Plinko: Building Provably Resilient Forwarding Tables

Brent Stephens, Alan L. Cox, Scott Rixner
Rice University
{brents,alc,rixner}@rice.edu

ABSTRACT

This paper introduces Plinko, a network architecture that uses a novel forwarding model and routing algorithm to build networks with forwarding paths that, assuming arbitrarily large forwarding tables, are provably resilient against t link failures, $\forall t \in \mathbb{N}$. However, in practice, there are clearly limits on the size of forwarding tables. Nonetheless, when constrained to hardware comparable to modern top-of-rack (TOR) switches, Plinko scales with high resilience to networks with up to ten thousand hosts. Thus, as long as t or fewer links have failed, the *only* reason packets of *any* flow in a Plinko network will be dropped are congestion, packet corruption, and a partitioning of the network topology, and, even after $t + 1$ failures, *most*, if not all, flows may be unaffected. In addition, Plinko is topology independent, supports arbitrary paths for routing, provably bounds stretch, and does not require any additional computation during forwarding. To the best of our knowledge, Plinko is the first network to have all of these properties.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design; C.2.2 [**Computer-Communication Networks**]: Network Protocols—*Routing protocols*; C.4 [**Performance Of Systems**]: Fault tolerance

General Terms

Algorithms, Design, Reliability

1. INTRODUCTION

Simply put, the principal task of a network is to deliver data between endpoints that are allowed to communicate. Failures can and will occur, and a resilient network is one that is able to correctly perform its principal task despite failures. The ideal network is fully resilient (∞-resilient), which means it can correctly deliver data between all endpoints despite an arbitrary number of failures, as long as there exists a path between the endpoints in the (new) underlying topology. Unfortunately, almost all networks today are far from the ideal, at least for periods of time.

This paper introduces Plinko, a new network architecture that is based upon new theoretical advances on building provably fully resilient forwarding tables that are . If it were possible to build arbitrarily large Plinko forwarding tables, then it would be possible to build a network that would only drop packets due to congestion, packet corruption due to link or switch failures, or a partitioning of the network topology. Additionally, in Plinko, packet forwarding continues at line rate despite failures. While Plinko requires features not available on current datacenter switch hardware, we present a design for implementing a Plinko switch based on common switch primitives. Arbitrarily large tables are clearly impossible, so we also analyze the resource requirements of Plinko switches configured into realistic datacenter topologies, concluding that Plinko scales with high resilience to networks with up to ten thousand hosts.

Building a resilient network is a fundamental challenge in networking with many diverse solutions. For the sake of discussion, we define two classes of resiliency: control-plane resilience and data-plane resilience. The key difference between the two classes is the time scale at which they operate. A fully resilient network that requires the control-plane will experience outages during the time it takes to compute and install a route, while a resilient data-plane uses preinstalled backup rules and requires no action to adapt.

Historically, control-plane resilience has been sufficient for most networks. However, failures are expected as the norm in modern datacenters [4], and incurring any additional latency is becoming increasingly unacceptable [1, 21, 23]. Further, a recent study of datacenter network failures found that current approaches to network resilience are only 40% effective in reducing the median impact of failure [4]. This prior work suggests that failures in a datacenter with control-plane resilience noticeably impact performance.

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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Hotnets '13, November 21–22, 2013, College Park, MD, USA.
Copyright 2013 ACM 978-1-4503-2596-7 ...$10.00.

Unfortunately, previous work on data-plane resilience has either offered limited resilience, used inefficient paths, incurred additional latency, or relied on graph specific properties, which limits their usefulness. For example, Feigenbaum *et al.* [2] proved that a fully resilient data-plane is impossible for hop-by-hop routing if the packet cannot be modified. ECMP, IP Fast Re-route [3], and MPLS Fast Re-route [16] only provide limited resilience. Packet recycling [10] uses very inefficient paths. R-BGP [7] relies on graph-specific properties. FCP [8] requires expensive software computation in response to failures. DDC [9] guarantees connectivity but at least temporarily incurs significant stretch and can suffer from forwarding loops, although an IP TTL may terminate the forwarding of a packet.

Unlike prior work on data-plane resilience, Plinko takes a simple exhaustive approach, when is made possible by the Plinko forwarding model. At a high level, Plinko iteratively protects the paths in the network against the failure of any additional link each round. For every link in every route built in the previous round, Plinko builds a backup route, if one exists, that protects against the case that the link fails. In the case of a failure, the switch local to the failure replaces the old route of a packet with a backup route, effectively bouncing the packet around in the network until it either reaches the destination or is dropped because no path exists[1].

