Practical PFC Deadlock Prevention in Datacenter Networks

ABSTRACT
to be written.

1. Introduction

In this paper, we present a simple and practical mechanism called Tagger to prevent deadlocks in large deployments of Remote Direct Memory Access (RDMA) over Ethernet (RoCE).

Public cloud providers like Microsoft [20] and Google [11] are deploying RoCE in their data centers to provide low latency, high throughput network transfers with minimal CPU overhead [20]. RoCE uses Priority Flow Control (PFC) to prevent packet losses due to buffer overflow at the switches. PFC allows a switch to temporarily pause its immediately upstream neighbor to prevent buffer overflow. While PFC is effective, it can cause numerous problems [20], including deadlocks [**?**, 6, 7].

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 [6]. 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 [6]

Deadlock in PFC-enabled network can occur in numerous ways [7], although in all cases, there is a circular buffer dependency (CBD) among the group of deadlocked switches. We will describe CBD more formally later in the paper. For now, the simple example shown in Figure 1 is sufficient – deadlock is formed, since switch A is blocked by switch B, which is blocked by C, which is paused by A.

A number of solutions to the deadlock problem are known, and they fall in two broad categories. The first category consists of solutions that detect the formation of the deadlock and then use various techniques to break it [14]. The problem with these solutions is that they do not address the root cause of the problem – and hence cannot guarantee that the deadlock would not immediately reappear. We do not consider these solutions any further in this paper.

The second category of solutions are designed to prevent the key necessary condition for deadlock formation – namely, the CBD. There are three ways to prevent CBD. Some solutions require centralized, SDN-style routing. These solutions are difficult to deploy in existing data centers, without wholesale infrastructure changes. The second category of solutions require major changes to the routing protocols [**?**], and thus cannot be implemented using commodity switches. Many of these solutions also require carefully controlling path of each packet – something that is simply not possible with decentralized routing in presence of link failures [19]. Finally, there are protocols that require creation of numerous priorities and buffer management according to those priorities. For example, if each of the flows in Figure 1 had its own priority and was buffered separately at each switch, there would be no deadlock. However, each priority class requires reservation of a certain amount of "headroom" buffer space, which limits the total number of priorities one can support. This headroom depends, among other factors, on the cable length – since the buffer must be large enough to absorb all packets in flight. Given the long cable lengths used in modern data centers, and the shallow buffers available on commodify switches, modern data center can typically support only two or three lossess priorities [6].

Given these shortcomings, to the best of our knowledge, no deadlock free routing solution has been deployed in production networks.

In contrast, our solution, Tagger, can be implemented on existing commodity switches, and does not require any changes to the existing routing protocol deployed in the data center. It also does not require an excessive number of priorities or queues.

The key insight behind Tagger is that CBD is caused by flows that take an "abnormal" path through the network. For example, a flow may end up on an "abnormal" path after BGP reacts to link failure. Not all abnormal paths cause CBD, however. Tagger consists of a clever tagging and buffer management scheme, such that packets that may cause CBD are automatically detected diverted to a different queues. When properly configured, this avoids CBD and hence deadlock. We show that for general topologies, TTL values can be used for tagging purpose, and we design a greedy algorithm that combines multiple tags into one lossless priority and effectively reduce the number of required queues. While the greedy heuristic is not optimal, for popular data center topologies such as as FatTree, we show that we can reduce the number of priorities even further by exploiting special structure of the topology.

We have implemented and tested Tagger on commodity Arista 7050 Switches that use the popular Broadcom chipsets.

Our key contributions are as follows: first, using traces from a large cloud provider's data center, we demonstrate the practical challenges in making RoCE deadlock-free. We

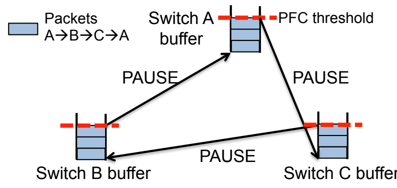


Figure 1: PFC-induced deadlock: simple illustration

also analyze Tagger using simulations and a testbed implementation. Second, we develop Tagger for general topologies and formally prove that it prevents deadlocks. Third, we optimize Tagger for popular data center topologies such as FatTree to achieve optimal results and efficiently support multiple application classes.

2. Background

We now provide a brief primer on RDMA, RoCE, the problem of deadlocks in data center networks, and prior work in this area.

RDMA and RoCE: Remote Direct Memory Access (RDMA) technology offers high throughput, low latency and low CPU overhead, by bypassing end-host networking stacks. Instead, Network Interface Cards(NICS) transfer data in and out of pre-registered memory buffers at the two end hosts. In modern data centers, RDMA is deployed using RDMA over Converged Ethernet V2 (RoCE) standard [1, 8]

PFC: RoCE needs a lossless L2 layer for optimal performance. This is accomplished in Ethernet networks using the Priority Flow Control (PFC) mechanism [2]. 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. See [4, 20] and §3.3 for details.

The PFC standard defines 8 classes[1], called priorities [2]. Packets in each priority are buffered separately, and PAUSE messages carry this priority. When a packet arrives at port i of switch S with priority j, it is enqueued in queue j of port i. If the queue length now exceeds the PFC threshold, a pause message (XOFF) 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 (XOFF) with priority j is received.

In short, PFC does flow control on a per-port, per-priority basis. This can cause a number of problems, such as unfairness and head-of-the-line-blocking [20]. Worse yet, it can lead to deadlocks.

Deadlock: PFC can lead to deadlocks, if paused links form a cycle, as shown in Figure 1. Once formed, deadlock is "permanent" in the sense that it will continue to exist even if no new traffic is injected into the loop. Deadlocks in PFC-based networks (or more generally, in credit-flow networks) are a well-known problem. It is not merely a theoretical problem – it has been reported in practice [6].

