Masking Failures from Application Performance in Data Center Networks with Shareable Backup

Dingming Wu*
Rice University
dw40@rice.edu

Yiting Xia*†
Facebook, Inc.
xia.yiting@gmail.com

Xiaoye Steven Sun
Rice University
xs6@rice.edu

Xin Sunny Huang
Rice University
xinh@rice.edu

Simbarashe Dzinamarira
Rice University
dzinamarira@rice.edu

T. S. Eugene Ng
Rice University
eugeneng@cs.rice.edu

ABSTRACT

Shareable backup is an economical and effective way to mask failures from application performance. A small number of backup switches are shared network-wide for repairing failures on demand so that the network quickly recovers to its full capacity without applications noticing the failures. This approach avoids complications and ineffectiveness of rerouting. We propose ShareBackup as a prototype architecture to realize this concept and present the detailed design. We implement ShareBackup on a hardware testbed. Its failure recovery takes merely 0.73ms, causing no disruption to routing; and it accelerates Spark and Tez jobs by up to 4.1× under failures. Large-scale simulations with real data center traffic and failure model show that ShareBackup reduces the percentage of job flows prolonged by failures from 47.2% to as little as 0.78%. In all our experiments, the results for ShareBackup have little difference from the no-failure case.

CCS CONCEPTS

• **Networks** → **Physical topologies**; **Network reliability**; **Data center networks**;

KEYWORDS

Data Center Network; Failure Recovery; Circuit Switching

*The first two authors contributed equally to the paper.
†Work done while at Rice University.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGCOMM '18, August 20–25, 2018, Budapest, Hungary

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-5567-4/18/08…$15.00
https://doi.org/10.1145/3230543.3230577

ACM Reference Format:
Dingming Wu, Yiting Xia, Xiaoye Steven Sun, Xin Sunny Huang, Simbarashe Dzinamarira, and T. S. Eugene Ng. 2018. Masking Failures from Application Performance in Data Center Networks with Shareable Backup. In *SIGCOMM '18: ACM SIGCOMM 2018 Conference, August 20–25, 2018, Budapest, Hungary.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3230543.3230577

1 INTRODUCTION

The ultimate goal of failure recovery in data center networks is to preserve application performance. In this paper, we propose *shareable backup* as a ground-breaking solution towards that goal. Shareable backup allows the entire data center to share a pool of backup switches. If any switch in the network fails, a backup switch can be brought online to replace it. The failover should be fast enough to avoid disruption to applications. With the power of shareable backup, it is possible for the first time to repair failures instantly instead of making do with a crippled network.

Shareable backup is a natural quest due to ineptness of rerouting, the mainstream solution to fault tolerance in data center networks [7, 8, 17–19, 26, 30, 38, 43]. While rerouting maintains connectivity, bandwidth is nonetheless degraded under failures. The rerouted traffic may contend with other traffic originally on the path, thus enlarging the effect of failure to a wider range of the network. Routing convergence is known to be slow [12], and even path re-computation on a centralized management entity is expensive [30, 38]. This latency is especially harmful to interactive applications with rigid deadlines. Rerouting also risks misconfigurations when updating routing tables, which may cause the network to dysfunction. Not to mention other overheads, e.g. slow failure propagation, longer alternative paths, excessive state exchange, etc.

With all these factors, the application performance may be jeopardized drastically. According to a failure study of a path-rich production data center, 10% less traffic is delivered for the median case of the analyzed failures, and 40% less for the worst 20% of failures [16]. Injecting these failures into

our simulation of a real data center setting (Section 6.7), 42% jobs get slowed down by at least 3× (Figure 9(b)), 51% jobs miss deadlines (Figure 10(b)), and 21.3% flows not on the path of failure still get affected because of rerouting (Table 4).

Shareable backup is desirable for its cost-effectiveness. The pool of backup switches needs not be large in practice, because failures in data centers are rare and transient. The above failure study shows most devices have over 99.99% availability; and failures usually last for only a few minutes [16]. With shareable backup, we for the first time achieve network-wide backup at low cost, which is impossible for the traditional 1:1 backup that requires a dedicated spare for each switch.

Shareable backup is achievable by circuit switches, which have been used to facilitate physical-layer topology adaptation in many novel network architectures [13–15, 23, 28, 32, 44, 50]. Theoretically, if the pool of backup switches and all the switches in the network are connected to a circuit switch, any switch can then be replaced as we change the circuit switch connections. However, a circuit switch has limited port count, and layering multiple circuit switches to scale up increases insertion loss. Rather than scaling up, recent proposals scale out low-cost modest-size circuit switches by distributed placement of them across the network [23, 50]. We adopt this approach to partition the network into smaller failure groups and realize shareable backup in each group.

In this work, we design a prototype architecture, named *ShareBackup*, to explore the feasibility of shareable backup on fat-tree [8], a typical network topology found in data centers [5, 36]. We have implemented ShareBackup and its competing solutions on a hardware testbed, a Linear Programming simulator, and a packet-level simulator. We have conducted extensive evaluations including TCP convergence, control system latency, bandwidth capacity, transmission performance at scale with real traffic and failure model, and benefits to Spark and Tez jobs on the testbed.

The key properties of ShareBackup are: (1) failure recovery only takes 0.73ms, latencies from hardware and control system combined; (2) it restores bandwidth to full capacity after failures, and routing is not disturbed; (3) for all our experiments, its performance difference with the no-failure case is negligible, proving its ability to mask failures from application performance; (4) under failures, it accelerates Spark and Tez jobs by upto 4.1× and reduces the percentage of job flows slowed down by failures in the large-scale simulation from 47.2% to 0.78%.

2 RELATED WORK

Data center network architectures rely on rich redundant paths for failure resilience [7, 8, 17–19, 38]. Among them, fat-tree is the most popular in practical use [8]. ShareBackup builds on top of fat-tree, so it is related to other proposals enhancing fault-tolerance of fat-tree networks. PortLand reroutes traffic to globally optimal paths based on a central view of the network at the fabric manager. F10 reduces delays from failure propagation and path re-computation by local rerouting at switches [26], at the cost of longer paths. It also adjusts wiring of fat-tree to form AB fat-tree, which provides diverse paths for local rerouting. Aspen Tree adds different degrees of redundancy to fat-tree to tune the local rerouting path length [43]. It either partitions the network or adds extra switches to have more paths. If keeping the host count, it requires at least one more layer, or 40% more switches. ShareBackup takes a completely different approach. Instead of rerouting, it deploys backup switches in the physical layer. We compare with PortLand, F10, and Aspen Tree in the evaluations to explore interesting properties of ShareBackup.

Besides architectural solutions, many works have been tackling failures in data centers from different angles. NetPilot and CorrOpt give operational guidance to manually mitigating the effect of failures [47, 52]. ShareBackup instead automatically replaces failed switches to restore full capacity of the network. Its recovery speed is also significantly faster, e.g. sub-ms vs. tens of minutes. Subways and Hot Standby Router Protocol suggest multi-homing hosts to several switches to avoid the single point of failure [27, 41], which consumes more ports on the hosts and switches. ShareBackup provides more efficient redundancy at the network edge without multi-homing, and we invent a more light-weight VLAN-based solution to make backup switches hot standbys with no additional latency (Section 4.4). In the context of rerouting, there is a large body of work on local fast failover [11, 22, 25, 29, 31, 33, 39, 51], some of which cause the explosion of backup routes and Plinko introduces a forwarding table compression algorithm accordingly [40]. ShareBackup does not depend on rerouting for failure recovery, so it avoids these complications and the forwarding tables are intrinsically small. On the application level, Bodik *et al.* propose intelligent service placement for both fault tolerance and traffic locality [10], and computation frameworks such as Spark [1] and Tez [2] restart tasks elsewhere when workers are lost. ShareBackup provides a more reliable network, so service placement has less constraints. Our experiment in Section 6.8 shows application-level resilience is insufficient: the performance is degraded by multi-folds if hosts are disconnected. Thus, in-network failure recovery is extremely important.

3 NETWORK ARCHITECTURE

ShareBackup has stringent requirements on cost and failure recovery delay, which guide our choice of circuit switch technologies. No existing circuit switch has enough ports to connect to all switches in the data center plus the pool of backup switches. Cascading multiple circuit switches wastes many

Algorithm 1 ShareBackup wiring algorithm

// Edge layer

for each $CS_{1,i,j}$ where $0 \le i < k$, $0 \le j < \frac{k}{2}$ **do**
  $Edge_{i,j} \in FG_{1,i}$
  for each p in $0 \le p < \frac{k}{2}$ **do**
    $\text{DOWN}_p \longleftrightarrow Host_{\frac{k}{2}\times p+j+i\times(\frac{k}{2})^2}$
    $\text{UP}_p \longleftrightarrow Edge_{i,p}$
    $\text{DOWN}_p \dashleftarrow\dashrightarrow \text{UP}_p$
  for each p in $\frac{k}{2} \le p < \frac{k}{2} + n$ **do**
    $\text{UP}_p \longleftrightarrow BS_{1,i,p-\frac{k}{2}}$

// Aggregation layer

for each $CS_{2,i,j}$ where $0 \le i < k$, $0 \le j < \frac{k}{2}$ **do**
  $Agg_{i,j} \in FG_{2,i}$
  for each p in $0 \le p < \frac{k}{2}$ **do**
    $\text{DOWN}_p \longleftrightarrow Edge_{i,p}$
    $\text{UP}_p \longleftrightarrow Agg_{i,p}$
    $\text{DOWN}_p \dashleftarrow\dashrightarrow \text{UP}_{(p+j)\%\frac{k}{2}}$
  for each p in $\frac{k}{2} \le p < \frac{k}{2} + n$ **do**
    $\text{DOWN}_p \longleftrightarrow BS_{1,i,p-\frac{k}{2}}$
    $\text{UP}_p \longleftrightarrow BS_{2,i,p-\frac{k}{2}}$

