Aspen Trees: Balancing Data Center Fault Tolerance, Scalability and Cost

Meg Walraed-Sullivan
Microsoft Research
Redmond, WA
megwal@microsoft.com

Amin Vahdat
Google, UC San Diego
Mountain View, CA
vahdat@cs.uscd.edu

Keith Marzullo
UC San Diego
La Jolla, CA
marzullo@cs.uscd.edu

ABSTRACT

Fault recovery is a key issue in modern data centers. In a fat tree topology, a single link failure can disconnect a set of end hosts from the rest of the network until updated routing information is disseminated to every switch in the topology. The time for re-convergence can be substantial, leaving hosts disconnected for long periods of time and significantly reducing the overall availability of the data center. Moreover, the message overhead of sending updated routing information to the entire topology may be unacceptable at scale. We present techniques to modify hierarchical data center topologies to enable switches to react to failures locally, thus reducing both the convergence time and control overhead of failure recovery. We find that *for a given network size*, decreasing a topology's convergence time results in a proportional decrease to its scalability (e.g. the number of hosts supported). On the other hand, reducing convergence time *without affecting scalability* necessitates the introduction of additional switches and links. We explore the tradeoffs between fault tolerance, scalability and network size, and propose a range of modified multi-rooted tree topologies that provide significantly reduced convergence time while retaining most of the traditional fat tree's desirable properties.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Network topology, Packet-switching networks; C.2.2 [**Network Protocols**]: Routing protocols

Keywords

Data Center Networks; Fault Tolerance; Network Redundancy

1. INTRODUCTION

Modern data center networks are often structured hierarchically. One of the most common topologies for such network fabrics is a multi-rooted fat tree [1, 4, 30], inspired by [24]. This topology is popular in part because it can support full bisection bandwidth, and it also provides diverse yet short paths between end hosts. In our experience operating large-scale data center infrastructure, and as shown in recent studies [9], a key difficulty in the data center is handling faults in these hierarchical network fabrics.

Despite the high path multiplicity between hosts in a traditionally defined fat tree, a single link failure can temporarily cause the loss of all packets destined to a particular set of hosts, effectively disconnecting a portion of the network. For instance, a link failure at the top level of a 3-level, 64-port fat tree tree can logically disconnect as many as 1,024, or 1.5%, of the topology's hosts. This can drastically affect storage applications that replicate (or distribute) data across the cluster; there is a significant probability that the failure of an arbitrary 1.5% of hosts could cause the loss of all replicas (or pieces) of a subset of data items, and the storage overhead required to prevent these types of loss could be expensive. Additionally, recent studies [9] show that one third of data center link failures disrupt ongoing traffic, causing the loss of small but critical packets such as acknowledgments and keep-alives. It is crucial then, that re-convergence periods be as short as possible.

However, it can take substantial time to update switches to route around failures. For instance, the time for global re-convergence of the broadcast-based routing protocols (e.g. OSPF and IS-IS) used in today's data centers [3, 29] can be tens of seconds [25, 26]. As each switch receives an update, its CPU processes the information, calculates a new topology and forwarding table, and computes corresponding updates to send to all of its neighbors. Embedded CPUs on switches are generally under-powered and slow compared to a switch's data plane [26, 28] and in practice, settings such as protocol timers can further compound these delays [22]. While more powerful switch CPUs are in the works, they are not yet a reality in the data center. The processing time at each switch along the path from a point of failure to the farthest switches adds up quickly. Packets continue to drop during this re-convergence period, crippling applications until recovery completes. Moreover, at data center scale, the control overhead required to broadcast updated routing information to all nodes in the tree can be significant.

Long convergence times are unacceptable in the data center, where the highest levels of availability are required. For instance, an expectation of 5 nines (99.999%) availability corresponds to about 5 minutes of downtime per year, or 30 failures, each with a 10 second re-convergence time. A fat tree that supports tens of thousands of hosts can have hundreds of thousands of links[1] and recent studies show that at best, 80% of these links have 4 nines availability [9]. In an environment in which link failures occur quite regularly, restricting the annual number of failures to 30 is essentially impossible.

Our goal is to eliminate excessive periods of host disconnection and packet loss in the data center. Since it is unrealistic to limit the number of failures sufficiently to meet availability requirements, we consider the problem of drastically reducing the re-convergence

[1]Even a relatively small 64-port, 3-level fat tree has 196,608 links.

CoNEXT'13, December 9-12, 2013, Santa Barbara, California, USA.

http://dx.doi.org/10.1145/2535372.2535383.

time for each individual failure, by modifying fat trees to enable *local failure reactions*. These modifications introduce redundant links (and thus a denser interconnect) at one or more levels of the tree, in turn reducing the number of hops through which routing updates propagate. Additionally, instead of requiring global OSPF convergence after a link failure, we send simple failure (and recovery) notification messages to a small subset of switches located near to the failure. Together, these techniques substantially decrease reconvergence time (by sending small updates over fewer hops) and control overhead (by involving considerably fewer nodes and eliminating reliance on broadcast). We name our modified fat trees *Aspen trees*, in reference to a species of tree that survives forest fires due to the locations of its redundant roots.

The idea of incorporating redundant links for added fault tolerance in a hierarchical topology is not new. In fact, the topology used in VL2 [11] is an instance of an Aspen tree. However, to the best of our knowledge, there has not yet been a precise analysis of the tradeoffs between fault tolerance, scalability, and network size across the range of multi-rooted trees. Such an analysis would help data center operators to build networks that meet customer SLAs while satisfying budget constraints. As [9] shows, this is missing in many of today's data centers, where even with added network redundancy, failure reaction techniques succeed for only 40% of failures.

We explore the benefits and tradeoffs of building a highly available large-scale network that reacts to failures locally. We first define Aspen trees, and present an algorithm that generates a set of Aspen trees given constraints such as the number of available switches or requirements for host support. We couple this design with a failure reaction protocol that leverages an Aspen tree's redundant links. To precisely specify the fault tolerance properties of Aspen trees, we introduce a *Fault Tolerance Vector (FTV)*, which quantifies reactivity by indicating the quality and locations of added fault tolerance throughout an Aspen tree.

Engineering topologies to support local failure reactions comes with a cost, namely, the tree either supports fewer hosts or requires additional switches and links. We formalize these tradeoffs in terms of an Aspen tree's FTV. Interestingly, improving fault tolerance by adding switches and links to a tree (while keeping host count fixed) has the potential to do more harm than good by introducing more points of failure. However, we show that the decreased convergence time enabled by local failure reaction more than makes up for the added opportunity for link failures. Finally, we use simulations to further explore the tradeoffs between fault tolerance, scalability, and network size for a variety of Aspen trees. Through analysis and simulations, we provide data center architects insight into the tradeoffs associated with Aspen trees, enabling them to design networks that balance their requirements for scale, cost and fault tolerance.

2. MOTIVATION AND CONTEXT

Before describing Aspen trees and their associated tradeoffs in detail, we first lay out some of the challenges inherent to failure recovery in multi-rooted fat tree topologies. We explore some existing approaches to mitigating the effects of link failures, in order to lend intuition to the rationale behind our Aspen tree design. As many modern data centers are structured as multi-rooted trees, we focus our attention on challenges inherent to these topologies.

In a traditional fat tree,[2] a single link failure can be devastating, causing all packets destined to a set of hosts to be dropped while updated routing state propagates to *every switch in the topology*. For

[2]We refer to the multi-fat tree presented in [1], in which channel capacities are uniform at all levels, as opposed to the fat tree in [24] in which link capacity increases moving up the tree.

