NetPilot: Automating Datacenter Network Failure Mitigation

Xin Wu
Duke University
xinwu@cs.duke.edu

Daniel Turner
U.C. San Diego
djturner@cs.ucsd.edu

Chao-Chih Chen
Microsoft
gechen@microsoft.com

David A. Maltz
Microsoft
dmaltz@microsoft.com

Xiaowei Yang
Duke University
xwy@cs.duke.edu

Lihua Yuan
Microsoft
lyuan@microsoft.com

Ming Zhang
Microsoft
mzh@microsoft.com

ABSTRACT

The soaring demands for always-on and fast-response online services have driven modern datacenter networks to undergo tremendous growth. These networks often rely on scale-out designs with large numbers of commodity switches to reach immense capacity while keeping capital expenses under check. The downside is more devices means more failures, raising a formidable challenge for network operators to promptly handle these failures with minimal disruptions to the hosted services.

Recent research efforts have focused on automatic failure localization. Yet, resolving failures still requires significant human interventions, resulting in prolonged failure recovery time. Unlike previous work, NetPilot aims to quickly *mitigate* rather than resolve failures. NetPilot mitigates failures in much the same way operators do – by deactivating or restarting suspected offending components. NetPilot circumvents the need for knowing the exact root cause of a failure by taking an intelligent trial-and-error approach. The core of NetPilot is comprised of an Impact Estimator that helps guard against overly disruptive mitigation actions and a failure-specific mitigation planner that minimizes the number of trials. We demonstrate that NetPilot can effectively mitigate several types of critical failures commonly encountered in production datacenter networks.

Categories and Subject Descriptors

C.2.3 [**Network Operations**]: Network management

Keywords

Datacenter networks, failure mitigation

1. INTRODUCTION

The growing demand for always-on and rapid-response online services has driven datacenter networks (DCNs) to an enormous size, often comprising tens of thousands of servers, links, switches, and routers. To reduce capital expenses and increase overall system reliability, datacenter (DC) designers are increasingly building their networks using broad layers of many inexpensive commodity hardware instead of large chassis switches. However, as the number of devices grows, the failure of network devices becomes the norm rather than the exception.

Diagnosing and repairing DCN failures in a timely manner has become one of the most challenging DC management tasks. Traditionally, network operators follow a three-step procedure to react to network failures: 1) detection; 2) diagnosis; and 3) repair. Diagnosis and repair are often time-consuming, because the sources of failures vary widely, from faulty hardware components to software bugs to configuration errors. Operators must sift through many possibilities just to narrow down potential root causes. Even though automated tools exist to help localize a failure to a set of suspected components [5, 19], operators still have to manually diagnose the root cause and repair the failure. These diagnosis and repair sometimes require third-party device vendors' assistance, further lengthening the failure recovery time. Because of the above challenges, it can take a long time to recover from disruptive failures even in well-managed networks. For instance, in April 2011, a failure in Amazon's AWS service impaired the operations of many cloud services for hours [29].

Realizing the problem above, we take a fundamentally different approach to tackle the failure recovery problem in large-scale DCNs. Specifically, we advocate a four-step process to react to failures: 1) detection; 2) *mitigation*; 3) diagnosis; and 4) repair. We argue that it is more important to mitigate failures than to fix them in real-time. Here "mitigate" means taking action(s) that alleviate the symptoms of a failure, possibly at the cost of temporarily reducing spare bandwidth or redundancy. Timely and effective failure mitigation enables a DCN to operate continuously even in the presence of failures, and allows operators to dive directly into failure diagnosis and repair.

This paper presents **NetPilot**, an *automated system* that adopts our four-step process to *quickly* mitigate failures in a large-scale DCN before operators diagnose and repair the root cause. NetPilot can significantly shorten the failure disruption time by mitigating failures without human intervention. It can also improve online user experience and lower potential revenue losses that stem from service downtime. Moreover, it can lower a DC's operational costs, as it reduces the number of emergent failures and the number of midnight calls to on-call operators.

A key observation that motivates NetPilot's design is that simple actions such as deactivation or restart, coupled with the redundancy that exists in a DCN (§ 2), can effectively mitigate most types of failures in a DCN. DCNs often have extra links and switches to accommodate peak traffic load and device failures. In many cases, simple actions such as deactivating or restarting an offending com-

SIGCOMM'12, August 13–17, 2012, Helsinki, Finland.

ponent can mitigate failures with little impact on the network's normal functions. This observation differentiates NetPilot's design from conventional failure diagnosis and repair approaches, which require detailed failure-specific knowledge.

Since NetPilot has mitigation as its end goal, it can operate without human intervention and without knowing the precise failure root cause. NetPilot automatically mitigates failures in DCNs through a trial-and-error approach. First, it detects a failure and identifies the set of suspected faulty components. Then, it intelligently iterates through the suspected devices and applies mitigation actions on them one by one, until it mitigates the failure or has exhausted all possible actions.

Realizing NetPilot requires overcoming two key technical challenges. First, when mitigation actions are applied blindly, they can actually further compromise the health of the network, *e.g.*, deactivating a switch may overload other switches during peak hours. NetPilot avoids this problem by employing an Impact Estimator to accurately predict the impact of mitigation actions and executing the actions within a pre-defined safety margin.

Second, although NetPilot can safely try numerous mitigation actions before successfully mitigating a failure, an excessive number of trials will unnecessarily lengthen the failure mitigation process. NetPilot addresses this challenge with an optimized mitigation planner that uses failure-specific information to localize a failure to the most likely faulty components, and orders the sequence of mitigation actions according to their potential benefits.

To the best of our knowledge, NetPilot is the first automated failure mitigation system for DCNs. Our contributions are:

- We study and classify the high-impact failures in production DCNs over a six-month period, and find that we can mitigate most of those failures by simple actions such as restart or deactivation. We also find that there is sufficient redundancy in a DCN to accommodate the impact of mitigation actions (§ 2,3).
- We design and implement NetPilot (§ 4,5) and deploy it in a testbed that resembles a real DCN topology. We also conduct simulations using data from a production DCN to evaluate NetPilot at a large scale. We experimentally validate the accuracy of the Impact Estimator, and find that it offers an error rate of less than 8%
- We use NetPilot to automatically mitigate three types of high-impact failures that operators often encounter in production DCNs. Besides reducing operational overhead, NetPilot decreases the median mitigation time from 2 hours to 20 minutes compared to current operational practice, significantly shortening a failure's impact on online services.
- We justify the design choices made in NetPilot. Compared to simple heuristics, NetPilot can succeed with fewer trials while maintaining safe operating conditions in the network.

2. REDUNDANCY IN DATACENTER NETWORKS

In this section, we motivate NetPilot's design with the observation that today's DCNs have plenty of redundancy at the device level, protocol level, and application level. In the next section, we discuss how NetPilot takes advantage of these redundancies to automatically mitigate failures.

A DCN's must balance between scale and cost to support tens of thousands of servers with high bandwidth and at low cost. Solutions have increasingly converged to one design paradigm: using many inexpensive commodity devices to scale up capacity and to reduce cost while deploying various types of redundancy to combat unreliability [11]. We describe three redundancy types below.

2.1 Device-Level Redundancy

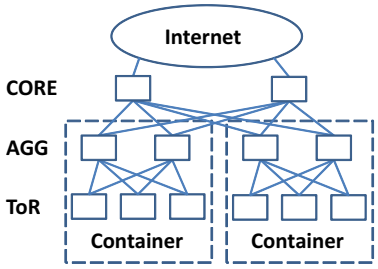

Figure 1: An example scale-out DCN topology.

A modern DCN typically uses a scale-out topology to create many redundant paths between two servers. Scale-out topologies such as a Fat-Tree [4] and Clos [8] can achieve full bi-section bandwidth using commodity switches that often have low port density.

Figure 1 shows an example scale-out topology. This topology has multiple layers: the top-of-rack (*ToR*) layer, the aggregation layer (*AGG*), and the core layer (*CORE*). A *container* is a conceptual management unit as well as a replicable building block sharing the same power and management infrastructure. A *CORE* router connects to multiple containers. For ease of exposition, in the rest of the paper, we use the terms *ToR*, *AGG*, and *CORE* to refer to a switch/router at the ToR, aggregation, or core layer, respectively.

This scale-out topology provides many paths, sometimes in the hundreds, between any two servers. Such path diversity makes the network resilient to single link, switch, or router failure. For example, in Figure 1, we find that deactivating a single link or device, with the exception of a *ToR*, will not partition the network. Even when a failed *ToR* causes network partition, it will only isolate the small number of servers connected to it.

2.2 Protocol-Level Redundancy

DCNs also use various protocol level technologies to meet traffic demands even when some devices fail. We briefly introduce three practical technologies that provide load balancing and fast failover at the link, switch, and path level. These technologies are widely available in off-the-shelf devices from major switch vendors.