// Core layer

for each $CS_{3,i,j}$ where $0 \le i < k$, $0 \le j < \frac{k}{2}$ **do**
  $Core_{\frac{i}{2}+\frac{k}{2}\times j} \in FG_{3,\frac{i}{2}}$
  for each p in $0 \le p < \frac{k}{2}$ **do**
    $\text{DOWN}_p \longleftrightarrow Agg_{i,p}$
    $\text{UP}_p \longleftrightarrow Core_{\frac{k}{2}\times p+j}$
    $\text{DOWN}_p \dashleftarrow\dashrightarrow \text{UP}_p$
  for each p in $\frac{k}{2} \le p < \frac{k}{2} + n$ **do**
    $\text{DOWN}_p \longleftrightarrow BS_{2,i,p-\frac{k}{2}}$
    $\text{UP}_p \longleftrightarrow BS_{3,j,p-\frac{k}{2}}$

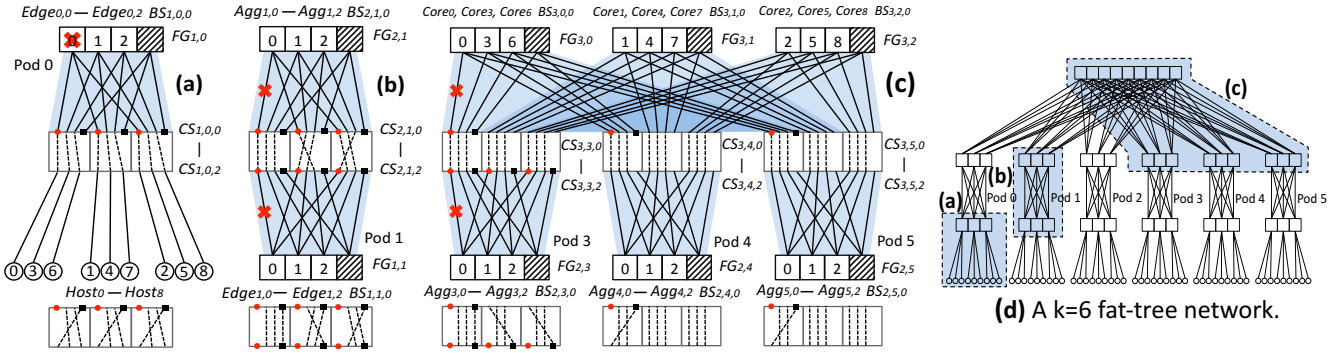

Figure 1: A $k = 6$ and $n = 1$ ShareBackup network. (a), (b), (c) correspond to shaded areas in (d). Devices are labeled according to the notations in Table 1. Edge and aggregation switches are marked by their in-Pod indices; core switches and hosts are marked by their global indices. Switches in the same failure group are packed together, which share a backup switch in stripes on the side. Circuit switches are inserted into adjacent layers of switches/hosts. The connectivity in shade is the basic building block for shareable backup. The crossed switch and connections represent example switch and link failures. Switches with failures are each replaced by a backup switch with the new circuit switch configurations shown at the bottom, where connections regarding the original red round ports reconnect to the new black square ports.

Table 1: List of notations

Notation	Meaning
k	Fat-tree parameter: switch port count and # Pods [8]
n	# backup switches shared by $\frac{k}{2}$ switches per failure group
$Host_j$	The jth host
$Edge_{i,j}$	The jth Edge switch in the ith Pod
$Agg_{i,j}$	The jth Aggregation switch in the ith Pod
$Core_j$	The jth Core switch
$CS_{l,i,j}$	The jth Circuit Switch in the ith Pod on the lth layer
$FG_{l,u}$	The uth Failure Group on the lth layer
$BS_{l,u,v}$	The vth Backup Switch in $FG_{l,u}$
UP_p	The pth UPward facing port of a circuit switch
$DOWN_p$	The pth DOWNward facing port of a circuit switch

intermediate ports, increases insertion loss thus requiring more powerful (and expensive) transceivers, and causes large end-to-end switching delay that slows down failure recovery. Instead, recent works promote partial configurability in small network regions using circuit switches with considerably low per-port cost and switching delay [23, 50]. For instance, a commercial 160-port 10Gbps electrical crosspoint switch costs $3 per port and has 70ns switching delay [23]; 32-port 25Gbps optical 2D-MEMS has been developed, with 40μs switching delay at an estimated cost of $10 per port [34, 46]. These technologies meet our demand.

These targeted circuit switches have modest port count, so we divide the network into smaller failure groups and deploy them in each group. Measurement studies show that failures in data centers are rare, uncorrelated, and spatially dispersed [16, 52]. ShareBackup's distributed design is a good match for these characteristics and can provide good coverage. Fat-tree has $\frac{k}{2}$ edge and aggregation switches per Pod. To align with the architecture, we cluster $\frac{k}{2}$ switches into a failure group and allow them to share n backup switches. All switches in a failure group, including the backup switches, must connect to the same circuit switch with the same wiring pattern. In this way, a backup switch can be brought online at run time to replace any failed switch or failed links associated with it within its failure group. This circuit switch should have at least $\frac{k}{2} \times (\frac{k}{2} + n)$ ports, which may exceed the port count of the targeted circuit switches for a large data center. We combine $\frac{k}{2}$ individual circuit switches side by side and design a wiring pattern to achieve equivalent functionality. Figure 1

gives intuitions of the architecture design. Algorithm 1 shows the wiring plan, with notations listed in Table 1.

Figure 1(a) illustrates the basic building block for ShareBackup with $n = 1$. The edge switches in the same Pod form a failure group (*line 2* in Algorithm 1). We place $\frac{k}{2}$ units of $(\frac{k}{2}+n)$ by $(\frac{k}{2}+n)$ circuit switches between the edge switches and the hosts. Every switch, regular and backup switch alike, connects to these $\frac{k}{2}$ circuit switches each with a link (*line 5 and 8*). As shown in Figure 3, these $\frac{k}{2}$ switches are chained together via 2 extra side ports, which is omitted in Figure 1 for simplicity. Hosts connect to the edge switches via straight-through connections on the intermediate circuit switches (*line 4 and 6*). The ports to backup switches are not connected internally. When a switch is down, the internal connections to it on all the circuit switches are reconfigured to connect to a backup switch, which replaces the failed switch completely. A switch whose links are down is replaced in the same manner so as to fix the link failures.

In Figure 1(b), the aggregation switches in the same Pod form a failure group (*line 10*). Edge and aggregation switches in their failure groups repeat the building block of connectivity in Figure 1(a) to another set of $\frac{k}{2}$ circuit switches (*line 12, 13, 16, and 17*). In a fat-tree Pod, an edge/aggregation switch connects to each and every aggregation/edge switch, so we use a rotational wiring pattern in the circuit switches (*line 14*) to achieve this shuffle connectivity, i.e. the different internal connections on $CS_{2,1,0}$ to $CS_{2,1,2}$.

Similarly, aggregation switches in each failure group shown in Figure 1(c) are connected upward to $\frac{k}{2}$ circuit switches with the wiring pattern in the building block. As the fat-tree example shows, the connections from aggregation switches in each Pod iterate through all the core switches in consecutive order. Because the aggregation switches are already connected to the circuit switches, we wire up the core switches and the circuit switches to achieve the fat-tree connectivity. In the Figure 1(c) example, core switches $Core_0$, $Core_1$, $Core_2$ connect to the first aggregation switch in each Pod ($Agg_{3,0}$, $Agg_{4,0}$, $Agg_{5,0}$) through different circuit switches in the Pod; $Core_3$, $Core_4$, $Core_5$ to the second aggregation switch in each Pod ($Agg_{3,1}$, $Agg_{4,1}$, $Agg_{5,1}$); and $Core_6$, $Core_7$, $Core_8$ to the third ($Agg_{3,2}$, $Agg_{4,2}$, $Agg_{5,2}$). As a summary of this pattern, the core switches connect to $\frac{k}{2}$ circuit switches with a stride of $\frac{k}{2}$, and we set up straight-through connections in the circuit switches. Similar to the building block for shareable backup in Figure 1(a), only switches connected to the same set of circuit switches can be put into a failure group. As a result, core switches whose indices are in $\frac{k}{2}$ intervals form a failure group. We give each failure group n backup switches and connect them up in the same way as regular switches.

In fat-tree, edge and aggregation switches are packaged into Pods for ease of deployment. In each ShareBackup Pod, there are n additional edge and aggregation switches respectively as backup switches, and 3 sets of $\frac{k}{2}$ circuit switches between adjacent layers of switches and hosts. It is straightforward to package the backup switches and the circuit switches into the original fat-tree Pods with simple changes of wiring as shown in Figure 1. Core switches and the backup core switches each connect to every Pod with one link. In practice, the core switches can be placed as in the original fat-tree, followed by the backup core switches. The reordering in Figure 1(c) is unnecessary. By streamlining the connectors from within each Pod, we can maintain the original Pod-host and Pod-core wiring patterns in fat-tree.

4 CONTROL PLANE

4.1 Fast Failure Detection and Recovery

Most previous fault-tolerant data center network architectures, such as PortLand [30] and F10 [26], mainly focus on link failures. According to a measurement study, however, switch failures account for 11.3% of the failure events in data centers and their impact is significantly more severe than link failures [16]. Therefore, ShareBackup aims to detect and recover both link and switch failures rapidly.