In [7] it was shown that Circular Buffer Dependency (CBD) is a *necessary* condition for deadlock formation. *Sufficient* condition for deadlock formation in PFC networks have yet to be fully understood [7].

Prior work on deadlock avoidance: Prior work on deadlock management falls in two categories: deadlock avoidance, or deadlock detection and resolution. Our focus in this paper is on deadlock avoidance. Since *sufficient* conditions for deadlock formation are not well characterized, deadlock avoidance schemes focus on preventing CBD for occurring. This is done either by limiting or modifying routing [17] to avoid CBD, or by careful buffer management [**?**].

However, these schemes fail to meet one or more of the three key challenges: (i) they cannot be deployed with existing routing, or, (ii) they do not deal with dynamic nature of data center networks, or, (iii) they require excessive switch buffers or number of priorities.

In the next section, we will describe these three challenges in more detail, and briefly review why previously proposed schemes for deadlock avoidance fail to meet them. We provide a more detailed review of related work in §8.

3. Challenges

3.1 Working with existing routing protocols

Most of the current proposals [9, 13, 16–18] for deadlock avoidance are based on custom routing protocols that go to great lengths to avoid CBD. Unfortunately, this makes it very difficult, if not impossible to deploy them in existing data center networks.

Modern data centers are built atop Ethernet and IP. Routing is accomplished in a variety of ways, including BGP [3, 5] or using SDN-like protocols [15]. No matter what routing protocols are used, data center operators tune them carefully satisfy numerous requirements such as manageability and fault tolerance. In addition, operators invest heavily in tools and technologies to monitor and maintain their networks.

Thus, it is very difficult for a data center operator to substantially change their routing infrastructure in order to avoid deadlocks. This task appears especially onerous because RoCEv2 itself can be deployed without any changes to routing protocols – RoCEv2 packets are encapsulated in normal UDP packets are routed like any normal IP packets.

Thus, unlike prior work, Tagger requires no changes to the underlying routing protocol.

3.2 Data Center Networks are Dynamic

Building a deadlock avoidance scheme that does not make any changes to routing protocols is easier said than done. The key problem is that the routing is dynamic – paths can change in response to link failures or load.

Take Figure 2 as an example. It shows a simplified (and small) version of network deployed in our data center. If

[1]Although, only one or two can be used in practice – see §3.3.
[2]The word priority is a misnomer. There is no implicit ordering among priorities – they are really just separate classes.

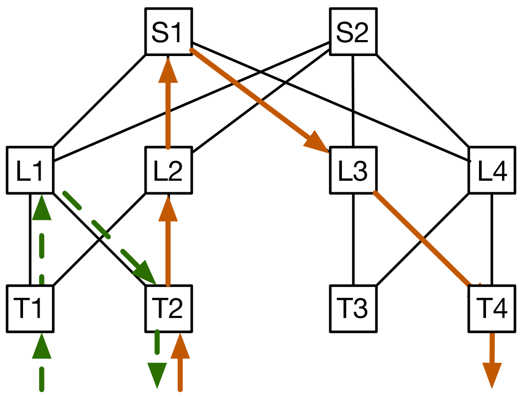


Figure 2: UP-DOWN routing in a Clos network.

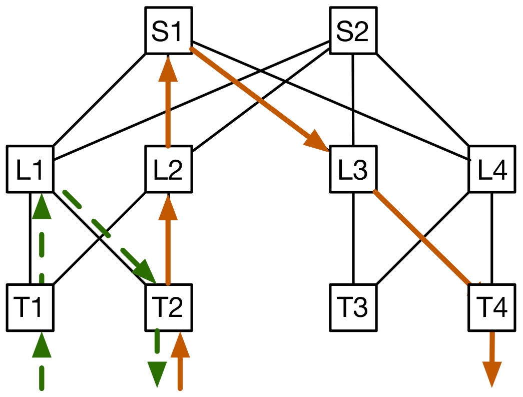


Figure 3: Violation of up-down routing can lead to deadlocks

packets always follow so-called up-down (also called valley-free [12]) routing, then deadlock cannot happen as CBD is not possible. 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.

However, packets may deviate from the UP-DOWN paths due to several reasons including link flapping and routing protocol flapping [10]. As have shown in [14], when the UP-DOWN property is broken and packets may reroute multiple times between two layers of switches, and deadlocks may form as a result, as shown in Figure 3.

By carefully tracking TTL values, we estimated that on average, 0.001% of the packets in our data center are forced to take a path that violates UP-DOWN routing. While this fraction appears small, given that trillions of packets traverse our network every day, the absolute number of packets that take such paths is quite high – which makes the threat of deadlocks, as discussed in [6, 7, 14] quite real.

3.3 Limited Number of Lossless Queues

One easy way to avoid deadlock without changing routing is to buffer packets from each flow separately from other flows at each hop – essentialy putting each flow in its own class, and applying per-hop backpressure only within the class. A more sophesticated scheme in [9] requires as many priorities as the diameter of the network.

One obvious problem with this idea is that the PFC standrad supports only 8 priority classes. But a bigger problem is that a switch can support only a small number of lossless queues.

The number of lossless queues that a switch can support is limited by two factors. First, the commodity switching ASICs typically support only a small number of queues (e.g., eight) and we need to use some of the queues for the lossy traffic. Second, to guarantee the lossless property, a switch needs to reserve certain amount of *headroom* from the memory pool. The size of the memory pool is of limited size, hence the number of lossless queues is further limited by the size of the memory pool.

The reserved headroom per port per lossless queue is to absorb the packets in flight from the time a receiver decides to send a PFC pause frame to its upstream sender to the time the sender stops transmitting after receiving the pause frame. See [6] for how PFC works. We describe how the headroom size is calculated in Appendix A.