At a first glance, it would appear that state explosion would cause Plinko to have exorbitant state requirements that would limit its applicability to all but the smallest networks. However, we show the surprising result that the exhaustive approach can provide high resilience on topologies with up to ten thousand hosts with hardware comparable to modern switches. This is possible in part because the design of the Plinko forwarding model allows multiple Plinko forwarding rules to be compressed into a single TCAM entry.

In this paper, we present practical advances to data-plane resilience. The contributions of this paper are as follows:

Provably t-resilient forwarding tables: We introduce a new forwarding model and algorithm that can build t-resilient forwarding tables, $\forall t \in \mathbb{N}$, supports arbitrary paths, is topology independent, bounds stretch, and does not require additional computation during forwarding. *To the best of our knowledge, no other forwarding model has all of these properties.* Additionally, the forwarding model is notable for using rules that are compressible.

Design of a Plinko Switch: We present an architecture for implementing a Plinko switch and a heuristic for compressing the Plinko forwarding entries.

Analysis of Resource Requirements: We quantify the trade-offs between resilience and forwarding table state. We demonstrate that even the resources available in current commodity TOR switches can provide enough resilience on datacenter topologies with ten thousand hosts to be able to guarantee protection against over 90% of all of the failures seen in Microsoft datacenters [4], with most larger failures resulting from maintenance.

The remainder of the paper is organized as follows. Section 2 presents the Plinko forwarding model and algorithm. Next, Section 3 discusses the design of a Plinko switch, and Section 4 evaluates the state requirements of a Plinko network configured into realistic datacenter topologies. We briefly discuss Plinko in Section 5 and work that is closely related to Plinko in Section 6. Finally, we conclude and suggest future work in Section 7.

[1]The way a packet is bounced around in the network in response to failures is reminiscent to us of "The Price is Right" game Plinko.

2. PLINKO

This section introduces the Plinko forwarding model and routing algorithm and contains our main theoretical result. We first give a high level insight into the operation of a Plinko network. Then we formally define the Plinko forwarding model, present our algorithm for Plinko routing, and provide intuition into its correctness.

2.1 Overview

The Plinko routing algorithm starts by establishing all-to-all communication by installing a default source route for all source/destination pairs of hosts at the source's local switch, with each source route being a list of next hops. Unless one of the edges in the path fails, the the switches along the source route will each pop a hop off of the source route and push the input hop onto a reverse source route also in the packet header. In the case that one of the edges in a path does fail, the switch local to the failure enables resilience by discarding the original route and using the packet's header, which includes both the destination and a reverse route, to try to look up an alternate path in the local forwarding table.

The rest of the routing algorithm iteratively increases resilience by installing new alternate routes to protect the paths built in the previous round against the failure of any one additional edge. For every edge in the paths built in the previous round, a backup route that matches the header of packets following the path is added to the node local to the edge. The new entry's output is either a path to the destination that does not traverse any of the failed edges already encountered by the packet or a drop rule if no such path exists. Because all possible failures of an additional edge are considered in each round, by the end of round t, paths have been built that are resilient to all possible failures of t edges.

Although this appears simple, building recursive backups is not always possible without modifying packets [2]. If a packet encounters multiple failures in standard Ethernet, a switch cannot determine the exact set of failures encountered by the packet, which can cause backup routes to form loops. Thus, the principal challenge in choosing an output path in Plinko is determining the set of failed links that have been encountered by a packet, which Plinko solves by introducing a new forwarding model that matches on the reverse path of a packet, which is accumulated in its header. Because different forwarding paths are used during different failures

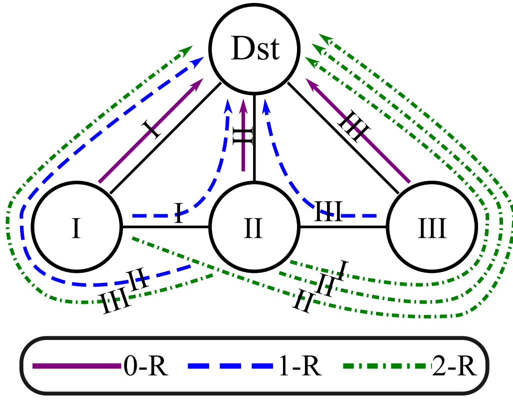


Figure 1: Fully resilient Plinko routes for Dst

and forwarding is deterministic, the reverse path identifies the set of failed edges already encountered by the packet.

Figure 1 illustrates the Plinko algorithm by showing fully resilient Plinko routes for the *Dst* node. In the figure, source routes are represented as arrows originating at the node containing the forwarding function entry. The level of resilience is shown by the linestyle and color, using 0-R, 1-R, and 2-R for patterns resilient to 0, 1, and 2 failures, respectively. For clarity, the paths are simply labeled with the original source node instead of the full reverse path of the packet.