Link Aggregation Control Protocol (LACP) abstracts multiple physical links into one logical link and transparently provides high aggregate bandwidth and fast failover at the link level. The resulting logical link is known as a Link Aggregation Group (LAG). LACP provides load balancing by multiplexing packets to physical links by hashing packet headers. Some LACP implementations allow a LAG to initiate from one physical switch but to terminate at multiple physical switches. A LAG can only load balance outgoing traffic, and it has no control over the incoming traffic.

Virtual switch is a logical switch composed of multiple physical switches. A network can use a virtual switch at the link or the IP layer to mask the failures of physical switches.

A virtual switch tolerates faults at the IP layer through an active/standby configuration. One switch is designated as the primary while the standby switch remains silent until it detects that the primary has failed. Two common implementations of IP layer virtual switches are the virtual redundancy router protocol (VRRP) [24] and hot standby router protocol (HSRP) [27]. Both VRRP and HSRP can be configured to provide load balancing.

A virtual switch at the link layer differs from its IP layer counterpart by allowing the physical switches to simultaneously forward traffic. Generically called Multi-Chassis LAG (MC-LAG), Virtual Port Channel (VPC) [2] and Split Multi-link Trunking [13] are two common implementations.

Full-mesh *CORE*s refer to the full-mesh interconnections between *CORE*s and containers, *i.e.*, every container connects to every core switch [10, 21]. The ECMP [12] routing protocols in full-mesh-*CORE*s topologies provide load balancing and fast failover for traffic between containers.

2.3 Application-Level Redundancy

Modern DCNs also deploy application-level redundancy for fault tolerance. Given that a *ToR* is a single point of failure for the servers connected to it (unless they are multi-homed), a common technique to increase failure resilience at the application level is to distribute and replicate applications under multiple *ToR*s. Therefore, stopping or restarting any switch including a *ToR* is unlikely to have more than an ephemeral impact on the applications.

3. REDUNDANCY WARRANTS AUTOMATED FAILURE MITIGATION

In this section, we analyze and classify the failure records in a six-month period from several production DCNs ($DCNs_p$). We then show that most failures are easy to detect, but difficult to diagnose or repair. However, we can mitigate them using simple actions such as deactivation or restart.

We obtained six months of failure records for $DCNs_p$, all of which were manually created by operators and contain detailed descriptions of critical failures. Network operators consider a failure critical if it is either visible to users or impacts revenue. Therefore each record represents a failure that required immediate investigation and response. The fields of interest from these records are: the data sources used to detect the failure, the techniques used to mitigate the failure, the final actions taken to repair the failure, and the start and end times.

Table 1 classifies $DCNs_p$'s critical failures. We find that the single largest source of failures are misconfigurations. Prior research made similar findings in other contexts, such as ISP networks [25]. Misconfigurations are common in DCNs due to the inherent complexity in managing configuration files in large-scale networks. Many of these misconfigurations, *e.g.*, incorrect ACL (Access Control List) rules, lead to lack of connectivity between certain hosts. Host-to-host pings can detect these misconfigurations, but fixing them is challenging and usually requires operators to manually debug the problems.

The next common type of failure is device software failures. One such example is a malfunctioning hash function that results in uneven link utilization among the physical links in a LAG. In some situations, the uneven utilization is so severe that one of the physical links becomes overloaded and discards packets. We can detect this type of failure by comparing the utilization of each link in the LAG. While we can detect software failures, diagnosing their root causes is generally nontrivial since operators know little about the inner workings of device binary code. Even for device vendors who have the source code, debugging the software is still challenging because failures cannot be easily reproduced in lab environments.

The third category of failures is hardware failures. Such failures occur frequently due to the large number of devices used in $DCNs_p$. For example, 13% of the failures are caused by a single problem, frame checksum (FCS) errors, which often significantly elevates host-to-host latencies. Like other failures, FCS errors can be detected by checking switch SNMP counters, *i.e.*, checking the number of corrupted frames received on each port. However, pinpointing the sources of FCS errors takes time because corrupted frames propagate throughout the network due to cut-through switching (§4.3.1). Even after identifying the correct source, operators have to manually replace the corrupted cable.

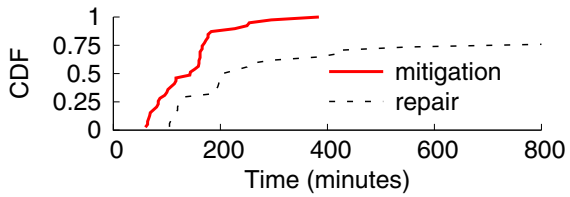

Figure 2: This figure shows the CDFs of how long it takes for $DCNs_p$'s operators to mitigate and to repair critical failures.

The last category is failures whose sources are unknown. In some failures, switches have high CPU utilization while their traffic load is low. In other cases, switches may suddenly cease to forward traffic. These failures often cause abnormally high latency or packet losses. Due to the intermittent and unpredictable nature of these failures, they are difficult to reproduce, diagnose, and repair.

3.1 Time-Consuming Failure Recovery

From studying the types of failures in $DCNs_p$, we see that completely repairing a failure may require debugging of code, software update, or hardware replacement. Further, this process may also involve multiple parties: network operators, software developers, hardware engineers, and external vendors. It is therefore difficult to automate. Figure 2 shows the CDFs of failure mitigation and repair times in $DCNs_p$. As can be seen, the failure repair times can exceed several days or even weeks.

3.2 Simple Mitigation Actions are Effective

Table 1 shows that simple actions such as deactivating or restarting a switch or port can *mitigate* most types of failures in $DCNs_p$ before the root cause can be repaired. For example, in the case of FCS errors, operators reduce latency by deactivating the corrupted links. In the case of overload triggered by load imbalance, operators restore load balance by restarting the offending switches.

In fact, operators of $DCNs_p$ already take *manual* mitigation actions to restore a network to a functioning state while diagnosing and repairing the failure. NetPilot's design goal of automating failure mitigation is partly motivated by the difficulty of manual mitigation. We compare the manual failure mitigation time with the repair time in Figure 2. As can be seen, the time it takes to mitigate failures (even manually) is much shorter than that to repair them. The median failure mitigation time is about two hours.

We note that not all failures can be mitigated by simple actions. Certain failures, *e.g.*, a global configuration error such as a misconfigured ACL, would require a network-wide reconfiguration to fix, and thus cannot be mitigated by restarting or deactivating a few offending devices. How to automatically mitigate those failures is beyond the scope of this paper.

3.3 Spare Capacity for Mitigation Actions

From the analysis above, we find that simple actions are highly effective in mitigating failures and also lead themselves to automation. An automated failure mitigation system can significantly reduce failure mitigation time, as well as the burden on operators.

However, one might be concerned that these simple mitigation actions may overload the network. To find out whether a DCN would have sufficient capacity for failure mitigation, we use the

Category	Detection	Mitigation	Repair	Percentage
software 21%	link layer loop	deactivate port	update software	19%
	imbalance triggered overload	restart switch		2%
hardware 18%	FCS error	deactivate port	replace cable	13%
	unstable power	deactivate switch	repair power	5%
unknown 23%	switch stops forwarding	restart switch	n/a	9%
	imbalance triggered overload	restart switch		7%
	lost configuration	restart switch		5%
	high CPU utilization	restart switch		2%
configuration 38%	errors on multiple switches	n/a	update configuration	32%
	errors on one switch	deactivate switch	update configuration	6%

Table 1: This table categorizes the high-impact failures in several production DCNs over a six-month period. All failures listed here either are visible to users or impact revenue.

maximum link utilization after deactivating a component to measure the amount of spare network capacity. We use the Impact Estimator to carry out this computation. (We will describe Impact Estimator in detail in §4.2.)

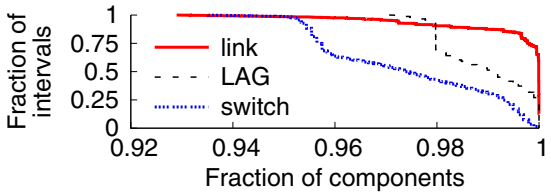

Figure 3: This figure shows the fraction of time intervals during which deactivating a link, a LAG, or a switch will not cause the maximum link utilization to exceed a target threshold (90%) versus the fraction of components for which this result holds.

We perform this computation using the traffic matrices aggregated over 10-minute intervals in one month for DCN_p, a single DCN in DCNs_p. Figure 3 shows the fraction of time intervals during which deactivating a link, a LAG, or a switch will not cause the maximum link utilization to exceed a target threshold (which is 90%) set by DCNs_p's operators and the fraction of components for which this holds. As can be seen, 99% of the links can be deactivated in over 90% of the intervals without overshooting the target threshold. These numbers are slightly lower for LAGs and switches, because a LAG or a switch contains multiple links. Overall, there is sufficient redundancy in DCN_p to tolerate a component deactivation most of the time.