From Appendix A, we can see that for a typical 32-port 40GbE Ethernet switch, it needs to reserve 2.76MB memory as the headroom to support one lossless queue. For a switch of 12MB memory, this is 23% of the total memory.

The headroom calculated in Appendix A is to make sure that packets cannot be dropped. In practice, the reservation size should be larger than that. This is because we need some additional reservation to make sure that the link is not under-utilized, when the receiver releases the sender from been paused. Furthermore, we need to reserve buffers for lossy traffic, which is still the dominating traffic in data centers. Consider all these constrains, the number of lossless queues can be supported by the current commodity switches typically is limited to two.

We note that new switching ASICs may be able to support more lossless queues by adding more memory, using smaller cell size (64-byte), reducing the pause frame response time. But the widely deployed switches support only two lossless queues. It is unlikely that the switches can support more than four or five lossless queues. Hence the solutions that use a large number of lossless queues are not practical.

We now describe how Tagger addresses these three challenges.

4. Tagger design

The design of Tagger is driven by the need to address the three challenges described in the previous section. Before delving into the details of the key algorithms in Tagger, it is useful briefly describe the physical implementation of Tagger. The implementation is described in detail in §6.

Tagger works by tagging packets. A packet arrives at the ingress port i of a switch S with a tag j. The arriving packet packet is enqueued in queue j. At departure, based on the values of i and j, the packet is possibly given a new tag, say k. The rest of the forwarding pipeline, as well as PFC PAUSE/Resume behavior (§2) operates without change.

The smartness of the system lies in generating the two mappings: from ingress port and incoming tag to the queue, and from ingress port and incoming tag to the new tag. These mappings are calculated offline, and installed in the switch. Thus, there is no additional overhead at run time – Tagger operates at line speed.

Tagger does not require any changes to the topology of the network or the routing. However, our key insight is that we can take the topology and the routes that must be lossless (we call them *lossless routes*) as input, and use them to generate the mappings described earlier.

The mappings generation is based on three main ideas. First, the switch configurations eliminate deadlock by reacting to the past path of each packet and move the packet into a safe priority just before CBD may appear. Second, the transition of priority is designed carefully so that a single tag in the packet header is sufficient for each switch to make such decisions. Finally, we make sure that Tagger requires only a small number of lossless queues. We now describe these ideas in detail.

4.1 Idea 1: Priority Transition Based on Micropaths

While Tagger works for any topology, we use most popular data center topology – Clos – as an example in this section.

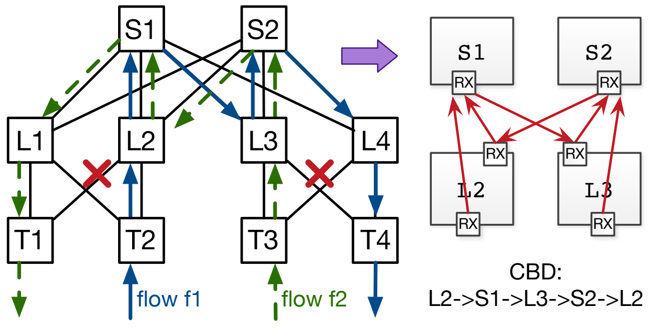
Figure 4(a) shows a simple Clos topology. Let's assume that the operator wants all normal paths to be lossless. In addition, let's assume that the operator is worried about rerouting due to link failures.

Typically, the "normal" paths will be the shortest paths – which for a CLOS network are the "up-down" paths described in the last section. In addition, due to link failures, we may have paths that are not up-down. Such paths can form CBD, and lead to deadlock, as shown in Figure 4(a)) [14].

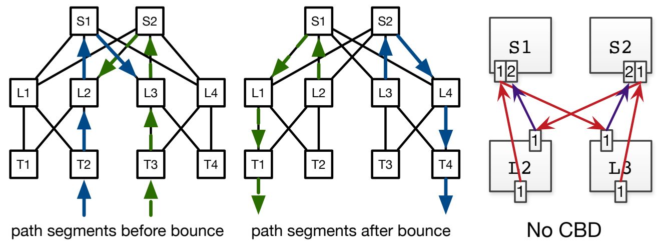
However, if we divide the paths into two segments, *before-bounce* and *after-bounce*, and assign them to different priority queues, there is no more CBD (Figure 4(b)). The insight we get from this example is that if the switch can detect that a packet may cause CBD in next hops, it can change the packet's priority to avoid CBD, thus avoiding deadlock.

Note that this is different from the prior solutions, which either assign a fixed priority based on pre-computed paths (and thus cannot react to dynamic routing), or change the priority every hop (and thus requires too many priority queues).

Instead, we divide the paths into a few segments, which we call *micropaths*, and change packet's priority only at a few transition point. It has the advantages from both types of prior work, *i.e.,* it requires a small number of priorities,



(a) 1-bounce paths creates CBD.



(b) CBD is eliminated with path segmenting and prioritizing.

Figure 4: Micropath based priority transition can eliminate CBD.

and allows the switch to react to each packet's path.

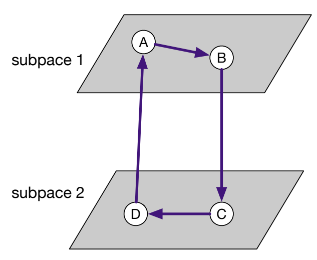
The key enabler of our approach is the topology and the set *lossless routes* specified by the operators as the input. Specifying lossless routes is not a significant burden: in our example, the operator may specify that all normal up-down paths paths with a single "bounce" are "lossless". Given the topology and the routing protocol it is easy to automatically enumerate all such paths [3].

