Tagger: Practical PFC Deadlock Prevention in Data Center Networks

Shuihai Hu[1,2], Yibo Zhu[1], Peng Cheng[1], Chuanxiong Guo[1]
Kun Tan[1*], Jitendra Padhye[1], Kai Chen[2]
[1] Microsoft [2]Hong Kong University of Science and Technology

ABSTRACT

Remote Direct Memory Access over Converged Ethernet (RoCE) deployments are vulnerable to deadlocks induced by Priority Flow Control (PFC). Prior solutions for deadlock prevention either require significant changes to routing protocols, or require excessive buffers in the switches. In this paper, we propose Tagger, a scheme for deadlock prevention. It does not require any changes to the routing protocol, and needs only modest buffers. Tagger is based on the insight that given a set of expected lossless routes, a simple tagging scheme can be developed to ensure that no deadlock will occur under any failure conditions. Packets that do not travel on these lossless routes may be dropped under extreme conditions. We design such a scheme, prove that it prevents deadlock and implement it efficiently on commodity hardware.

CCS CONCEPTS

• **Networks → Data center networks**; *Data path algorithms*;

KEYWORDS

Data Center Networks, RDMA, Deadlock Prevention, Tag

ACM Reference Format:
Shuihai Hu, Yibo Zhu, Peng Cheng, Chuanxiong Guo, Kun Tan, Jitendra Padhye, and Kai Chen. 2017. Tagger: Practical PFC Deadlock Prevention in Data Center Networks. In *Proceedings of CoNEXT '17*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3143361.3143382

1 INTRODUCTION

Public cloud providers like Microsoft and Google are deploying Remote Direct Memory Access (RDMA) over Converged Ethernet (RoCE) in their data centers to enable low latency, high throughput data transfers with minimal CPU overhead [38, 54]. Systems like Pilaf [37], Farm [15], TensorFlow [3], and CNTK [2] rely on RDMA/RoCE for enhanced performance.

RoCE uses Priority Flow Control (PFC) to prevent packet drops due to buffer overflow at the switches. PFC allows a switch to temporarily pause its upstream neighbor. While PFC is effective, it can lead to deadlocks [27, 29, 49]. Deadlocks are caused by circular buffer dependency (CBD) [29], *i.e.,* the occupied buffers are waiting for each other in a loop.

While CBD can be caused by a routing loop, routing loop is not required – flows that travel on loop-free paths can create buffer dependencies that lead to CBD. A simple but contrived example is shown in Figure 1. We will discuss more realistic scenarios (e.g. Figure 3) later. See [29] for several other examples.

The deadlock problem is not merely theoretical – our conversations with engineers at large cloud providers confirm that they have seen the problem in practice and at least one provider has reported it publicly [27]. Deadlock is a serious problem because a deadlock is not transient – once a deadlock forms, it does not go away even after the conditions (e.g. a temporary routing loop due to link failure) that caused its formation have abated [27]. Worse, a small initial deadlock may cause the PFC frames to propagate and create a global deadlock, and shutdown the whole network.

Current solutions to the deadlock problem fall in two categories. The first category consists of solutions that *detect* the formation of the deadlock and then use various techniques to *break* it [45]. These solutions do not address the root cause of the problem, and hence cannot guarantee that the deadlock would not immediately reappear.

The second category of solutions are designed to *prevent* deadlocks, by avoiding CBDs in the first place.

In §3, based on the data and experience from a large cloud provider's data centers, we show that any practical deadlock prevention scheme must meet three key challenges. These include: (i) it should require no changes to existing routing protocols or switch hardware, (ii) it must deal with link failures and associated route changes, and (iii) it must work with limited buffer available in commodity switches.

Prior proposals for deadlock prevention fail to meet one or more of these challenges. Most of them [9, 12–14, 16, 17, 19, 24, 31, 40, 44, 47, 48, 48, 49, 52] are focused on designing routing protocols that are significantly different from what are supported by commodity Ethernet switches. Many of these schemes also require carefully controlling the paths – something that is simply not possible with decentralized routing in presence of link failures [53]. Finally, some schemes [7, 22, 32, 42], require creation of numerous priorities and buffer management according to those priorities. However, modern data center networks, built using commodity switches, can realistically support only two or three lossless priorities [27].

In this paper, we present Tagger, which meets all three challenges described above. Tagger is based on a simple observation: in a data center, we can ask the operator to supply a list of paths that must be lossless. Packets that do not travel on lossless paths may be dropped under extreme circumstances. We call these expected lossless paths (ELPs). Enumerating ELPs is straightforward for

[*] Now at Huawei.

CoNEXT '17, December 12–15, 2017, Incheon, Republic of Korea
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5422-6/17/12…$15.00
https://doi.org/10.1145/3143361.3143382

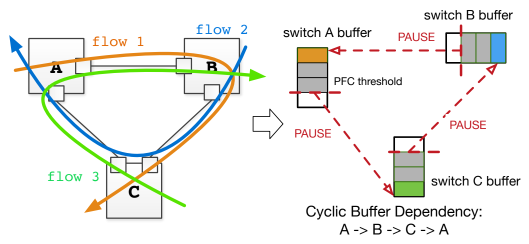


Figure 1: A simple (but contrived) example to illustrate CBD formation without routing loop.

"structured" topologies like Clos [11], FatTree [4] or Bcube [26], and not onerous even for randomized topologies like Jellyfish [46].

Using ELPs, we create a system of match-action rules to "tag" packets. The switches use these tags to enqueue packets in different lossless queues. The tags carried in packets are manipulated in a way such that CBD never forms due to packets traveling on paths in ELP. If packets ever deviate from paths in ELP (e.g. due to link failures or routing errors) they are automatically placed in a lossy queue to ensure that they do not trigger PFC. Operators have full flexibility to add more redundant paths into ELP, bringing the possibility of falling in the lossy queue to nearly 0. Once ELP is given, Tagger guarantees that there will be no deadlock - even under unforeseen link failures or routing errors, like loops!

Tagger works for any routing protocol because there are no restrictions on what paths can be included in the ELP, tagging rules are static, and are specified only in terms of local information (tag, ingress port and egress port) available at each switch.

The number of lossless queues and the number of tag match-action rules required by Tagger are small. Even for a Jellyfish topology with 2000 switches, Tagger requires just three lossless queues per switch. In fact, we prove that for Clos topology, Tagger is optimal in terms of number of lossless queues required. We also show that using two lossless queues practically guarantees lossless, and we further show how to minimize the number of match-action rules required to implement Tagger.

We have implemented and tested Tagger on commodity Arista switches with Broadcom chips. The implementation requires carefully addressing the problem of priority transition (§7). Our simulations and experiments show Tagger imposes negligible performance penalty on RDMA traffic.

2 BACKGROUND

RDMA and RoCE: RDMA technology offers high throughput, low latency and low CPU overhead, by bypassing host networking stack. It allows Network Interface Cards (NICS) to transfer data between pre-registered memory buffers at end hosts. In modern data centers, RDMA is deployed using RDMA over Converged Ethernet V2 (RoCE) standard [30]

PFC: RoCE needs a lossless fabric for optimal performance. This is accomplished in Ethernet networks using the Priority Flow Control (PFC) mechanism [1]. Using PFC, a switch can pause an incoming link when its ingress buffer occupancy reaches a preset threshold. As long as sufficient "headroom" is reserved to buffer packets that are in flight during the time takes for the PAUSE to take effect, no packet will be dropped due to buffer overflow [10, 54].

The PFC standard defines 8 classes, called priorities[1]. Packets in each priority are buffered separately. PAUSE messages carry this priority. When a packet arrives at port i of switch S with priority j, it is enqueued in the ingress queue j of port i. If the ingress queue length exceeds the PFC threshold, a pause message is sent to the upstream switch connected to port i. The message carries priority j. The upstream switch then stops sending packets with priority j to switch S on port i until a resume message with priority j is received.

Deadlock: PFC can lead to deadlocks when paused queues form a cycle. Deadlock cannot happen if there is no Circular Buffer Dependency (CBD). Thus deadlock avoidance schemes, including this work, focus on avoiding CBD. We now describe the three key challenges that any practical deadlock avoidance scheme must meet.

3 CHALLENGES

3.1 Work with existing routing protocols and hardware

Data center routing protocols have to satisfy a variety of complex requirements regarding fault tolerance, and security [6]. Operators also invest heavily in tools and technologies to monitor and maintain their networks; and these tools are tailored for the routing protocols that are already deployed. Thus, operators are unwilling to deploy brand-new routing protocols like [12–14, 16, 17, 19, 24, 40, 44, 47, 48, 52] or hardware just for deadlock avoidance – especially when RoCEv2 (encapsulated in standard UDP packets) itself can be deployed without any changes to routing.

However, the widely-used routing protocols, like BGP and OSPF, never attempt to avoid CBD since they are designed for lossy networks. Moreover, modifying these protocols to avoid CBD is not trivial. They are inherently asynchronous distributed systems – there is no guarantee that all routers will react to network dynamics (§3.2) at the exact same time. This unavoidably creates transient routing loops or CBDs, enough for lossless traffic to create deadlocks. In such cases, we cannot ensure both losslessness and deadlock-freedom.