4. NetPilot DESIGN

NetPilot quickly mitigates failures without knowing their actual root causes in four steps. The first step (S_1) is *failure detection*, in which it constantly monitors the network and detects any potential failure. The second step (S_2) is *mitigation planning*. When NetPilot detects a failure, it will propose a set of suspected components, determine the appropriate mitigation actions, and order these actions based on the likelihood of success or potential impact. The third step (S_3) is *impact estimation*. To avoid taking any action that would further degrade network health, NetPilot estimates the impact of each action and discards the actions that are considered unsafe. The last step (S_4) is *plan execution*. NetPilot will successively execute each mitigation action. If an action successfully mitigates the failure, NetPilot marks the failure as mitigated. Otherwise, NetPilot will roll back the action and try the next action.

In this section, we focus on two main technical challenges: impact estimation (S_3) and mitigation planning (S_2). We postpone the discussion of failure detection (S_1) and plan execution (S_4) to the next section. Note that impact estimation *must* be accurate in order for NetPilot to avoid actions that could further degrade network health. However, NetPilot can work properly even *without* precisely localizing a failure or ordering the mitigation actions.

4.1 Impact Metrics

A chief design goal for NetPilot is to avoid taking any mitigation action that could further degrade a DCN's health. Typically, for a given traffic matrix over a time interval T, we can assess a DCN's health via three metrics: *availability*, *packet losses* and *end-to-end latency*. The availability and packet losses of a DCN can be quantified by the fraction of servers with network connectivity to the Internet ($online\_server\_ratio$) and the total number of lost packets ($total\_lost\_pkt$) during the interval T respectively. Quantifying latency is tricky because it is difficult to predict how intra-DC network latency would change after a mitigation action. Given this problem, we use the maximum link utilization (max_link_util) across all links during the interval T as an indirect measure of network latency. Because the propagation delay is small in a DCN (no more than a few milliseconds), low link utilization implies small queuing delay and thus low network latency. Next, we will explain how to predict these metrics after a mitigation action.

4.2 Estimating Impact

The Impact Estimator aims to estimate a mitigation action's impact on a DCN. Answering this question is crucial for ensuring the safety of mitigation actions. The Impact Estimator takes an action A and a traffic matrix TM as two input variables and computes the expected impact of A under TM. Since it is straightforward to compute $online\_server\_ratio$ given a network topology, we focus on estimating max_link_util and $total\_lost\_pkt$ in the rest of the discussion.

We can get the max_link_util and $total\_lost\_pkt$ by collecting SNMP counters in a DCN. However, predicting these two metrics after a mitigation action is nontrivial because the action could change the traffic distribution in the network. In a DCN with no centralized routing control, we cannot precisely predict how packets are routed to their destinations, unless we know all the packet headers, forwarding tables, and load balancing hash functions.

Our approach to address this challenge is based on two important facts that shape the traffic distribution in DCNs. First, there are far more flows than the diversity of paths in DCNs [10, 17]. Second, hash-based flow-level load balancing is widely used at the link level, switch level, and path level in production DCNs [10, 12].

These two facts make packet headers and load balancing hash functions, which are difficult to obtain in real-time, unnecessary in predicting traffic distribution. Our intuition is that hashing many flows onto a relatively small number of paths leads to even load balancing [10]. As a result, a coarse-grained TM plus forwarding

tables should enable us to estimate the real traffic distribution with reasonably high accuracy.

We choose to represent a TM at the granularity of *ToR*-to-*ToR* traffic demands instead of server-to-server, because this representation dramatically reduces the size of TM while not affecting the computation of traffic distribution at the *AGG* or *CORE* layers.

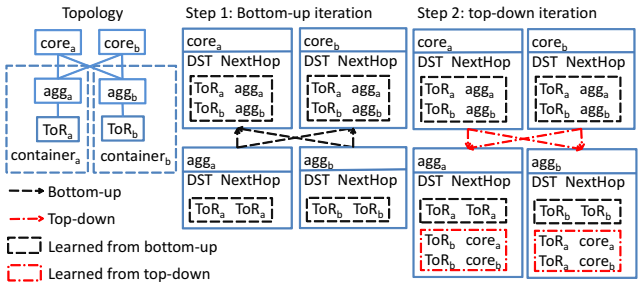

Figure 4: A switch learns the equal cost next hops to any *ToR* after two iterations in a hierarchical DCN topology.

Besides TMs, we also need the forwarding tables to know the next hops to any given destination. As explained in §2, a DCN typically follows a hierarchical structure with traffic traversing valley-free paths. This inspires us to infer the forwarding tables in a similar manner, as illustrated in Figure 4. In the first bottom-up iteration, every switch learns the routes to its descendant *ToR*s from its direct children. In the second top-down iteration, every switch learns the routes to the non-descendant *ToR*s. After these two iterations, every switch builds up the full forwarding table to any *ToR*s in the network.

Algorithm $node.Forward(load)$

1: **if** $load.dst == node$
2: return; // reach the destination;
3: $nxtHops = node.Lookup(load.dst)$
4: **for** each node n in $nxtHops$
5: **for** each link l between $node$ and n
6: $subload.dst = load.dst$;
7: $subload.volume = \frac{load.volume}{|nxtHops|} \times \frac{1}{|\text{links between } node \text{ and } n|}$;
8: $n.Forward(subload)$;

We use the term $load$ to refer to the traffic demand between two *ToR*s. Algorithm $node.Forward$ presents how a $node$ forwards a $load$ in detail. Line 3 returns all the next hops (*nxthops*) to a destination. Assuming even load balancing for traffic crossing adjacent levels in the network hierarchy, Lines 4-8 first evenly split $load$ among the *nxthops*, and then for each next hop, the traffic is evenly split among the physical links. The second traffic split is necessary due to the presence of LAGs (described in §2). By running this algorithm on each $load$ in TM and aggregating the contribution of each $load$ on each link, we predict all the link utilizations.

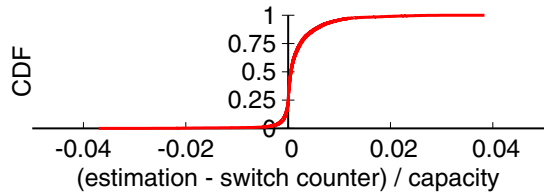

Figure 5: The differences between the link utilizations read from SNMP counters and those computed by Impact Estimator are within 4%.

We evaluate Algorithm $node.Forward$ in an 8000-server production DCN to see whether the Impact Estimator is accurate. We log one-month's socket events on all the servers and aggregate the logs into *ToR*-to-*ToR* traffic matrices at a 10-minute granularity. We also collect the link utilizations during the same month via SNMP at a 10-minute granularity. Figure 5 shows the CDF of the relative differences between the estimated link utilizations and the measured ground truth. As can be seen, the Impact Estimator works very well, and has a maximum error rate of $\pm 4\%$.

So far, we have explained how the Impact Estimator works under a known network topology and TM. To predict the impact of an action, we need to know the new topology and TM after the action is committed. Although inferring the new topology is straightforward, predicting the new TM can be tricky because a mitigation action might affect the traffic demand from minutes up to days. For a restart action which takes only several minutes, we use the TM in the most recent time interval (*e.g.*, 10 minutes) to predict the action's impact during the restart period, assuming the TM is unlikely to change dramatically in such a short time. For a deactivation action that may last days, due to a faulty component needing to be replaced, we ideally desire to use the TM's in the future days to predict the impact during the deactivation period. However, traffic prediction is a research topic by itself, and is beyond the scope of this paper. Instead, we use the TMs in the most recent n days before a deactivation action to predict the impact in the future n days, assuming that the traffic demands are stable over $2n$ days when n is small. In our evaluation (§6.5), we find that this simple heuristic works reasonably well.

4.3 Planning Mitigation

Given that NetPilot takes a trial-and-error approach toward failure mitigation, it needs a mitigation planner to localize suspected components and prioritize mitigation actions to minimize the number of trials. One simple solution is to use existing work [5, 15, 16] to localize failures and then iteratively try deactivating or restarting the suspected components. Although this simple, failure-agnostic solution might work, we choose to develop a mitigation planner that uses failure-specific knowledge to achieve finer-grained localization and more meaningful ordering of mitigation actions (*i.e.*, based on success likelihood). This in turn leads to fewer trials and shorter mitigation times. The downside is that NetPilot needs a planning module for each type of failure. However, we consider this tradeoff worthwhile since there are relatively few types of critical failures in DCNs (as shown in Table 1).

In this section, we first describe in detail mitigation planning for three types of failures: *FCS errors*, *link-down*, and *uneven-split*. We will then discuss the other failure types (listed in Table 1), which are easier to handle compared to these three types.