By properly dividing lossless routes into multiple subspaces of micropaths, we can ensure that CBD is eliminated *within* each subspace. We will discuss the details of the algorithm are described later in this section. But first, we must consider another issue: Even though proper micropath partition ensures no CBD *within* each priority, packets can still cause CBD *across* priorities, as shown in Figure 5(a). Switch must know the past path of each packet in order to decide the priority that avoids CBD. Obviously, the tag size is limited, so we can't carry the entire history of the packet in the tag. Thus, we need the next idea.

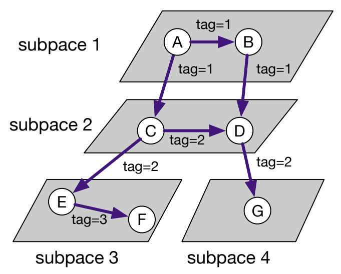
4.2 Idea 2: Tag for Ordered Micropath Subspaces

As shown in Figure 5(b), we enforce the order of transitioning among micropath subspaces and tag only based on current subspace. With this, the packets do not need to record the whole history of past micropaths in headers. A

[3]What happens if there are so many link failures that some packets end up taking a two-bounce path? In such case, our tagging scheme moves the packets to a lossy queue, which follows the standard drop-tail discipline. Such queues, obviously, do not deadlock



(a) CBD among subspaces.



(b) Ordered subspaces.

Figure 5: Ordered subspaces ensure deadlock-free priority transition.

fixed length header field (most commonly, DSCP) is sufficient to carry the tag. The tag values are one-one mapped to the micropath subspaces, which one-one maps to priority queues on each switch.

Thus, with Tagger, the switch enqueues each packet into priority queue based on its tag, and changes the tag if the switch decides to change the packet's priority at the next hop. We can prove that the network is deadlock-free if the tag system, which defines micropath subspaces and transition, satisfies the two requirements below:

1. A packet's tag is unchanged or changed at each hop of its path. When the tag is changed, it must be changed monotonically along any packet's path (e.g., always increasing).

2. For any given tag, all micropaths within the corresponding micropath subspace do not form cyclic buffer dependency.

Proof of deadlock freedom:

Claim: Any tag system that satisfies the above two requirements is deadlock-free.

Proof sketch: We prove by contradiction. Suppose we can find a topology and a set of lossless routes, for which there exists a subspace partition and priority transition that satisfies the above two requirements, but is not deadlock-free. Then, as the tag system is not deadlock-free, we can find a set of micropaths that form a CBD.

Case 1: All these micropaths are in the same subspace. According to requirement 2, micropaths within the same subspace do not form CBD. Contradicted.

Case 2: These micropaths are in different subspaces. We sort these micropaths according to the subspaces they belong to, and choose a micropath in the smallest subspace. Since these micropaths can form a CBD, starting from the source end of the chosen micropath, we can always find a circular path back to the chosen micropath when traversing along these micropaths.

According to requirement 1, once we enter a larger subspace, we can never go back. So this circular path should only include micropaths in the smallest subspace. But according to requirement 2, micropaths within the same subspace do not form a CBD. Hence no circular path can be found within the smallest subspace. So this circular path does not exist. Contradicted.

Thus, we conclude that there do not exist a set of micropaths that form a CBD. This contradicts the supposition that the tag system is not deadlock-free. Hence, any tag system that satisfies the above two requirements is deadlock-free.

Putting these two ideas together we can generate a system of tags and tag conversion tables for each switch. However, we must pay heed to the fact that a switch cannot support more than one or two lossless priorities in practice. Thus, we must minimize the number of tags we use.

4.3 Idea 3: Minimizing the Number of Tags

The number of tags (or, corresponding micropath subspaces) is essentially the number of lossless queues required on each switch. We must reduce it to below the switch hardware limit. In this section, we focus on the number of lossless queues for just *one* application class. We first formalize the tagging system.

Tagging system. 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 system. With a tagging system, the *tagged graph* $G(V,E)$ is generated following below rules.

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

2. In G, there exists an edge $(A_i, x) \rightarrow (B_j, y)$ *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.

The tags define a partition of the tagged graph, $\{G_k\}$, where $G_k = \{(A_i, k)|\forall A, i\}$. Each G_k is a *micropath subspace* and has a unique lossless priority. On switches, we configure ACL rules to match on tags and assign lossless priorities according to $\{G_k\}$. In addition, each edge corresponds to a switch action of setting the tag for the next hop at the egress.

Handling packets that do not follow the lossless routes. If a packet does not match any tag-setting actions, it means the packet is not expected in the lossless routing. Such packets are assigned a special tag, and all switches match on this tag and assign lossy priority. This rule sits at the bottom of switch egress ACL, acting as a default safeguard that avoids unexpected buffer dependency.

To guarantee deadlock-free, two requirements directly follow Section 4.2. First, any G_i does *not* have a cycle. This is because each edge in G_i is essentially a buffer dependency – whether A_i can dequeue packets depending on whether B_j

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

Table 1: Notations in the formalized description.

has paused upstream. A cycle in G_i means cyclic buffer dependency. Second, There is no lossless route going from G_i to G_j if $i < j$, because we enforce the order of micropath subspaces.

Brute-force tagging system. For general graph without structure information, a straight forward tagging system is to monotonically decrease the tag (thus, the priority) every hop, as described in Algorithm 1. It is easy to verify that the above requirements are met, so deadlock is eliminated with this tagging system. However, it requires too many lossless queues in a large network since it depends on the diameter of the topology.