In this paper, instead of making drastic changes to routing protocols, we explore a different design tradeoff. Our system, Tagger, works with any unmodified routing protocols and guarantees deadlock-freedom by giving up losslessness only in certain rare cases. We favor this approach because it is more practical to deploy, and has negligible performance penalty.

3.2 Data center networks are dynamic

Given that we aim to work with existing routing infrastructures, we must address the issue that most routing schemes are dynamic – paths change in response to link failures or other events.

Figure 2 shows a simplified (and small) version of network deployed in our data center, with commonly used up-down routing

[1]The word priority is a misnomer. There is no implicit ordering among priorities – they are really just separate classes.

Date	Total No.	Rerouted No.	Reroute probability
11/01/2016	11381533570	148416	1.3e-5
11/02/2016	11056408780	130815	1.2e-5
11/03/2016	10316034165	104472	1.0e-5
11/04/2016	10273000622	92555	0.9e-5
11/05/2016	10230003382	102872	1.0e-5
11/06/2016	10491233987	106266	1.0e-5
11/07/2016	9608289622	100916	1.1e-5

Table 1: Packet reroute measurements in more than 20 data centers.

(also called valley-free [41]) scheme. In up-down routing, a packet first goes UP from the source server to one of the common ancestor switches of the source and destination servers, then it goes DOWN from the common ancestor to the destination server. In UP-DOWN routing, the following property holds: when the packet is on its way UP, it should not go DOWN; when it is on its way DOWN, it should not go UP. Thus, in *normal cases*, there can be no CBD and hence no deadlock.

However, packets can deviate from the UP-DOWN paths due to many reasons, including link failures, port flaps etc., which are quite common in data center networks [33, 53]. When the up-down property is violated, packets "bouncing" between layers can cause deadlocks [45]. See Figure 3.

In our data centers, we see hundreds of violations of up-down routing per day. Such routes can persist for minutes or even longer. In the next, we present our measurement results in more than 20 data centers.

Measurements of violations of up-down routing. Our measurement works as follows. We instrument the servers to send out IP-in-IP packets to the high-layer switches. The outer source and destination IP addresses are set to the sending server and one of the high layer switches, and the inner source and destination IP addresses are set to the switch and the sending server, respectively. The high-layer switches are configured to decapsulate those IP-in-IP packets that are targeting themselves in hardware.

After decapsulation, the outer IP header is discarded, and the packet is then routed using its inner header. We set a TTL value, 64 in this paper, in the inner IP header. As the packet is forwarded back to the server, the TTL is decremented per hop. For a three-layer Clos network, there are three hops from the highest layer switches to the server. Hence normally the TTL value of the received packets should be 61. If, however, the TTL value of a received packet is smaller than 61, say 59, we know the received packet was not taking the shortest path, and the packet must have taken a reroute path.

For every measurement, a server sends out $n = 100$ IP-in-IP probing packets, if the received TTL values are not equal, we know packet reroute happened for this measurement. We then calculate the reroute probability of the measurements as $\frac{M}{N}$, where M is the number of measurements that experienced packet reroute, and N is the total number of measurements. The measurement results are shown in Table 1.

The most important conclusion we can draw from Table 1 is that packet reroute does happen in data center networks. The reroute probability is around 10^{-5}. Though 10^{-5} is not a big number, given the large traffic volume and the large scale data center networks,

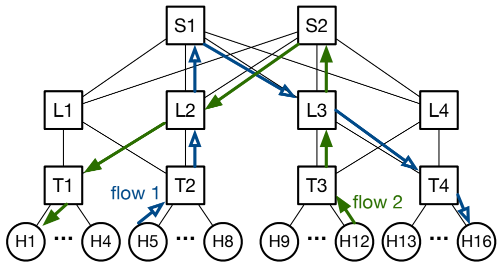


Figure 2: Clos topology with two up-down flows.

the deadlocks due to packet reroute as discussed in [27, 29, 45] do not just exist in paper designs. They are real!

3.3 Limited number of lossless queues

One idea to solve deadlock is to assign dynamic priority to packets. The priority of a packet increases as the packet approaches its destination [32]. Such a design requires as many priorities as the largest path length in the network. However, there are two practical problems. First, given the network dynamics, the largest path length may not be all that small (§3.2). Second, the PFC standard supports only 8 priorities. Worse yet, commodity switches can realistically support only two or three lossless priorities [27]. The problem is that to guarantee losslessness, a switch needs to reserve certain amount of *headroom* for absorbing the packets in flight during the time it takes for the PAUSE message to take effect.

The switch buffers are made of extremely fast and hence extremely expensive memory, so their size is not expected to increase rapidly even as link speeds and port counts go up. Some of this buffer must also be set aside to serve lossy traffic (i.e. normal TCP traffic), which still constitutes a majority of traffic in data centers. At the same time, the PFC response time is subject to physical limits and cannot be arbitrarily reduced. Thus, even newest switching ASICs are not expected to support more than four lossless queues. Hence the solutions that require a large number of lossless queues are not practical.

4 TAGGER FOR CLOS TOPOLOGY

While Tagger works for any topology, we first describe the core ideas using the popular Clos network topology.

4.1 Definitions

Expected Lossless Paths (ELP): This is a set of paths that the network operator requires to be lossless. For example, in a Clos network, the operator may want that all up-down paths or all shortest paths to be lossless. Any loop-free route can be included in ELP.

Lossless class: A lossless class is a service that the network provides for applications. The network guarantees that packets in a lossless class will not be dropped due to buffer overflow as long as they traverse paths in ELP.

Tag: Tag is a small integer assigned / associated with a packet. The tag is embedded in a packet. A packet belonging to one lossless class may change its tag value, which means that a lossless class may have multiple tag values.

Switch model: For ease of discussion, we use a simplified switch model in this section. A switch has multiple ports. Each port has up to n lossless queues (typically $n \leq 8$), and at least one lossy queue[2]. The queue number corresponds to the PFC priority. Switch classifies arriving packets into ingress queues based on *tags*. Each tag maps to a single priority value. If a tag does not match any priority value, the packet is added to a lossy queue. Before forwarding a packet, the switch can rewrite the tag according to user-specified, match-action rules based on the InPort (ingress port number), OutPort (egress port number) and the original tag value.

We now describe the core idea behind Tagger.

4.2 Tagging on bounce

We start with a simple scenario. Consider Figure 2. Let's assume that the operator defines ELP simply - just shortest up-down paths as they exist in this network. Both the green and blue flows follow such paths and there is no deadlock.

Now, let's assume that as shown in Figure 3, two links fail, which forces both flows to travel on paths that are not in the ELP. We call these paths 1-bounce paths, as the path violates the up-down rule once ($L2$ to $S1$ for green flow, $L3$ to $S2$ for blue flow). As shown, this can lead to CBD, and hence may cause deadlock.

One way to avoid such CBD is to put any packets that have deviated from the ELP in a lossy queue. Being assigned to lossy queue means that such packets will not trigger PFC. Since the paths in ELP are shortest up-down paths they are deadlock free, and the bounced packets won't trigger PFC, the network will stay deadlock free even if packets bounce.

Being assigned to lossy queue does not mean that the packets are immediately (or ever) dropped – they are dropped *only if* they arrive at a queue that is full. We only need to ensure that these wayward packets do not trigger PFC.

Thus, if each switch can detect that an arriving packet has traveled (sometime in past) on a "bouncy" path, it can put that packet in lossy queue, and there will be no deadlock.

How can a switch determine whether the packet has bounced, using just local information, in presence of dynamic routing?

Consider the green flow in Figure 3. We want switch $S1$, the first switch after bounce, to detect the bounce and put the packet in a lossy queue.

One way for $S1$ (and any switches afterwards) to recognize a bounced packet is by TTL. Since the ELP consists of shortest paths, a bounced packet will have "lower than expected" TTL. However, TTL values are set by end hosts (more in §7), so a more controllable way is for $L2$ to provide this information via a special tag (e.g. DSCP) in the packet.

Note that for any shortest, up-down path, $L2$ would have never forward any packet that arrived from $S2$ to $S1$. So, if $L2$ "tags" packets that have arrived from $S2$ that it is forwarding to $S1$, then $S1$, *and all other switches along the path after* $S1$ know that the packet has travelled on a non-ELP path.

Note that $L2$ needs only local information to do the tagging. We can initially assign a *NoBounce* tag to every packet. $L2$ then simply

[2]All queues share a single memory pool.

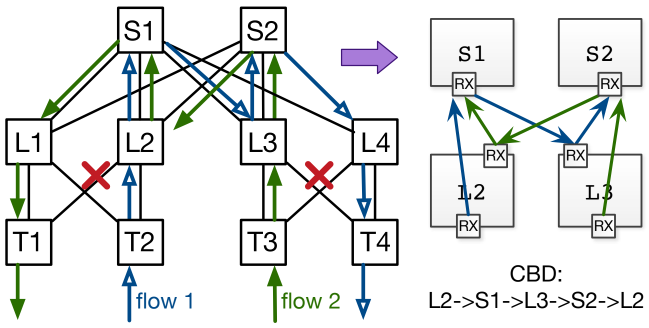


Figure 3: 1-bounce path creates CBD. The green flow (T3 to T1) bounces at L2 due to failure of L2-T1. The blue flow (T2 to T4) bounces at L3 due to failure of L3-T4. Note that the paths are loop-free, and yet there is a CBD.

needs to check ingress and egress port for each packet: it changes the tag from *NoBounce* to *Bounced* if a packet arriving from ingress port connected to $S2$ exits on egress port connected to $S1$. It is easy to see that these rules can be pre-computed since we know the topology, and the set of paths that we want to be "lossless".