In Figure 1, there are two 2-R entries for node *II*. This is because failure information is only known locally by Plinko forwarding hardware, i.e., switches only learn of the failure of local links, and the 1-R path [*II-I*, *I-Dst*] can fail at either the *II-I* link or the *I-Dst* link and there is still an operational path to *Dst*. Therefore, if links *II-Dst* and *I-Dst* are failed, switch *II* will still attempt to forward packets along the [*II-I*, *I-Dst*] path based on its local information. Only once packets reach switch *I* will the failed link *I-Dst* be learned, at which point the packet needs to be forwarded along the path [*I-II*, *II-III*, *III-Dst*]. While this leads to path stretch in our example, the stretch tends to be minimal in practice.

2.2 Model

We use a network model that is extended from that used by Feigenbaum *et al.* [2]. Formally, a network is modeled as an undirected graph $G = (V, E)$, where $V = \{1, \ldots, n\}$ and $\{u, v\} \in E$. We define $E_v = \{\{u, v\} \in E; u, v \in V\}$ as the set of edges local to vertex $v \in V$. The powerset 2^{E_v} is a bitmask representing the failure status of the edges in E_v. E_v^* is the set of all paths starting at v, and $P_v^d \subseteq E_v^*$ is the set of all paths from v to d. $E(p[h])$ and $V(p[h])$ are the edge and source vertex of the h-th hop in a path p, respectively, and $p[h:1]$ is the reverse path of p from $V(p[h])$ to i.

In our model, each node $v \in V$ has a *forwarding function* $f_v(d, rp, bm) \to p$ that maps a packet's destination $d \in V$ and reverse path $rp \in E_v^*$, which we assume are located in headers, as a function of a bitmask $bm \in 2^{E_v}$ of the node's state to an arbitrary path $p \in P_v^d$ to the destination. For convenience, we also represent the forwarding function as

Forwarding Entries $(d, rp, F_v, e) \to$	p
(*Dst*, [ ], ∅, *II-Dst*) →	[*II-Dst*]
(*Dst*, [ ], {*II-Dst*}, *II-I*) →	[*II-I*, *I-Dst*]
(*Dst*, [ ], {*II-Dst, II-I*}, *II-III*) →	[*II-III*, *III-Dst*]
(*Dst*, [*II-I*], {*II-Dst*}, *II-III*) →	[*II-III*, *III-Dst*]
(*Dst*, [*II-III*], {*II-Dst*}, *II-I*) →	[*II-I*, *I-Dst*]

Table 1: *The forwarding function for* Dst *at node* II.

$f_v(d, rp, F_v, e) \to p$, where F_v is a set of local edges that must be failed and e is an edge that must be up, so as to compactly specify a set of bitmasks over the local incident edges. We call each entry in the forwarding function a route r, which is a structure that contains four fields: a destination $r.d \in V$, the output path $r.p \in P_v^d$, the reverse path of packets the entry matches $r.rp \in E_v^*$, and the set of failed links already encountered by packets that match the entry $r.fl \subseteq E$. We chose to use the reverse path of a packet in the forwarding function because it can be used to identify the set of failed edges already encountered by a packet, and we expect that rules with similar reverse paths have similar output paths, which allows for rules to be compressed.

As a concrete example, Table 1 shows the forwarding function for node *II* in Figure 1. Although we present the forwarding function entries symbolically, it is also possible to represent each entry as an opaque blob of bits suitable for matching in hardware.

In this paper, we are primarily concerned with the resilience of the forwarding pattern given that a set of edges $F \subseteq E$ has failed. We define $G^F = (V, E \setminus F)$ as the new graph defined if $F \subseteq E$ of the edges are removed, and $G^i = \{G^F : |F| = i\}$ as the set of all possible graphs formed by the failure of i edges. We define a *forwarding path* as a path in G^F that is defined by the forwarding pattern f. To formalize the degree of resilience of a forwarding pattern f, we say that a forwarding pattern is t-resilient if $\forall G^F \in G^i, \forall i \leq t$, (1) there exists a forwarding path from a node v to d in G^F if any route exists from node v to d in G^F, $\forall v, d \in V$, and (2) there are no infinitely long forwarding paths defined by f in G^F.

2.3 Algorithm

The algorithm for t-resilient routing in the Plinko forwarding model is shown in Algorithm 1. The rest of this section provides intuition into its correctness.