4.3.1 Frame Checksum Errors

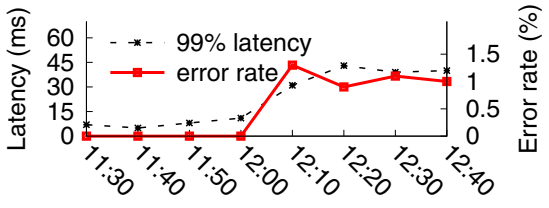

Figure 6: The 99-percentile application-level latency increases significantly due to one corrupted link in DCN$_p$.

Many of the links in a DCN are optical. When foreign material such as dust gets between an optical cable and its connector,

the packets traversing the optical link can suffer bit flips. This causes a frame to mismatch its checksum. As shown in Table 1, FCS errors occur frequently in DCNs$_p$ and can significantly degrade performance. Figure 6 shows how a corrupted link impacts the application-level latency in DCN$_p$. Around 12:00pm, a link connecting a *ToR* and an *AGG* switch began to corrupt 1% of all packets. This 1% corruption rate leads to a 4.5 times increase in the 99^{th} percentile latency for applications running under that *ToR*.

Although replacing the faulty cable is the ultimate solution, this could take days depending on staff availability. Operators can mitigate the failure by disabling the faulty link before it is replaced. However, identifying the faulty link is challenging due to the wide use of cut-through switching [1] in DCNs. Because cut-through switches start forwarding a frame before they can verify its checksum, switches can distribute corrupted packets across the network before the corrupted packets are detected locally.

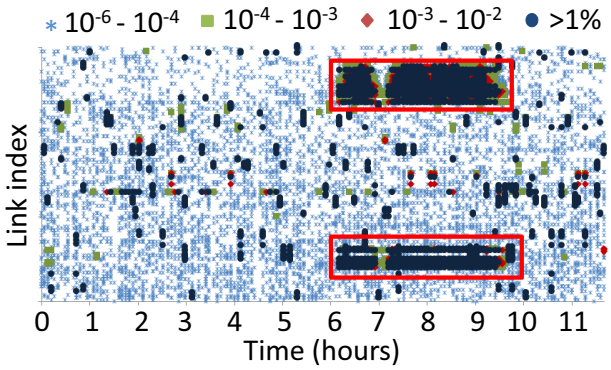

Figure 7: Each point shows a link's error rate. Darker dots indicate higher error rates. We highlight two areas that have the highest error rates (>1%) in 28 links. They are caused by two corrupted links in DCN$_p$, and it took the operators 3.5 hours and 11 trials to deactivate the two offending links.

Figure 7 exemplifies how cut-through switching affects FCS errors in DCN$_p$. The operators began to observe many corrupted packets around hour 6.5, when 28 ports' error rates exceed 1%. Over the next 3.5 hours, operators were busy deactivating the suspected ports one-by-one to determine the faulty links. Finally around hour 10, they found and deactivated the two offending ports and the link error rates returned to normal afterward.

Mitigating FCS errors. Our solution is based on two observations. First, *errors are conserved* on cut-through switches that have no faulty links, *i.e.*, the number of incoming corrupted packets should match the number of outgoing corrupted packets. This observation holds because packet losses are uncommon and broadcast/multicast packets account for only a tiny fraction of the total traffic in DCNs$_p$. Second, the error rate of each faulty link is small and the number of simultaneous faculty links is small. Therefore, it is unlikely that multiple faulty links contribute to the corruption of one packet.

Based on these two observations, we design an FCS error propagation model to localize faulty links. We use x_l to denote link l's corruption rate, p_l and e_l for the total number of packets and the number of corrupted packets traversing l respectively, and m_{kl} for the fraction of packets coming from link k that also traverse link l. Note that the number of corrupted packets coming from link l is equal to the number of packets corrupted by l plus the number of packets corrupted by other links that traverse l. By ignoring the packets corrupted by multiple links, We have:

$$e_l = \sum_{k \neq l} p_k \, x_k \, m_{kl} + p_l \, x_l \quad (1)$$

We use the same technique as that of the Impact Estimator to compute m_{kl}. e_l, p_k and p_l can be obtained from SNMP counters. Thus, the linear equations (1) provide the same number of constraints as the number of variables (x_l's). If we get a unique solution, the faulty links are those with non-zero x_ls. If the solutions are not unique, we simply pick one with the smallest number of non-zero x_ls based on the fact that the number of simultaneous faulty links is usually small. Our evaluation shows that this approach works well in practice with very few false positives (§6.3).

4.3.2 *Link-down and Uneven-split*

Even when the network has the capacity to handle the offered load, link overloading may still occur due to load imbalance or link failure, leading to packet losses and high latencies in DCNs.

Link-down: When one link in a LAG_x is down, the LAG_x will redistribute the traffic to the remaining links. Since this process is transparent to higher layer protocols, traffic demands remain the same over LAG_x. Thus, LAG_x can become overloaded. One mitigation strategy is to deactivate the entire LAG_x and have the traffic re-routed via other LAGs to the *nxthops* (defined in §4.2). Another strategy is to deactivate all the LAGs (including LAG_x) to the *nxthops* and re-route the traffic to other switches.

Uneven-split: Due to software or hardware bugs, a switch may unevenly split traffic among the *nxthops* or the links in a LAG. In DCNs$_p$, we sometimes observe extreme traffic imbalance such as when one link in a lag carries 5Gb/s more traffic than any of the other links in the LAG. While the exact root causes might be unknown, operators have found that restarting the LAG or switches on either end rebalances the traffic (at least for some period of time).

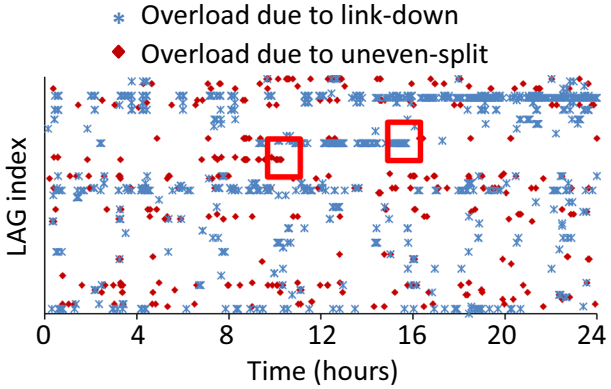

Figure 8: Each point indicates a link overloading incident in a LAG caused by load imbalance in a production DCN. We highlight two incidents. The first occurred around hour 10 was due to uneven-split and was mitigated by restarting a switch. The other occurred around hour 16 was due to link-down and was mitigated by deactivating a LAG.

Figure 8 illustrates that both types of failures are common in DCN$_p$. We collect all the link utilizations every 10 minutes for one day in DCN$_p$. In this figure, each dot represents a LAG that meets the following two conditions in a 10-minute interval: 1) at least one link is overloaded (utilization $> 90\%$); and 2) at least one link is broken (*link-down*) or the difference between the maximum and mean link utilizations exceeds 5% (*uneven-split*). We choose

5% as the load imbalance threshold because Figure 5 not only suggests that accurate impact estimation is feasible but also suggests that large load variance ($> 5\%$) within a LAG is a strong indication of the traffic imbalance problem. We highlight two incidents in Figure 8: one *uneven-split* failure mitigated by a switch restart around hour 10 and another *link-down* failure mitigated by a LAG deactivation around hour 16.

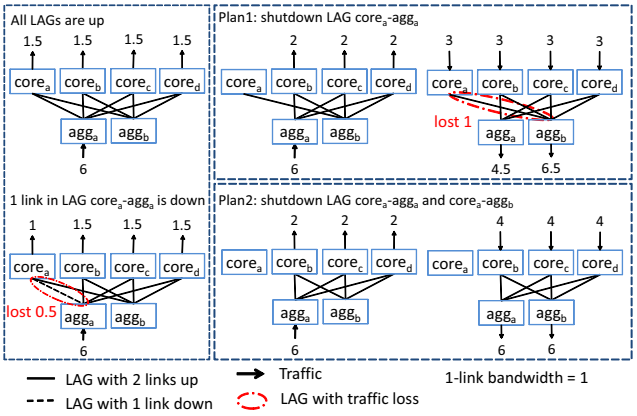

Figure 9: The first plan deactivates one LAG but still causes downward traffic loss. The second plan deactivates two LAGs without causing any traffic loss.