While this scenario is quite simple, we chose it because it clearly illustrates the core idea behind Tagger – given a topology and an ELP we can create a system of tags and *static* rules that manipulate these tags to ensure that there will not be CBD, even when the underlying routing system packets on paths that are not in the ELP.

Of course, this basic idea is not enough. First of all, packets may bounce not just from the Leaf layer, but at any layer. Second, recall from §3.2 that "bounces" are a fact of life in data center networks. The operator may not want to put packets that have suffered just a single bounce into a lossy queue – we may want to wait until the packet has bounced more than once before assigning it to a lossy queue. This means that ELP will consist of more than shortest up-down paths, and the paths in ELP may be prone to CBD! Third, we must ensure that we don't end up using more lossless queues than the switch can support. To this end, we show how to combine tags.

4.3 Reducing the number of lossless queues

Consider again the network in Figure 2. Let's say the operator wants to make sure that in face of link failures, packets are not immediately put into a lossy queue. The operator is willing to tolerate up to k bounces. So, the ELP consists of all shortest paths, and all paths with up to k bounces. Do we need to assign a distinct tag and a corresponding priority queue for each bouncing point?

To answer this question, we leverage our knowledge of Clos topology. Consider a packet that bounces twice. The path between the first bounce and the second bounce is a normal up-down path. Therefore, these path segments do not have CBD even if we combine them into a single priority queue. We can use a single tag to represent these segments altogether, and map the tag to a globally unique priority queue.

This completes the design of Tagger for Clos topology. Packets start with tag of 1. We configure all ToR and Leaf switches such that every time they see a packet coming down and then going up

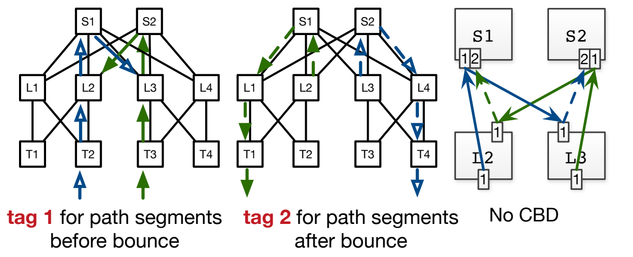

S1
S2
L1
L2
L3
L4
T1
T2
T3
T4
tag 1 for path segments before bounce
tag 2 for path segments after bounce
No CBD

Figure 4: Illustration of Tagger for ELP of all shortest up-down paths, and all 1-bounce paths. For clarity, only lossless queues are shown.

(therefore, bouncing) for any reasons, they increase the tag by one. Spine switches do not need to change tags since packets never go up from there.

All switches put packets with tag i to i^{th} lossless queues. Since ELP includes paths with up to k bounces, the switches need to have $k+1$ lossless queues. If a packet bounces more than k times (e.g. due to numerous link failures, or loop), it will carry a tag larger than $k+1$. All switches will put such packets into a lossy queue.

Figure 4 illustrates the tagging algorithm in action, for ELP consisting of all shortest up-down paths, plus all 1-bounce paths. Packets, when traveling on path segments before bounce carry a tag value of 1, and the tag is set to 2 after the bounce. This ensures that the packets are queued in separate lossless queues, and thus there is no CBD. In other words, we show a system for $k = 2$. The lossy queue for packets that bounce more than once is omitted for clarity.

This design satisfies all three challenges described in §3. We do not change the routing protocol. We work with existing hardware. We deal with dynamic changes, and we do not exceed the number of available lossless queues.

4.4 Deadlock freedom and optimality

We now provide brief proof sketches to prove that the above algorithm is deadlock free, and optimal in terms of number of lossless priorities used.

Tagger is deadlock-free for Clos networks: Tagger has two important properties. First, for any given tag and its corresponding priority queue, there is no CBD because each tag only has a set of "up-down" routing paths. Second, every time the tag changes, it changes monotonically. This means that the packet is going unidirectionally in a DAG of priority queues. This is important because otherwise CBD may still happen across different priorities. We conclude that no CBD can form either within a tag or across different tags. The network is deadlock-free since CBD is a necessary condition for deadlock. A formal proof, which applies to any topology, is given in §5.

Tagger is optimal in terms of lossless priorities used: We show that to make all k bounces paths lossless and deadlock-free, at least $k+1$ lossless priorities are required. We use contradiction. Assume there exists a system that can make k bounces paths lossless and deadlock-free with only k lossless priorities. Consider a flow that loops between two directly connected switches $T1$ and $L1$ for $k+1$ times, which means it bounces k times at $T1$. With Pigeonhole principle, we know that at least two times during the looping, the packet must have the same lossless priority. This means there exists a CBD, and deadlock can happen when having sufficient traffic demand [29]. Contradiction.

Much of the discussion in this section used the specific properties of Clos networks, and the specific ELP set. We now show how to generalize Tagger for any topology and any ELP set.

Symbol	Description
A_i	Switch A's i^{th} ingress port
(A_i, x)	A node in tagged graph
$(A_i, x) \to (B_j, y)$	A tagged edge
V	All tagged nodes
E	All tagged edges
$G(V, E)$	Tagged graph
T	Largest tag in $G(V, E)$
G_k	Partition of $G(V, E)$ for priority k

Table 2: Notations in the formalized description.

5 GENERALIZING TAGGER

We begin by formalizing the description of the tagging system using notations in Table 2.

Let A_i represent a unique ingress port in the network, *i.e.,* switch A's i^{th} ingress port. We use a *tagged graph* $G(V,E)$ to uniquely represent a tagging scheme. Given a tagging scheme, the *tagged graph* $G(V,E)$ is defined as:

(1) G contains a node, (A_i, x), *iff.* port A_i may receive packets with tag x, and these packets must be lossless. V is the set of all such nodes.

(2) G contains an edge $(A_i, x) \to (B_j, y)$ *iff.* switch A and B are connected, *and* switch A may change a packet's tag from x to y before sending to B (the case $x = y$ also counts). E is the set of all such edges.

Given a tag k, we also define $\{G_k\}$, with vertices $V(G_k)$ and edges $E(G_k)$:

$$V(G_k) = \{(A_i, k)|\forall A, i\}$$

$$E(G_k) = \{v_0 \to v_1 | \forall v_0, v_1 \in V(G_k), v_0 \to v_1 \in E(G)\}$$

Each tag k is mapped to a unique lossless priority.

Each node has a rule to match on a tag on an ingress port, and assign the packet to corresponding lossless queue. In addition, each edge corresponds to a switch action of setting the tag for the next hop.

If a packet arrives at A_i with tag x, and is destined for port B_j, and there is no corresponding edge in $G(V,E)$, it means that the packet has traversed on a path that is not in ELP. Such packets are assigned a special tag, and all switches assign this tag to lossy priority[3].

In the rest of the section, we will describe how to generate the tagging graph – i.e. the tagging rules. But first, we prove that the tagging scheme described by such a graph is deadlock free, as long as the graph meets two requirements.

(1) Any G_k for G *must not* have a cycle. This is because each edge in G_k is essentially a buffer dependency – whether A_i can dequeue packets depends on whether B_j has paused it. A cycle in G_k means cyclic buffer dependency.

[3]This rule is always the last one in the TCAM rule list, acting as a safeguard to avoid unexpected buffer dependency. See §7.

Algorithm 1 A brute-force tagging system that decreases the tag by one on every hop.

```
Input: Topology and $ELP$
Output: A tagged graph $G(V, E)$
$V \leftarrow Set()$;
$E \leftarrow Set()$;
for each path $r$ in $ELP$ do
    $tag \leftarrow 1$;
    for each hop $h$ in $r$ do
        $V \leftarrow V \cup \{(h, tag)\}$;
        $E \leftarrow E \cup \{lastHop \rightarrow (h, tag)\}$;
        $tag \leftarrow tag + 1$;
return $G(V, E)$;
```

(2) There must be no link going from G_x to G_y if $x > y$. This means we enforce the order of G_x and G_y.

These requirements are essentially generalization of the properties discussed in §4.4.

THEOREM 5.1. *Any tag system, defined by $G(V, E)$, that satisfies the above two requirements is deadlock-free.*

PROOF. We prove by contradiction. Suppose there exists a tag system, whose tagged graph $G(V, E)$ satisfies the above two requirements, but is not deadlock-free. This means $G(V, E)$ has a cycle $v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_0$. If traffic traverses all hops in the cycle, the cycle leads into a CBD and can form deadlock.

Case 1: All the nodes in the cycle have the same tag t. According to the first requirement, G_t does *not* have a cycle. Contradicted.

Case 2: The nodes in the cycle have at least two different tags, t_0 and t_1. Without loss of generality, we assume $t_0 < t_1$, and v_i has tag t_0, v_j has tag t_1. Because v_i and v_j belongs to a cycle, there must exist a path going from v_j to v_i. Since $t_0 < t_1$, along the path there must exist a hop where the tag decreases. However, according to the second requirement, such a hop cannot exist. Contradicted.