The first condition necessary for t-resilience is the existence of a forwarding path between all hosts after t arbitrary failures as long as a path remains in the underlying topology. To satisfy this condition, Plinko starts by building a single default route for every pair of hosts. Then, in rounds, Plinko increases the level of resilience by one by installing exactly one backup path for every edge built in the previous round, with the default paths being protected in the first round. This is possible because the reverse path of a packet includes the source hop of the packet, and, from a given source–be it

Algorithm 1 – Plinko routing for t-resilient forwarding

Input: network topology $G = (V, E)$ where $V = \{1, \ldots, n\}$
Output: a forwarding pattern $f = (f_1, \ldots, f_n)$. $\forall v \in V$, $f_v(d, rp, F_v, e) \rightarrow p$, $d \in V$, $rp \in E_v^*$, $F_v \subseteq E_v$, $e \in E_v \setminus F_v$, and $p \in P_v^d$

1. **Build the Default Routes:** $\forall v, d \in V, do$:
   - Let $p \in P_v^d$, $e = E(p[1])$, and r be a new route.
   - Set $r.d := d$, $r.p := p$, $r.rp := \varnothing$, and $r.fl := \varnothing$.
   - Set $f_v(r.d, r.rp, r.fl \cap E_v, e) := r.p$.
2. **Iteratively Fix Routes for an Additional Failure Each Round:** For round i in $\{1, \ldots, t\}$, do:
   (a) **Consider every edge in all the paths $r.p$ built in round $i - 1$.**
   - Let $R_{i-1} = \{r : |r.fl| = (i - 1)\}$.
   - $\forall r \in R_{i-1}$ and $\forall h \in \{1, \ldots, |r.p|\}$, do:
     i. **Build a backup path for $r.p$ assuming that the link at hop h failed.**
     - Let $v = V(r.p[h])$, $e = E(r.p[h])$, and $rp = r.p[h : 1]$.
     - If $\exists np \in P_v^d$ s.t. $np \cap (r.fl \cup \{e\}) = \emptyset$ do:
       – Let $e_{np} = E(np[1])$ and nr be a new route.
       – Set $nr.d := r.d$, $nr.p := np$, $nr.rp := rp + r.rp$ and $nr.fl := r.fl \cup \{e\}$.
       – Set $f_v(nr.d, nr.rp, nr.fl \cap E_v, e_{np}) := nr.np$.

the original switch or the switch local to a failure–there is only one possible path given the local failures. This means that the reverse path of any packet can be used to infer both the original source and the exact set of failures the packet has encountered. With this ability, Plinko can build backup routes for every path from the previous round that are guaranteed to not use any of the failed links already encountered by the packet, thus protecting the forwarding pattern against the failure of any single additional link.

The second condition for resilience is that paths terminate. In Plinko, the reverse and forward paths of a route are finite, and the reverse path of a packet increases as it is forwarded through the network. Therefore, the reverse path of a packet is guaranteed to eventually grow to a point such that there is no route in the network that can possibly match it.

2.4 Proof

Theorem 1. *For every network there exists a t-resilient Plinko forwarding pattern, $\forall t \in \mathbb{N}$.*

Lemma 1. *If routing is deterministic, then a reverse path $rp \in E_i^*$ uniquely identifies the set $F \subseteq E$ of failed edges previously encountered by a packet.*

Note that deterministic routing means that if two packets have the same destination, reverse path, and local edge bitmask, then they are guaranteed to match the same forwarding entry and that a packet only encounters a failed edge if that edge is along the path the packet is following. The proof of Lemma 1 proceeds by contradiction. First, assume that there is a reverse path rp that can represent two different sets of failed links, F_1 and F_2. Let $e \in (F_1 - F_2)$ be a failed edge only present in one of the sets represented by rp. Because failed edges are only encountered when a packet is destined to be forwarded over the edge, then rp represents the reverse path of packets that were intended to be forwarded over e and packets that were never destined to be forwarded over e. However, because routing is deterministic, all packets that started at the same node with the same destination are guaranteed to hit the same forwarding entries which provide identical forward paths, so either all of the packets with reverse path rp were intended to be forwarded over e or none of them were, so there is a contradiction.

Given Lemma 1, the proof that Algorithm 1 builds a t-resilient forwarding pattern, which proves Theorem 1 by construction, requires two parts. The first part is to show that packets do not loop in the network. Because the forwarding entries all match on finite reverse paths and all forward paths are finite, a packet is guaranteed to be dropped after it has traversed more hops than the sum of the longest reverse path and the longest forward path. The second step is to show that, given t arbitrary failures, there exists a forwarding path from node i to d if they are connected in the topology. This proof follows by induction.