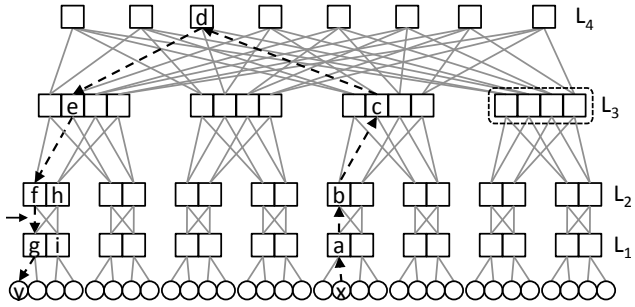


Figure 1: Packet Travel in a 4-Level, 4-Port Fat Tree

instance, consider a packet traveling from host x to host y in the 4-level, 4-port fat tree of Figure 1 and suppose that the link between switches f and g fails shortly before the packet reaches f. f no longer has a downward path to y and drops the packet. In fact, with the failure of link $f-g$, the packet would have to travel through h to reach its destination. For this to happen, x's ingress switch a would need to know about the failure and to select a next hop accordingly, as every (shortest) path from b to y traverses link $f-g$.

In other words, knowledge of a single link failure needs to propagate to all of the lowest level switches of the tree, passing through every single switch in the process. Each switch performs expensive calculations that grow with the size of the topology, leading to substantial delay before the last switches are updated. Isolated link failures are both common and impactful in the data center [9], so timely reaction is critical.

A number of existing routing techniques can help in avoiding packet loss without requiring global re-convergence. For instance, bounce routing protocols work around failures by temporarily sending packets away from a destination. In the example of Figure 1, such a protocol might send the packet from f to i. Switch i can then bounce the packet back up to h, which still has a path to g. Thus, bounce routing leverages more sophisticated software to support the calculation and activation of extra, non-shortest path entries and to avoid forwarding loops. Unfortunately, in networks that employ link-level pause (e.g. lossless fabrics like InfiniBand), bounce routing based on local information can lead to deadlock [6, 17].

It is also possible to "bounce" a packet in the other direction, back along its path to the nearest switch that can re-route around a failed link, similar to the technique of DDC [26]. In DDC, the packet of Figure 1 would need to travel from f back up to the top of the tree and then back down to a before it could be re-routed towards h. DDC provides "ideal connectivity," ensuring that a packet is not dropped unless the destination is physically unreachable, at the cost of (temporarily) introducing long paths. F10 [27] also bounces packets on failure, using a novel asymmetrical wiring scheme to limit the lengths of bounce paths. F10 includes three cascading failure reaction mechanisms, for failures of different durations. We more closely compare Aspen trees to DDC and F10 in § 10.

Our approach, on the other hand, offers an alternative to bouncing packets in either direction. We modify a fat tree by introducing redundancy at one or more levels. This allows local failure reaction without introducing long paths or requiring global re-convergence, at a cost in either scale or network size.

3. ASPEN TREE OVERVIEW

We define an Aspen tree as a set of k-port switches connected to form a multi-rooted hierarchy of n levels, with hosts connected to the leaf switches in this hierarchy. An n-level, k-port Aspen tree can

differ from a traditionally-defined n-level, k-port fat tree in that the interconnect between adjacent levels of the tree may be denser, that is, there may be more links between switches at adjacent levels than in a fat tree. The density of this interconnect can vary from level to level in an Aspen tree. Thus for any n and k, there exists a *set* of topologically distinct Aspen trees. A traditionally-defined n-level, k-port fat tree is member of the set of n-level, k-port Aspen trees; the fat tree of Figure 1 is one such example. We are not proposing a particular instance of Aspen trees as an ideal topology in the general case. Instead, our goal is to show the range of possible Aspen trees given requirements for host support, network size, failure reaction time, and overhead.

We denote switches' levels in an n-level, k-port Aspen tree with L_1 through L_n (as marked in Figure 1) and we refer to the hosts' level as L_0. Each switch has k ports, half of which connect to switches in the level above and half of which connect to switches below. Switches at L_n have k downward-facing ports. We group switches at each level L_i into *pods*.[3] A pod includes the maximal set of L_i switches that all connect to the same set of L_{i-1} pods below, and an L_1 pod consists of a single L_1 switch. An example L_3 pod is circled in Figure 1.

In order to limit our attention to a tractable set of options, we introduce a few restrictions on the set of Aspen trees that we will discuss. First, we consider only trees in which switches at each level are divided into pods of uniform size. That is, all pods at L_i must be of equal size, though L_i pods may have different sizes than pods at $L_{f:f\neq i}$. Similarly, within a single level, all switches have equal fault tolerance (i.e. equal numbers of links) to neighboring pods in the level below, but the fault tolerance of switches at L_i need not equal that of switches at $L_{f:f\neq i}$. In general, we use L_i fault tolerance to refer to links between switches at L_i and L_{i-1}. In a traditional fat tree there are S switches at levels L_1 through L_{n-1} and $\frac{S}{2}$ switches at L_n; we retain this property in Aspen trees. (In Figure 1, k is 4 and S is 16.) We do not consider multi-homed hosts, given the associated addressing complications.

4. DESIGNING ASPEN TREES

We now describe our method for generating trees with varying fault tolerance properties; this lends intuition to the set of Aspen trees that can be created, given a set of constraints such as network size or host support. Intuitively, our approach is to begin with a traditional fat tree, and then to disconnect links at a given level and "repurpose" them as redundant links for added fault tolerance at the same level. By increasing the number of links between one subset of switches at adjacent levels, we necessarily disconnect another subset of switches at those levels. These newly disconnected switches and their descendants are deleted, ultimately resulting in a decrease in the number of hosts supported by the topology.

[3] In some literature, these are called *blocks* [12, 23].

Figure 2 depicts this process pictorially. In Figure 2(a), L_3 switch s connects to four L_2 pods: q=$\{q_1,q_2\}$, r=$\{r_1,r_2\}$, t=$\{t_1,t_2\}$, and v=$\{v_1,v_2\}$. To increase fault tolerance between L_3 and L_2, we will add redundant links from s to pods q and r. We first need to free some upward facing ports from q and r, and we choose the uplinks from q_2 and r_2 as candidates for deletion because they connect to L_3 switches other than s.

Next, we select L_3 downlinks to repurpose. Since we wish to increase fault tolerance between s and pods q and r, we must do so at the expense of pods t and v, by removing the links shown with dotted lines in Figure 2(b). For symmetry, we include switch w with s. The repurposed links are then connected to the open upward facing ports of q_2 and r_2, leaving the right half of the tree, hosts *and* switches, disconnected and ready for deletion, as shown in Figure 2(c). At this point, s is connected to each L_2 pod via two distinct switches and can reach either pod despite the failure of one such link. We describe this tree as 1-*fault tolerant* at L_3.[4]

For a tree with a given depth and switch size, there may be multiple options for the fault tolerance to add at each level, and fault tolerance can be added to any subset of levels. Additionally, decisions made at one level may affect the available options for other levels. In the following sections, we present an algorithm that makes a coherent set of these per-level decisions throughout an Aspen tree.