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

```
Input: Topology and routing paths $R$ that must be lossless
Output: A tagged graph $G(V, E)$
$V \leftarrow Set()$;
$E \leftarrow Set()$;
$maxTag \leftarrow$ longestPath($R$)+1;
for each path $r$ in $R$ do
    $tag \leftarrow maxTag$;
    for each hop $h$ in $r$ do
        $V \leftarrow V \cup \{(h, tag)\}$;
        $E \leftarrow E \cup \{lastHop \to (h, tag)\}$;
        $tag \leftarrow tag - 1$;
return $G(V, E)$;
```

Greedy algorithm. Leveraging the brute-force tagging system as a start point, we design Algorithm 2 minize the number of lossless queues. It works by greedily combining as many nodes, from brute-force tagging system, as possible into each micropath subspaces under CBD-free constraint. To ensure the monotonic property, we start from combing the nodes with largest tag to smallest tag in the brute-force tagging system. Obviously, the monotonic property will still hold after combination.

We assign a new tag t' (different from the brute-force tag) for each micropath subspace, and generate switch configurations based on the algorithm output. The worst case scenario is as bad as using the original input, brute-force tags, which require as many priority queues as the length of longest lossless route. However, in Section 7, we show that this algorithm works reasonably well for generic topology. highlight: For example, Jellyfish with 1000 nodes require only X priorities for *one* application class.

5. Optimal Solution for Structured Topology

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

```
Input: The brute-force tagged graph $G(V, E)$
Output: A new tagged graph $G'(V', E')$ that has small $|\{G'_k\}|$
Initialize $V'$, $E'$, $V_{tmp}$, $E_{tmp}$ as empty $Set()$;
$t' \leftarrow 0$;
for $t \leftarrow maxTag$ down to $minTag$ 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
        $t' \leftarrow t' + 1$;
        $V_{tmp} \leftarrow$ {nodes in $V'$ with tag $t' + 1$};
        $E_{tmp} \leftarrow$ {edges in $V'$, both ends have tag $t' + 1$};
return $G'(V', E')$;
```

5.1 Caveats of the Generic Greedy Algorithm

Although the generic Algorithm 2 has significantly reduced the number of lossless queues, it has two main caveats.

First, it may not always return the optimal solution. For example, we again consider the simple three-layer Clos network in Figure **??**. Assuming non-bounce packets and one-bounce packets must be lossless, we know the optimal tagging system only requires *two* lossless priorities. However, the greedy algorithm will output *three* micropath subspaces. The reason is that greedy algorithm does not combine bounces that happen at different hops. As a result, it has to assign a separate prirotiy for each case, as shown in Figure **??**.

Second, the generic algorithm is inefficient in reusing priorities for multiple application classes. For example, the generic algorithm output is two priorities per application class, and the operators have up to two prioritie on switches. In this case, only *one* lossless application class is supported. However, if we can find a smarter tagging system that ensures basic connectivity with only the first micropath, we can compress *two* application classes into two switch priorities, as described in Section 5.2.

The fundamental reason of these problems is that generic algorithm does not fully utilize the inherently characteristics of structured topology. In this section, we will show the smarter tagging systems for popular topology including Clos and BCube. The smarter tagging system addresss both of the problems above.

5.2 Optimizing Clos Topology

The tagging algorithm for Clos topology is straightforward. We first define the baseline UP-DOWN routing as first micropath space, with tag 0. Every time the packet bounces, we increase the tag by one, until the packet is assigned to a lossy queue. Algorithm 3 shows the process. For Clos topology, we can simply replace Algorithm 1 by Algorithm 3, and Algorith 2 is not needed any more. With k lossless priorities, packets with up to $k-1$ bounces are all lossless. It's easy to prove that, to handle $k-1$ bounces, at least k lossless priorities are required.

Algorithm 3 The optimal tagging system for Clos topology.

```
Input: Clos topology and routing paths $R$ that must be lossless
Output: A tagged graph $G(V,E)$
$V \leftarrow Set()$;
$E \leftarrow Set()$;
for each path $r$ in $R$ do
    $tag \leftarrow 0$;
    for each hop $h$ in $r$ do
        $V \leftarrow V \cup \{(h, tag)\}$;
        $E \leftarrow E \cup \{lastHop \rightarrow (h, tag)\}$;
        if $h$ is up-facing && nextHop is down-facing then
            $tag \leftarrow tag + 1$;
return $G(V,E)$;
```

Priority reuse. Above tagging system allows us to run k application classes with k lossless priorities. It works as follows. We start the first class with priority 0 (tag 0), and change to one higher priority (tag) every time the packet bounces. We can then start the second class with tag 1, and change tags just like the first class. Then third class starts with tag 2, and so on. We can prove that such priority reuse will not create CBD.

The trade-off is that the second class supports one less bounce, the third class supports two less, and so on. Under fixed constraints on the number of priorities on switches, how many application classes run in parallel? we leave the choice to operators. In practice, we run two application classes with two priority queues on switch. The second class is still lossless for most of the time, as numbers in Section 3.2 show.

On the other hand, the generic algorithm cannot support this at all. The problem is the lossless routing will become disconnected if any one priority (a micropath segment) is missing. Therefore, when the second application class starts with tag 1, the traffic between at least some hosts becomes *always* lossy.

5.3 Optimizing BCube Topology

5.4 High-level Takeaways

6. Implementation

The output of our algorithm can already be implemented in the commodity switches. In this section, we describe the details of our implementation, starting by explaining how the commodity shared-buffer switch maintains the queues and handles PFC.

6.1 Switch Model

In this part, we abstract the switch model on which we implement our algorithm output. Most commodity shared-buffer switches comply with this model. As shown in Fig. 6, our switch model consists of five modules: switch port, ingress pipeline, buffer management module, egress pipeline and PFC module. The function of these modules are list as follows.