Base case: After step 1, it is clear that a 0-resilient forwarding pattern has been built. Every vertex has a destination forwarding pattern with a single entry that contains a route to the destination. If there are no failures in the network, then the default routes will correctly deliver traffic to the destination.

Inductive case: Starting with a t-resilient destination forwarding pattern, one iteration of step 2 will produce a $(t+1)$-resilient destination forwarding pattern. The proof is by contradiction. We first assume that there is a set of failed edges $F \subseteq E$ where $|F| = t+1$ such that there is not a forwarding path from a node v to the destination even though v is connected to the destination in the network topology. Note that the failure of a single edge causes a path to become invalid, so without loss of generality there must exist an edge e such that there exists a forwarding function entry that maps to a path that traverses e and the node local to e does not have a backup forwarding function entry installed for the reverse path in case e fails. From Lemma 1, we know that the reverse path of an entry uniquely identifies the set of failed links encountered by packets that match the entry. By the definition of t-resilience, all entries that have encountered less than t failures are resilient to the failure of any additional link, so the forwarding entry affected by the failure of e must be for a reverse path that has already encountered t failures. However, in step 2 we add a backup route, if one exists, for every edge along the paths used by every forwarding entry that has already encountered t failures, so there cannot exist an edge e that causes a node v to be disconnected in the forwarding pattern unless the topology is partitioned. Thus, there is a contradiction.

In addition to proving the resilience of Plinko, we can also bound the hop count of a forwarding path in Plinko, which bounds the stretch and the latency of packets forwarded by Plinko. The bound is shown in Theorem 2.

Theorem 2. *Let h be the length of a forwarding path for a Plinko forwarding pattern. If the length of longest path*

used by any forwarding function entry is lp and there are f failures, then $h \leq lp * (f + 1)$.

The proof of Theorem 2 proceeds by contradiction. First, assume that there exists a forwarding path defined by the forwarding pattern that encounters f failures and travels more than $lp * (f + 1)$ hops. Note that all paths lead to the destination, so unless a failure is encountered, a packet will be delivered to the destination after at most lp hops. There are then two cases where a packet can travel more than $lp * (f + 1)$ hops: (1) a packet encounters more than f failures and (2) a packet follows a path longer than lp hops, both of which are contradictions.

If we make the assumption that shortest paths are always used, then we can further bound the hop count. Recall that G^i is the set of all possible graphs formed by the failure of i edges, and let $D(G)$ denote the diameter of graph G. The hop count is then bounded by Theorem 3:

Theorem 3. *Let h be the length of a forwarding path. If shortest paths are used and f links have failed, then $h \leq \sum_{i=0}^{f} max(\{D(G) : G \in G^i\})$*

The proof of Theorem 3 is similar to that for Theorem 2, so it is omitted for space.

While Theorem 2 and Theorem 3 are written in terms of hop count, the switching latency at each hop in a real network is almost always a bounded function, so bounding the hop count also bounds total network latency.

3. IMPLEMENTATION

While the theoretical result of the resilience of the Plinko forwarding model is an important advance, the forwarding model and routing algorithm do not provide complete insight into how to implement a Plinko network. Although a full implementation of Plinko is beyond the scope of this work, this section discusses various factors involved with building a Plinko switch.

There are four main components necessary for building a Plinko network. The first is source routing, the second is building the reverse path, the third is the control plane, and the fourth is implementing the Plinko forwarding function. The first three components have been well studied, so we elide most of the details of their implementation, assuming that we implement source routing and reverse path building similar to the Axon [18] and use OpenFlow [11] to implement the control plane.

The principal difficulty in implementing a Plinko switch is then in efficiently implementing the Plinko forwarding function, and we only consider hardware implementations due to the increased latency of software forwarding. Also, although typical switches contain many different types of hardware tables, we currently only consider implementing Plinko with a single TCAM, hardware tables that match headers against wildcard patterns, outputting only the highest priority entry. The amount of TCAM state required by Plinko will be the dominant limiting factor of a Plinko network, and, like most hardware tables, TCAM state is measured by the number of bytes per entry and the total number of entries, which when combined give the total number of bytes of state.

10GbE Switch	HP Procurve 5400zl	Intel FM6000	IBM G8264
TCAM Size	35KB	110KB	~140KB

Table 2: 10 Gbps TOR Switch TCAM Sizes

Table 2 provides some context by showing the TCAM sizes of current TOR Ethernet datacenter switches [5, 6, 20]. Given current trends, we expect that these tables sizes will increase in future generations, especially since recent research has designed a switch with 4.75MB of TCAM state [14]. While these switches may not be immediately suitable for implementing Plinko, TCAMs are generic, so it is safe to assume that a Plinko switch could be built with a comparable amount of state.