Failure detection and recovery are handled by a management entity, e.g. one or more dedicated network controllers. For switch failures, we require switches to send keep-alive messages continuously to the management entity. After missing keep-alive messages from a switch for a pre-defined time period, the management entity allocates an available backup switch to failover to and reconfigures the circuit switches associated with the failure group. As shown in Figure 1(a), in these circuit switches, original connections to the failed switch should reconnect to the backup switch.

We adopt the rapid failure detection mechanism in F10 [26] for link failures, where switches keep sending packets to each other (or to hosts) to test the interface, data link, and forwarding engine. When a link is down, it takes time to determine which end has lost connectivity. For the purpose of fast failure recovery, the switches on both sides of the failed link are replaced. The cause of failure is analyzed later by the procedure in Section 4.3. The management entity gets notifications of link failures from switches and hosts, and reconfigures the circuit switches in the same way as it addresses switch failures on both ends. Figure 1(b) and 1(c) show examples of this approach.

4.2 Distributed Network Controllers

ShareBackup requires the management entity to be implemented as distributed network controllers. A single controller is not capable of collecting frequent heartbeats from all the switches in the network. Like F10, the probing interval for failure detection can be as low as a few ms. For example, in

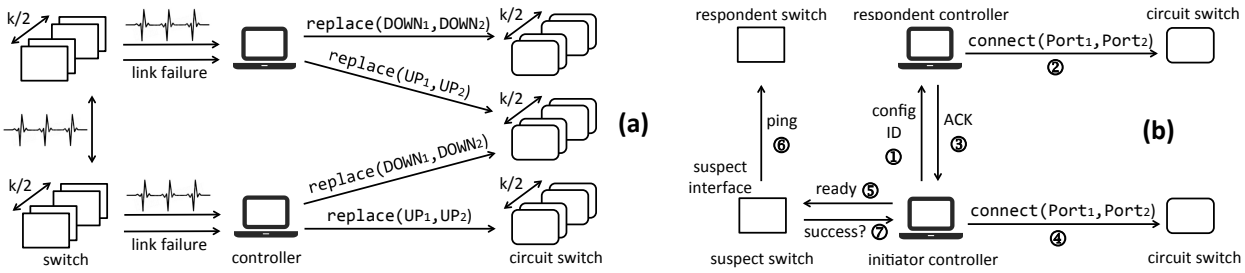

Figure 2: Communication protocol in the control system. (a): Failure detection and recovery. (b): Diagnosis of link failure.

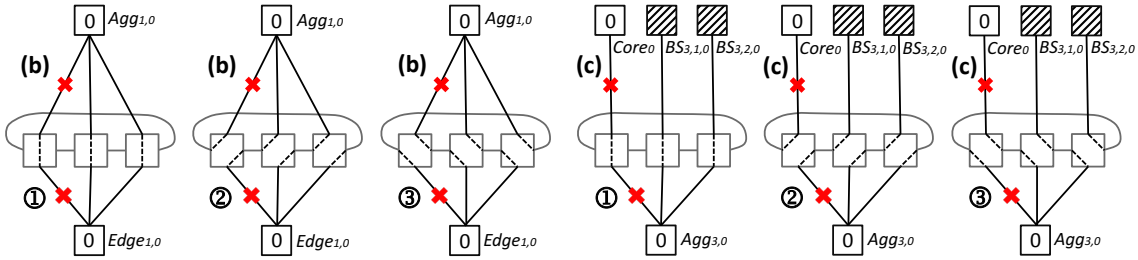

Figure 3: Circuit switch configurations for diagnosis of link failures shown by examples (b) and (c) in Figure 1. Circuit switches in a Pod are chained up using the side ports. Only "suspect switches" on both sides of the failed link and some related backup switches are shown. Through configurations ①, ②, and ③, the "suspect interface" on both "suspect switches" associated with the failure can connect to 3 different interfaces on one or multiple other switches.

a $k = 48$ fat-tree with 2880 switches, a heartbeat message every 5ms leads to 576k queries per second, which exceeds the capacity of a single controller. Distributed controllers also isolate the impact of controller failures to a small portion of the network. Controllers only store local state, so adding redundant controllers can further enhance resiliency with low state-exchange overhead. Finally, with distributed placement of network controllers, switches and circuit switches can be physically close to their controller, which effectively reduces the message latency of failure detection and recovery.

Figure 2(a) shows the failure detection and recovery process using distributed network controllers. Based on the layout of the ShareBackup architecture, an intuitive way of controller placement is to assign each failure group a dedicated controller. Each controller receives heartbeat messages from $\frac{k}{2}$ switches only. As shown in Figure 1, a failure group in the core layer corresponds to $\frac{k}{2}$ circuit switches beneath it, while one in the edge and aggregation layers corresponds to two sets of $\frac{k}{2}$ circuit switches above and beneath it, so each controller reconfigures k circuit switches at maximum. The load for each controller is thus very light even for a large network. The controllers do not share state, so the communication among them is also minimal. Due to the simple functionality, controllers can be placed on existing servers with a separate management network, or on dedicated low-cost hardware, such as the Arduino [3] and Raspberry Pi [6] platforms. Multiple controllers can also reside on the same machine to realize different degrees of distribution/centralization.

Most circuit switches nowadays use the TL1 software interface to setup a connection, whose input and output ports should be specified explicitly, i.e. *connect(in_port,out_port)*. The network controller needs to maintain the current connections of the circuit switches so as to switch to new connections. In Figure 1(b) and 1(c), a circuit switch connects to switches from the failure groups above and below, so it can be reconfigured by both controllers. Inconsistent reconfiguration from different controllers may result in wrong connections. To address this issue, we change the API to *replace(old_port,new_port)* and free controllers from bookkeeping of the circuit configurations. Using this new API, a controller only reconfigures ports on one side of the circuit switch and is ignorant of changes on the other side. Before the reconfiguration, the old port (red ports in Figure 1) connects to the failed switch and the new port (black ports in Figure 1) connects to the backup switch. After the change, the old port's role will be replaced by the new port, i.e., the new port will connect to the old port's original peering port. In case of concurrent requests from different controllers, the circuit switch reconfigures one side at a time, so the order of execution does not affect the end result.

4.3 Offline Auto-Diagnosis

Most link failures are due to a failure that occurs on one end. After both switches are replaced, we run automatic failure diagnosis in the background to find which "suspect interface" (and the "suspect switch" it belongs to) has caused the problem. We chain up circuit switches in the same layer of a Pod

as a ring through the side ports. Figure 3 shows the circuit switch configurations, through which the suspect interface on either end of the failed link can connect to 3 different interfaces, either on the same switch (as $Agg_{1,0}$, $Edge_{1,0}$, and $Core_0$) or on different switches (as $Agg_{3,0}$).

The network controllers for the involved switches coordinate to change the circuit switch configurations and enforce the switches to exchange testing messages. A suspect interface that has connectivity in at least one configuration is redressed as healthy, so is the corresponding suspect switch. Because auto-diagnosis only involves suspect switches already taken offline and backup switches not in use, this process is completely independent of the functioning network.

Figure 2(b) illustrates the controller coordination process. The two suspect interfaces are tested one by one. Their corresponding controllers elect one to be the initiator and the other one as passive respondent. The initiator cycles through the configurations to test the suspect interface on its side, after which the initiator and the respondent reverse roles to test the other suspect interface. As shown in Figure 2(a), in our distributed control system, each controller is responsible for reconfiguring a small subset of circuit switches and is only allowed to control the ports on its own side. So, both controllers need to participate in the circuit setup. The respondent controller learns the target connections from the initiator controller via the configuration ID. Here, we use the original TL1 interface, i.e. *connect(input_port, output_port)*, to connect to the side ports. As an offline process, failure diagnosis can be preempted by failure recovery. It is paused if the involved backup switch, such as $BS_{3,1,0}$ and $BS_{3,2,0}$ in Figure 3(c), needs to be used when another failure happens. The initiator controller thus proceeds only after receiving confirmation that the respondent side is not being preempted at the moment. It will continue with the next configuration if reaching the ACK timeout. In the end, the tested interface pings the other end and terminates the diagnosis process if it has connectivity.

Failure diagnosis requires both sides have at least one healthy interface, so that both suspect interfaces can be tested. If this condition is not met, both suspect switches are considered faulty. Since all hosts are actively in use, the offline failure diagnosis is not supported between hosts and edge switches. We assume switches are at fault for link failures to hosts. If the problem is not fixed after replacing the switch, we mark the switch as healthy and trouble-shoot the host.

After a failed switch is repaired or a suspect switch is exonerated, it is unnecessary to switch back to the original connectivity. Backup switches and regular switches are equal in functionality, so we keep the backup switch online and turn the replaced switch into a backup switch for future use. This design saves the reconfiguration overhead and avoids disruptions in the network. The network controller keeps track of the current backup switches in their failure groups.

4.4 Live Impersonation of Failed Switch

Traffic is redirected to the backup switch in the physical layer after a failed switch is replaced. The backup switch needs to impersonate the failed switch by using the same routing table. Fat-tree uses Two-Level Routing, where each switch has a pre-defined routing table [8]. To avoid the additional delay of inserting forwarding rules into the backup switch, we aim to preload the routing table and make the backup switch a hot standby. Regular switches recovered from failures can work as backup switches, so every switch needs to store the routing tables of all the switches in the failure group. The challenge is to resolve the conflicts between different routing tables.