- **Switch port**: The module to receive and send packets.
- **Ingress pipeline**: The module to execute packet processing tasks before a packet enters switch buffer. These tasks include packet header parsing, L2/L3 lookup, ingress ACL processing, packet tunneling, packet mirroring, etc. This module has multiple match-action stages to fulfill its tasks. Our (tag, port)-to-priority mapping function is implemented as one match-action stage here.
- **Buffer management module**: The module to manage the switch buffer. This module has m ingress queues and m egress queues. Assuming the switch has n ports and supports k priority classes. Each port has k corresponding ingress queues and k corresponding egress queues to buffer packets of k priority classes. It is easy to know m = n*k. In the module, there is also a scheduler to forward packets to egress queues based on the packet metadata calculated at ingress pipeline.

  This module maintains separate counters to track the length of each ingress queue and each egress queue (denoted as ingress counter and egress counter). The ingress counter increases when a packet enters the ingress queue, and decreases when a packet leaves the switch buffer. Egress counter works similarly. Note that when a packet is forwarded to an egress queue from some ingress queue, the corresponding ingress counter will not decrease at this stage as this packet still remains in the switch buffer.
- **Egress pipeline**: The module to execute packet processing tasks after a packet leaves switch buffer. This module also has multiple match-action stages to do tasks such as egress ACL processing, packet tunneling and packet mirroring. Our (tag, port)-to-newtag mapping function is implemented as one match-action stage here.
- **PFC engine**: The module to perform PFC function. If the switch receives a PFC PAUSE frame at some output port on some specific priority class, this module will pause the corresponding egress queue immediately. If the occupancy of some ingress queue exceeds the PFC threshold, this module will trigger the corresponding input port to send a PFC PAUSE frame to pause the upstream device on the corresponding priority.

6.2 ACL Rules for Tag Handling

To ensure packets enter the correct lossless space at each hop, our solution requires the switch to classify packets into

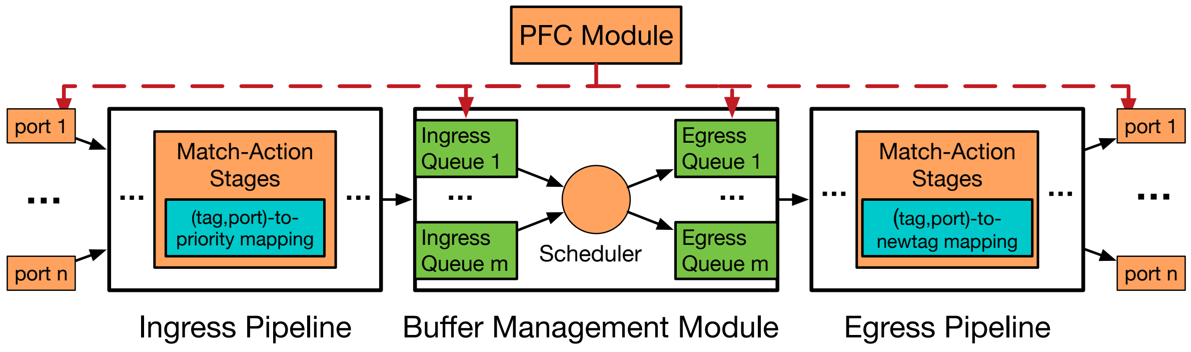
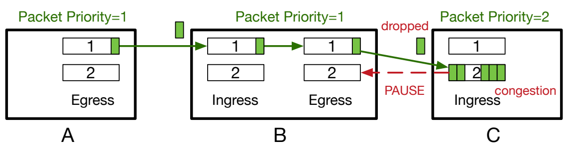
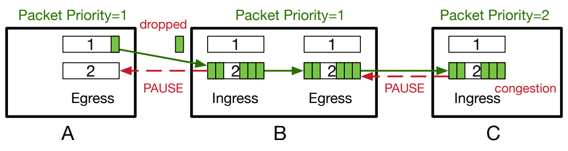


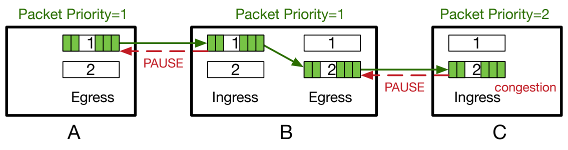
Figure 6: Switch model.



(a) Ingress priority = 1, egress priority = 1 → packet drop.



(b) Ingress priority = 2, egress priority = 2 → packet drop.



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

Figure 7: Decoupling ingress priority from egress priority is necessary for ensuring a lossless network.

different priority classes based on (port, tag) information. This can be done by installing some ACL rules at the ingress pipeline. As shown in Fig. 6, these rules will match the port # and value of tag, and assign packets to different priority calsses accordingly.

To optimize the number of priorities needed for Clos and BCube topologies, we need to support flexible manipulation of tag value. This can be done by installing some ACL rules at the egress pipeline. As shown in Fig. 6, these rules will calculate a new tag value for any outgoing packet based on the port # and old tag value, so the vaue of tag can be updated at each hop.

6.3 Transition of Lossless Queue

Current PFC mechanism cannot prevent packet loss in some cases when packets change their priority classes along the path. An example can be found in Fig. 7. More description about this example will be added later.

As shown in Fig. 7(c), if we want a switch to pause its upstream device correctly, for packets of the same flow, the egress queue at last hop and the ingress queue at current hop must be of the same priority class. This means that we need to decouple ingress priority from egress priority in our implementation.

Then how to do this? We notice that Broadcom chipset based commodity switches, such as Arista 7060 switch, have alternative variables that can be used to decide the priority class of a packet inside the buffer management module. In normal case, only 1 variable will be used, and ingress priority is always the same as egress priority.