Case 1 and Case 2 cover all possible scenarios. Thus, we conclude that there does not exist a $G(V, E)$ that satisfies the two requirements but is not deadlock-free. □

5.1 Generating $G(V, E)$

In the next, we describe our algorithm to generate a deadlock-free $G(V, E)$ for any given topology, and the ELP set.

For general graph without structure information, a straightforward tagging system [32] is to monotonically increase the tag (thus, the priority) at every hop, as described in Algorithm 1.

It is easy to verify that the graph generated by this algorithm meets the two requirements specified earlier, and thus it guarantees deadlock freedom. Figure 5 shows a small example, including the topology, the ELP set, the generated graph, and the corresponding rule lists for each node.

Of course, with just this basic algorithm, we may end up with too many tags (i.e. lossless priorities) – in fact, as many as the longest path length in lossless routes. This is why we need three lossless priorities for the simple example in Figure 5(b). In a three-layer Clos network, the longest up-down path has 5 hops, so Algorithm 1 will use 5 priorities just to support up-down routing. We now show how to combine tags to reduce the number of lossless queues needed.

Algorithm 2 Greedily minimizing the number of tags by merging brute-force tags.

```
Input: The brute-force tagged graph $G(V, E)$ with largest tag $T$
Output: A new tagged graph $G'(V', E')$ that has small $|\{G'_k\}|$
Initialize $V'$, $E'$, $V_{tmp}$, $E_{tmp}$ as empty $Set()$;
$t' \leftarrow 1$;
for $t \leftarrow 1$ to $T$ do
    for each $(A_i, t)$ in $V$ whose tag is $t$ do
        $V_{tmp} \leftarrow V_{tmp} \cup \{(A_i, t')\}$;
        $E_{tmp} \leftarrow E_{tmp} \cup$ {edges of $(A_i, t)$, change $t$ to $t'$};
        if $G_{tmp}(V_{tmp}, E_{tmp})$ is acyclic then
            $V' \leftarrow V' \cup \{(A_i, t')\}$;
            $E' \leftarrow E' \cup$ {edges of $(A_i, t)$, change $t$ to $t'$};
        else
            $V' \leftarrow V' \cup \{(A_i, t' + 1)\}$;
            $E' \leftarrow E' \cup$ {edges of $(A_i, t)$, change $t$ to $t' + 1$};
            $V_{tmp} \leftarrow V_{tmp} \setminus \{(A_i, t')\}$;
            $E_{tmp} \leftarrow E_{tmp} \setminus$ {edges of $(A_i, t')$};
    if $V'$ contains nodes of tag $t' + 1$ then
        $V_{tmp} \leftarrow$ {nodes in $V'$ with tag $t' + 1$};
        $E_{tmp} \leftarrow$ {edges in $V'$, both ends have tag $t' + 1$};
        $t' \leftarrow t' + 1$;
return $G'(V', E')$;
```

5.2 Reducing the number of lossless queues

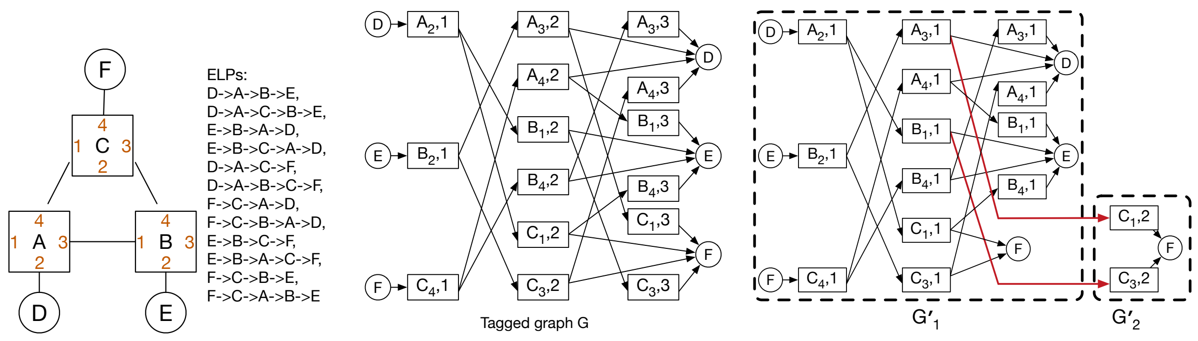
Algorithm 2 uses a greedy heuristic to combine the tags generated by Algorithm 1 to reduce the number of lossless queues required. It greedily combines as many nodes in $G(V, E)$ as possible into each path segment under CBD-free constraint. To ensure the monotonic property, we start from combining the nodes with smallest tag, 1 and proceed linearly to consider all tags up to T, which is the largest tag number used in $G(V, E)$.

The new tag t' also starts from 1. In every iteration, we check all nodes with the same tag value t. V_{tmp} and E_{tmp} is the "sandbox". For every node, we add it to V_{tmp} and E_{tmp} and check whether adding it to $G'_{t'}$ will lead to a cycle within $G'_{t'}$. If not, we re-tag the node to be t'. Otherwise, we re-tag the node to be $t' + 1$. Re-tagging the node to be $t' + 1$ does not cause a cycle in $G'_{t'+1}$, because all nodes in $G'_{t'+1}$ so far have the same old tag of t, which means there is no edge between them. At the end of each iteration, if there are nodes being re-tagged as $t' + 1$, we move on to add nodes into $G'_{t'+1}$ in the next iteration. This ensures that the monotonic property will still hold after combination.

In Figure 5(c) we see Algorithm 2 in action to minimize the $G(V, E)$ from Figure 5. We see that the number of tags is reduced to *two*.

5.3 Analysis

Algorithm runtime: Algorithm 2 is efficient. Recall that T is the largest value of tag in $G(V, E)$. Let S, L and P be the number of switches, the number of links and the number of ports a switch has in the original topology, respectively. Then, $G(V, E)$ can have at most $L \times T$ nodes. Each node will be examined exactly once for checking whether G_{tmp} is acyclic. Checking whether G_{tmp} is acyclic with a newly added node requires a Breadth-First Search,



(a) Topology and ELP set. (b) Output tagged graph by Algorithm 1. (c) Output tagged graph by Algorithm 2.

Figure 5: Walk-through example of the algorithms. Each rectange in (b) and (c) is a (port,tag) pair.

Tag	InPort	OutPort	Newtag
1	2	3	2
1	2	4	2
2	3	2	3
2	3	4	3
2	4	2	3
2	4	3	3
3	3	2	4
3	4	2	4
others	others	others	lossy tag

(a) rules installed in A

Tag	InPort	OutPort	Newtag
1	2	1	2
1	2	4	2
2	1	2	3
2	1	4	3
2	4	1	3
2	4	2	3
3	1	2	4
3	4	2	4
others	others	others	lossy tag

(b) rules installed in B

Tag	InPort	OutPort	Newtag
1	4	1	2
1	4	3	2
2	1	3	3
2	1	4	3
2	3	1	3
2	3	4	3
3	1	4	4
3	3	4	4
others	others	others	lossy tag

(c) rules installed in C

Table 3: Tag rewriting rules under Algorithm 1. Tag "4" will only appear on destination servers.

Tag	InPort	OutPort	Newtag
1	2	3	1
1	2	4	1
1	3	2	1
1	3	4	2
1	4	2	1
1	4	3	1
others	others	others	lossy tag

(a) Rules installed in A

Tag	InPort	OutPort	Newtag
1	2	1	1
1	2	4	1
1	1	2	1
1	1	4	2
1	4	1	1
1	4	2	1
others	others	others	lossy tag

(b) Rules installed in B

Tag	InPort	OutPort	Newtag
1	4	1	1
1	4	3	1
1	1	3	1
1	1	4	1
1	3	1	1
1	3	4	1
2	1	4	2
2	3	4	2
others	others	others	lossy tag

(c) Rules installed in C

Table 4: Tag rewriting rules generated by Algorithm 2 (without compression).

with runtime complexity of $O(|V_{tmp}| + |E_{tmp}|)$. $|V_{tmp}|$ is bounded by the number of links L, and $|E_{tmp}|$ is bounded by the number of pairs of incident links $L \times P$, in the network. Thus, the total runtime complexity is $O(L \times T \times (L + L \times P))$. Note that T itself is bounded by the length of the longest path in *ELP*.

Number of tags: Algorithm 2 is not optimal, but works well in practice. For example, it gives optimal results for BCube topology without requiring any BCube-specific changes – a k-level BCube with default routing only needs k tags to prevent deadlock. The results are promising even for unstructured topology like Jellyfish. Using Algorithm 2, a 2000-node Jellyfish topology with shortest-path routing requires only 3 tags to be deadlock-free (§8).

Number of rules: From the conceptual switch model, a switch needs InPort (ingress port number), OutPort (egress port number), and the current Tag to decide the next Tag. Hence it seems the number of rules needed per switch is $n(n-1) \times \frac{m(m-1)}{2}$, where n is the number of switch ports and $m = |G'_k|$ is the number of Tags. We will show in § 7 that the number of rules can be compressed to $n \times \frac{m(m-1)}{2}$, by taking advantage of the bit masking technique supported in commodity ASICs. Table 4 shows the rules before compression.