In fat-tree, all the core switches and all the aggregation switches in the same Pod have the same routing table. Therefore, in the aggregation and core layers of our network, switches in a failure group only keep a common routing table. For in-bound traffic, edge switches in a Pod, also a failure group, have the same set of $\frac{k}{2}$ forwarding entries that match on the suffix of the end host addresses. For out-bound traffic, each of these edge switches has $\frac{k}{2}$ different entries. We use VLANs for differentiation. We first edit the original fat-tree routing tables by assigning every edge switch in the Pod a unique VLAN ID and adding it to the out-bound routing table entries. The edited routing tables from all the edge switches are then combined together and stored in every switch in the failure group. A host knows which edge switch it should connect to, so it tags out-going packets with the VLAN ID of the edge switch. No matter what switches in the failure group are active, by matching the VLAN ID, packets can always refer to the correct routing table. This combined routing table from $\frac{k}{2}$ edge switches has $\frac{k}{2}$ in-bound entries and $\frac{k^2}{4}$ out-bound entries. This total number is within the TCAM capacity of commercial switches even for large-scale fat-tree networks. For instance, the table contains only 1056 entries for a $k = 64$ fat-tree with over 65k hosts.

5 DISCUSSION

5.1 ShareBackup Failure Handling

ShareBackup uses a separate control network for failure recovery, which raises the question as to how ShareBackup handles failures in the control network itself. Below, we discuss failures on different components of the control system.

Circuit switch failures. Circuit switches are highly reliable physical layer devices [35]. They have bare-minimum control software receiving infrequent reconfiguration requests. When the control software fails, a circuit switch becomes unresponsive but keeps the existing circuit configuration—the data plane is not impacted. However, because ShareBackup

can no longer reconfigure the circuit switch, the affected failure group of the network has to fall back to traditional rerouting when switch/link failures occur later. When a circuit switch hardware fails, such as port-downs, the connected switches will experience real link failures. ShareBackup will regard them as regular link failures and replace the affected switches. In the rare case that a circuit switch is completely down (e.g., power outage), the controller will receive a large number of link failure reports in a short period of time. It will stop failure recovery and request for human intervention. Since each switch is connected to $\frac{k}{2}$ circuit switches, each of them only loses $\frac{2}{k}$ capacity during the downtime. We note that hardware/power failures are relatively rare in practice—most of the failures are from the software layer such as switch software/firmware bugs or configuration errors [47]. Circuit switches are generally free from these type of failures.

Control channel failures. A controller in ShareBackup needs to communicate with both circuit switches and other controllers. Control channel failures in the first case are equivalent to circuit switch software failures and are thus treated by ShareBackup in the same way. For the second case, our offline auto-diagnosis requires the coordination of two controllers to reconfigure circuit switches simultaneously. If the communication between the two controllers fails, the initiator controller will stop auto-diagnosis after a timeout and call for human intervention. We note that offline auto-diagnosis is only performed on switches not in use and does not impact the production traffic. Finally, when a link failure happens, switches on both sides of the link need to be replaced, which is handled by two separate controllers. If one side of the replacement is not successful, the other side will continue experiencing connection error on the replaced switch. To handle this issue, a backup switch will fall back to rerouting if connection error is not resolved after replacement.

Controller failures. Our distributed controllers are intrinsically robust. Each controller only keeps a small number of runtime states of current circuit configurations, and it is straightforward to protect against controller failures by state replication on a shadow controller.

5.2 Cost Analysis

We make key design decisions in ShareBackup to reduce extra cost. Concurrent failures are rare in data centers [16], so the ideal case is to have a single backup switch shared by the entire network. However, as discussed at the beginning of Section 3, this requires cascaded circuit switches with high cost, insertion loss, and switching delay. As a compromise, we deploy low-cost circuit switches with short switching delay, e.g. electrical crosspoint switch or optical 2D-MEMS, in separate failure groups. Our targeted circuit switches have modest port count. As Figure 1 shows, we combine them to cover more switches and form larger failure groups. Our design achieves a reasonably low backup ratio at low circuit switch cost. The additional cabling cost is minimal, because the circuit switches, either electrical or optical, are passive and do not require active elements, e.g. optical transceiver or amplifier for copper. The extra cost introduced by ShareBackup on a $k = 48$ fat-tree network with 27648 servers is estimated to be 6.7% [48]. And this cost does not scale up as the network size increases—the larger the failure groups, the smaller the backup ratio and hence the lower the extra cost.

ShareBackup can be partially deployed at different network layers or pods, which helps further reduce cost. In today's data centers, a host connects to one ToR switch only. If ToR or host link failures happen, hosts are disconnected and we have to rely on application frameworks to restart the work elsewhere. Our testbed experiment in Figure 11 shows Spark and Tez jobs get delayed by upto 4.1× in such case. Therefore, ShareBackup is especially powerful at the edge layer. In a fat-tree network, there are $\frac{k^2}{4}$ parallel paths in the core layer, but only $\frac{k}{2}$ in the aggregation layer. ShareBackup is more helpful in the aggregation layer, as rerouting may cause greater congestion with fewer paths to balance the load. Partial deployment is straightforward in ShareBackup thanks to the separate failure groups. We give a complete solution in this paper, but network operators have the freedom to deploy backup switches in certain areas of the network according to application requirements and monetary budget.

5.3 Benefits to Network Management

When switches are routinely taken out for upgrade or maintenance, backup switches can neatly take their place to avoid downtime. Misconfigurations account for a large proportion of failures in data centers [16], and they are hard to reason and fix. ShareBackup can help mitigate the effect and diagnose the problem. The configurations of backup switches can be verified when they are idle. If a switch is misconfigured, it can failover to the backup switch whose configurations are guaranteed to be correct. Then complicated diagnosis can be executed offline. With judicious use of hardware, our offline diagnosis in Section 4.3 helps identify which interface has caused a link failure. In today's data centers, failure diagnosis and repair are mostly handled manually and take hours at least. Even pioneering work like NetPilot takes 20 minutes only to mitigate failures [47]. Our implementation demonstrates later in Section 6.5 that ShareBackup automatically repairs failures in sub-ms and diagnoses failures in sub-second, which is a breakthrough for data center management.

5.4 Alternatives in the Design Space

An interesting question is whether PortLand and F10 will outperform ShareBackup if allowing the same deployment

Table 2: Road map of experiment setups.

Experiment	Section	Workload	Failure model	Platform
TCP disruption	6.4	Single flow	Single	Testbed
Control plane latency	6.5	-	Single	Testbed
Practical bandwidth	6.6	iPerf flows	Rand layered	Testbed
Theoretical bandwidth	6.6	Synthetic traffic	Rand layered	LP simulator
FCT/CCT slowdown	6.7	Coflow trace	Real	Packet simulator
Job deadlines	6.7	Deadline trace	Real	Packet simulator
Throughput-intensive app	6.8	Word2Vec, Sort	Rand layered	Testbed
Latency-sensitive app	6.8	TPC-H	Rand layered	Testbed

cost, e.g., employing the same number of ShareBackup's backup switches as their production switches. However, tree networks are known to lack expandability. It is hard, if not impossible, to add only a small proportion of switches with the same port count to a fully populated tree network. Even if more switches could be added, they would lock up bandwidth to fixed locations. Failures at highly unpredictable locations might still cause bandwidth loss. In contrast, ShareBackup can move backup switches to wherever needed. Unstructured networks have been proposed for easier expansion [21, 38, 42], but the performance under failures is yet to be explored. Admittedly, these topologies have rich bandwidth and diverse paths, but the path length hugely varies, causing risk of path dilation. We can add switches to either provision bandwidth at the price of degraded performance under failures, or to provide guaranteed performance while keeping the backups idle most of the time. We choose the latter, and we believe shareable backup is an effective way to reduce the idle rate.

6 IMPLEMENTATION AND EVALUATION

A prior study has demonstrated the cost advantage of ShareBackup: using 1 backup switch per failure group, ShareBackup only costs 6.7% more than fat-tree; and it is still more cost-effective than other redundancy-featured architectures with 4 backup switches per failure group [48]. However, this work lacks implementation of the ShareBackup system and performance evaluation against alternative solutions.

In this paper, we conduct comprehensive evaluations about the ShareBackup performance using both testbed implementation and large-scale simulations. Table 2 is a road map showing the setup of each experiment. First, we explore key properties of the ShareBackup system on the testbed, including failure recovery delay, TCP behavior during the transient state, and overhead of the control system. Since the major advantage of ShareBackup against other fault-tolerant network architectures is to restore bandwidth after failures, we next compare their bandwidth capacity with both Linear Programming simulations and testbed experiments. Next, we evaluate ShareBackup's transmission performance using packet-level simulations with practical routing and transport protocols. To simulate real-world scenarios, we use traffic traces and a failure model from production data centers. Finally, we run

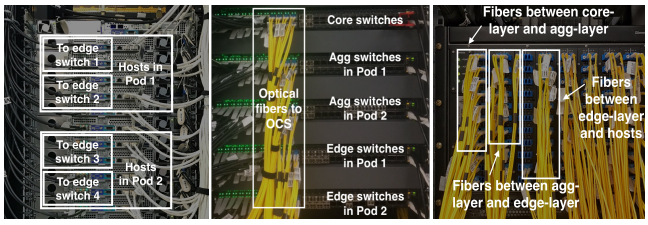

Figure 4: A testbed of $k = 4$ $n = 1$ ShareBackup with 2 Pods.

Spark and Tez jobs on our testbed to measure the performance improvement to real data center applications.