We configure the switch to use different variables, denoted as ING_PRIO and EG_PRIO, to decide ingress priority and egress priority separately. To set correct value for both NG_PRIO and EG_PRIO, we install some ACL rules at the ingress pipeline to associate ING_PRIO and EG_PRIO with (port, tag) information. These rules establish separate mappings from (port, tag) to ingress priority and egress priority, respectively.

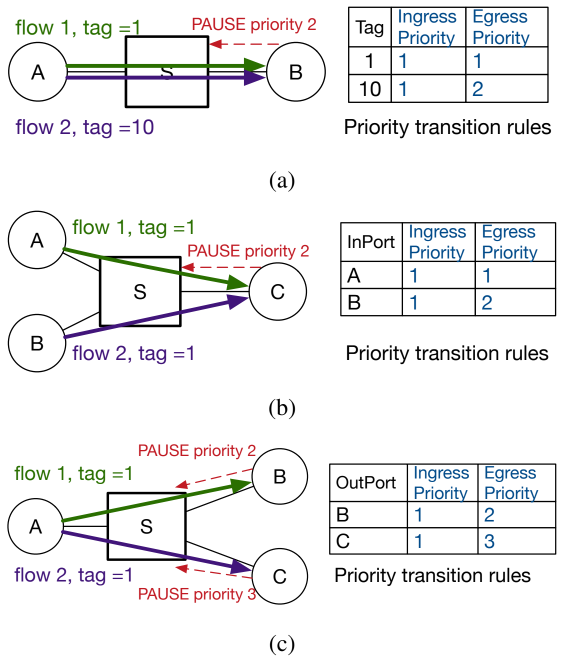
7. Evaluation

We evaluate Tagger using a combination of testbed experiments and numerical experiments. Our evaluation centers around four key questions:

- **Can Tagger prevent deadlock when deadlock-prone misconfiguration or network failure happens?**
- **Is Tagger scalable for large scale datacenters?**
- **Can Tagger work at full line rate with little performance overhead?**
- **Does the mechanism of priority reusing cause any inefficiency or unfairness problem?**

7.1 Lossless transition between priority classes

Purpose: demonstrate our implementation ensures lossless transition between priority classes.



Tag	Ingress Priority	Egress Priority
1	1	1
10	1	2

InPort	Ingress Priority	Egress Priority
A	1	1
B	1	2

OutPort	Ingress Priority	Egress Priority
B	1	2
C	1	3

Figure 8: Experiment scenarios to demonstrate lossless transition between priority classes.

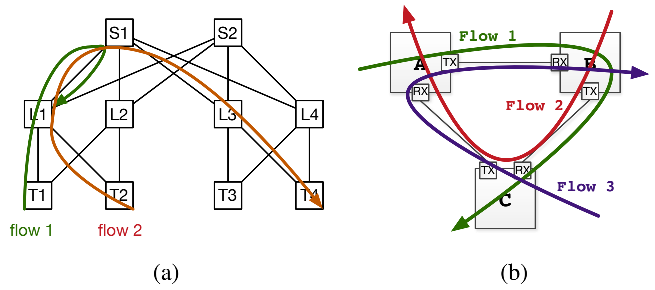


Figure 9: Scenarios of validation experiment.

Scenario-1: As shown in Fig. 8(a), by pausing priority 2 at server B, flow 2 is paused correctly without any packet loss, while flow 1 is not affected.

Scenario-2: As shown in Fig. 8(b), by pausing priority 2 at server C, flow 2 is paused correctly without any packet loss, while flow 1 is not affected.

Scenario-3: The scenario is shown in Fig. 8(c). By pausing priority 2 at server B, flow 1 is paused correctly without any packet loss. By pausing priority 3 at server C, flow 2 is paused correctly without any packet loss.

7.2 Validation

Purpose: demonstrate our solution can prevent deadlock.

Scenario-1: As shown in Fig. 9(a), In a Clos network, we generate 2 flows across different ToRs. Flow 1 enters a routing loop due to misconfiguration, and forms a deadlock. Without Tagger, flow 2 will also get paused due to propagation of PFC PAUSE. With Tagger, there is no deadlock and flow 2 is not affected by the routing loop.

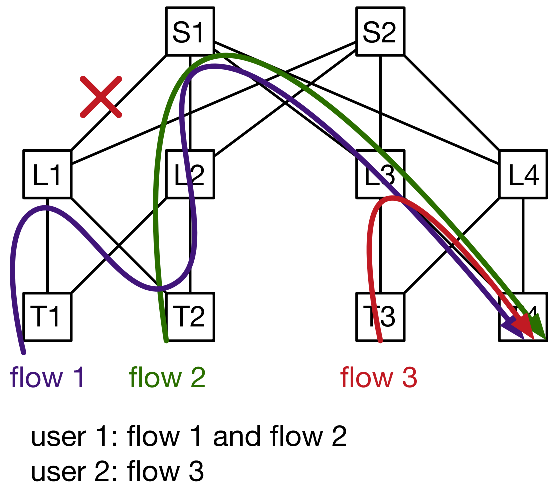


Figure 10: Scenario of priority reusing experiment.

Scenario-2: As shown in Fig. 9(b), we inject 3 flows into three paths that contain a CBD. Without Tagger, deadlock occurs and three flows are all paused. With Tagger, there is no deadlock and three flows achieve good throughput.

7.3 Overhead

7.3.1 Resource Consumption

Purpose: demonstrate our solution only consumes a small number of priorities and ACL entries.

Scenario: for Fattree, BCube and Jellyfish, we calculate the following metrics under different network scales to show Tagger is scalable in terms of resource consumption.

1. # of lossless priorities;
2. # of total ACL rules;
3. # of ACL rules at the bottleneck switch.

7.3.2 Performance overhead

Purpose: demonstrate performance overhead of Tagger is small.