Mitigating a *link-down* or *uneven-split* requires some care due to the complexity of the traffic matrix and topology, as exemplified in Figure 9. Each pair of switches is connected by a LAG consisting of two physical links with a combined capacity of two units. There are six units of upward traffic from agg_a to the $cores$ and twelve units of downward traffic from $cores$ to $aggs$. Suppose one link between agg_a and $core_a$ is down, halving the corresponding LAG capacity, resulting in 0.5 unit of upward traffic loss. One obvious mitigation strategy (Plan 1) is to deactivate the entire LAG between agg_a and $core_a$. Although this prevents the upward traffic loss, it causes one unit of downward traffic loss between $core_a$ and agg_b. The correct strategy is to deactivate the LAG between $core_a$ and agg_b as well (Plan 2). This will shift the downward traffic via $core_a$ to the other $cores$ and prevent traffic loss in both directions.

Mitigating *link-down*. NetPilot mitigates *link-down* failures by estimating the impact of all possible deactivation actions and carrying out the ones with the least impact, *i.e.*, minimizing maximum link utilization. Because a link could be down for n days, NetPilot needs to estimate an action's impact during the downtime. To do so, NetPilot uses the traffic matrices of the most recent n days (§4.2) as an approximation. Such a computation is difficult for human operators to perform because the number of mitigation actions and traffic matrices to consider in concert could be quite large.

Mitigating *uneven-split*. NetPilot mitigates *uneven-split* failures by restarting LAGs or switches. To limit the temporal interruptions during restarts, NetPilot prioritizes the restart sequence based on a restart's estimated impact, while also assuming a component cannot carry any traffic during restart. Since restarting one component usually takes only a few minutes, NetPilot uses the traffic matrix in the most recent time interval (*e.g.*, 10 minutes) as an approximation of the traffic matrix during the restart. After exhaustively calculating the impact for every possible restart, the planner will first carry out the action with the least estimated impact. If this action does not mitigate the failure, the planner will re-prioritize the remaining options based on the latest traffic matrix.

4.3.3 *Other Types of Failures*

FCS error, *link-down*, and *uneven-split* are by no means all the failures that NetPilot can mitigate. We carefully review all the critical failures in Table 1 and find 62% of them can be localized via available data sources (such as SNMP counters and syslogs) and can be mitigated via deactivation or restart. The only exceptions are the failures due to configuration errors (38%). Although configuration errors on a single switch can be mitigated by deactivating the misconfigured switch, identifying if a configuration error involves one or multiple switches still requires human intervention. We briefly discuss how to apply NetPilot to mitigate other failures:

Link layer loop: Due to switch software bugs, link layer protocols sometimes never converge and cause severe broadcast storms. This failure can be localized by identifying the switches which become suddenly overloaded but experience little traffic demand increase. The mitigation strategy is to deactivate one of the afflicted ports or switches to restore a loop-free physical topology.

Unstable power: Failures due to unstable power are localized by searching syslogs for unexpected switch-down events. They can be mitigated by deactivating the switches impacted by unstable power.

Failures due to unknown reasons: Such failures account for 23% of all critical failures. Even if their root causes are unknown, they can be easily localized to a single switch and mitigated by a restart. For example, a switch that stops forwarding can be identified once the difference between its received and delivered bytes exceeds a threshold. It is also straightforward to identify a switch that loses its configuration or suffers from high CPU utilization.

5. NetPilot IMPLEMENTATION

NetPilot's primary implementation challenge is reliability. As a failure mitigation system, NetPilot itself must be robust to failures. We build NetPilot as a pipeline of five independent processes, as shown in Figure 10. These processes include a failure detector (§ 5.1), failure aggregator (§ 5.2), planner (§ 5.3), impact estimator (§ 5.4), and plan executor (§ 5.5). Each process records its relevant state to a replicated database so that the state can survive server crashes. Operators can also use the recorded state to determine ex post facto why NetPilot took specific actions.

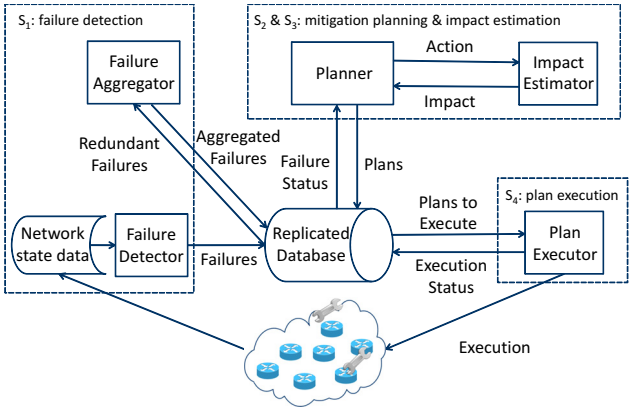

Figure 10: NetPilot implementation overview.

5.1 Failure Detector

The failure detector uses three data sources to detect failures: SNMP traps [22], switch and port counters, and syslogs [20]. The detector then applies failure-specific criteria to evaluate whether a failure has occurred. For example, the failure detector looks at the *bytes-in* and *dropped-packets* counters of a port to determine if a

link is overloaded. In our implementation, values from above data sources are processed every five minutes,

When the failure detector detects a failure, it updates the database with the following information: the type of detected failure, data sources used to detect the failure, and the components that exhibit abnormal behaviors. Note that these components are not necessarily the faulty components, because the failure effects may propagate to healthy components, *e.g.*, a broken link may cause overload and hence packet losses at other links.

5.2 Failure Aggregator

Because the failure detector runs continuously and NetPilot takes a trial-and-error approach, we expect that the same failure will be detected multiple times before it is mitigated. NetPilot therefore needs a mechanism to decide whether a detected failure instance is a new or ongoing failure.

The failure aggregator compares a newly reported failure instance against all the ongoing failures recorded in the database. If it determines that the newly reported instance has not been mitigated before – determined by the failure type and components involved – it updates the database and marks the failure as ready for mitigation. If it has seen the failure and the planner is taking a mitigation action, it marks the instance as requiring no further action. If it has seen the failure and the planner has taken a mitigation action for the failure, it flags the failure as unsuccessfully mitigated. The planner may then try the next mitigation action if there is one available.

The failure aggregator does not remove the failure instance created by the failure detector, but simply marks that it has been processed so that an operator can examine the initial failure detection as well as the choices made by the failure aggregator later on.

5.3 Planner

The planner takes three steps to choose a mitigation action. First, it employs failure-specific modules to localize a failure to a set of suspected components. Second, it generates the appropriate mitigation actions against all suspected components. Third, it uses the impact estimator to estimate the impact of these actions, ranks them based on their impact or success likelihood, and then executes the best one. At the end of each step, the planner updates the database with its computation results for post-analysis.

5.4 Impact Estimator

The impact estimator implements the algorithm presented in §4.2. It uses the run-time DCN topology and historical TMs to compute $online\_server\_ratio$, max_link_util, and $total\_lost\_pkt$. We extract the run-time topology from device configurations and running state (*i.e.*, up/down). It includes both the physical and logical device connections such as a LAG that comprises multiple physical links and a virtual switch that comprises multiple physical switches. The traffic matrices are continuously collected via socket event logs on each server and are aggregated to ToR-to-ToR traffic matrices at a 10-minute granularity.

5.5 Plan Executor

Once the planner chooses a mitigation action, the *plan executor* is engaged to take the action. The executor translates the action into a series of commands recognized by switches. As the commands are vendor-specific, we create a vendor-specific *configlet* file that includes the commands for each mitigation action. A *configlet* file parameterizes configuration arguments such as port number, so it can be reused to take the same action on different switches or ports. We also implement a library that allows the executor to send commands to switches via both in-band and out-of-band channels. After an action is taken, the executor updates the database to record the time when the action was taken and whether the action was successfully applied to the switch.

5.6 Interactions with Operators

NetPilot is fully capable of mitigating failures without human intervention. Nonetheless, NetPilot is explicitly designed to record the inputs and outputs of each mitigation step in a manner that is readily accessible to operators. Operators can later examine the decisions at each step. This design helps them debug and understand counterintuitive mitigation actions. Moreover, it helps reveal failures that are repeatedly mitigated for only a short period of time.

6. EVALUATION

In this section, we show that NetPilot can quickly and automatically mitigate several types of critical failures in DCNs. After presenting our experimental methodology, we first conduct three end-to-end experiments to highlight that, compared with the approach used by today's operators in $DCNs_p$, NetPilot can mitigate failures faster and with less disruption. Then we conduct more detailed experiments to illustrate the three reasons why NetPilot outperforms the status quo: effective failure localization, accurate impact estimation and action prioritization that minimizes disruption.

6.1 Experimental Methodology

We conduct four types of experiments to study various key aspects of NetPilot: 1) *failure-replay,* for failures with operation logs, we feed the device counters and traffic matrices when failures occurred in DCN_p into NetPilot and compare NetPilot's actions with the actions that were actually taken by the operators; 2) *heuristic-replay,* for failures without operation logs, we feed the device counters and traffic matrices when failures occurred in DCN_p into NetPilot and an *operator's approach* and compare their actions; 3) *testbed,* we inject traffic and failures into a testbed and compare the actions taken by NetPilot with those taken by operators; 4) *simulation,* we use large scale simulations to compare NetPilot and the operator's approach in the face of multiple simultaneous failures.