Optimality: Algorithm 2 may not return the optimal solution. Consider the example shown in Figure 6. If ELP set consists of shortest and "1-bounce" paths, we know the optimal tagging system only requires *two* lossless queues. However, the greedy algorithm will output a tagging system that requires *three* lossless queues. The reason is that Algorithm 2 does not combine bounces that happen when the packet is going up and when the packet is coming down.

For example, as shown in Figure 6, the bounce of green flow will force Algorithm 2 to create a new tag for the first two hops, since the third hop, which is a bouncing hop, may lead to CBD. However, the blue flow bounces at the last two hops and will force Algorithm 2 to create another new tag. Thus, Algorithm 2 generates three tags, requiring three lossless queues.

The fundamental reason for this is that generic algorithm does not fully utilize the inherent characteristics of structured topology like Clos. We have not been able to find an optimal solution to

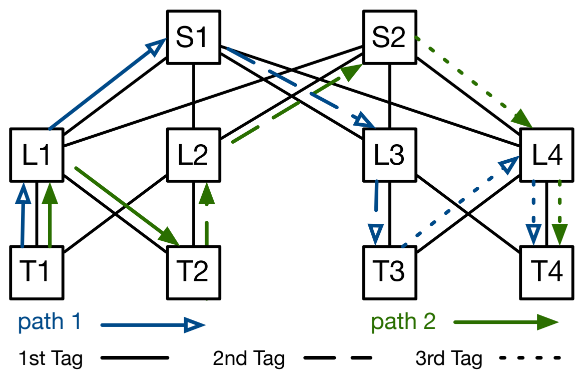


Figure 6: Algorithm 2 does not output optimal result for Clos with 1-bounce paths.

this problem (nor have we been able to prove that the problem is NP-hard) – although we can create topology-specific solutions, as seen in §4.

However, we do note that the number of tags in the solution generated by Algorithm 2 is an upper bound on the optimal solution. Without any assumptions, the worst case is the same as using the brute-force solution, which requires as many tags as the length of longest lossless route, T. However, if we know that the smallest cycle in lossless routes is longer than l, the output number of tags is bounded by $\lceil T/l \rceil$. We omit the proof.

6 DISCUSSION

Multiple application classes: Sometimes, system administrators need to use multiple lossless priorities to keep different traffic classes from impacting each other. For example, in [54] congestion notification packets were assigned a separate lossless class to ensure that these packets would not be held up by data traffic.

A näive way to use Tagger in such cases is to treat each application (or traffic class) separately. For example, in §4.3, we showed that for the Clos network, if ELP contained paths with no more than M bounces lossless, we need $M + 1$ priorities. If there are N applications, the näive approach would use $N * (M + 1)$ priorities.

However, we can use fewer priorities by trading off some isolation. The first application class starts with tag 1, and uses tags up to $M + 1$. The second class starts with tag 2, and also increases tags by 1 at each bounce. Thus, the second class uses tags $2 \ldots M + 2$. Thus, for N application classes need just $M + N - 1$ tags. This can be further reduced by making some application classes to tolerate fewer bounces than others.

Note that there is still no deadlock after such mix. First, there is still no deadlock within each tag, because each tag is still a set of "up-down" routing. Second, the update of tags is still monotonic. We omit formal proof for brevity.

The reduced isolation may be acceptable, since only a small fraction of packets experience one-bounce and may mix with traffic in the next lossless class. This technique can be generalized for the output of Algorithm 2.

Specifying ELP: The need to specify expected lossless paths is not a problem in practice. For Clos networks, it is easy to enumerate paths with any given limit on bouncing. In general, as long as routing is traffic agnostic, it is usually easy to determine what routes the routing algorithm will compute – e.g. BGP will find shortest AS path etc. If an SDN controller is used, the controller algorithm can be used to generate the paths under a variety of simulated conditions. ECMP is handled by including all possible paths.

We stress again that there are no restrictions on routes included in ELP, apart from the common-sense requirement that each route be loop-free. Once ELP is specified, we can handle any subsequent abnormalities.

Use of lossy queue: Some may dislike the fact that we may eventually push a wayward packet into a lossy queue. We stress that we do this only as a last resort, and we reiterate that it does not mean that the packets are automatically or immediately dropped.

Topology changes: Tagger has to generate a new set of tags if ELP is updated. ELP is typically updated when switches or links are added to the network. If a FatTree-like topology is expanded by adding new "pods" under existing spines (i.e. by using up empty ports on spine switches), none of the older switches need any rule changes. Jellyfish-like random topologies may need more extensive changes.

Note that switch and link failures are common in data center networks [53], and we have shown (Figure 3) that Tagger handles them fine.

Flexible topology architectures: Tagger can support architectures like Helios [18], Flyways [28] or Projector [23], as long as ELP set is specified.

PFC alternatives: One might argue that PFC is not worth the trouble it causes; and we should focus on getting rid of PFC altogether. We are sympathetic to this view, and are actively investigating numerous schemes, including minimizing PFC generation (e.g. DCQCN [54] or Timely [38]), better retransmission in the NIC, as well as other novel schemes.

Our goal in this paper, however, is to ensure safe deployment of RoCE using PFC and existing RDMA transport. Tagger fixes a missing piece of the current RoCE design: the deadlock issue caused by existing routing protocols which were designed for lossy networks. Besides the research value of providing a deadlock free network, Tagger protects the substantial investments which we and many others already made in production data centers.

Deployment of Tagger: To deploy Tagger in production data centers, we only need to install some Tagger rules at the switches. As will be explained in Section 7, these rules can be directly expressed with TCAM entries, and hence have no discernible impact on throughput and latency.

7 IMPLEMENTATION

Tagger can be implemented by basic match-action functionality available on most modern commodity switches. However, correct implementation requires a key insight into the way PFC PAUSE frames are handled.

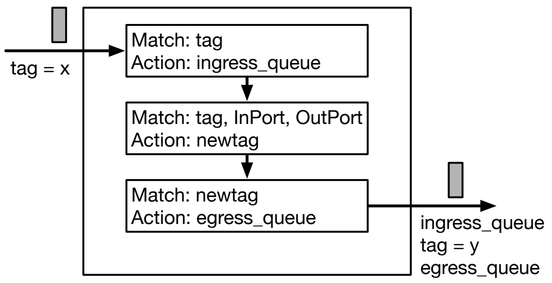
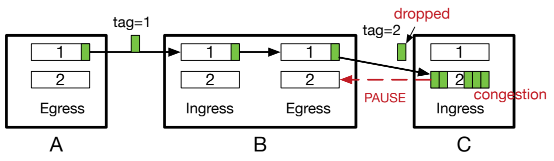
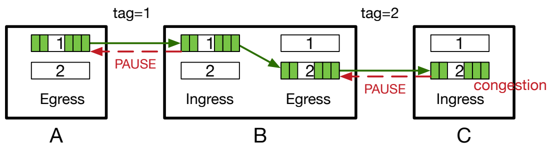


Figure 7: Tagger match-action rules



(a) Ingress priority = egress priority → packet drop.



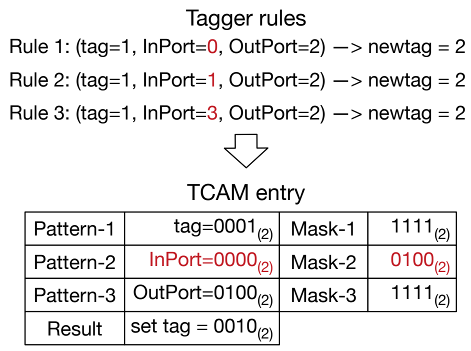
(b) Ingress priority = 1, egress priority = 2 → no drop.

Figure 8: Decoupling ingress priority from egress priority at switch B is necessary for lossless priority transition.

Match-Action rules: Tagger needs to perform two operations at every hop, i.e., *tag-based priority queueing* and *tag rewriting*. These two operations are implemented using a 3-step match-action pipeline (Figure 7). First, Tagger matches the value of tags and classifies packets into ingress queues based. Second, Tagger matches (tag, InPort, OutPort) and rewrites the value of tag. The *third* step, wherein the packet is placed in an egress queue based on the *new* tag value, is needed to ensure correct PFC operation, as described below.

Handling priority transition: By default, a switch will enqueue a departing packet in the egress queue of the same priority class as its ingress queue, as shown in Figure 8(a). In this example, Switch B is configured to perform priority transition for packets received from switch A and destined for switch C. Packets exit egress queue 1 at switch B, but with priority 2. When ingress queue 2 of switch C becomes congested, the PFC PAUSE from switch C to switch B carries priority 2, and cannot pause the egress queue 1 of switch B. This default behavior can lead to packet loss.

Therefore, we must map the packet to the egress queue based on its new priority (Figure 8(b)). This avoids packet loss, since the PFC from switch C correctly pauses the queue on which the packet with the new tag would be exiting.