6.1 Testbed

Our prototype network is a $k = 4$ $n = 1$ ShareBackup with 2 Pods, that is 2 Pods of a $k = 4$ fat-tree where each failure group's 2 switches share 1 backup switch. Specifically, it is a non-blocking network with 12 active switches, 6 backup switches, and 8 hosts. Figure 4 shows the physical deployment of this logical network. All links are 10Gbps. The switches locate on 5 48-port OpenFlow packet switches: one partitioned into core switches and their backups, and the others each into the active and backup switches in the same layer of a Pod. The circuit switches are logical partitions of a 192-port 3D-MEMS optical circuit switch (OCS). The hosts are individual machines each with 6 3.5GHz dual-hyperthreaded CPU cores and 128GB RAM. They run Linux 3.16.5 with TCP Cubic. To make the testbed more manageable, we connect hosts to the OCS via an extra hop on packet switches.

We deploy distributed network controllers as described in Section 4.2. Our OCS uses the standard TL1 interface. To support the proposed interface function, i.e. *replace( )* in Figure 2, we let controllers talk to each logical circuit switch (an OCS partition) through an agent. The agent stores the connections of its own circuit switch, through which it translates the controller queries via our new interface into the TL1 command to control the corresponding ports on the OCS. The source code of our switch image is unaccessible, so we are unable to realize the failure detection mechanism in Section 4.1. Since failure detection is not our main contribution, we bypass it by creating failures at will. We disable forwarding rules to introduce failures and make the controllers react after a dummy detection latency of 10ms. This artifact is easily solvable, since the BFD protocol for fast failure detection is readily available for many commercial switches. We set VLANs at end hosts to enable live impersonation of failed switches according to Section 4.4. We focus on the online failure recovery in the testbed implementation. The offline diagnosis of link failures is evaluated separately in Section 6.5.

The switching delay of our OCS is several milliseconds, orders of magnitude higher than that of the targeted circuit switches, e.g. 70ns for electrical crosspoint switch [23] and

40μs for 2D-MEMS [46]. To evaluate the performance accurately, we also emulate the ideal circuit switch using electrical packet switch (EPS), which observes similar switching delay. The bipartite connections on circuit switches are realized as straight-through forwarding rules between input and output ports on the EPS. In case of failures, controllers change the forwarding rules to redirect traffic to the backup switch. Although rule insertion/deletion introduces extra latency, this is at the best of our effort given the limited hardware.

6.2 Simulation

For both simulations below, the simulated network is a $k = 16$ fat-tree, which consists of 320 switches and 1024 hosts. We assign each failure group 1 backup switch, that is 40 additional switches to the network.

Linear Programming Simulation: We abstract the network as a graph and cripple a varying number of links and switches. We solve the maximum concurrent multi-commodity flow problem [24] given different traffic patterns using a Linear Programming (LP) solver, which is a well-adopted approach in topology analysis [37, 38, 49]. This formulation is to maximize the minimum throughput among all the flows, showing the worst case under the effect of failures. The result assumes optimal routing under perfect load balancing.

Packet-Level Simulation: We developed a simulator that supports TCP, fat-tree's Two-Level Routing, and dynamic failure events. The failure recovery delay is based on the measurement result on our testbed and reported numbers for the compared architectures [26, 30, 43]. Our simulation is a big improvement to a similar study previously [48]. First, their work is limited to converged steady state after failures, while we consider the failure recovery process. Second, their results are biased against rerouting solutions. It uses ECMP routing, which may create more hot spots after rerouting due to hash collisions, and thus exaggerate the effect of failures. It also randomly drops packets from the buffer when congestion happens, so packet retransmission will hugely increase the flow completion time. In contrast, Two-Level Routing eliminates randomness by assigning each flow a deterministic path, and load balancing further mitigates hot spots. The random drop behavior is disabled. Our simulation enables realistic and fair comparisons against rerouting solutions.

6.3 Experimental Setup

6.3.1 *Network Architectures Compared.*

PortLand [30]: We abstract PortLand as a fat-tree [8] network in the LP formulation. In the packet-level simulator and the testbed, we use Two-Level Routing as the default routing method without failures and improve PortLand's global rerouting with near-optimal load balancing under failures. In the simulator, we reroute traffic heuristically according to the bandwidth utilization of alternative paths. In the testbed, for every possible failure, we hard-code the rerouted paths for affected flows such that each link carries roughly the same number of flows. Our optimized version of PortLand gives a throughput upper bound for rerouting.

F10 [26]: We build F10's AB fat-tree. In the packet-level simulator and the testbed, we use Two-Level Routing as the default failure-free routing and perform 3-hop local rerouting under failures. In the simulator, we randomly reroute impacted flows to available local paths. In the testbed, limited by the network scale, there is only one alternative path for each flow. The LP solver enforces global optimal routing, so it evaluates the capacity of the network topology alone, which serves as a very loose upper bound for actual F10.

Aspen Tree [43]: We maintain the host count as in the above networks and pick the Aspen Tree configuration that minimizes extra cost and failure convergence time, that is adding an extra layer of switches below the core switches. Two-level routing does not apply to Aspen Tree because of the redundant layer. Instead, we use ECMP to distribute flows evenly in each layer. Under failures, we reroute traffic locally if alternative paths exist or push back to upstream switches otherwise. Like in F10, the LP simulations perform global optimal routing on the topology to show the capacity upper bound. Aspen Tree is not supported in our testbed due to the extra hardware required.

6.3.2 *Failure Models.*

Random Layered: The LP analysis requires an easy-to-reason failure model that helps understand the effect of failure locations, so we generate random switch and link failures separately in different layers of the network. We simplify this model in the testbed experiments: we create one link failure at a time, since switch failures and concurrent link failures are fatal for our small-scale testbed.

Real: We reproduce real-world failures in our packet-level simulator according to a failure study in production data centers [16]. We create dynamic switch and link failures in the network. Failure locations are derived from the probability of failures per switch/link type (Figure 6 and 7 in [16]), and the arrival time and duration of failures are based on the corresponding distributions (Figure 8 and 9 in [16]).

6.3.3 *Traffic Patterns.*

The LP solver runs on steady traffic, so we drive the computation with the following widely-used synthetic traffic patterns.

Permutation: Every host sends a single flow to a unique server other than itself at random. This pattern creates uniform traffic across the network.

Stride: Every host sends a single flow to its counterpart in the next Pod. This traffic pattern creates heavy contention in the network core.

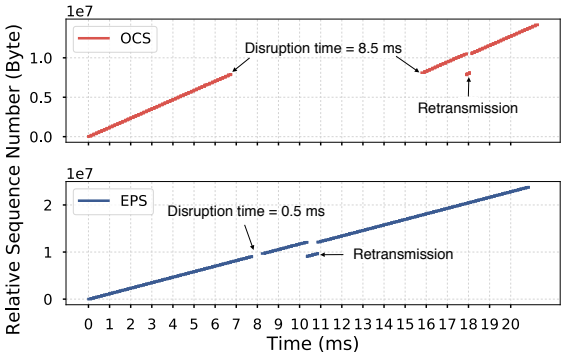

Figure 5: Trace of TCP sequence number during failure recovery.

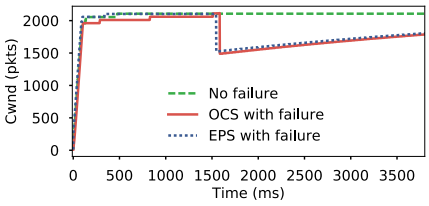

Figure 6: TCP congestion window size during failure recovery.

Hot Spot: Every 100 hosts form a cluster, in which one host broadcasts to all the others. It simulates the multicast phase in many machine learning applications.

Many-to-Many: Every 20 hosts form a cluster with all-to-all traffic. This traffic pattern simulates the shuffle phase in MapReduce jobs.

There are two pervasive types of applications in data centers: throughput-intensive and latency-sensitive [9, 20]. We feed the packet-level simulator with data center traffic traces from these applications to create realistic settings.

Coflow: We obtain the trace in a Facebook data center from the coflow benchmark [4]. It contains correlated flows known as coflows that reflect communications in MapReduce jobs. We observe highly skewed multicast, shuffle, and incast traffic in the trace: some coflows involve a large number of machines and have high traffic volume. For each rack-to-rack flow in the trace, we create flows between hosts under the source and destination edge switches to saturate switch uplinks.

Deadline: We follow the method in D^3 to generate partition-aggregate traffic in interactive web applications [45]. For each query, we randomly choose 1 host as the aggregator and 40 hosts as workers. Workers respond after a random jitter between (0, 10ms] to simulate the local computation. The network utilization varies between 10% and 30%. The deadline is set to be 2× the response time in the failure-free network.

6.3.4 Real Applications.

We run Spark and Tez on our testbed as representative applications in data centers. Among the 8 hosts, the first works as the master node and all the others as slave nodes. We create the following throughput-intensive and latency-sensitive jobs.

Spark Word2Vec: This iterative machine learning job uses high dimensional vectors to represent words in documents. In each iteration, the master node broadcasts the updated model to all workers. We configure Spark to broadcast ~500MB data in each iteration in a BitTorrent fashion. This phase thus observes heavy all-to-all traffic. The data to be transmitted is readily available in memory.

Tez Sort: This job is a distributed sorting algorithm based on the MapReduce programming model. The aggregate input data size is 100GB. This job has a heavy shuffle phase, where all the nodes as mappers send data to a subset of nodes as reducers. We store the data on a RAM disk to prevent the hard drive being the bottleneck of data read/write.

Spark TPC-H: This decision support benchmark consists of a suite of database queries that help answer important business questions. The queries are running against a 160GB database on each worker. The processing power of the decision-making system is reflected by the number of queries per hour, so the query latency is critical to performance.