4.1 Aspen Tree Generation

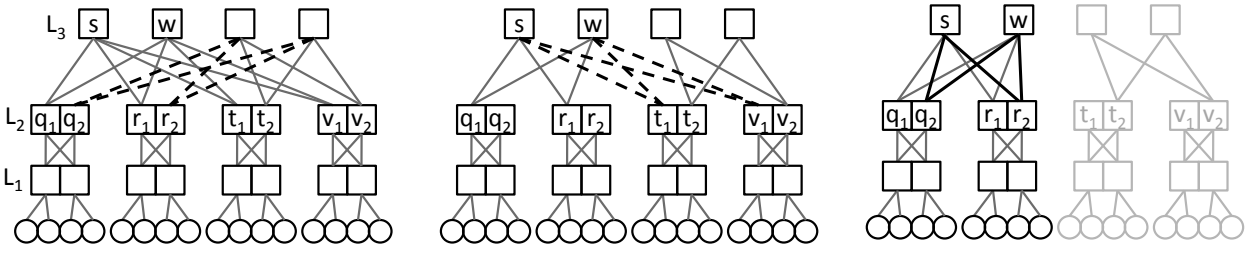
Intuitively, we begin at the top level of the tree, L_n, and group switches into a single pod. We then select a value for the fault tolerance between L_n and the level below, L_{n-1}. Next, we move to L_{n-1}, divide the L_{n-1} switches into pods, and choose a value for the fault tolerance between L_{n-1} and L_{n-2}. We repeat this process for each level moving down the tree, terminating when we reach L_1. At each level, we select values according to a set of constraints that ensure that all of the per-level choices together form a coherent topology.

4.1.1 Variables and Constraints

Before presenting the details of our algorithm, we first introduce several variables and the relationships among them. Recall that an Aspen tree has n levels of switches, and that all switches have exactly k ports. In order for the uplinks from L_i to properly match all downlinks from L_{i+1}, to avoid over-subscription, the number of switches at all levels of the tree except L_n must be the same.[5] We denote this number of switches per level with S. Each L_n switch has twice as many downlinks (k) as the uplinks of an L_{n-1} switch ($\frac{k}{2}$) and so for L_{n-1} uplinks to match L_n downlinks, there are $\frac{S}{2}$ L_n switches.

[4] In Figure 2(c) we could have connected s (w) to q_1 (q_2) via a second link rather than to q_2 (q_1). This is a special case of what we call *striping*, which we discuss in § 7.

[5] Aspen trees can be generalized to any number of (variable size) switches at each level. For brevity, we consider only symmetric trees.



(a) Freeing Uplinks from L_2 (b) Selecting L_3-L_2 Links to Repurpose (c) Reconnecting Redundant Links

Figure 2: Modifying a 3-Level, 4-Port Fat Tree to Have 1-Fault Tolerance at L_3

At each level L_i, our algorithm groups switches into pods and selects a fault tolerance value to connect L_i switches to L_{i-1} pods below. We represent these choices with four variables: p_i, m_i, r_i and c_i. The first two encode pod divisions; p_i indicates the number of *pods* at L_i, and m_i represents the number of *members* per L_i pod. Relating p_i and m_i to the number of switches per level, we have:

$$p_i m_i = S, 1 \leq i < n \qquad p_n m_n = \frac{S}{2} \tag{1}$$

The variables r_i and c_i relate to per-level fault tolerance. r_i expresses the *responsibility* of a switch and is a count of the number of L_{i-1} pods to which each L_i switch connects. c_i denotes the number of *connections* from an L_i switch s to each of the L_{i-1} pods that s neighbors. Since we require (§ 3) that switches' fault tolerance properties are uniform within a level, a switch's downward links are spread evenly among all L_{i-1} pods that it neighbors. Combining this with the number of downlinks at each level, we have the constraint:

$$r_i c_i = \frac{k}{2}, 1 < i < n \qquad r_n c_n = k \tag{2}$$

Each constraint listed thus far relates to only a single level of the tree, so our final equation connects adjacent levels. Every pod q below L_n must have a neighboring pod above, otherwise q and its descendants would be disconnected from the graph. This means that the set of pods at $L_{i:i\geq 2}$ must "cover" (or rather, be responsible for) all pods at L_{i-1}:

$$p_i r_i = p_{i-1}, 1 < i \leq n \tag{3}$$

An Aspen tree is formally defined by a set of per-level values for p_i, m_i, r_i and c_i, such that constraint Equations 1 through 3 hold, as well as by a *striping* policy for specifying switch interconnection patterns. We defer a discussion of striping until § 7.

4.1.2 Aspen Tree Generation Algorithm

We now use Equations 1 through 3 to formalize our algorithm, which appears in pseudo code in Listing 1. The algorithm calculates per-level (line 6) values for p_i, m_i, r_i, c_i and S (lines 1-5) given the number of downlinks (line 7) at each level.

We begin with the requirement that each L_n switch connects at least once to each L_{n-1} pod below. This effectively groups all L_n switches into a single pod, so p_n=1 (line 8). We consider each level in turn from the top of the tree downwards (lines 9, 14). At each level, we choose appropriate values for fault tolerance variables c_i and r_i (lines 10-11) with respect to constraint Equation 2.[6] Based on the value of r_i, we use Equation 3 to determine the number of pods in the level below (line 12). Finally, we move to the next level, updating the number of downlinks accordingly (lines 13-14).

The last iteration of the loop calculates the number of pods at L_1 (line 12). Since each L_1 switch is in its own pod, we know that S=p_1 (line 15). We use the value of S with Equation 1 to calculate m_i values (lines 16-18). If at any point, we encounter a non-integer value for m_i, we have generated an invalid tree and we exit (lines 19-20).

Note that instead of making decisions for the values of r_i and c_i at each level, we can choose to enumerate all possibilities. Rather than creating a single tree, this generates an exhaustive listing of all possible Aspen trees given k and n.

4.2 Aspen Trees with Fixed Host Counts

The algorithm in Listing 1 creates an Aspen tree given a fixed switch size (k), tree depth (n), and desired fault tolerance values (c_2...c_n). The number of hosts that the tree supports is an output

[6] Alternatively, we could accept as an input, desired per-level fault tolerance values <ft_n,...ft_2>, setting each $c_i = ft_i + 1$.

Listing 1: Aspen Tree Generation Algorithm

```
   input : k, n
   output: p, m, r, c, S
1  int p[1...n] = 0
2  int m[1...n] = 0
3  int r[2...n] = 0
4  int c[2...n] = 0
5  int S
6  int i = n
7  int downlinks = k
8  p[n] = 1
9  while i ≥ 2 do
10     choose c[i] s.t. c[i] is a factor of downlinks
11     r[i] = downlinks ÷ c[i]
12     p[i − 1] = p[i]r[i]
13     downlinks = k/2
14     i = i − 1
15 S = p[1]
16 m[n] = S ÷ 2
17 for i = 1 to n − 1 do
18     m[i] = S ÷ p[i]
19     if m[i] ∉ ℤ then report error and exit
20 if m[n] ∉ ℤ then report error and exit
```

value. We present the algorithm in this way in order to match the intuition of Figure 2. It is instead possible to create an Aspen tree by fixing the host count of a corresponding fat tree and adding more levels of switches in order to accommodate higher fault tolerance. With a fixed host count, S remains identical to that for the corresponding fat tree, so we begin with the fact that p_1=S and work upwards, selecting c_i and r_i values according to the desired fault tolerance.[7] A concern with this technique is that the addition of switches (and interconnecting links) introduces more points of failure. We show in § 8.2 that the benefits of local failure reaction outweigh the increased likelihood of a packet encountering a failure.

Another way to increase a tree's fault tolerance without reducing host support is to replace each k-port switch with a larger $x\times k$-port switch, where $x\in\mathbb{Z}$. As this option requires a complete replacement of existing hardware, we expect it to be less appealing to network operators, and we do not discuss it further.