Tagger rules

Rule 1: (tag=1, InPort=0, OutPort=2) —> newtag = 2

Rule 2: (tag=1, InPort=1, OutPort=2) —> newtag = 2

Rule 3: (tag=1, InPort=3, OutPort=2) —> newtag = 2

TCAM entry

Pattern-1	tag=$0001_{(2)}$	Mask-1	$1111_{(2)}$
Pattern-2	InPort=$0000_{(2)}$	Mask-2	$0100_{(2)}$
Pattern-3	OutPort=$0100_{(2)}$	Mask-3	$1111_{(2)}$
Result	set tag = $0010_{(2)}$		

Figure 9: Rule compression with bit masking. Port numbers are bitmaps. The first bit from right represents port 0. The second bit represents port 1, and so on.

Rule compression: The match-action rules of Tagger are implemented with TCAM. TCAM entries consist of *Pattern*, *Mask*, and *Result*. They refer to the pattern to be matched, the mask bits associated with the pattern and the action that occurs when a lookup hits the pattern, respectively. One TCAM entry can have several Pattern-Mask pairs to match multiple packet header fields simultaneously, e.g., an entry like (Pattern-1, Mask-1, Pattern-2, Mask-2, Result) matches two fields simultaneously and fires only if both matches succeed.

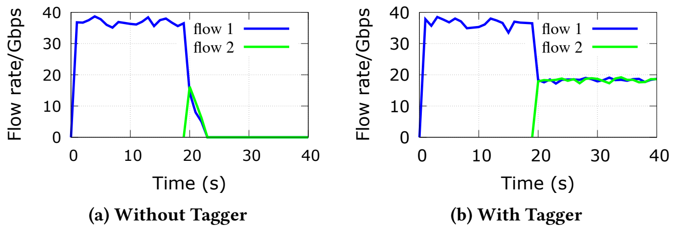
Rules with the same Result can be compressed into one TCAM entry, if their Patterns can be aggregated using bit masking. Consider the three rules in Figure 9. These rules are identical except the InPort field in Pattern.

On commodity ASICs, port numbers in TCAM are bitmaps, not binary values. To match a single port, we can simply set the corresponding bit in the pattern to 1, and set the mask to all 1's. However, we may match multiple ports with one rule. We set the pattern to all 0's, and set the corresponding bits in the mask to 0. As shown in Figure 9, to match InPorts 0, 1 and 3, we set Pattern-2 to "0000" and Mask-2 to "0100". In this case, only the packet received from InPorts 0, 1 or 3 will match Pattern-2 after doing bit masking with Mask-2. Thus, the three rules are compressed into a single TCAM entry.

Recall from §5 that without any compression, we need $n(n-1)m(m-1)/2$ rules per switch. The number of rules can be compressed to $nm(m-1)/2$ by aggregating InPorts. The result can be further improved by doing joint aggregation on tag, InPort and OutPort.

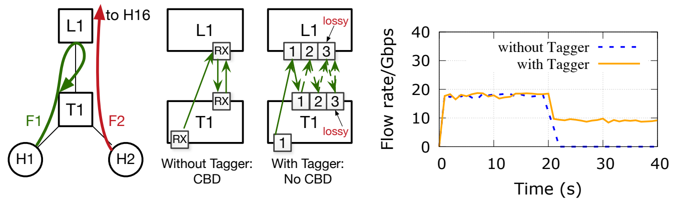
Broadcom implementation: We implemented Tagger on commodity switches based on Broadcom ASICs. We use DSCP field in IP header as the tag. The DSCP-based ingress priority queuing (step 1), ingress ACL and DSCP rewriting (step 2), and ACL-based egress priority queuing (step 3) are well supported by the commodity ASICs and do not require any ASIC changes. Everything is implemented using available and documented functionality.

We considered using TTL instead of DSCP to tag packets, but TTL is automatically decremented by the forwarding pipeline, which complicates the rule structure.



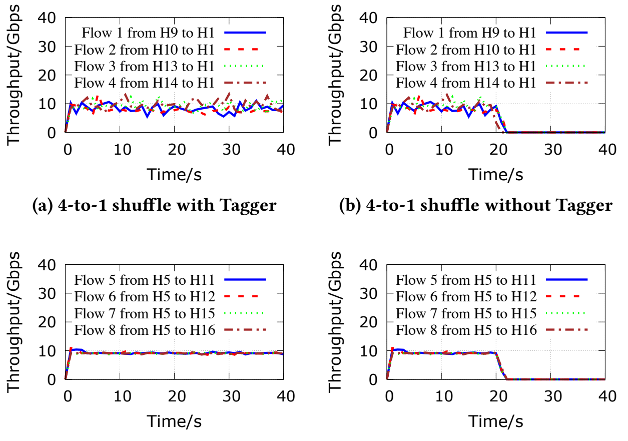
(a) Without Tagger (b) With Tagger

Figure 10: Clos deadlock due to 1-bounce paths



(a) Scenario (b) Rate of flow 2

Figure 11: Deadlock due to routing loop



(a) 4-to-1 shuffle with Tagger (b) 4-to-1 shuffle without Tagger

(c) 1-to-4 shuffle with Tagger (d) 1-to-4 shuffle without Tagger

Figure 12: PFC PAUSE propagation due to deadlock

8 EVALUATION

We evaluate Tagger using testbed experiments, numerical analysis and NS-3 simulations. We consider three questions: (*i*) Can Tagger prevent deadlock? (*ii*) Is Tagger scalable for large data center networks?, and (*iii*) Does Tagger have a performance penalty?

Testbed: Our testbed (Figure 2) consists of a Clos network with 10 Arista 7060 (32x40Gb) switches and 16 servers with Mellanox 40Gb ConnectX-3 Pro NICs.

Switches	Ports	Longest ELP	Lossless Priorities	Max Rules
100	32	5	2	40
500	64	6	3	76
1,000	64	6	3	88
2,000	64	7	3	98
2,000 (*)	64	7	4	135

Table 5: Rules and priorities required for Jellyfish. Half the ports on each switch are connected to servers. ELP is shortest paths for first four entries. ELP for last entry includes additional 20,000 random paths.

8.1 Deadlock prevention

We have already *proved* that Tagger prevents deadlock. Thus, experiments in this section are primarily illustrative. We have also done extensive simulations, which we omit for brevity.

Deadlock due to one bounce: We recreate the scenario shown in Figure 3, where 1-bounce paths lead to CBD. In this experiment, we start the blue flow at time 0, and the green flow at time 20. Figure 10 shows the rate of the two flows with and without Tagger. Without Tagger, deadlock occurs and rate of both flows are reduced to 0. With Tagger, and ELP set to include shortest paths and 1-bounce paths, there is no deadlock and flows are not paused.

Deadlock due to routing loop: As shown in Figure 11(a), we generate 2 flows across different ToRs, i.e., F_1 from H1 to H15 and F_2 from H2 to H16. At time = 20s, we install a bad route at L1 to force F_1 enter a routing loop between T1 and L1. The path taken by F_2 also traverses link T1-L1. ELP is set to include the shortest paths and 1-bounce paths.

Figure 11(b) shows the rate of F_2 with and without Tagger. Without Tagger, deadlock occurs and F_2 is paused due to propagation of PFC PAUSE. With Tagger, there is no deadlock and F_2 is not paused (but rate is affected by the routing loop). Note that throughput of F_1 is zero, as packets are dropped due to TTL expiration. The key takeaway here is that Tagger was able to successfully deal with a routing loop.

PAUSE propagation due to deadlock: Once deadlock occurs, PFC PAUSE will propagate and may finally pause all the flow running in the datacenter network. In this experiment, we run a many-to-one data shuffle from H9, H10, H13 and H14 to H1, and a one-to-many data shuffle from H5 to H11, H12, H15 and H16 simultaneously. We then manually change the routing tables so that the flow from H9 to H1 and the flow from H5 to H15 take 1-bounce paths. This creates CBD as discussed earlier.

In Figure 12, we plot the throughput of all 8 flows with and without Tagger. Without Tagger, all flows get paused due to PFC PAUSE propagation and throughput is reduced to zero. With Tagger, flows are not affected by link failures.

8.2 Scalability

Can Tagger work on large-scale networks, while commodity switches can support only a limited number of lossless queues (§3)? We have already shown that on an arbitrarily large Clos topology, Tagger requires $k+1$ lossless priorities to support paths with up to k bounces. We now consider other topologies.

	8-to-1 scenario	16-to-1 scenario	24-to-1 scenario
Baseline: RoCE without PFC	1.25×10^{-2}	6.6×10^{-2}	1.27×10^{-1}
RoCE + Tagger (1 lossless Q)	0	6.87×10^{-7}	1.61×10^{-6}
RoCE + Tagger (2 lossless Qs)	0	0	0

Table 6: Packet loss rate under web search workload

Jellyfish topology is an r-regular random graph, characterized by the number of switches, the number of ports a switch has (n) and the number of ports used to connect with other switches (r). In our experiment, we let r = n/2. Remaining ports are connected to servers. We construct ELP by building destination-rooted shortest-path spanning trees at all the servers. Table 5 shows the results.

Tagger requires only four classes for a network with 2000 switches, even when 20,000 random routes are used in addition to the shortest paths, and at most[4] 135 match-action rules per switch. Modern commodity switches can support 1-4K rules, so this is not an issue.

We also considered smaller (100 switches, 32 ports) Jellyfish topologies with up to 16-shortest paths between every switch pair. We need only 2 lossless priorities, and no more than 47 rules per switch.