Although the forwarding function entries in Algorithm 1 have a simple mapping to TCAM entries, mapping the Plinko forwarding function to a TCAM introduces an optimization problem: given a forwarding function, find a smaller set of TCAM entries and priorities that are equivalent to the forwarding function.

In our implementation, we use a combination of two heuristics to solve the TCAM packing optimization problem. The first heuristic, which we developed, greedily merges forwarding table entries, and the second heuristic is bit weaving [12]. We evaluated the two heuristics independently, and found that bit weaving on its own significantly underperforms the first heuristic, although combining the two provides modest improvements.

The first heuristic is based off of two observations. The first observation is that higher priority entries in a TCAM have to be finer in granularity so as to avoid matching packets intended for a lower priority entry. The second observation is that there is a greater chance for state reduction by favoring aggregating the largest set of rules that share a common output path first. Based on these two observations, the first heuristic first sorts the rules in descending order based on the size of the set of rules that share the same output path and then greedily attempts to aggregate entries into rules in a new TCAM. For each of the forwarding function entries, if modifying a rule already in the new TCAM to also match the entry from the old table does not result in a conflict with any of the lower priority TCAM rules in the new table, then the modification is performed. Otherwise, a TCAM rule that is identical to the forwarding function entry is created at the highest priority in the new table.

After applying the first packing heuristic, we then perform bit weaving [12] on the resulting TCAM. Bit weaving finds a bit swapping for a group of adjacent rules that allow the rules to be expressed as a LPM table. Next, it applies techniques for compressing LPM tables and then merges com-

patible rules into a ternary string. Lastly, the bit swapping is undone to restore the original rules.

Although prior work has demonstrated that installing only a single path for each source/destination pair can still provide good load balancing [20], NetLord [13] introduced techniques applicable to Plinko that can enable ECMP while simultaneously reducing state. The TCAM state is reduced by using encapsulation to perform routing on the switch topology instead of the host topology. ECMP is enabled by adding hosts to multiple VLANs. The physical hosts can then perform load balancing by either using deterministic hashing, as in ECMP, packet spraying, or MPTCP [22] to send packets on different VLANs. We consider Plinko implementations both with and without NetLord.

4. EVALUATION

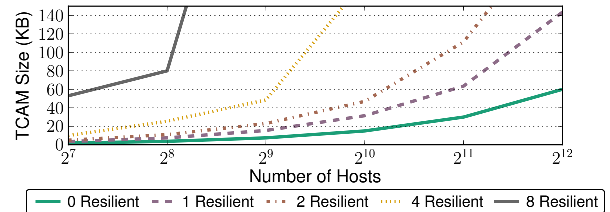
The primary aspect of Plinko that we evaluate is the forwarding table state requirements, which we only report as the total required kilobytes because the width of the entries were always less than 30 bytes per entry, smaller than the TCAM widths of modern switches. We find that Plinko, if used with NetLord, scales at 4-resilience to networks with 8K hosts. We also evaluated stretch, but we omit the results due to space, noting that stretch was typically negligible.

For our evaluation, we build all-to-all paths between the hosts in the topology. Networks that require less connectivity are expected to require proportionally less state. While Plinko supports arbitrary paths, we chose to build shortest paths in our evaluation so as to not artificially increase the required state per switch. Also, we reuse the paths of existing entries whenever possible to provide more opportunities for TCAM entry compression.

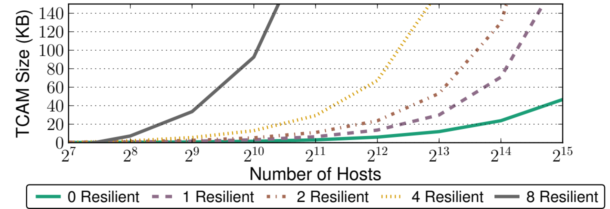
We use two types of realistic datacenter topologies in our evaluation of Plinko. The first type of topology is a $1:5$ bisection bandwidth Jellyfish topology [19], and the second type is a $1:1$ bisection bandwidth extended generalized fat tree (EGFT) [15]. For the EGFT topologies, we searched the space of all possible EGFTs to find the one with the fewest possible number of switches.

When considering TCAM state, we report only the maximum state required at any switch in the network. Current datacenter topologies, including the two that we consider, are designed to be implemented with (close to) identical TOR switches, so reporting the maximum captures the state requirements necessary for TOR switch to implement Plinko. Also, we make the pessimistic assumption that all entries need to be padded to the length of the longest entry. On some switches, such as the Intel FM6000 series, this is not the case, so the actual TCAM state would be reduced [6].