5. ASPEN TREE PROPERTIES

An n-level, k-port Aspen tree is defined by a set of per-level values for p_i, m_i, r_i, and c_i; these values together determine the per-level fault tolerance, the number of switches needed and the number of hosts supported.

5.1 Fault Tolerance

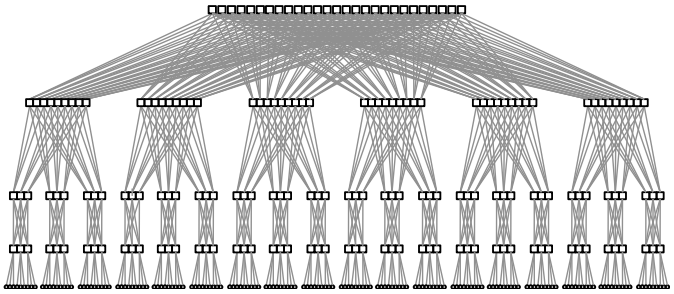
The fault tolerance at each level of an Aspen tree is determined by the number of connections c_i that each switch s has to pods below. If all but one of the connections between s and a pod q fail, s can still reach q and can route packets to q's descendants. Thus the fault tolerance at L_i is $c_i - 1$.

To express the overall fault tolerance of a tree, we introduce the *Fault Tolerance Vector (FTV)*. The FTV lists, from the top of the tree down, individual fault tolerance values for each level, i.e.

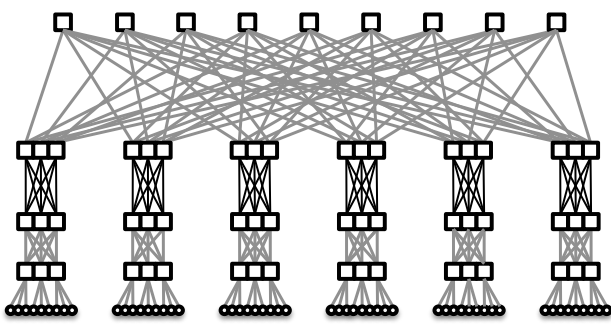
[7] The desired fault tolerance values are needed a priori in order to determine the number of levels to add to the corresponding fat tree; Aspen trees with x levels of redundant links have $n + x$ total levels.

Fault Tolerance		S	Switches	Hosts	Hierarchical Aggregation			
FTV	DCC				L_4	L_3	L_2	Overall
<0,0,0>	1	54	**189**	**162**	**3**	**3**	**3**	**27**
<0,0,2>	3	18	63	54	3	3	1	9
<0,2,0>	3	18	**63**	**54**	**3**	**1**	**3**	**9**
<0,2,2>	9	6	21	18	3	1	1	3
<2,0,0>	3	18	**63**	**54**	**1**	**3**	**3**	**9**
<2,0,2>	9	6	21	18	1	3	1	3
<2,2,0>	9	6	21	18	1	1	3	3
<2,2,2>	27	2	7	**6**	**1**	**1**	**1**	**1**

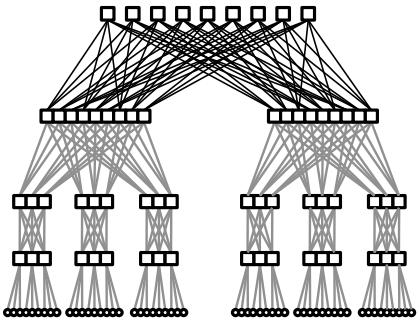
(a) All Possible 4-Level, 6-Port Aspen Trees
(Bold rows correspond to topologies pictured.)



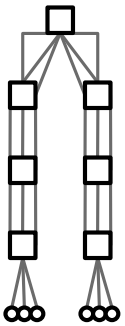
(b) Unmodified 4-Level 6-Port Fat Tree: FTV=< 0,0,0 >



(c) FTV=< 0,2,0 >



(d) FTV=< 2,0,0 >



(e) FTV=< 2,2,2 >

Figure 3: Examples of 4-Level, 6-Port Aspen Trees

$<c_n - 1,...c_2 - 1>$. For instance, an FTV of <3,0,1,0> describes a five level tree, with four links between every L_5 switch and each neighboring L_4 pod, two links between an L_3 switch and each neighboring L_2 pod, and only a single link between an L_4 (L_2) switch and neighboring L_3 (L_1) pods. The FTV for a traditional fat tree is <0,...,0>.

Figure 3 presents four sample 4-level, 6-port Aspen trees, each with a distinct FTV. Figure 3(a) lists all possible n=4, k=6 Aspen trees, omitting those with a non-integer value for m_i at any level. At one end of the spectrum, we have the unmodified fat tree of Figure 3(b), in which each switch connects via only a single link to each pod below. On the other hand, in the tree of Figure 3(e), each switch connects three times to each pod below, giving this tree an FTV of <2,2,2>. Figures 3(c) and 3(d) show more of a middle ground, each adding duplicate connections at a single (different) level of the tree.

5.2 Number of Switches Needed

In order to discuss the number of switches and hosts in an Aspen tree, it is helpful to begin with a compact way to express the variable S. Recall that our algorithm begins with a value for p_n, chooses a value for r_n, and uses this to generate a value for p_{n-1}, iterating down the tree towards L_1. The driving factor that moves the algorithm from one level to the next is Equation 3. "Unrolling" this chain of equations from L_1 upwards, we have:

$$\begin{aligned} & p_1 = p_2 r_2 \\ p_2 = p_3 r_3 \ & \rightarrow p_1 = (p_3 r_3) r_2 \\ & \dots \\ p_{n-1} = p_n r_n \ & \rightarrow p_1 = (p_n r_n) r_{n-1} \dots r_3 r_2 \\ p_n = 1 \ & \rightarrow p_1 = r_n r_{n-1} \dots r_3 r_2 \\ \forall i : 1 \le i < n, \ & p_i = \prod_{j=i+1}^{n} r_j \end{aligned} \tag{4}$$

We use Equations 2 and 4 and the fact that S is equal to the number of pods at L_1 to express S in terms of the tree's per-level's c_i values:

$$S = p_1 = \prod_{j=2}^{n} r_j = r_n \times \prod_{j=2}^{n-1} r_j = \frac{k}{c_n} \times \prod_{j=2}^{n-1} \frac{k}{2c_j} = \frac{k^{n-1}}{2^{n-2}} \times \prod_{j=2}^{n} \frac{1}{c_j}$$

To simplify the equation for S, we introduce the *Duplicate Connection Count (DCC)*, which when applied to an FTV, increments each entry (to convert per-level fault tolerance values into corresponding c_i values) and multiplies the resulting vector's elements into a single value.[8] For instance, the DCC of an Aspen tree with FTV <1,2,3> is 2 × 3 × 4=24. We express S in terms of the DDC as:

$$S = \frac{k^{n-1}}{2^{n-2}} \times \frac{1}{DCC} \tag{5}$$

Figure 3(a) shows the DCCs and corresponding values of S for each Aspen tree listed, with $S=\frac{54}{DCC}$.

This compact representation for S makes it simple to calculate the total number of switches in a tree. Levels L_1 through L_{n-1} each have S switches and L_n has $\frac{S}{2}$ switches. This means that there are $(n-\frac{1}{2})S$ switches altogether in an Aspen tree. Figure 3(a) lists the number of switches in each example tree.