BCube [26] is server-centric topology, constructed from servers with n ports, n^k switches with $k+1$ ports. BCube(8, 3) with ELP of 4 shortest paths requires 4 lossless priorities, and 41 rules per switch. F10 [33] is a fault-tolerant FatTree-like topology. With three-level network of 64 port switches, and ELP of all shortest and 1-bounce paths, we need just 2 lossless priorities and 164 rules per switch.

To conclude, in terms of number of lossless classes and ACLs, Tagger scales well for modern data center topology.

Generating tagging rules is a one-time activity. Still, runtime of Algorithm 2 is of possible interest. Figure 13 shows the runtime for Jellyfish topologies of different sizes. Even with 10000 switches, Algorithm 2 takes just 19.6 hours on a commodity desktop machine.

8.3 Impact on performance

Operators may have the following two concerns when deploying Tagger in production data centers. First, the use of lossy queue as a last resort may cause RoCE to suffer from severe packet loss. Second, making every packet traverse the Tagger rules installed at switches may delay the packet processing and downgrade throughput. In the next, we evaluate the performance impact of Tagger regarding the above two aspects.

Impact of using lossy queue as a last resort: In Figure 14, we measured the percentage of bounced flows under varying link failure rate with flow-level simulations. In our simulator, we model a datacenter network with FatTree topology (with switch port number k = 8, 16, 32 and 64). Every flow is initially routed over a random shortest path. At the switches, we pre-install a set of candidate next-hops for each destination [5]. If a link fails, for every affected flow, the direct connected switch will locally reroute the flow to a random candidate next-hop.

In our simulations, we generate 1 million flows with random source and destination. We count the number of flows bounced once, twice and more than twice under varying link failure rate.

Figure 14(a), (b) and (c) show the percentage of flows bounced once, twice and more than twice, respectively. There are two take-aways. First, when links fail, the behavior of local rerouting has a good chance to cause DOWN-UP bounce for tree-based networks. Second, even under high link failure rate, a flow is rarely bounced twice or more.

We also evaluate the packet loss rate of the lossy queue under stressful traffic using NS-3 simulations. We choose the setting of FatTree(k=8) with 20% link failure rate [6]. We establish many-to-one traffic scenarios by letting servers under different ToRs send traffic to a common destination server. The flows are generated according to web search [5] and data mining [25] workload with 0.99 average load on bottleneck link. At switches, we configure WRR scheduling among lossless and lossy queues with equal weight.

The result under web search workload is shown in Table 6. We omit the similar result under data mining workload. In our simulations, we only consider the congestion loss in the lossy queue. We didn't include the packet loss caused by link failures before rerouting takes effect, as it is not our focus.

We use "RoCE without PFC" as the baseline, where all the flows are in the lossy priority class. Compared with the baseline, Tagger with 1 lossless queue has a much lower packet loss rate (only 10^{-7}-10^{-6}). This is mainly because only $\sim$ 16% of flows are bounced once and enter the lossy priority class. For Tagger with 2 lossless queues, we don't observe any packet loss as only $\sim$ 3% of flows are bounced more than once. The takeaway is as follows: The use of lossy queue as a last resort will not cause severe packet loss, because only a small part of flows will be bounced into the lossy priority class under failures. In practice, making two-bounce paths lossless is good enough to achieve losslessness.

Impact of Tagger rules: On datapath, packets have to traverse the rules installed by Tagger. These rules installed in TCAM, and hence have no discernible impact on throughput and latency. We installed different number of Tagger rules on T1, and measured average throughput and latency between H1 and H2 over several runs. Figure 15 confirms that throughput and latency are not affected by the number of rules.

9 RELATED WORK

Deadlock-free routing. Many Deadlock-free routing designs have been proposed. See [12–14, 16, 17, 19, 21, 24, 40, 44, 47, 48, 52] for representative schemes. Generally, these designs prevent deadlock by imposing restrictions on the routing paths or enforcing certain packet rerouting policies. We classify them into two categories.

The first category is *deterministic routing based approach*, in which the routing path is not affected by the traffic status, and

[4]Different switches require different number of rules due to the random nature of the topology.

[5]In the simulator, we aggregate the destinations to reduce the number of sets needed.
[6]In practice, it is unlikely to have such a high link failure rate. We choose this setting as a stress test for Tagger.

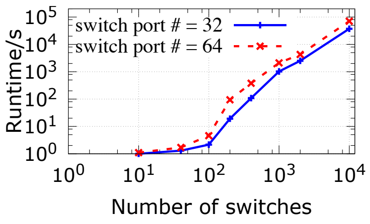
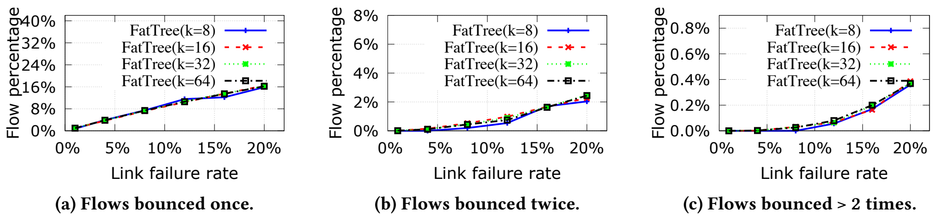
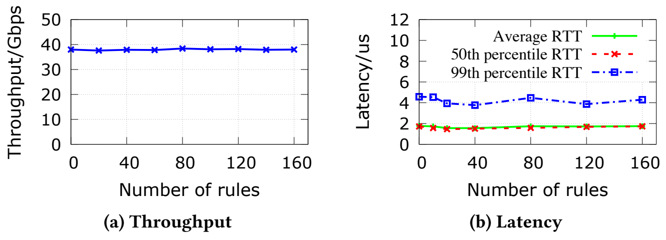


Figure 13: Runtime of Algorithm 2



(a) Flows bounced once. (b) Flows bounced twice. (c) Flows bounced > 2 times.

Figure 14: The percentage of bounced flows under varying link failure rate



(a) Throughput (b) Latency

Figure 15: Tagger rules have no impact on throughput and latency

there is no CBD. For example, the solution proposed by Dally and Seitz [13] split each physical channel into a group of ordered virtual channels[7], and constructed CBD-free routing by restricting packets over decreasing order of virtual channels. TCP-Bolt [49] and DF-EDST [48] are two recent works under this category. They both built edge-disjoint spanning trees (EDSTs) to construct CBD-free routing paths. DF-EDST further built a deadlock-free tree transition acyclic graph, such that the transition among some EDSTs can be allowed. These designs either work only for specific topologies [13] or are not compatible with existing routing protocols including OSPF and BGP [48, 49].

The second category is *adaptive routing based approach*. The key idea is to pre-install "escape" paths at the switches to cover all possible destinations. The switches can reroute packets to the "escape" paths in the presence of congestion so that deadlock can be avoided. However, no commodity switching ASICs so far support the dynamic rerouting based on traffic / queue status required by the adaptive routing designs. Furthermore, a certain amount of buffer needs to be reserved at the switches for the use of pre-installed "escape" paths.

Intel Omni-Path. Intel Omni-Path architecture [8] uses the concept of Service Channels (SC) for routing deadlock avoidance. Unlike Tagger, Ommi-path uses a centralized fabric manager to manage the network [8], including setting up SCs. This is not feasible at data center scale.

Buffer management for deadlock prevention. It has been shown that by increasing the packet priority hop-by-hop, and putting packets of different priority into different buffers, deadlock can be avoided [7, 22, 32, 42]. These designs, however, need a large number of lossless queues, which is the longest path length in the network.

Deadlock detection and recovery. The solutions under this category [34, 36, 45, 50, 51] mainly include two steps. First, employing a heuristic mechanism to detect suspected deadlocks, e.g., monitoring the throughput and queue occupancy of each switch port. Second, recovering deadlock by dropping or temporarily rerouting some of the packets involved in the deadlock. The main problem with these solutions is that they do not resolve the root cause of the detected deadlock, and hence cannot prevent the reappearing of the same deadlock.

Deadlock-free routing reconfiguration. Several schemes [20, 35, 39, 43] have been proposed for ensuring deadlock-free during routing reconfiguration. Tagger can be used to help any routing protocol to be deadlock-free, as Tagger is decoupled from the routing protocols.

Summary. Tagger is different from prior work. Tagger's innovations are its ELP concept, and the algorithms that pre-generate the static tagging rules and minimize the number of priority classes. As a result, Tagger works with any routing protocol, and with existing hardware.

10 CONCLUSION

We have presented Tagger for deadlock prevention in data center networks. By carrying tags in the packets and installing pre-generated match-action rules in the switches for tag manipulation and buffer management, Tagger guarantees deadlock-freedom. Tagger decouples itself from routing protocols by introducing the expected lossless path (ELP) concept, hence it works well with any existing routing protocol, distributed or centralized. Tagger works for general network topologies. We further showed that Tagger achieves optimality for the well-known Clos/FatTree topology, in terms of the number of lossless queues and the number of match-action rules. Tagger can be implemented using existing commodity switching ASICs and is readily deployable.