Figure 2a shows the number of bytes of TCAM state required for different levels of resilience on the EGFT topology. We omit the similar figure from the Jellyfish topology, noting that both the Jellyfish and the EGFT have similar state requirements, although the EGFT uses fewer hosts per switch and requires more switches. With edge protection,



(a) EGFT TCAM state



(b) Predicted Jellyfish TCAM state with 4-way ECMP NetLord

Figure 2: TCAM sizes of different Plinko networks

Plinko can provide a 4-resilient, 2-resilient, and 1-resilient forwarding pattern with less than 140KB of state for networks with 1024, 2048, and 4096 hosts, respectively. To provide context on the number of failures in a datacenter, a recent study [4] found that 59% of correlated failures only involve a single link, and 90% of correlated failures involve 4 links or fewer, with the bulk of the larger failure groups resulting from planned maintenance, which can be handled explicitly. Given these results, we can see that Plinko is able to support high levels of resiliency for modest size datacenters on hardware comparable to today's switches.

As discussed in Section 3, using NetLord with Plinko can reduce state requirements. To quantify the benefit of using NetLord with Plinko, we calculate the state given four routes per destination at each switch then fit them to second degree polynomials to predict the state requirements for topology sizes larger than 4096 hosts, which we have not fully considered. Figure 2b, which plots the prediction, shows that using NetLord with Plinko is expected to provide a 4-resilient, 2-resilient, and 1-resilient forwarding pattern with less than 140KB of TCAM state for networks with approximately 8K, 16K, and 27K physical hosts, respectively.

To demonstrate the effectiveness of TCAM compression, we present the compression ratios from the 4096 host Jellyfish topologies in Figure 2b. We saw that the compression ratio was 1.48, 2.08, 4.12, and 11.00 for 1-, 2-, 4-, and 8-resilience, respectively.

5. DISCUSSION

Although it may seem as though Plinko requires significant computation to install routes, implementing Plinko as an online algorithm is simple. When a host joins the network, Plinko only needs to compute and install 0-resilient routes for the host before connectivity is established, after

which the routes for additional resilience can then be computed and installed lazily. The task of computing 0-resilient routes is fundamental to all networks, so Plinko does not incur unnecessary latency.

Similarly, handling changes of the switch topology can be done safely and efficiently. Removing a switch is equivalent to the failure of its links, but after a topology change, we would like for other switches to use routes that do not attempt to traverse the now removed switch. Permanent topology changes from both switch removals and additions can be handled by periodically recomputing routes, as computational resources become available. Packet loss during updates can be completely avoided by performing provably consistent network updates [17].

Next, we note that a Plinko network that provides t-resilience does not immediately fail when there have been more than failures. Instead, the forwarding pattern shows a graceful degradation in performance because only paths that encounter $t + 1$ failures lose connectively.

Lastly, Plinko is well suited for enabling energy-proportional networking. With knowledge of both the traffic matrix and all of the paths in the network, a controller can find out the maximal set of switches that can be suspended, subject to resiliency constraints, and suspended switches simply appear as failed switches to the operating switches. The controller is also able to intelligently re-enable switches in the network as the load increases because it can compute the throughput benefits that can be gained by enabling different switches in the network, subject to the current traffic matrix and routes.

6. RELATED WORK

In addition to the work of Feigenbaum *et al.* [2], which formalized resilience and provided some initial results, Plinko is also related to other work. Due to the extent of the approaches to resilience, we limit our discussion to a few projects that are closely related to Plinko.

The most closely related work to Plinko is Failure Carrying Packets, or FCP [8]. Like Plinko, FCP uses failure information accumulated in a packet's header to guarantee a fully resilient network, although in FCP, only the IDs of failed links are added to the packet, while Plinko accumulates the full reverse path of the packet. The key difference between Plinko and FCP is that, unlike Plinko, FCP requires the control plane of the router local to a failure to dynamically recompute routes and perform forwarding on a per-packet basis, which can add latency and reduce the maximum forwarding rate. Plinko builds upon the work of FCP by precomputing all of the backup routes and installing them in a forwarding table.

Packet Re-cycling (PR) [10] is also closely related to Plinko. Although PR can pre-compute forwarding tables that are fully resilient and only require and additional $log_2(d)$ bits of data in a packet's headers, PR cannot use arbitrary paths. Additionally, path lengths in PR are typically far from minimal in the presence of failures. This is because PR routes around failures in a manner akin to solving a labyrinth by the right-hand rule.