5.3 Scalability

The most apparent cost of adding fault tolerance to an Aspen tree using the method of § 4.1 is the resulting reduction in the number of hosts supported. In fact, each time the fault tolerance of a single level is increased by an additive factor of x with respect to that of a traditional fat tree, the number of hosts supported by the tree is decreased by a *multiplicative* factor of x. To see this, note that the maximum number of hosts supported by the tree is simply the number

[8]The DCC counts distinct paths from an L_n switch to an L_1 switch.

of L_1 switches multiplied by the number of downward facing ports per L_1 switch. That is:

$$hosts = \frac{k}{2} \times S = \frac{k^n}{2^{n-1}} \times \frac{1}{DCC} \quad (6)$$

As Equation 6 shows, changing an individual level's value for c_i from the default of 1 to $x > 1$ results in a multiplicative reduction by a factor of $\frac{1}{x}$ to the number of hosts supported. This tradeoff is shown for all 4-level, 6-port Aspen trees in Figure 3(a) and also in the corresponding examples of Figures 3(b) through 3(e). The traditional fat tree of Figure 3(b) has no added fault tolerance and a corresponding DCC of 1. Therefore it supports the maximal number of hosts, in this case, 162. On the other hand, the tree in Figure 3(e) has a fault tolerance of 2 between every pair of levels. Each level contributes a factor of 3 to the tree's DCC, reducing the number of hosts supported by a factor of 27 from that of a traditional fat tree. Increasing the fault tolerance at any single level of the tree affects the host count in an identical way. For instance, Figures 3(c) and 3(d) have differing FTVs, as fault tolerance has been added at a different level in each tree. However, the two trees have identical DCCs and thus support the same number of hosts. This is the key insight that leads to our recommendations for middle ground topologies in § 8.1.

Another component of a tree's scalability is its hierarchical aggregation, or the number of L_{i-1} pods that fold into each L_i pod. This property contributes to the efficiency of communication and labeling schemes that rely on shared a label prefixes for compact forwarding state [30, 35]. In these schemes, it is desirable to group as many L_{i-1} switches together as possible under each L_i switch.

As with host count, there is a tradeoff between fault tolerance and hierarchical aggregation. This is because the number of downward-facing ports available at each switch (k) does not change as the fault tolerance of a tree is varied. If the c_i value for a switch s is increased, the extra links must come from other downward neighbors of s, necessarily reducing the number of pods to which s connects below.

We express the hierarchical aggregation at level L_i of an Aspen tree as $\frac{m_i}{m_{i-1}}$. It is difficult to directly relate fault tolerance and hierarchical aggregation to one another at a single level, because aggregation is a multi-level concept. To increase the aggregation at L_i we must either increase m_i or decrease m_{i-1}, which in turn reduces the aggregation at either L_{i+1} or L_{i-1}. Because of this, we consider aggregation across the entire tree, using the product of per-level values. Though imprecise, this gives intuition about the tradeoff between fault tolerance and hierarchical aggregation.

$$\frac{m_n}{m_{n-1}} \times \frac{m_{n-1}}{m_{n-2}} \times \ldots \times \frac{m_3}{m_2} \times \frac{m_2}{m_1} = \frac{m_n}{m_1} = \frac{S}{2}$$

Therefore, hierarchical aggregation relates to an Aspen tree's FTV in an identical manner to that of host count. Figure 3(b) has the maximal possible hierarchical aggregation at each level (in this case, 3) while Figure 3(e) has no hierarchical aggregation at all. The additional fault tolerance at a single level of each of Figures 3(c) and 3(d) costs these trees a corresponding factor of 3 in overall aggregation. Figure 3(a) lists hierarchical aggregation values for all possible n=4, k=6 Aspen trees. As hierarchical aggregation behaves identically to host support, we omit further discussion of the property for brevity.

6. LEVERAGING FAULT TOLERANCE

We leverage added fault tolerance links in Aspen trees by considering an insight similar to that of failure-carrying packets [22]: the tree consists of a relatively stable set of deployed physical links, and a subset of these links are up and available at any given time. Our approach is to run global re-convergence at a slower time-scale than traditional OSPF or IS-IS deployments, and to use a separate

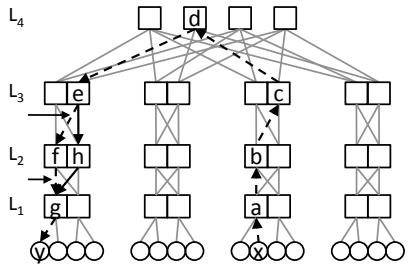


Figure 4: 4-Level, 4-Port Aspen Tree with FTV=<0,1,0>

notification protocol to react to transient link failures and recoveries. We call this protocol the *Aspen Reaction and Notification Protocol (ANP)*. ANP notifications are sent upwards to ancestors located *near to* a failure, rather than being broadcast throughout the entire tree as in OSPF or IS-IS. More importantly, these notifications are simpler to compute and process than the calculations required for global re-convergence. By decreasing the number of hops through which updates propagate and the processing time at each hop, we significantly reduce the tree's re-convergence time.[9]

Recall from the example of § 2 that a traditional fat tree has no choice but to drop a packet arriving at a switch incident on a failed link below. In Figure 1, a packet sent from host x to host y is doomed to be lost the instant that x's ingress switch a selects b as the packet's next hop. This is because all (shortest) paths from b to y pass through link $f - g$. Extra fault tolerance links can serve to push this dooming decision farther along the packet's path, reducing the chance that the packet will be lost due to a failure that occurs while it is in flight.

Figure 4 shows an n=4, k=4 Aspen tree, modified from the 4-level, 4-port fat tree of Figure 1 to have additional fault tolerance links between L_3 and L_2, that is, its FTV is <0,1,0>. These added links give a packet sent from x to y an alternate path through h, as indicated by the bold arrows. e can route packets via h rather than f, when link f-g fails. The purpose of ANP is to quickly notify the appropriate switches in the event of a failure, so that the redundant links in an Aspen tree can be leveraged to route around the failures.

We begin by examining various failure scenarios and establishing the means by which ANP can enable switches to route around a failure in each case. To determine the set of ancestors that should receive a failure notification, we must consider the effect of a link failure along an in-flight packet's intended path. Shortest path routing will send packets up and back down the tree, so we consider the upward and the downward path segments in turn.

If a link along the upward segment of a packet's path fails, the packet's path simply changes on the fly. This is because each of a switch's uplinks leads to some nonempty subset of L_n switches. In § 4, we introduced the requirement that all L_n switches connect at least once to all L_{n-1} pods, so all L_n switches ultimately reach all hosts. As such, a packet can travel upward towards any L_n switch, and a switch at the bottom of a failed link can simply select an alternate upward-facing output port in response to the failure, without sending any notifications.

The case in which a link fails along the downward segment of a packet's intended path is somewhat more complicated. Consider a failure between L_i and L_{i-1} along a packet's intended downward path. Fault tolerance properties below L_i are not relevant, as the packet needs to be diverted *at or before* reaching L_i in order to avoid

[9]Note that even with localized failure reaction there will still be background control traffic for normal OSPF behavior, but this traffic will not be on the critical path to re-convergence.

the failure. However, if there is added fault tolerance at or above L_i, nearby switches can route around the failure according to the following cases:

Case 1: c_i>1. *The failure is at a level with added fault tolerance.* This case corresponds to the failure of link $e-f$ in Figure 4. When the packet reaches switch e, e realizes that the intended link $e-f$ is unavailable and instead uses its second connection to $f's$ pod, through h. By definition of a pod, h has downward reachability to the same set of descendants as f and therefore can reach g and ultimately, the packet's intended destination, y. e does not need to send any notifications in order to facilitate this new routing pattern; it simply forwards packets destined for y through h rather than f upon discovering the failure of link $e-f$.