Scenario-1: We let a flow traverse m hops in the network, and perform tagging and priority manipulation at every hop. We can evaluate the performance overhead of Tagger by measuring the throughput and latency when using different values of m.

Scenario-2: we increase the # of ACL rules installed in a switch, and observe the performance overhead of the flows passing over this switch.

7.4 Impact of priority reusing

Purpose: demonstrate the mechanism of priority reusing will not downgrade link utilization and throughput of flows (but may introduce unfairness problem).

Scenario: As shown in Fig. 10(a), a Clos network serves two users belonging to different application classes. For user 1, priority 1 is used for traffic following shortest paths, and priority 2 is used for traffic following 1-bounce paths. For user 2, priority 2 is used for traffic following shortest paths, and priority 1 is used for traffic following 1-bounce paths.

Assuming WRR policy among different egress queues. Assuming traffic of these two users shares a bottleneck link. We create a link failure to let flow 1 of user 1 enter the second lossless space. Flow 1 then competes link bandwidth in the same egress queue with flow 3 of user 2 at the bottleneck link L3-T4.

By measuring the link utilization and throughput of all flows, we can demonstrate that link utilization of the bottleneck link and overall throughput of all flows is not downgraded, but priority reusing does cause some unfairness problem among different suers.

compared scheme: no priority reusing among users (4 priorities are used)

7.5 Hierarchical lossless space improves application performance

This experiment will be moved to Section 3 as an motivation experiment.

Purpose: demonstrate that for Clos network, applications can achieve better performance when both shortest paths and 1 bounce paths are lossless.

Scenario: In a Clos network, we run both throughput-intensive and latency-sensitive applications. We randomly generate k failures in the network to have some of the flows follow 1-bounce paths.

compared schemes:

1. both shortest paths and 1 bounce paths are lossless.
2. shortest paths are lossless. 1 bounce paths are lossy.
3. Only shortest paths are allowed.

8. Related Work

RoCE and the need for PFC. Here we emphasize that (at least current) RoCE cannot work well without PFC. The results in DCQCN and RDMA at scale (SIGCOMM'16) paper show packet drops are very harmful for RoCE performance.

We may add a note saying that PFC has its fundamental advantage (e.g., no retransmission), if we could make it work well in practice, using PFC would be the right approach for RoCE. Our paper is trying to make this happen.

Circuit switching-based approaches. Those solutions from HPC and InfiniBand work by preemption. This does not work in Ethernet and in practice.

Brute-force buffer layering solutions. They use a lot of lossless queues. No one has discussed how they can be implemented in reality.

Summary: our differences. We believe that deciding the priority of packets along the path is better than changing routing configuratoins.

9. Conclusion

A. PFC headroom calculation

The PFC headroom needed per port per lossless queue can be calculated by considering the time interval needed for a receiver to pause its upstream sender. The time interval is composed of the following 6 periods for the lossless class p:

The time to send a PAUSE frame t_1. Once a pause frame is generated, it may be blocked by a packet ahead of it which just starts transmitting. Since Ethernet is non-preemptive, the pause fame has to wait for the completion of the previous packet. Hence in the worst-cast, $t_1 = \frac{L_{mtu}+L_{pfc}}{B}$, where L_{mtu} is the MTU size, and L_{pfc} is the size of a PFC pause frame, and B is the link rate.

The PAUSE frame propagation time t_2. This value is decided by the cable length between the sender and receiver.

The PAUSE frame receiving time at the sender $t3$. $t_3 = \frac{L_{pfc}}{B}$.

The PFC response time t_4. This is the amount of time needed for the sender to respond after the pause frame is received.

The time for the sender to stop transmitting t_5. Again, because Ethernet is non-preemptive, when the sender decides to stop transmitting, it needs to finish the packet already gets started. In the worst-case, the packet size can be L^p_{mtu}, which is the maximum packet size for that lossless class p. Hence $t_5 = \frac{L^p_{mtu}}{B}$.

The time for the pipe to be drained t_6. We know $t_6 = t_2$.

At a first glance, the headroom size should be $B \times \sum t_i$. But there are some additional details. The switching ASICs typically divide a packet into small cells of equal size for internal packet storage and processing. The cell size (C) is typically larger than the smallest Ethernet packet size (64 bytes). For one 64-byte packet, one cell is allocated. So in the worst-case, the needed headroom size is:

$$S_{hdr} = C\lceil\frac{(t_1+t_2+t_3+t_4+t_6)B}{64}\rceil + C\lceil\frac{t_5B}{64}\rceil$$

For a typical 40GbE RoCEv2 setup, we have $L_{mtu} = 1500$ bytes, $L_{pfc} = 64$ bytes, $t_2 = t_6 = 1us$ (for about 200 meters cable length), $t_4 = 2.75us$, $L^p_{mtu} = 1100$ bytes, $C = 208$ bytes. For a commodity switch with 32 full duplex 40GbE ports, the total headroom size needed is 2.76MB for supporting one lossless queue.

10. References

[1] Rdma over converged ethernet (roce). `http://www.mellanox.com/page/products_dyn?product_family=79`.

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

[3] Alexey Andreyev. Introducing data center fabric, the next-generation Facebook data center network.

[4] Cisco. Priority Flow Control: Build Reliable Layer 2 Infrastructure.

[5] 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.

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

[7] 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.

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

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

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

[11] 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*.

[12] 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.

[13] 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*.

[14] 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.

[15] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. *ACM SIGCOMM Computer Communication Review*, 45(4):183–197, 2015.

[16] Tor Skeie, Olav Lysne, and Ingebjørg Theiss. Layered shortest path (lash) routing in irregular system area networks. In *IPDPS '02*.

[17] 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.

[18] 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.

[19] 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*.

[20] 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*.