6.1.1 Operator's Approach

For failures with operation logs created by operators, we can directly compare NetPilot's actions with the actual operator's actions. For failures without operation logs, we build a model of how operators would mitigate the failures based on discussions with $DCNs_p$'s operators and our observations. We first carefully review the failures with operation logs and enumerate the typical action sequences for the types of failures discussed in §4.3. We then have operators explain the reasoning behind each sequence of actions. We summarize the operator's heuristics for mitigating each type of failure and estimating impact below.

Estimating impact: When deactivating a link or switch, all of its traffic will fail over evenly among its redundant components. Components not in the same redundancy group have no change in traffic. Because manual computation is slow and error-prone, operators cannot afford to compute the traffic changes that are more than one hop away from the deactivated component.

Mitigating FCS errors: Identify the switches having significantly more corrupted packets of outgoing than incoming, and then deactivate their ports in the descending order of error rate.

Mitigating *uneven-split*: Try restarting LAGs first and then switches, because operators believe restarting a switch is likely to have greater impact.

Mitigating *link-down*: Compute the impact of each possible action under the latest traffic loads using the impact estimation heuristic above, and then execute the action with the least impact.

6.1.2 *Experimental Setup*

Our data for the failure-replay and heuristic-replay experiments come from DCNs$_p$, several large production DCNs using the scale-out topology for which operators log critical failures with details and mitigation actions. We collected six months of device counters via SNMP at 5-minute intervals. We also logged the socket events on all servers during the same six months and aggregated the logs into ToR-to-ToR traffic matrices at a 10-minute granularity.

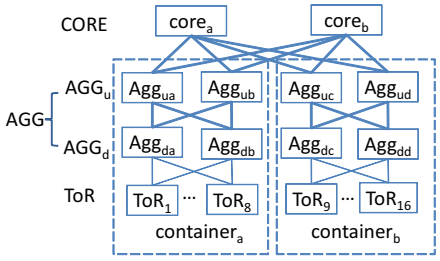

Figure 11: The testbed topology.

As shown in Figure 11, our testbed is a mini version of DCN$_p$'s scale-out topology with all the important characteristics preserved. It has a hierarchical structure with two containers. The connections between containers and *CORE*s form a full mesh running BGP ECMP for load balancing. We use multi-chassis LAG to virtualize the AGG_d switch pairs and VRRP to virtualize the AGG_u switch pairs in the same container. Each connection above AGG_d is a LAG with four 10Gb/s physical links and each connection between a *ToR* and an AGG_d is one 10Gb/s physical link. Unlike the topology in Figure 1, the testbed topology has 1) two *AGG* levels: AGG_u and AGG_d, and 2) one traffic generator instead of multiple servers under each ToR. The traffic generators can either inject arbitrary traffic or replay real server traffic traces captured in DCN$_p$.

Our simulator for the simulation experiments use the same topology and traffic matrix data as the failure-replay experiments. However, we inject hypothetical failures and compare the actions taken by NetPilot and the operator's approach.

6.2 End-to-End Failure Mitigation

In this section, we compare the state-of-the-art manual failure mitigation by operators with NetPilot for three failure incidents in DCNs$_p$. We are limited by the number of available operation logs to conduct a large-scale comparison. Nevertheless, from the failure incidents we examined, we expect that NetPilot's improvement shown in these examples is highly representative.

FCS error. Figure 7 shows the timeline of operators mitigating an FCS error incident. The operators iteratively tried deactivating links that had a significant error rate. Without an Impact Estimator, they performed manual calculations to ensure the remaining links would not become overloaded when they deactivated a link. It took a team of experienced operators nearly 3.5 hours and 10 unsuccessful trials to deactivate the two faulty links. In contrast, a failure-replay experiment shows that NetPilot can pinpoint and deactivate the two troublesome links in less than 15 minutes without any human intervention.

Overload due to *link-down*. Figure 8 depicts an incident of overloaded link caused by *link-down*. This incident persisted for almost 6 hours before the operators mitigated it by deactivating a LAG. The mitigation action was repeatedly delayed because the operator's manual impact estimation was inaccurate and informed them that deactivating the LAG would be worse than taking no action at all. After six hours of continually re-running their impact estimation, the operators deactivated the LAG around hour 16.

In contrast, a failure-replay experiment shows that NetPilot finds an alternative action that would have mitigated the incident soon after the failure was detected, which is almost 5.5 hours ahead of the operator's action. This is possible only because NetPilot's Impact Estimator is faster and more accurate and thus can exhaustively explore all options. This improved accuracy allows NetPilot to take actions that operators would have erroneously excluded.

Overload due to *uneven-split*. A known bug in the software that runs on the AGG_us causes them to occasionally stop generating routing updates. This in turn causes the *CORE*s to cease forwarding traffic to the afflicted AGG_us even though the links are up. Because it is difficult to consistently reproduce the exact same scenario in a testbed experiment, we emulate it by misconfiguring an AGG_u's ACL to block the TCP connections to the *CORE*s. Under this setting, *CORE*s' routing table entries with the afflicted AGG_u as the next hop will expire and most of the traffic will be sent to the other AGG_u in the same afflicted container.

NetPilot detects this problem as an *uneven-split* incident and attempts to mitigate it by restarting switches. Because we only install the ACL rules in the running configuration and not the startup configuration, the problem will be mitigated upon switch restart.

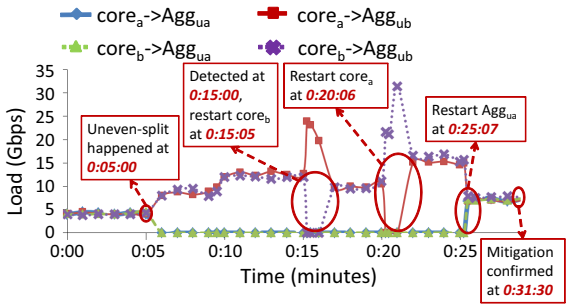

Figure 12: NetPilot mitigates link overloading due to *uneven-split* after restarting three switches.

Figure 12 shows the end-to-end mitigation process. We inject the *uneven-split* failure at minute 5 of the trace. Because switch counters are pushed to NetPilot once every 5 minutes, NetPilot responds at a granularity of 5 minutes. The first problem is identified at minute 15 because the failure aggregator waits for two consecutive counter values before declaring a failure. Approximately 5 seconds later, the planner finishes generating a list of possible actions, estimating the impact of each action via the Impact Estimator, and executing the one with the least impact. When the next set of counters gets pushed to NetPilot, the planner notices that the problem has not been mitigated and starts the next round of planning, excluding the actions that have already been taken. Five minutes later, the planner again notices the failure persists. As a result, it starts a third round of planning and mitigates the failure after restarting the third switch. Even though the first two actions are incorrect, NetPilot can mitigate the failure in approximately 20 minutes, which is still much faster than engaging the operators and manually restarting the suspected switches.

6.3 Fine-grained Failure Localization

In this section, we show that NetPilot can expedite the mitigation process using its fine-grained failure localization. We compare NetPilot's FCS model described in §4.3.1 against three al-

ternative approaches: 1) *Greedy deactivation*, in which links are deactivated in the order of decreasing error rate; 2) *Operator's approach*, which is described in §6.1.1; and 3) *Exhaustive search*, in which we try deactivating every possible combination of the links on the switches that violate the error conservation constraint, from one up to three links in each combination, in the order of decreasing combined error rate. If deactivating one combination fails to mitigate the failure, we roll back the ineffective link deactivations and try deactivating the next combination.

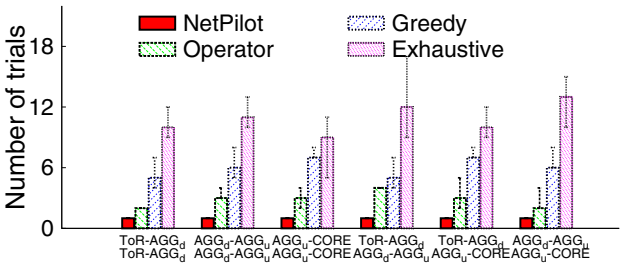

Figure 13: This figure compares the number of trials needed by different approaches to localize two simultaneously corrupted links in our testbed.

We design the first set of testbed experiments to show the number of attempts needed by each of the four algorithms to mitigate FCS errors. In each of the six experiments, we pick two links in the testbed and use a commercial FCS error injector to corrupt 1% of all packets that traverse the two links. Two simultaneous faulty links in a large-scale DCN is common, as there are tens or hundreds of thousands of links. We repeat each experiment ten times using different traffic matrices. The histogram in Figure 13 shows the median number of trials for each experiment, while the error bars mark the maximum and minimum number of trials.