Case 2: c_i=1,c_{i+1}>1. *The closest added fault tolerance is at the level immediately above the failure.* This corresponds to the failure of link $f-g$ in Figure 4. In this case, if the packet travels all the way to f it will be dropped. But if e learns of the failure of $f-g$ before the packet's arrival, it can select the alternate path through f's pod member h. To allow for this, when f notices the failure of link $f-g$, it should notify any parent (e.g. e) that has a second connection to f's pod (e.g. via h).

Case 3: c_i=1,$c_{j:j>i+1}$>1 *The nearest level with additional links is more than one hop above.* Figure 5 shows an example of this case, in which L_2 link $f-g$ fails and the closest added fault tolerance is at L_4 (e.g. at f's ancestor d). Upon the packet's arrival, d selects i as the next hop, so that the packet travels along the path $d-i-h-g-y$. While the fault tolerance is located further from the failure than in case (2), the goal is the same: f notifies any ancestor (e.g. d) that has a downward path to another member of f's pod (e.g. h).

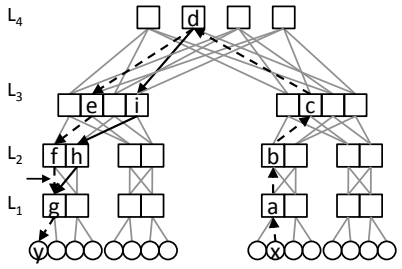


Figure 5: 4-Level, 4-Port Aspen Tree with FTV=<1,0,0>

To generalize our *Aspen Reaction and Notification Protocol*, when a link from L_i switch s to L_{i-1} neighbor t fails, s first determines whether it has non-zero fault tolerance. If so, it subsequently routes all packets intended for t to an alternate member of t's pod. Otherwise, s passes a notification (indicating the set of hosts H that it no longer reaches) upwards. Upon receipt of an ANP notification from a descendant, ancestor switch a updates its routing information for those hosts in H to which it has alternate paths and forwards a notification upwards for those hosts in H to which a does not have alternate paths. The process is similar for link recovery.

7. WIRING THE TREE: STRIPING

In § 4, we described the generation of Aspen trees in terms of switch count and placement, and the number of connections between switches at adjacent levels. Here, we consider the *striping*, or organization of connections between switches. We have deferred this discussion until now because of the topic's dependence on the techniques described in § 6 for routing around failures.

Striping refers to the distribution of connections between an L_i pod and neighboring L_{i-1} pods. For instance, consider the striping

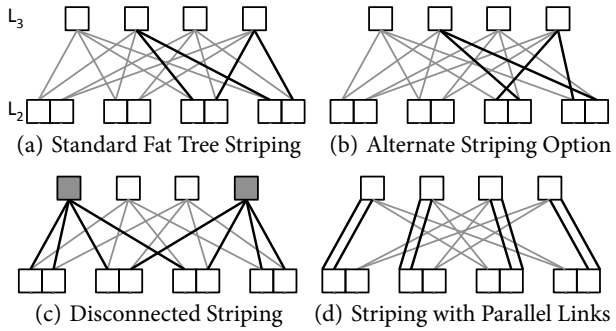


Figure 6: Striping Examples for a 3-Level, 4-Port Tree (Hosts and L_1 switches are omitted for space and clarity.)

pattern between L_3 and L_2 in the 3-level tree of Figure 6(a). The leftmost (rightmost) switch in each L_2 pod connects to the leftmost (rightmost) two L_3 switches. On the other hand, Figure 6(b) shows a different connection pattern for the switches in the rightmost two L_2 pods, as indicated with the bold lines.

Striping patterns can affect connectivity, over-subscription ratios, and the effectiveness of redundant links in Aspen trees. Some striping schemes disconnect switches at one level from pods at the level below. In fact, we made a striping assumption in § 4 to avoid exactly this scenario, by introducing the constraint that each L_n switch connects to each L_{n-1} pod at least once. The striping scheme of Figure 6(c) violates this constraint, as the two shaded L_3 switches do not connect to all L_2 pods. Some striping patterns include parallel links, as in Figure 6(d). Each L_3 switch connects twice to one of its neighboring L_2 pods, via parallel connections to a sole pod member.

Introducing additional fault tolerance into an Aspen tree increases the number of links between switches and pods at adjacent levels, thus increasing the set of possibilities for distributing these connections. Recall that ANP relies on the existence of ancestors common to a switch s incident on a failed link and alternate members of s's pod; an acceptable striping policy must yield such common ancestors. This means that striping patterns should not consist entirely of duplicate, parallel links. For instance, in Figure 5, the fact that d has connections to e's pod via two *distinct* pod members (e and i) is the key property that allows d to successfully route around the failure of link $f-g$. If instead d connected to e's pod by two duplicate links to e, d would have to drop all packets destined for y.

In general, any striping policy that yields the appropriate common ancestors discussed in § 6 is acceptable for Aspen trees. More formally, ANP relies on the following characteristics of its underlying topology: *For every level L_i with minimal connectivity to L_{i-1}, if $L_{f:f>i}$ is the closest fault tolerant level above L_i, each L_i switch s shares at least one L_f ancestor a with another member of s's pod, t.*

8. DISCUSSION

In this section, we consider an instance of Aspen trees that provides a significant reduction in convergence time at a moderate cost (e.g. 80% faster convergence with 50% host loss). We also more carefully explore the topic of introducing more points of failure when adding depth to a tree. Finally, we discuss the issues of concurrent failures and some limitations of Aspen trees.

8.1 Practical Aspen Trees

We showed in § 6 that the most useful and efficient fault tolerance is (1) above failures and (2) as close to failures as possible. While the most fault-tolerant tree has an FTV with all maximal (and non-zero)

entries, this may come at too high a scalability or network cost. To enable usable and efficient fault tolerance, in FTVs with non maximal entries it is best to cluster non-zero values towards the left while simultaneously minimizing the lengths of series of contiguous zeros. For instance, if an FTV of length 6 can include only two non-zero entries, the ideal placement would be <x,0,0,x,0,0>, with x>0. There are at most two contiguous zeros, so updates propagate a maximum of two hops, and each 0 has a corresponding x to its left, so no failure leads to global propagation of routing information.

One Aspen tree in particular bears special mention. Given our goal of keeping fault tolerance at upper tree levels, the largest value-add with minimal cost is the addition of extra links at the single level of the tree that can accommodate all failures, i.e. the top level. A tree with only L_n fault tolerance and an FTV of <1,0,0,...> supports half as many hosts as does a traditional fat tree of the same depth. The average convergence propagation distance for this tree is less than half of that for a traditional fat tree, and more importantly, updates only travel upward rather than fanning out to all switches in the tree. For instance, an Aspen tree with n=4, k=16 and FTV=<1,0,0> supports only half as many hosts as an n=4, k=16 fat tree, but converges 80% faster. In fact, the topology used for VL2 [11] is an instance of an Aspen tree with an FTV of <1,0,0,...>. While this type of tree represents an interesting point in the design space, we reiterate that our goal with Aspen is to explore fault tolerance, scalability, and network cost tradeoffs rather than to select a "best tree." We aim to give data center operators the insight they need in order to tune their networks to their individual requirements.

8.2 Aspen Trees with Fixed Host Counts

The algorithm of Listing 1 builds Aspen trees by first selecting the network size (in terms of switch size, k, and tree depth, n) and desired fault tolerance, and then determining the number of hosts supported as compared to that of a traditional fat tree (Equation 6). If we instead fix both the number of hosts supported and the desired fault tolerance, network size becomes the dependent variable.