6.4 Transient State Analysis

We examine the TCP behavior during ShareBackup's failure recovery. A sender host transmits TCP packets at line rate to a receiver host. At the receiver, we capture packets with Wireshark to get the sequence number and record TCP congestion window size with the *tcp_probe* kernel module while injecting a link failure along the path. We get similar results with the variance of sender and receiver locations. Figure 5 and Figure 6 show one instance of the results.

In Figure 5, the OCS implementation and the EPS emulation experiences 8.5ms and 0.5ms disruption time respectively. This delay is contributed by OCS/EPS reconfiguration, i.e. resetting OCS circuits or changing EPS forwarding rules. Interestingly, we observe less packet loss on the OCS testbed even though it has relatively longer disruption time. Further investigations reveal that our packet switch by default stops forwarding packets when their destination port is detected as down. Those packets are buffered in switch memory and sent out after the port comes up. The EPS emulation does not have such *port-down* period and packets are then continuously sent out and dropped in the transient state. As described at the beginning of Section 3, our targeted circuit switch technologies have much lower switching delay than the EPS. They function like the OCS and will cause the port-down event. In practical implementations, we expect shorter disruption time than the EPS and similar or less packet loss than the OCS.

In Figure 6, neither the OCS implementation nor the EPS emulation hit the retransmission timeout. For both of them, TCP can proceed smoothly and recover lost packets rapidly. This result validates our design of fast in-network failure

Table 3: Break-down of failure recovery and diagnosis delay (ms).

	Failure Recovery				Failure Diagnosis	
	total	communication	computation	reconfig	preemption	no preemption
OCS	8.73	0.22	0.01	8.5	502.1	487.3
EPS	0.73	0.22	0.01	0.5	359.2	352.6

repair that is transparent to applications. Our testbed experiments in the rest of the paper are based on the OCS implementation. As we will show later, regardless of the relatively long disruption time, the performance of our ShareBackup implementation is still similar to a failure-free network.

6.5 Responsiveness of Control Plane

Our testbed is limited in scale, and the above implementation ignores failure diagnosis. To understand the efficiency of the control plane in a large data center, we abstract the involved entities as individual processes and realize the communication protocol in Figure 2 on a $k = 48$ fat-tree network. There are 2880 switch processes, 120 controller processes, and 3456 agent processes for circuit switches. The communications are implemented using the server-client model with TCP sockets.

Link failures are more complicated than switch failures for the control system and are followed by the offline failure diagnosis, so we show the failure recovery and diagnosis delay for link failures in Table 3 as the worst-case performance of the control system. The failure recovery delay is broken down into the time for communications in Figure 2(a), computation at the distributed controllers, and OCS/EPS reconfiguration discussed in the above subsection. Circuit switch agents are not necessary if circuit switches support the proposed reconfiguration function, i.e. *replace( )* in Figure 2. Thus, the communication delay can be further reduced in real implementation. The computation delay is minimal, as controllers only map the failed switch and the backup switch to their circuit switch ports so as to reset circuits. In the EPS emulation, the end-to-end delay of failure recovery is only 0.73ms, which will be even lower if using the targeted circuit switches with shorter switching latency and the modified reconfiguration function. F10 and PortLand reported 1ms and 65ms convergence delay [26, 30]. ShareBackup is more efficient than these state-of-the-art solutions because it does not involve change of forwarding rules and computation for rerouting.

Our implementation of failure diagnosis cycles through all the configurations in Figure 3, although the process may terminate earlier in reality. Even with our very simple implementation, failure diagnosis can finish in hundreds of milliseconds. If the diagnosis process is preempted by failure recovery in one of the tested configurations, the duration only increases slightly. As discussed in Section 5 (3), this sub-ms failure recovery and sub-second failure diagnosis are breakthroughs to data center management, compared to the common practice based on manual efforts nowadays.

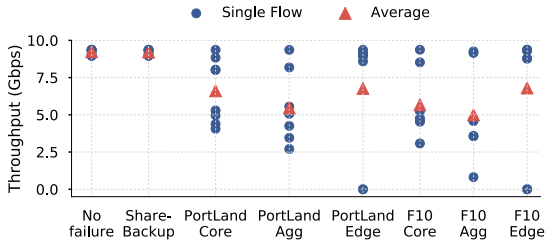

Figure 7: iPerf throughput of 8 flows saturating all links on testbed.

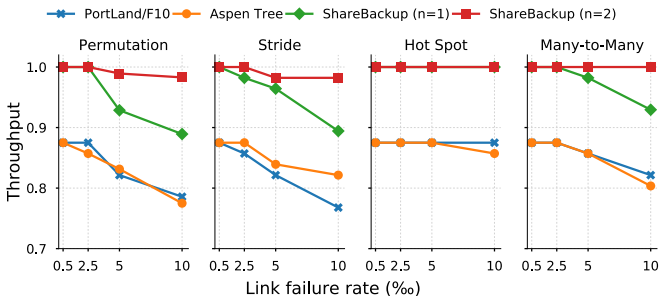

Figure 8: Minimum flow throughput under edge-aggregation link failures normalized against the no-failure case on the LP simulator with global optimal routing for all networks.

6.6 Bandwidth Advantage

On the testbed, we show the bandwidth difference among the various network architectures with an iPerf throughput experiment. We create an instance of permutation traffic, where Two-Level Routing places the 8 flows onto different paths without contention. This traffic pattern saturates all links. In Figure 7, Sharebackup achieves the same performance as the no-failure case, showing ShareBackup's ability to restore bandwidth quickly after failures. In contrast, for PortLand and F10, the worst-case flow throughput decreases dramatically as the failure approaches edge links. In a fat-tree (or AB fat-tree) network, there are $\frac{k^2}{4}$ parallel paths in the core layer, $\frac{k}{2}$ in the aggregation layer, yet only 1 in the edge layer right above hosts. As a result, rerouting causes greater congestion with fewer paths to balance the load at the edge. F10 is less tolerant to failures than PortLand, because its local rerouting uses longer paths and makes congestion even worse.

This trend holds for our LP simulations as well. Due to space limitation, we only show the results for link failures in the aggregation layer in Figure 8. ShareBackup outperforms the other architectures by 13% to 25%. It achieves similar throughput as the case without failures given 2 backup switches per failure group; while 1 backup switch falls short sometimes when concurrent failures happen in the same failure group. Note that this is a stress test. In data centers, most devices have over 99.99% availability, and concurrent failures are very rare [16]. So, 1 backup switch per failure group is sufficient in most cases. Portland and F10 have the same numbers, as the LP solver performs optimal routing on them

Table 4: Percentage of impacted flows/coflows in Figure 9.

	ShareBackup	PortLand	F10	Aspen Tree
Directly impacted flows	0.63%	3.92%	3.91%	3.89%
Indirectly impacted flows	0	16.01%	21.29%	16.23%
Directly impacted coflows	0.78%	17.32%	18.31%	18.48%
Indirectly impacted coflows	0	18.95%	28.89%	19.22%

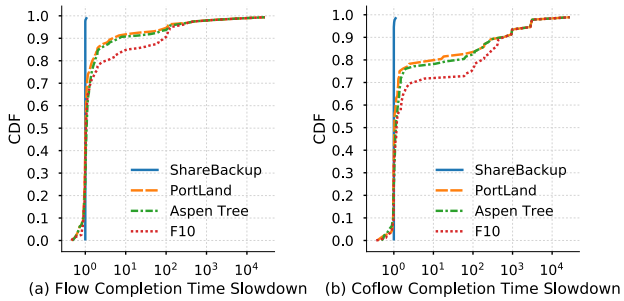

Figure 9: CDF of completion time slowdowns on packet simulator.

both. Although Aspen Tree uses more switches for redundancy, it fails to add more bandwidth to the network, so its performance is no better than PortLand and F10.

Under edge link failures, the impacted flows have zero throughput in all architectures, whereas ShareBackup still gives full capacity. If link failures happen in the core layer, these architectures have very similar throughput, since there are abundant alternative paths for rerouting. The results for switch failures are the same as link failures. Our formulation calculates the minimum flow throughput as the worst-case analysis. Switch failures affect more flows but do not change the minimum value. From these observations, we conclude that ShareBackup is most powerful in the edge layer, then the aggregation layer. As discussed in Section 5 (1), ShareBackup supports layer-wise partial deployment to save cost.

6.7 Transmission Performance at Scale

For throughput-intensive jobs in the Coflow trace, the transmission performance is largely determined by the network's bandwidth capacity shown in the above section. Figure 9 plots the distributions of flow completion time (FCT) and coflow completion time (CCT) normalized against the no-failure case, i.e. slowdowns. Overall, Sharebackup has negligible performance degradation, whereas PortLand, F10 and Aspen Tree experience multi-fold slowdowns for >20% flows and >30% coflows. The impact of failure is magnified at the coflow level, since a small number of straggler flows is all it takes to negatively impact the CCT. F10 performs notably worse than PortLand and Aspen Tree, because its local rerouting uses longer paths (path dilation), resulting in more flows being impacted. Aspen Tree slightly underperforms PortLand, because Aspen Tree's local rerouting also has a small path dilation. As an application can only proceed after the entire coflow has finished, ShareBackup is significantly more effective in masking failures from applications.

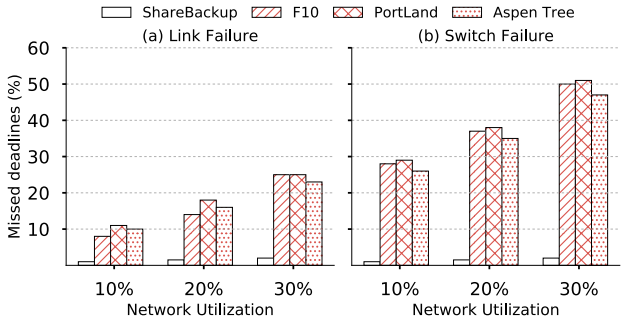

Figure 10: Percentage of jobs missing deadlines on packet simulator.