In almost all cases, NetPilot can accurately locate the two corrupted links in one trial. In two out of the sixty experiments, NetPilot identifies three links, the two malfunctioning links and one false positive link. This is far more effective than the operator's approach that will cumulatively deactivate two to five links before disabling the corrupted links. On the other hand, because the exhaustive search will roll back the ineffective link deactivations, it will eventually mitigate the FCS errors without having any healthy link deactivated. However, the disruption caused to the network by the tens of trials is significant.

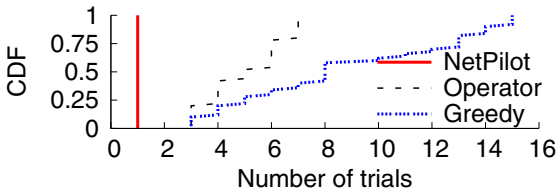

Figure 14: This figure shows the CDFs of the number of trials needed by different approaches to localize three simultaneously corrupted links. NetPilot significantly outperforms the others.

In the testbed experiments, we can only corrupt two links simultaneously due to the limitations of the FCS error injector. We use simulation to simulate more than two corrupted links in a much larger network, *i.e.*, DCN$_p$. We perform fifty simulations. In each simulation we randomly pick three links and set their corruption rates to be uniformly distributed between 1% and 5% (typically observed in DCN$_p$). NetPilot's FCS model can uniquely localize the three corrupted links in 96% of the cases. For the remaining 4%, NetPilot localizes the three corrupted links plus one false positive link. Figure 14 shows that compared to NetPilot, other approaches require far more trials to mitigate just three simultaneously corrupted links. We omit the results from exhaustive search because its search space is too large and the simulations cannot finish in a reasonable amount of time.

To study the localization accuracy of NetPilot's FCS model in the real world, we replay the traffic matrices and switch counters from 78 DCNs$_p$'s FCS error instances on NetPilot. NetPilot can generate unique solutions and accurately localize the corrupted links in over 90% of the instances. In the remaining 10% of the instances, the solutions are not unique because the failure-time traffic matrices do not provide sufficient constraints for the linear equations, *e.g.*, we cannot tell if a link is corrupted when there is no traffic on it. For these cases, we pick the solution with the smallest number of non-zero corruption rates, as discussed in Section 4.3.1. We find this approach works well with the maximum one-link false positives.

6.4 Accurate Impact Estimation

In §4.2, we showed that NetPilot can accurately estimate link utilizations when no mitigation action is taken. We now conduct testbed experiments to show that NetPilot can accurately predict link utilizations as well as packet losses after device deactivations. We compare NetPilot with the operator's approach under five types of component deactivation: a randomly selected physical link, a LAG between an *AGG$_d$* and an *AGG$_u$*, a LAG between an *AGG$_u$* and a *CORE* switch, an *AGG$_d$* switch, and an *AGG$_u$* switch. For each deactivation type, we repeat the experiments under 144 different traffic matrices generated as follows. First, we collect the ToR-to-ToR traffic matrices from DCN$_p$ at a 10-minute granularity for one day. Then we use a modulo-16 hash function (16 is the number of *ToR*s in the testbed) to map the *ToR*s in DCN$_p$ to the *ToR*s in the testbed. Finally, we map the ToR-to-ToR traffic matrices from DCN$_p$ to the testbed and scale down the traffic volume by a scaling factor. We use a scaling factor that leads to no packet loss in the testbed to study how well NetPilot estimates link utilizations. We use a different scaling factor that leads to packet losses to study how well NetPilot estimates packet losses.

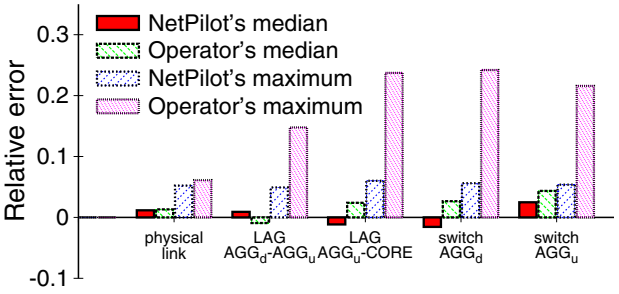

Figure 15: This figure compares the median and maximum relative errors of NetPilot's estimations of the maximum link utilization after deactivating various network components with those of the operator's manual approach. NetPilot has low estimation errors.

Figure 15 compares the relative errors of NetPilot's link utilization prediction with that of the operator's approach when there is no loss in the testbed. The relative error is defined as the difference between a predicted value and the value read from a switch counter normalized by a link's capacity: $\frac{prediction - switch\_counter}{link\_capacity}$.

NetPilot' median and maximum relative errors are less than 2.5% and 5% respectively. Although the median relative errors of the

operator's approach are only slightly higher, its maximum relative errors can often exceed 20%. The main reason is that the operator's approach cannot predict the significant traffic load increase on links that are multiple hops away from a deactivated component.

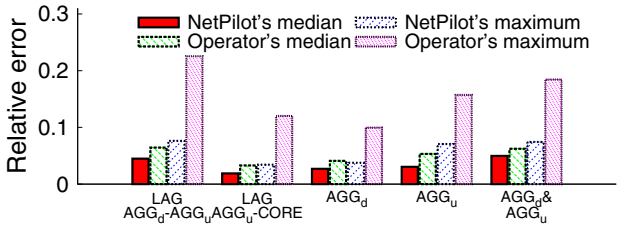

Figure 16: This figure compares the median and maximum relative errors of NetPilot's packet loss estimations with those of the operator's approach when deactivating various network components. Again NetPilot achieves low estimation errors.

Figure 16 is similar to Figure 15 except that we scale the traffic matrices to lead to packet losses in the network. We also replace the case of deactivating a single physical link with the case of deactivating both an AGG_d and AGG_u, since the former rarely leads to packet losses. Again NetPilot's loss prediction has median and maximum relative errors below 5% and 8% respectively. Yet, the maximum relative error of the operator's approach is always more than double NetPilot's.

6.5 Effective Action Planning

When mitigating a failure, NetPilot must carefully choose the order of actions to minimize network disruption. In this section, we show that correctly ordering mitigation actions is challenging and often contradicts the operator's intuition.

We first compare NetPilot with the operator's approach when mitigating *uneven-split* failures. We collect all the link utilizations at a 10-minute granularity from DCN$_p$ for one year and identify 151 *uneven-split* incidents by applying two criteria: 1) the difference between the maximum and mean link utilizations in the same LAG exceeds 5% (the same threshold used in §4.3.2); 2) the difference above lasts at least thirty minutes. Because some of these incidents did not cause link overload and thus were not investigated by operators, we do not know which component was responsible. Therefore, we randomly assign one "responsible" component to each incident and conduct heuristic-replay experiments.

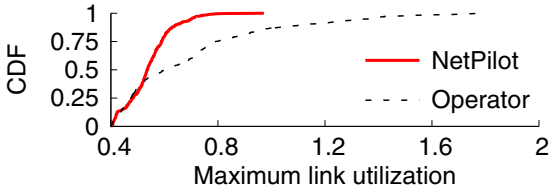

Figure 17: This figure compares the CDFs of the maximum link utilizations when NetPilot and operators restart components to mitigate the *uneven-split* failures. NetPilot causes lower maximum link utilizations during the failure mitigation period.

In these experiments, we emulate the operator's approach by first restarting LAGs and then switches (§6.1.1), whereas NetPilot exhaustively compares all possible actions under the latest TM and picks the one with the least impact (§4.3.2).

We use the CDFs of the maximum link utilizations under both approaches during the failure mitigation periods to compare NetPilot with the operator's approach, as shown in Figure 17. Because we randomly assign faulty components, it takes NetPilot and the operator's approach on average the same number of trials to successfully mitigate an *uneven-split* failure. Therefore, the failure mitigation periods under both approaches are roughly the same. The approach with lower maximum link utilizations is a better approach. For ease of presentation, we represent packet losses as link utilizations greater than 100%. As can be seen, for 60% of the incidents, NetPilot is far less disruptive than the operator's approach. NetPilot never overloads links while the operator's approach would lead to traffic losses in 20% of the incidents.

Unlike *uneven-split* failures, *link-down* failures are mitigated by deactivations in which the deactivated components may remain down for days. The operator's approach uses the latest TM right before the actions to estimate the impact of deactivations, and to choose the actions with the least negative impact. NetPilot has two main advantages over this approach: 1) it can use multiple TMs that better approximate future TMs in the deactivation periods to estimate impact; and 2) its impact estimation is more accurate (§ 6.4).