An Aspen tree with non-zero fault tolerance needs more levels of switches to support the same number of hosts than does a fat tree with identically sized switches. This raises the question of whether the decreased convergence time in an Aspen tree outweighs the increased probability of a packet encountering a link failure along one of the added links. To evaluate this, we first calculate the number of links added to turn a fat tree into a corresponding Aspen tree with non-zero fault tolerance and an identical number of hosts. We then calculate the average convergence time of each tree across failures at all levels. Finally, for each tree, we multiple this average convergence time by the number of links in the tree to determine the tree's *convergence cost*. This gives a single term that accounts for both the total number of links in each tree (and thus the number of possible points of failure) and the cost of each such failure.

Figure 7 compares this *convergence cost* of an n-level fat tree to that of an Aspen tree with the same host count, for the range of n that we expect to see in practice and for a varying number of levels with added fault tolerance. The graph shows that when an n-level fat tree is extended with up to x=$n-2$ new levels that have non-zero fault tolerance, the resulting $(n+x)$-level Aspen tree always has a lower convergence cost than the corresponding fat tree.

Therefore, while a packet encounters more links in its path through an Aspen tree than it would in the corresponding fat tree with the same host count, the probability that the packet can be re-routed around a failure rather than dropped more than makes up for this introduction of more points of failure.

Note that increasing the depth of an Aspen tree also increases the path length for the common case packet that does not encounter a failure. In the data plane, this added latency is on the order of ns or μs. However, this may not be acceptable in all scenarios, and therefore it is crucial that a data center operator use only the minimal added fault tolerance absolutely necessary for correct operation when building an Aspen tree with fixed host count.

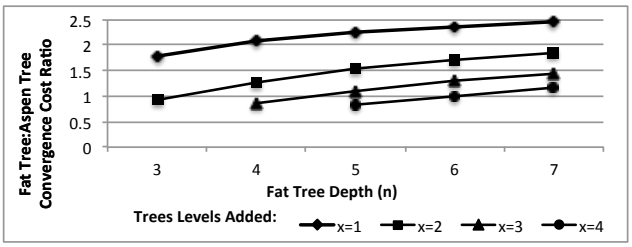


Figure 7: Fat Tree versus Aspen Tree Convergence Costs

8.3 Concurrent Failures

For clarity, we discuss Aspen trees and ANP in the context of of single link failures. However, in most cases, our techniques apply seamlessly to multiple simultaneous link failures. In fact, failures far enough apart in a tree have no effect on one another and can be considered individually. Additionally, ANP can enable re-routing even when failures are close to one another, as long as the striping policy of § 7 is satisfied. It is possible that in some pathological cases, compound failures can lead to violations the striping policy of § 7, ultimately causing packet loss. We leave a complete analysis of compound failure patterns for future work.

8.4 Aspen Tree Limitations

As § 5 shows, there is a non-trivial cost associated with decreasing re-convergence time from that of a traditionally-defined fat tree. This cost can be in terms of the scalability of the tree (e.g. host support) or the network size. As discussed in § 4.2 and 8.2, decreasing re-convergence time by increasing a tree's network size can lead to longer path lengths.

As such, we encourage data center operators to balance the tradeoffs of Equations 5 and 6 to design trees that satisfy their requirements with minimum cost. Hence, an understanding of these tradeoffs is crucial prior to deployment; Aspen trees are static topologies and cannot be reconfigured through software. To change properties such as a tree's FTV, a data center operator would have to physically add and/or remove network elements and links. Additionally, quick failure reaction in Aspen trees requires the deployment of a new protocol, ANP, as well as software reconfiguration of existing global re-convergence protocols to run at a slower time scales and off of the critical path to re-convergence.

While ANP reduces the re-convergence period after link failure or recovery, it does not provide zero re-convergence time for all trees. There is a window of vulnerability after a failure or recovery while ANP notifications are sent and processed, and packet loss can occur during this window. An Aspen tree with all non-zero FTV entries would have instant re-convergence, but we expect that this comes at too high a cost to be practical.

Finally, there are pathological tree configurations in which additional fault tolerance cannot help to avoid packet loss. In such cases, we have bottleneck pods, i.e. pods with only a single switch, at high levels in the tree. If a failure occurs immediately below a bottleneck pod, no amount of redundancy higher in the tree can help as there are no alternate pod members to route around the failure. We do not expect to see such trees in practice.

9. EVALUATION

We now explore more closely the tradeoffs between convergence time, scalability, and network size in Aspen trees. We first consider the convergence time and scalability across Aspen trees with fixed network sizes, as described in § 4.1. We then show the increase in network size necessary to decrease convergence time without sacrificing host support, as discussed in § 4.2.

9.1 Convergence versus Scalability

An Aspen tree with added fault tolerance, and therefore an FTV with non-zero entries, has the ability to react to failures locally. This replaces the global re-convergence of broadcast-based routing protocols with a simple failure notification protocol, ANP. ANP notifications require less processing time, travel shorter distances, and are sent to fewer switches, significantly reducing re-convergence time and control overhead in the wake of a link failure or recovery.

If the fault tolerance at a level L_f is non-zero, then switches at L_f can route around failures that occur at or below L_f, provided a switch incident on an L_i failure notifies its L_f ancestors to use alternate routes. So, the convergence time for a fault between L_i and L_{i-1} is simply the set of network delays and processing times for an ANP notification at each switch along an $(f-i)$-hop path. Adding redundant links at the closest possible level L_f above expected failures at L_i minimizes this convergence time.

The cost of adding fault tolerance to a fixed-depth fat tree is in the tree's scalability, in terms of both host support and hierarchical aggregation. Each FTV entry $x > 0$ reduces the maximum possible number of hosts by a multiplicative factor of $x + 1$.

We analytically evaluate the benefits of reducing re-convergence time with respect to the scalability costs, using the equations derived in § 5. We begin with a small example with n=4 and k=6 in order to explain the evaluation process. For each possible 4-level, 6-port Aspen tree, we consider the FTV and correspondingly, the distance that updates travel in response to a failure at each level. We calculate this distance by simply using the number of hops between a failure and the nearest ancestor level with non-zero fault tolerance. For instance if there is non-zero fault tolerance between L_i and L_{i-1}, then the update propagation distance for failures at L_i is 0 and the distance for failures at L_{i-2} is 2. If there is no area of non-zero fault tolerance above a level, we are forced to revert to global re-convergence, and the update propagation distance is that required to reach the furthest switches in the tree. We express the average convergence time for a tree as the average of this propagation distance across failures at all levels of the tree.[10]

We consider the scalability cost of adding fault tolerance by counting the number of hosts missing in each Aspen tree as compared to a traditional fat tree with the same depth and switch size. We calculate these numbers using Equation 6 (§ 5.3). We elect to consider hosts removed, rather than hosts remaining, so that the compared measurements (convergence time and hosts removed) are both minimal in the ideal case and can be more intuitively depicted graphically. Figure 8 shows this convergence versus scalability tradeoff; for each possible FTV option, the figure displays the average convergence time (in hop count) across all levels, alongside the number of hosts missing with respect to a traditional fat tree.[11] To normalize, values are shown as percentages of the worst case.

Thus, we have a spectrum of Aspen trees. At one end of this spectrum is the tree with no added fault tolerance links (FTV=<0,0,0>) but with no hosts removed. At the other end are trees with high fault tolerance (all failure reactions are local) but with over 95% of the hosts removed. In the middle we find interesting cases: in these, not every failure can be handled locally, but those not handled locally can be masked within a small and limited number of hops. The convergence times for these middle-ground trees are significantly less than that of a traditional fat tree, but substantially fewer hosts are removed than for the tree with entirely local failure reactions.

