken to limit the heavy hitters.

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 [19]. 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 [19], [51]. One approach for handling such an initial trigger server is to prevent the NIC from generating pause frames and disable lossless mode at the switch port. Alternatively, the connected switch can make use of dynamic buffer sharing, which improves the utilization of available buffer space and reduces PFC pause frame propagation.

V. Preliminary Results

We prototype ITSY in the P4 language with BMv2 software switches [36] in the MiniNet environment for validation. The experiments are performed in the CloudLab platform [15], each node has 4 cores and 32GB of RAM. We evaluate ITSY for scenarios that the initial trigger is on the loop and out of the loop, using a $k = 4$ fat-tree topology.

Memory overhead: The memory overhead of ITSY mainly comes from the port-based causality data structure and the maintained information of checking message. Both of them are determined by the number of ports in the network. In our experiments, each switch has 4 ports and the memory overhead in this case is less than 10^{-1}KB. If we deploy ITSY in a large-scale network with $10,000$ 64-port switches, 14-bit SwitchID and 6-bit PortID are required to support the uniqueness requirement. The corresponding memory overhead is less than 1KB per switch. Overall, the total memory requirement of ITSY is quite small and it can be easily deployed in today's programmable data planes that usually have tens of MB of memory [6].



(a) W/O resolving initial trigger (b) Resolving initial trigger

Fig. 5: Benefits of resolving initial trigger out of the loop

Detection effectiveness: We evaluate the effectiveness of ITSY by creating deadlocks in two different scenarios—the initial trigger is on the loop and out of the loop respectively. We introduced two failed links so rerouting could finally lead to a deadlock loop. Traffic flows are generated across ToR switches and the resulting congestion could spread to the upstream switches and then to switches on the deadlock loop. We repeat each scenario 10 times. The average detection time for the *on loop* scenario is 0.8ms and 1.4ms for the *out of loop* scenario. 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, this situation is detected by the temporal consistency check without declaring any deadlock. There was no false positive or false negative.

Benefits of resolving the initial trigger: To validate 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 of different flows and results are shown in Figure 5. Although the baseline can recover from the deadlock for a while, if the traffic pattern does not change, the deadlock will reappear. In contrast, ITSY breaks the deadlock and resolves the initial trigger simultaneously, which prevents the recurrence of the deadlock.

VI. 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. ITSY can be implemented entirely in the data plane, which achieves low overhead and reacts quickly to deadlocks. For ongoing work, we plan to develop a fully automatic deadlock resolution system and apply it to more topologies as well as more complex deadlock scenarios.

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

[3] K. Anjan and T. M. Pinkston. An efficient, fully adaptive deadlock recovery scheme: Disha. In *Proceedings of the 22nd annual international symposium on Computer architecture*, pages 201–210, 1995.

[4] 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 Conference*, pages 29–43, 2016.

[5] M. Bansal, J. Mehlman, S. Katti, and P. Levis. Openradio: a programmable wireless dataplane. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 109–114, 2012.

[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 Review*, 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] W. J. Dally and C. L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. 1988.

[12] A. Dixit, P. Prakash, Y. C. Hu, and R. R. Kompella. On the impact of packet spraying in data center networks. In *2013 Proceedings IEEE INFOCOM*, pages 2130–2138. IEEE, 2013.

[13] 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.

[14] 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.

[15] 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.

[16] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 469–483, 2015.

[17] K.-T. Förster, R. Mahajan, and R. Wattenhofer. Consistent updates in software defined networks: On dependencies, loop freedom, and blackholes. In *2016 IFIP Networking Conference (IFIP Networking) and Workshops*, pages 1–9. IEEE, 2016.

[18] J. Geng, J. Yan, and Y. Zhang. P4qcn: Congestion control using p4-capable device in data center networks. *Electronics*, 8(3):280, 2019.

[19] 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.

[20] 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 conference*, pages 38–49, 2011.

[21] D. Hancock and J. Van der Merwe. Hyper4: Using p4 to virtualize the programmable data plane. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pages 35–49, 2016.

[22] 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.

[23] 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.

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

[25] 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 Review*, 44(4):539–550, 2014.

[26] 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.

[27] D. Lee, S. J. Golestani, and M. J. Karol. Prevention of deadlocks and livelocks in lossless, backpressured packet networks, Feb. 22 2005. US Patent 6,859,435.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] J. M. Martínez, P. Lopez, J. Duato, and T. M. Pinkston. Software-based deadlock recovery technique for true fully adaptive routing in wormhole networks. In *Proceedings of the 1997 International Conference on Parallel Processing*, pages 182–189. IEEE, 1997.

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

[34] Microsoft. The microsoft cognitive toolkit. https://docs.microsoft.com/en-us/cognitive-toolkit/.

[35] 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 Review*, 45(4):537–550, 2015.

[36] P4lang. P4 behavioral model. https://github.com/p4lang/behavioral-model.

[37] H. Park and D. P. Agrawal. Generic methodologies for deadlock-free routing. In *Proceedings of International Conference on Parallel Processing*, pages 638–643. IEEE, 1996.

[38] 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.

[39] A. Ramrakhyani, P. V. Gratz, and T. Krishna. Synchronized progress in interconnection networks (spin): A new theory for deadlock freedom. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 699–711. IEEE, 2018.

[40] 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.

[41] 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.

[42] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*, pages 164–176, 2017.

[43] T. Skeie, O. Lysne, and I. Theiss. Layered shortest path (lash) routing in irregular system area networks. In *Proceedings 16th International Parallel and Distributed Processing Symposium. IPDPS 2002*, pages 8–pp. Citeseer, 2002.

[44] B. Stephens and A. L. Cox. Deadlock-free local fast failover for arbitrary data center networks. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9. IEEE, 2016.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] J. Wu. A fault-tolerant and deadlock-free routing protocol in 2d meshes based on odd-even turn model. *IEEE Transactions on Computers*, 52(9):1154–1169, 2003.

[50] 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 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 419–430, 2012.

[51] 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 Review*, 45(4):523–536, 2015.

[52] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, et al. Packet-level telemetry in large datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 479–491, 2015.