Although we have shown NetPilot's impact estimation is more accurate in the previous subsection, we further use a failure-replay experiment to show that more accurate impact estimation can lead to less disruptive mitigation actions. We replay 97 *link-down* incidents observed in DCNs$_p$ during one year. For each incident, NetPilot uses the same TM as the operator does to estimate the impact of a deactivation action. We then compare the resulting maximum link utilizations right after NetPilot's deactivation actions with those from the real failure traces.

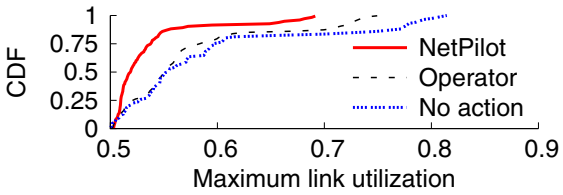

Figure 18: This figure shows the CDFs of the maximum link utilizations right after each method takes its deactivation action to mitigate *link-down* failures in DCN$_p$. NetPilot outperforms other methods because its impact estimation is more accurate.

Figure 18 shows the comparison results. As can be seen, NetPilot's actions are noticeably better than the actual actions taken by the operators or taking no action at all. These results suggest that NetPilot's higher estimation accuracy can lead to less disruptive mitigation actions.

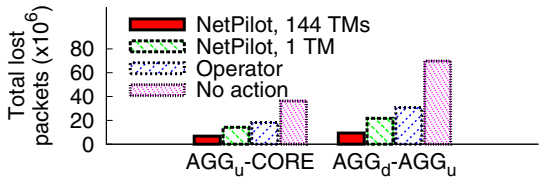

Figure 19: This figure compares the accumulated packet losses in a 24-hour period right after a deactivation action is taken in our testbed. When using multiple historical TMs to approximate future TMs, NetPilot leads to the fewest packet losses.

Next, we run testbed experiments to show that using multiple TMs to approximate future TMs during a device's deactivation period can also lead to less disruptive mitigation actions. In these experiments, we inject *link-down* failures in the testbed by shutting down half of the physical links in a LAG either between an

AGG_d and an AGG_u or between an AGG_u and a *CORE*. We then use the 144 TMs from the preceding 24 hours, rather than just the TM right before a deactivation action, to plan mitigation actions. We then measure the packet losses during the following 24 hours using the TMs in those hours.

Figure 19 presents the packet losses under each method. As can be seen, NetPilot has the fewest packet losses when using multiple historical TMs to approximate future TMs.

7. RELATED WORK

The design and implementation of NetPilot parallel the ideas from various fields. We group the related work into three categories: failure diagnosis, failure recovery, and what-if analysis.

Failure diagnosis: Automated failure diagnosis is a well-studied topic. Some systems identify failures in IP networks with active probes [6]. Others take measurement data from both end hosts and the network and build probabilistic models to localize the most likely components that are responsible for the observed failure data [15, 19]. Recent work has focused on developing systems that can pinpoint any failure that decreases application performance, whether it be hardware-related or software-related [5, 16]. There is also a large body of work on distributed system diagnosis [3, 7, 23]. NetPilot distinguishes itself by not attempting to find the exact root cause of a failure.

Failure recovery: The idea of automated failure recovery in DCs is not new. Isard [14] proposed an automated server management system called Autopilot based on the concept of recovery oriented computing [9]. When Autopilot detects a server is misbehaving, it takes one of three recovery actions: restart, reimage, or RMA (return merchandise authorization).

R3 [31] is a recovery service that can quickly mitigate link failures by pre-computing forwarding table updates for the loss of each link. Outside the networking domain, Total Recall [18] is a distributed storage system that adapts the amount of redundancy to compensate for host availability changes. Saxons is a peer-to-peer overlay service that can heal itself in the event of a partition [26]. NetPilot has different requirements from the systems above: failures that span multiple devices and constantly changing traffic loads make pre-computation prohibitively expensive. Also, NetPilot must minimize the adverse impact caused by mitigation.

What-if analysis: In order to make an informed decision, it is crucial for NetPilot to reason about the impact of possible mitigation actions – in essence a what-if analysis on the network. Recent work has explored the subject of what-if analysis for content distribution networks, with Tariq *et al.* [28] proposing a statistical approach and Wang *et al.* [30] proposing an empirical approach. Unlike the work above, NetPilot uses a what-if analysis technique that takes advantage of the unique properties of DCNs.

8. CONCLUSION

NetPilot is a system that automatically mitigates DCN failures. It is a departure from the status quo that relies heavily on human intervention. We believe that our work is critical to managing modern DCNs given the ballooning number of devices in these DCNs and the trend towards commodity hardware. NetPilot works by identifying a candidate set of afflicted components that are likely to cause a problem and iteratively taking mitigation actions targeting each one until the problem is alleviated. A key insight that makes this approach viable is the redundancy presented in modern DCN topologies. This redundancy reduces the potential for any single deactivated or rebooted component to disrupt a network. Our experiments show that NetPilot can successfully detect and mitigate several common types of failures both in a testbed and in a real production DCN.

9. ACKNOWLEDGMENTS

The authors would like to thank our shepherd Brad Karp and anonymous reviewers for their help in shaping this paper into its final form. We would also like to thank the infrastructure team at Microsoft Bing whose help was invaluable.

This work was largely done at Microsoft. Part of Xin Wu and Xiaowei Yang's work was supported by NSF awards CNS-0845858 and CNS-1040043.

10. REFERENCES

[1] Cut-Through and Store-and-Forward Ethernet Switching for Low-Latency Environments. `http://www.cisco.com/en/US/prod/collateral/switches/ps9441/ps9670/white_paper_c11-465436.pdf`.

[2] Virtual PortChannels: Building Networks without Spanning Tree Protocol. `http://www.cisco.com/en/US/prod/collateral/switches/ps9441/ps9402/white_paper_c11-516396.html`.

[3] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. SOSP '03.

[4] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM '08*.

[5] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *SIGCOMM '07*.

[6] D. Banerjee, V. Madduri, and M. Srivatsa. A Framework for Distributed Monitoring and Root Cause Analysis for Large IP Networks. In *SRDS '09*.

[7] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based failure and evolution management. In *NSDI '04*.

[8] W. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[9] D. P. et. al. Recovery Oriented Computing: Motivation, Definition, Techniques, and Case Studies. Technical report, Berkeley Computer Science, 2002.

[10] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: a scalable and flexible data center network. In *SIGCOMM '09*.

[11] U. Hoelzle and L. A. Barroso. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers.

[12] C. Hopps. Analysis of an Equal-Cost Multi-Path Algorithm. RFC 2992, 2000.

[13] IEEE. 802.3ad Link Aggregation Standard.

[14] M. Isard. Autopilot: automatic data center management. *SIGOPS Oper. Syst. Rev.*, 41:60–67, April 2007.

[15] S. Kandula, D. Katabi, and J.-P. Vasseur. Shrink: a tool for failure diagnosis in IP networks. In *MineNet '05*.

[16] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl. Detailed diagnosis in enterprise networks. In *SIGCOMM '09*.

[17] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of data center traffic: measurements & analysis. In *IMC '09*.

[18] R. B. Kiran, K. Tati, Y. chung Cheng, S. Savage, and G. M. Voelker. Total Recall: System Support for Automated Availability Management. In *NSDI '04*.

[19] Kompella, R.R and Yates, Jennifer, and Greenberg, Albert and Snoeren, Alex. IP Fault Localization Via Risk Modeling. In *NSDI '05*.

[20] C. Lonvick. The BSD Syslog Protocol. RFC 3164, 2001.

[21] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. PortLand: a scalable fault-tolerant layer 2 data center network fabric. In *SIGCOMM '09*.

[22] R. Presuhn. Management Information Base (MIB) for the Simple Network Management Protocol (SNMP). RFC 3418, 2002.

[23] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat. WAP5: black-box performance debugging for wide-area systems. In *WWW '06*.

[24] S. Nadas, Ericsson. Virtual Router Redundancy Protocol (VRRP) Version 3 for IPv4 and IPv6. *Internet RFC 5798*, 2010.

[25] A. Shaikh, C. Isett, A. Greenberg, M. Roughan, and J. Gottlieb. A case study of OSPF behavior in a large enterprise network. In *IMW '02*.

[26] K. Shen. Structure management for scalable overlay service construction. In *NSDI '04*.

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

[28] M. Tariq, A. Zeitoun, V. Valancius, N. Feamster, and M. Ammar. Answering what-if deployment and configuration questions with wise. In *SIGCOMM '08*.

[29] A. A. Team. Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region. `http://aws.amazon.com/message/65648/`.

[30] Y. Wang, C. Huang, J. Li, and K. Ross. Estimating the performance of hypothetical cloud service deployments: A measurement-based approach. In *INFOCOM '11*.

[31] Y. Wang, H. Wang, A. Mahimkar, R. Alimi, Y. Zhang, L. Qiu, and Y. R. Yang. R3: resilient routing reconfiguration. In *SIGCOMM '10*.