ACKNOWLEDGEMENTS

This work was partially supported by China 973 Program under Grant No.2014CB340300 and HK GRF-16203715. Shuihai Hu was intern with Microsoft Research Asia when he worked on this project. We would like to thank our shepherd Costin Raiciu and the anonymous reviewers for their helpful feedback and suggestions.

[7]A virtual channel is equivalent to a priority queue in PFC in the store-and-forward setting.

REFERENCES

[1] Ieee. 802.11qbb. Priority based flow control, 2011.

[2] The Microsoft Cognitive Toolkit. https://github.com/Microsoft/CNTK/wiki, 2017.

[3] Martín Abadi et al. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, 2016.

[4] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A Scalable, Commodity Data Center Network Architecture. SIGCOMM ’08.

[5] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). SIGCOMM ’10.

[6] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Don’T Mind the Gap: Bridging Network-wide Objectives and Device-level Configurations. SIGCOMM ’16.

[7] Dimitri Bertsekas and Robert Gallager. *Data Networks*. Prentice Hall, 1992.

[8] Mark S. Birrittella, Mark Debbage, Ram Huggahalli, James Kunz, Tom Lovett, Todd Rimmer, Keith D. Underwood, and Robert C. Zak. Intel Omni-Path Architecture Enabling Scalable, High Performance Fabrics. In *Hot Interconnects*, 2015.

[9] Jacek Blazewicz, Daniel P. Bovet, Jerzy Brzezinski, Giorgio Gambosi, and Maurizio Talamo. Optimal centralized algorithms for store-and-forward deadlock avoidance. *IEEE transactions on computers*, 43(11):1333–1338, 1994.

[10] Cisco. Priority Flow Control: Build Reliable Layer 2 Infrastructure. http://www.cisco.com/c/en/us/products/collateral/switches/nexus-7000-series-switches/white_paper_c11-542809.pdf.

[11] Charles Clos. A Study of Non-Blocking Switching Networks. *Bell Labs Technical Journal*, 32(2):406–424, 1953.

[12] William J. Dally and Hiromichi Aoki. Deadlock-Free Adaptive Routing in Multicomputer Networks Using Virtual Channels. *IEEE Transactions on Parallel and Distributed Systems*, 4, April 1993.

[13] William J Dally and Charles L Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on computers*, C-36, May 1987.

[14] Jens Domke, Torsten Hoefler, and Wolfgang E. Nagel. Deadlock-Free Oblivious Routing for Arbitrary Topologies. IPDPS ’11.

[15] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, pages 401–414, 2014.

[16] J. Duato and T. M. Pinkston. A General Theory for Deadlock-Free Adaptive Routing Using a Mixed Set of Resources. *IEEE Trans. Parallel Distrib. Syst.*, 2001.

[17] Jos Duato. A New Theory of Deadlock-Free Adaptive Routing in Wormhole Networks. *IEEE Transactions on Parallel and Distributed Systems*, 4, December 1993.

[18] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Hajabdolali Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Helios: a hybrid electrical/optical switch architecture for modular data centers. *ACM SIGCOMM Computer Communication Review*, 40(4):339–350, 2010.

[19] Jose Flich, Tor Skeie, Andres Mejia, Olav Lysne, Pierre Lopez, Antonio Robles, Jose Duato, Michihiro Koibuchi, Tomas Rokicki, and Jose Carlos Sancho. A survey and evaluation of topology-agnostic deterministic routing algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 2012.

[20] Alan Gara, Matthias A Blumrich, Dong Chen, GL-T Chiu, Paul Coteus, Mark E Giampapa, Ruud A Haring, Philip Heidelberger, Dirk Hoenicke, Gerard V Kopcsay, et al. Overview of the Blue Gene/L system architecture. *IBM Journal of Research and Development*.

[21] David Gelernter. A DAG-Based Algorithm for Prevention of Store-and-Forward Deadlock in Packet Networks. *IEEE Trans. Compu.*, C-30, October 1981.

[22] M. Gerla and L. Kleinrock. Flow Control: A Comparative Survey. *IEEE Trans. Commun.*, COM-28, April 1980.

[23] Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Nikhil Devanur, Janardhan Kulkarni, Gireeja Ranade, Pierre-Alexandre Blanche, Houman Rastegarfar, Madeleine Glick, and Daniel Kilper. Projector: Agile reconfigurable data center interconnect. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pages 216–229. ACM, 2016.

[24] Christopher J. Glass and Lionel M. Ni. The Turn Model for Adaptive Routing. *SIGARCH Comput. Archit. News*, 1992.

[25] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: A scalable and flexible data center network. In *SIGCOMM*, 2009.

[26] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. BCube: A high performance, server-centric network architecture for modular data centers. In *SIGCOMM*, 2009.

[27] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitendra Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *SIGCOMM ’16*.

[28] Daniel Halperin, Srikanth Kandula, Jitendra Padhye, Paramvir Bahl, and David Wetherall. Augmenting data center networks with multi-gigabit wireless links. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 38–49. ACM, 2011.

[29] Shuihai Hu, Yibo Zhu, Peng Cheng, Chuanxiong Guo, Kun Tan, Jitendra Padhye, and Kai 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. ACM, 2016.

[30] Infiniband Trade Association. Supplement to InfiniBand Architecture Specification Volume 1 Release 1.2.2 ANNEX A17: ROCEV2 (IP ROUTABLE ROCE)), 2014.

[31] InfiniBandcntk. InfiniBand Trade Association, InfiniBand Architecture, Specification. http://www.infinibandta.com, 2001.

[32] Mark Karol, S Jamaloddin Golestani, and David Lee. Prevention of deadlocks and livelocks in lossless backpressured packet networks. *IEEE/ACM Transactions on Networking*, 2003.

[33] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. F10:A Fault-Tolerant Engineered Network. In *NSDI*, 2013.

[34] P. López and J. Duato. A Very Efficient Distributed Deadlock Detection Mechanism for Wormhole Networks. HPCA ’98.

[35] Olav Lysne, Timothy Mark Pinkston, and Jose Duato. A methodology for developing deadlock-free dynamic network reconfiguration processes. part ii. *IEEE Transactions on Parallel and Distributed Systems*.

[36] Juan Miguel Martínez, Pedro Lopez, José Duato, and Timothy Mark Pinkston. Software-based deadlock recovery technique for true fully adaptive routing in wormhole networks. In *Parallel Processing, 1997., Proceedings of the 1997 International Conference on*.

[37] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *USENIX Annual Technical Conference*, pages 103–114, 2013.

[38] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. In *SIGCOMM ’15*.

[39] Timothy Mark Pinkston, Ruoming Pang, and José Duato. Deadlock-free dynamic reconfiguration schemes for increased network dependability. *IEEE Transactions on Parallel and Distributed Systems*.

[40] V. Puente, R. Beivide, J. A. Gregorio, J. M. Prellezo, J. Duato, and C. Izu. Adaptive Bubble Router: A Design to Improve Performance in Torus Networks. ICPP ’99.

[41] Sophie Y Qiu, Patrick Drew McDaniel, and Fabian Monrose. Toward valley-free inter-domain routing. In *2007 IEEE International Conference on Communications*, pages 2009–2016. IEEE, 2007.

[42] E. Raubold and J. Haenle. A Method of Deadlock-free Resource Allocation and Flow Control in Packet Networks. In *ICCC*, Auguest 1976.

[43] Thomas L Rodeheffer and Michael D Schroeder. *Automatic reconfiguration in Autonet*, volume 25. ACM, 1991.

[44] Jose Carlos Sancho, Antonio Robles, and Jose Duato. An effective methodology to improve the performance of the up*/down* routing algorithm. *IEEE Transactions on Parallel and Distributed Systems*.

[45] Alex Shpiner, Eitan Zahavi, Vladimir Zdornov, Tal Anker, and Matty Kadosh. Unlocking credit loop deadlocks. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pages 85–91. ACM, 2016.

[46] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P. Brighten Godfrey. Jellyfish: Networking data centers randomly. In *NSDI*, 2012.

[47] Tor Skeie, Olav Lysne, and Ingebjørg Theiss. Layered Shortest Path (LASH) Routing in Irregular System Area Networks. In *Prof. of IPDPS*, 2012.

[48] Brent Stephens and Alan L. Cox. Deadlock-Free Local Fast Failover for Arbitrary Data Center Networks. In *IEEE Infocom*, 2016.

[49] Brent Stephens, Alan L Cox, Ankit Singla, John Carter, Colin Dixon, and Wesley Felter. Practical dcb for improved data center networks. In *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*, pages 1824–1832. IEEE, 2014.

[50] Anjan K. V. and Timothy Mark Pinkston. An Efficient, Fully Adaptive Deadlock Recovery Scheme: DZSHA. In *ISCA*, 1995.

[51] Anjan K. Venkatramani, Timothy Mark Pinkston, and José Duato. Generalized Theory for Deadlock-Free Adaptive Wormhole Routing and Its Application to Disha Concurrent. IPPS ’96.

[52] Jie 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.

[53] Xin Wu, Daniel Turner, Chao-Chih Chen, David A. Maltz, Xiaowei Yang, Lihua Yuan, and Ming Zhang. Netpilot: Automating datacenter network failure mitigation. In *SIGCOMM ’12*.

[54] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. In *SIGCOMM ’15*.