These results are corroborated by Table 4. As demonstrated in Section 6.5, ShareBackup recovers from a failure in sub-ms. So, very few flows and coflows get impacted during the small transition period. Other architectures, however, have upto 25.2% flows and 47.2% coflows impacted. Note that a large portion of flows/coflows are indirectly impacted, which are not hit by failures but have contention with the rerouted flows. Rerouting spreads the effect of failures to innocent flows, thus converting the local failure to global performance degradation. In comparison, ShareBackup's principle of fixing failures at where they happen effectively localizes the problem and provides more predictability to application performance.

For latency-sensitive flows in the *Deadline* trace, Figure 10 shows the percentage of jobs that miss deadlines under failures. In ShareBackup, failures only cause less than 2% deadline miss, with slight increase as the network utilization grows. ShareBackup handles switch and link failures in the same way, so the results are similar. Rerouting-based solutions perform much worse in comparison. They are sensitive to network utilization and failure type, with the worst-case job miss rate reaching 51%. Although F10 has similar failure recovery delay as ShareBackup, its local rerouting renders heavy traffic contention. PortLand's global rerouting is more efficient, but there is still bandwidth loss and path re-computation takes as long as 65ms [30]. Jointly affected by these two factors, F10 outperforms PortLand slightly. Aspen Tree pushes back most impacted downstream traffic to the core layer and reroutes from there, balancing rerouting delay and path dilation. Its performance thus falls between PortLand and F10.

6.8 Benefits to Real Applications

The performance of the bandwidth-intensive applications on the testbed is shown in Figure 11. We do not differentiate link failures in the aggregation and core layers because they result in similar performance. The trend is consistent with the bandwidth difference in Figure 7. It confirms that ShareBackup masks failures from application performance: for both applications, the communication time and job completion time constantly stay the same as the no-failure case. Big-data frameworks consume most time on computations.

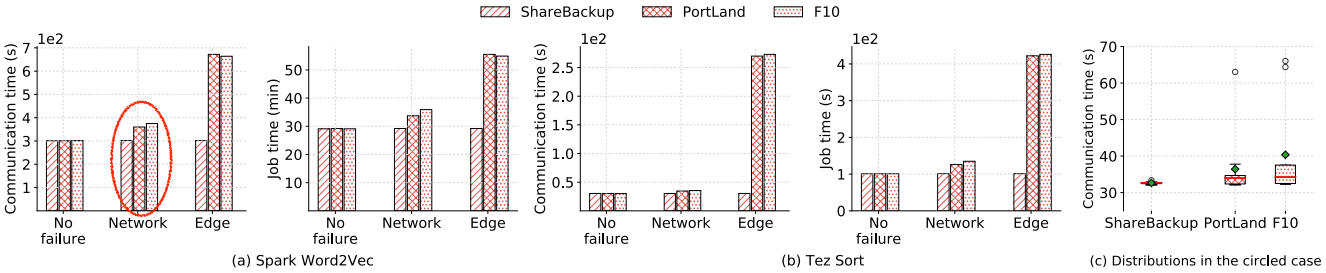

Figure 11: Performance of the Spark Word2Vec and Tez Sort applications with a single edge (edge-host) or network (core-aggregation and aggregation-edge) link failure on the testbed.

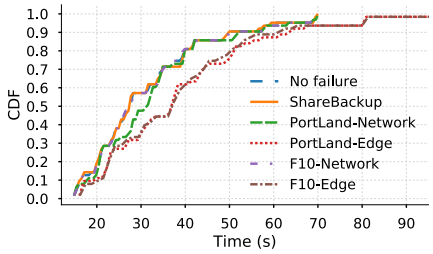

Figure 12: CDF of query latency in the Spark TPC-H application with a single edge (edge-host) or network (core-aggregation and aggregation-edge) link failure on the testbed.

Inter-node communications between computations are influenced by node synchronization, data serialization, garbage collection, etc. Considering all these factors, the bandwidth advantage of ShareBackup can still translate into over 12% less communication time and 23% less job completion time under in-network failures. If an edge link fails, workers may get lost and the master needs to relaunch tasks. The communication phase and thus the entire job finish multi-fold slower. In our experiments, we even encountered cases where the job crashed. Failures near the hosts are disastrous to applications, and ShareBackup is an especially useful remedy.

Figure 11(c) zooms in to the distribution from multiple iterations of data broadcast for the Spark Word2Vec case. Under failures, ShareBackup has almost the same communication and job duration throughout the runs, while the variation is huge for PortLand and F10. In this Word2Vec job with BitTorrent-like traffic, receivers retrieve data blocks depending on availability. Rerouting slows down data retrieval on different degrees at the worker nodes. The change of data availability shapes traffic further later on, leads to a long-tail in completion time. This observation validates the point in Section 6.7 that rerouting enlarges the effect of failure while ShareBackup preserves predictability. We again show ShareBackup can mask failures from application performance: for many applications, it provides performance guarantee even for the worst case.

This long-tail phenomenon also exists in the Spark TPC-H application. The CDF of query latency in Figure 12 reflects TPC-H's performance metric: the number of queries finished in a time period. PortLand and F10 under edge link failures are on average 25% lower than the rest cases using that metric, and their job completion time determined by the last query is 37.4% and 38.1% longer respectively. Hosts are disconnected in this case, and the job relies on Spark's own resilience mechanism, i.e. task relaunch, to proceed. Traffic is light in this application. Traffic contention from rerouting is not heavy enough to cause congestion, so PortLand and F10 have similar performance as ShareBackup in most cases. Nonetheless, ShareBackup is still necessary for edge link failures.

7 CONCLUSION

The advancement of circuit switching technology makes it possible to assign backup switches on demand at runtime. ShareBackup is the first effort to realize this concept of shareable backup in data center networks. Circuit switches have inherent tradeoffs between cost, switching latency, and port count. ShareBackup aims at a cost-effective network architecture for rapid failure recovery, so port count has to be restricted. The choice of modest-size circuit switches drives ShareBackup's distributed design of both the network architecture and the control system. We find this design a good match for the rare, uncorrelated, and spatially dispersed failures in data centers. With co-design of architecture and control system, backup switches can work as hot standbys without primary-backup coordinations or online change of forwarding rules. Besides failure recovery, special setups of circuit switches can automate and speed up failure diagnosis. Extensive system implementations and evaluations demonstrate ShareBackup can effectively mask failures from application performance. This powerful concept of shareable backup goes beyond the specific ShareBackup architecture. We encourage more research efforts in this promising direction.

ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers and our shepherd Yibo Zhu for their thoughtful feedback. This research was sponsored by the NSF under CNS-1422925 and CNS-1718980.

REFERENCES

[1] [n. d.]. Apache Spark, https://spark.apache.org. https://spark.apache.org

[2] [n. d.]. Apache Tez, https://tez.apache.org/. https://tez.apache.org/

[3] [n. d.]. Arduino, https://www.arduino.cc. https://www.arduino.cc

[4] [n. d.]. Coflow-Benchmark, https://github.com/coflow/coflow-benchmark. https://github.com/coflow/coflow-benchmark

[5] [n. d.]. Introducing data center fabric, the next-generation Facebook data center network, url = https://code.facebook.com/posts/360346274145943/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/.

[6] [n. d.]. Raspberry Pi, https://www.raspberrypi.org. https://www.raspberrypi.org

[7] Jung Ho Ahn, Nathan Binkert, Al Davis, Moray McLaren, and Robert S. Schreiber. November 2009. HyperX: Topology, Routing, and Packaging of Efficient Large-scale Networks. In *SC '09*. Portland, Oregon, USA, Article 41, 11 pages. https://doi.org/10.1145/1654059.1654101

[8] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. August 2008. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM '08*. Seattle, Washington, USA, 63–74. https://doi.org/10.1145/1402958.1402967

[9] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitu Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. August 2010. DCTCP: Efficient Packet Transport for the Commoditized Data Center. In *SIGCOMM'10*. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.187.5830

[10] Peter Bodík, Ishai Menache, Mosharaf Chowdhury, Pradeepkumar Mani, David A. Maltz, and Ion Stoica. August 2012. Surviving Failures in Bandwidth-constrained Datacenters. In *SIGCOMM '12*. Helsinki, Finland, 431–442. https://doi.org/10.1145/2342356.2342439

[11] Michael Borokhovich, Liron Schiff, and Stefan Schmid. 2014. Provable data plane connectivity with local fast failover: Introducing openflow graph algorithms. In *Proceedings of the third workshop on Hot topics in software defined networking*. ACM, 121–126.

[12] Matthew Caesar, Martin Casado, Teemu Koponen, Jennifer Rexford, and Scott Shenker. 2010. Dynamic Route Recomputation Considered Harmful. *SIGCOMM Comput. Commun. Rev.* 40, 2 (April 2010), 66–71. https://doi.org/10.1145/1764873.1764885

[13] Kai Chen, Ankit Singla, Atul Singh, Kishore Ramachandran, Lei Xu, Yueping Zhang, Xitao Wen, and Yan Chen. April 2012. OSA: An Optical Switching Architecture for Data Center Networks with Unprecedented Flexibility. In *NSDI '12*. San Joes, CA.