DDC [9], or data-driven connectivity, is a closely related project that provably provides *ideal forwarding-connectivity*, which only guarantees that a packet will reach its destination as long as the network remains physically connected. DDC achieves ideal forwarding-connectivity by performing provably safe link-reversals. DDC repeats link-reversal operations until the forwarding DAG converges , which, for n switches, is guaranteed to occur after $O(n^2)$ reversal operations.

The key differences between Plinko and DDC are that DDC can temporarily incur significant latency, and, in DDC, packets on the side of a partition that is not connected to the packet's destination will persistently be forwarded in loops until either the control plane detects the partition and deletes routes, a TTL in the packet, if any, expires, or the packet is dropped due to congestion. In contrast, Plinko guarantees that packets will be dropped in the event of a partition.

7. CONCLUSIONS

In conclusion, we introduced Plinko, a network that uses a new provably resilient forwarding model. We evaluated the state requirements of a simple implementation of Plinko, and found that it can build a 4-resilient network for topologies with up to about ten thousand hosts.

As future work, we plan to explore implementing Plinko with a pipeline of multiple hardware tables similar to those found in today's switches, which could improve scalability. Additionally, FCP [8] introduced another provably resilient forwarding model, although they did not consider implementing it in hardware. We plan to consider hardware implementations of FCP, which we will compare and contrast against Plinko. Lastly, we plan to extend our evaluation to include stretch, forwarding throughput, and additional failure models.

8. REFERENCES

[1] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *SIGCOMM*, 2010.

[2] J. Feigenbaum, P. B. Godfrey, A. Panda, M. Schapira, S. Shenker, and A. Singla. On the resilience of routing tables. In *Brief announcement, 31st Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, July 2012.

[3] P. Francois and O. Bonaventure. An evaluation of ip-based fast reroute techniques. In *CoNext*, 2005.

[4] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *SIGCOMM*, 2011.

[5] HP Procurve OpenFlow support. http://www.openflow.org/wp/

wp-content/uploads/2011/04/HP_Procurve_OpenFlow_support.pdf.

[6] Intel Ethernet Switch FM6000 Series - Software Defined Networking. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ethernet-switch-fm6000-sdn-paper.pdf.

[7] N. Kushman, S. Kandula, D. Katabi, and B. M. Maggs. R-bgp: staying connected in a connected world. In *NSDI*, 2007.

[8] K. Lakshminarayanan and T. Anderson. Achieving convergence-free routing using failure-carrying packets. In *SIGCOMM*, 2007.

[9] J. Liu, A. Panda, A. Singla, P. B. Godfrey, M. Schapira, and S. Shenker. Ensuring connectivity via data plane mechanisms. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2013.

[10] S. S. Lor, R. Landa, and M. Rio. Packet re-cycling: eliminating packet losses due to network failures. In *HotNets*, 2010.

[11] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, 2008.

[12] C. R. Meiners, A. X. Liu, and E. Torng. Bit weaving: a non-prefix approach to compressing packet classifiers in TCAMs. *IEEE/ACM Trans. Netw.*, 20(2), Apr. 2012.

[13] J. Mudigonda, P. Yalagandula, J. C. Mogul, B. Stiekes, and Y. Pouffary. NetLord: a scalable multi-tenant network architecture for virtualized datacenters. In *SIGCOMM*, 2011.

[14] Nick McKeown. Protocol Independence. http://www.opennetsummit.org/pdf/2013/presentations/nick_mckeown.pdf.

[15] S. Ohring, M. Ibel, S. Das, and M. Kumar. On generalized fat trees. *Parallel Processing Symposium, International*, 0:37, 1995.

[16] P. Pan, G. Swallow, and A. Atlas. RFC 4090 Fast Reroute Extensions to RSVP-TE for LSP Tunnels, May 2005.

[17] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. SIGCOMM, 2012.

[18] J. Shafer, B. Stephens, M. Foss, S. Rixner, and A. L. Cox. Axon: A flexible substrate for source-routed Ethernet. In *ANCS*, 2010.

[19] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey. Jellyfish: Networking data centers randomly. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2012.

[20] B. Stephens, A. Cox, W. Felter, C. Dixon, and J. Carter. PAST: Scalable ethernet for data centers. In *CoNext*, 2012.

[21] B. Vamanan, J. Hasan, and T. N. Vijaykumar. Deadline-aware datacenter tcp (d2tcp). In *SIGCOMM*, 2012.

[22] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, implementation and evaluation of congestion control for multipath TCP. *NSDI*, 2011.

[23] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. DeTail: reducing the flow completion time tail in datacenter networks. In *SIGCOMM*, 2012.