[14] K. Chen, X. Wen, X. Ma, Y. Chen, Y. Xia, C. Hu, and Q. Dong. 2015. WaveCube: A Scalable, Fault-tolerant, High-performance Optical Data Center Architecture. In *2015 IEEE Conference on Computer Communications (INFOCOM)*. 1903–1911. https://doi.org/10.1109/INFOCOM.2015.7218573

[15] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Hajabdolali Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. August 2010. Helios: A Hybrid Electrical/Optical Switch Architecture for Modular Data Centers. In *SIGCOMM '10*. New Delhi, India, 339–350. https://doi.org/10.1145/1851182.1851223

[16] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. 2011. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. In *Proceedings of the ACM SIGCOMM 2011 Conference (SIGCOMM '11)*. ACM, New York, NY, USA, 350–361. https://doi.org/10.1145/2018436.2018477

[17] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. 2009. VL2: A Scalable and Flexible Data Center Network. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication (SIGCOMM '09)*. ACM, New York, NY, USA, 51–62. https://doi.org/10.1145/1592568.1592576

[18] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. August 2009. BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers. In *SIGCOMM '09*. Barcelona, Spain, 63–74. https://doi.org/10.1145/1592568.1592577

[19] Chuanxiong Guo, Haitao Wu, Kun Tan, Lei Shi, Yongguang Zhang, and Songwu Lu. August 2008. DCell: A Scalable and Fault-Tolerant Network Structure for Data Centers. In *SIGCOMM '08*. Seattle, Washington, USA, 75–86. https://doi.org/10.1145/1402958.1402968

[20] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. November 2009. The Nature of Data Center Traffic. In *IMC '09*. Chicago, Illinois, USA, 202–208. https://doi.org/10.1145/1644893.1644918

[21] Simon Kassing, Asaf Valadarsky, Gal Shahaf, Michael Schapira, and Ankit Singla. 2017. Beyond Fat-trees Without Antennae, Mirrors, and Disco-balls. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. ACM, Los Angeles, CA, USA, 281–294. https://doi.org/10.1145/3098822.3098836

[22] Karthik Lakshminarayanan, Matthew Caesar, Murali Rangan, Tom Anderson, Scott Shenker, and Ion Stoica. 2007. Achieving convergence-free routing using failure-carrying packets. *ACM SIGCOMM Computer Communication Review* 37, 4 (2007), 241–252.

[23] Sergey Legtchenko, Nicholas Chen, Daniel Cletheroe, Antony Rowstron, Hugh Williams, and Xiaohan Zhao. 2016. XFabric: A Reconfigurable In-Rack Network for Rack-Scale Computers. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA, 15–29. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/legtchenko

[24] Tom Leighton and Satish Rao. November 1999. Multicommodity Max-flow Min-cut Theorems and Their Use in Designing Approximation Algorithms. *J. ACM* 46, 6 (November 1999), 787–832. https://doi.org/10.1145/331524.331526

[25] Junda Liu, Aurojit Panda, Ankit Singla, Brighten Godfrey, Michael Schapira, and Scott Shenker. 2013. Ensuring Connectivity via Data Plane Mechanisms.. In *NSDI*. 113–126.

[26] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. 2013. F10: A Fault-Tolerant Engineered Network. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX, Lombard, IL, 399–412. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/liu_vincent

[27] Vincent Liu, Danyang Zhuo, Simon Peter, Arvind Krishnamurthy, and Thomas Anderson. 2015. Subways: A Case for Redundant, Inexpensive Data Center Edge Links. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT '15)*. ACM, Heidelberg, Germany, Article 27, 13 pages. https://doi.org/10.1145/2716281.2836112

[28] Yunpeng James Liu, Peter Xiang Gao, Bernard Wong, and Srinivasan Keshav. August 2014. Quartz: A New Design Element for Low-latency DCNs. In *SIGCOMM '14*. Chicago, Illinois, USA, 283–294. https://doi.org/10.1145/2619239.2626332

[29] Suksant Sae Lor, Raul Landa, and Miguel Rio. 2010. Packet re-cycling: eliminating packet losses due to network failures. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. ACM, 2.

[30] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. 2009. PortLand: A Scalable Fault-tolerant

Layer 2 Data Center Network Fabric. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication (SIGCOMM '09)*. ACM, New York, NY, USA, 39–50. https://doi.org/10.1145/1592568.1592575

[31] Pan P., Swallo G., and Atlas A. 1998. Fast Reroute Extensions to RSVP-TE for LSP Tunnels. *RFC 4090* (1998).

[32] George Porter, Richard Strong, Nathan Farrington, Alex Forencich, Pang Chen-Sun, Tajana Rosing, Yeshaiahu Fainman, George Papen, and Amin Vahdat. August 2013. Integrating Microsecond Circuit Switching into the Data Center. In *SIGCOMM '13*. Hong Kong, China, 447–458. https://doi.org/10.1145/2486001.2486007

[33] Mark Reitblatt, Marco Canini, Arjun Guha, and Nate Foster. 2013. FatTire: Declarative Fault Tolerance for Software-defined Networks. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN '13)*. ACM, Hong Kong, China, 109–114. https://doi.org/10.1145/2491185.2491187

[34] Michael Schlansker, Michael Tan, Jean Tourrilhes, Jose Renato Santos, and Shih-Yuan Wang. 2013. Configurable optical interconnects for scalable datacenters. In *Optical Fiber Communication Conference and Exposition and the National Fiber Optic Engineers Conference (OFC/NFOEC), 2013*. IEEE, 1–3.

[35] Tae Joon Seok, Niels Quack, Sangyoon Han, Wencong Zhang, Richard S Muller, and Ming C Wu. 2015. Reliability study of digital silicon photonic MEMS switches. In *Group IV Photonics (GFP), 2015 IEEE 12th International Conference on*. IEEE, 205–206.

[36] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. August 2015. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *SIGCOMM '15*. ACM, London, United Kingdom, 183–197. https://doi.org/10.1145/2785956.2787508

[37] Ankit Singla. 2015. *Designing Data Center Networks for High Throughput*. Ph.D. Thesis. University of Illinois at Urbana-Champaign.

[38] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P. Brighten Godfrey. April 2012. Jellyfish: Networking Data Centers Randomly. In *NSDI '12*. San Jose, California, USA, 1–14. arXiv:1110.1687 http://arxiv.org/abs/1110.1687

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

[40] Brent Stephens, Alan L. Cox, and Scott Rixner. 2016. Scalable Multi-Failure Fast Failover via Forwarding Table Compression. In *Proceedings of the Symposium on SDN Research (SOSR '16)*. ACM, Santa Clara, CA, Article 9, 12 pages. https://doi.org/10.1145/2890955.2890957

[41] Li T., Cole B., Morton P., and Li D. 1998. Cisco Hot Standby Router Protocol (HSRP). *RFC 2281* (1998).

[42] Asaf Valadarsky, Gal Shahaf, Michael Dinitz, and Michael Schapira. 2016. Xpander: Towards Optimal-Performance Datacenters. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies (CoNEXT '16)*. ACM, Irvine, California, USA, 205–219. https://doi.org/10.1145/2999572.2999580

[43] Meg Walraed-Sullivan, Amin Vahdat, and Keith Marzullo. 2013. Aspen Trees: Balancing Data Center Fault Tolerance, Scalability and Cost. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT '13)*. ACM, New York, NY, USA, 85–96. https://doi.org/10.1145/2535372.2535383

[44] Guohui Wang, David G. Andersen, Michael Kaminsky, Konstantina Papagiannaki, T. S. Eugene Ng, Michael Kozuch, and Michael Ryan. August 2010. c-Through: Part-time Optics in Data Centers. In *SIGCOMM '10*. New Delhi, India, 327–338. https://doi.org/10.1145/1851182.1851222

[45] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. 2011. Better Never Than Late: Meeting Deadlines in Datacenter Networks. In *Proceedings of the ACM SIGCOMM 2011 Conference (SIGCOMM '11)*. ACM, Toronto, Ontario, Canada, 50–61. https://doi.org/10.1145/2018436.2018443

[46] M. C. Wu, O. Solgaard, and J. E. Ford. 2006. Optical MEMS for Lightwave Communication. *Journal of Lightwave Technology* 24, 12 (December 2006), 4433–4454. https://doi.org/10.1109/JLT.2006.886405

[47] Xin Wu, Daniel Turner, Chao-Chih Chen, David A. Maltz, Xiaowei Yang, Lihua Yuan, and Ming Zhang. August 2012. NetPilot: Automating Datacenter Network Failure Mitigation. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '12)*. Helsinki, Finland, 419–430.

[48] Yiting Xia, Xin Sunny Huang, and T. S. Eugene Ng. December 2017. Stop Rerouting! Enabling ShareBackup for Failure Recovery in Data Center Networks. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks (HotNets '17)*. Palo Alto, CA, 171–177.

[49] Yiting Xia and T. S. Eugene Ng. November 2016. Flat-tree: A Convertible Data Center Network Architecture from Clos to Random Graph. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNets '16)*. Atlanta, GA, 71–77. https://doi.org/10.1145/3005745.3005763

[50] Yiting Xia, Xiaoye Steven Sun, Simbarashe Dzinamarira, Dingming Wu, Xin Sunny Huang, and T. S. Eugene Ng. 2017. A Tale of Two Topologies: Exploring Convertible Data Center Network Architectures with Flat-tree. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. ACM, New York, NY, USA, 295–308. https://doi.org/10.1145/3098822.3098837

[51] Baohua Yang, Junda Liu, Scott Shenker, Jun Li, and Kai Zheng. 2014. Keep forwarding: Towards k-link failure resilient routing. In *INFOCOM, 2014 Proceedings IEEE*. IEEE, 1617–1625.

[52] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas Anderson. 2017. Understanding and Mitigating Packet Corruption in Data Center Networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. ACM, Los Angeles, CA, 362–375. https://doi.org/10.1145/3098822.3098849