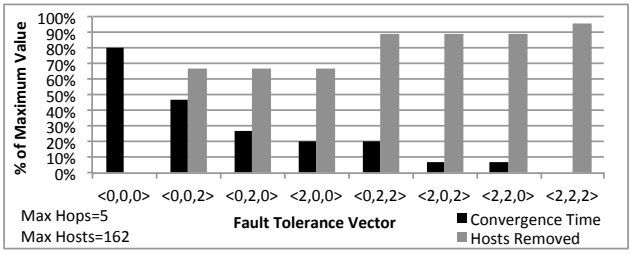


Figure 8: Convergence vs. Scalability: n=4, k=6 Aspen Trees

We observe that there are often several trees with the same host count but with differing convergence times. This is shown in the second, third and fourth entries of Figure 8, in which the host counts are all $\frac{1}{3}$ of that for a traditional fat tree, but the average update propagation distance varies from 1 to 2.3 hops. A network designer constrained by the number of hosts to support should select a tree that yields the smallest convergence time for the required host support. Similarly, there are cases in which the convergence times are identical but the host count varies, e.g. FTVs <2,0,0> and <0,2,2>. Both have average update propagation distances of 1, but the former supports 54 hosts and the latter only 18.

In practice, we expect trees with $3 \leq n \leq 7$ levels and $16 \leq k \leq 128$ ports per switch, in support of tens of thousands of hosts. Figure 9 shows examples of larger trees. At these sizes, there are many possible Aspen trees for each combination of n and k and we often find that numerous trees (FTVs) all correspond to a single [host count, convergence time] pair. We collapsed all such duplicates into single entries, and because of this, we removed the FTV labels from the graphs of Figure 9 for readability.

Figure 9(a) shows the same trend as does Figure 8, but since there are more data points, the results are perhaps more apparent. As we move from left to right in the graphs, we remove more hosts. Again, the host removal bars are grouped into steps; each individual number of hosts removed corresponds to several different values for average convergence time. Figure 9(b) show trees with larger switches and smaller depth, keeping our results in line with realistic data center sizes. The shallower depth limits the number of possible trees, so there are fewer entries than in Figure 9(a). Overall, Figure 9 confirms that with only modest reductions to host count, the reaction time of a tree can be significantly improved.

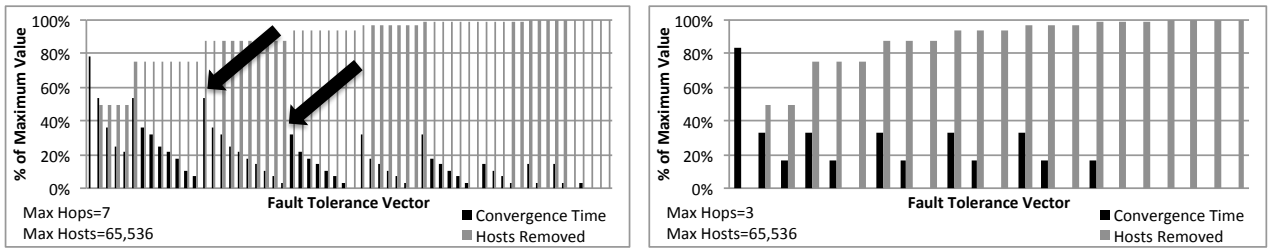
9.2 Convergence versus Network Size

We next consider improving a fat tree's fault tolerance by increasing its network size while keeping host count constant, as introduced in § 4.2. We examine the various Aspen tree options for a given host count, measuring for each tree the financial cost (in terms of added switches), re-convergence time (based on the distance between failures and redundant links), and control overhead (based on the number of switches that react to a failure).

For these evaluations, we implemented both ANP and a link-state protocol based on OSPF, which we call *LSP*. We built the two protocols in Mace [7, 19], a language for distributed systems development that includes an accompanying model checker [18] and simulator [20] that allow us to consider a variety of different Aspen trees and failure models. We used safety and liveness properties in the

[10] We exclude 1^{st} hop failures as Aspen trees cannot mitigate these without introducing multi-homing for hosts.

[11] Because we average convergence times across tree levels, no individual bar in the graph reaches 100% of the maximum hop count.



(a) Convergence Time vs. Hosts Removed: n=5, k=16 Aspen Trees Arrows indicate varying convergence time with single host count

(b) Convergence Time vs. Hosts Removed: n=3, k=64 Aspen Trees

Figure 9: Convergence vs. Scalability Tradeoff for Data Center Scale Aspen trees

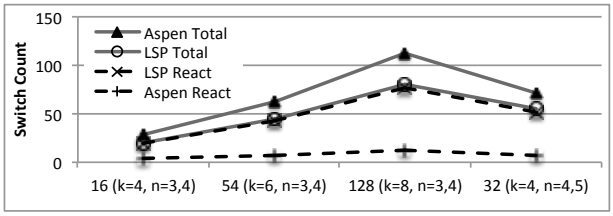
Mace model checker to verify the correctness of our implementations of ANP and LSP, and we used the Mace simulator to compare the failure reaction time and overhead of the two protocols. We chose LSP as a baseline routing protocol for comparison; we leave a detailed comparison of the tradeoffs between routing complexity and failure reactivity in more complex protocols (e.g. bounce routing, DDC [26], and F10 [27]) as future work.

We built a topology generator that takes as inputs the tree depth (n), switch size (k), and FTV, and creates an n-level Aspen tree of k-port switches matching the input FTV. We used this generator to create input topologies for Mace as follows: for varying values of k and n, we created an n-level, k- port fat tree and a corresponding (n+1)-level, k-port Aspen tree with FTV <x,0,0,...>, with x such that both trees have identical host counts. We highlight this particular FTV for the reasons introduced in § 8.1.

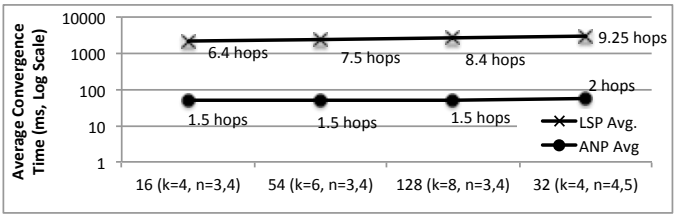
For each pair of trees, we initially ran LSP to set up routes for the topology and verified the accuracy of these routes using the Mace model checker. Using the Mace simulator, we then failed each link in each tree several times and allowed the corresponding recovery protocol (for fat trees, LSP and for Aspen trees, ANP) to react and update switches' forwarding tables. We recorded the minimum, maximum, and average numbers of switches involved and re-convergence times across failures for each tree.

Figure 10(a) shows the total number of switches in each fat tree and corresponding Aspen tree, along with the average number of switches involved in each failure reaction. Because the minimums and maximums tracked the averages quite closely, we graph only the averages for ease of visibility. The x-axis gives the number of hosts in the tree, the switch size (k), and the depth (n) of the fat tree and Aspen tree, respectively. In order to change an n-level, k-port fat tree into an Aspen tree with FTV <x,0,0,...>, we increase the number of switches at L_n from $\frac{S}{2}$ to S and add a new level, L_{n+1}, with $\frac{S}{2}$ switches. In other words, we add S new switches to the tree. This is a fixed percentage of the total switches in the tree for any given n, and corresponds to 40%, 29% and 22% increases in total switch count, for 3, 4 and 5-level fat trees, respectively, or a $\frac{2}{k}$ increase in the switch-to-host ratio. Figure 10(a) expresses both the cost of the Aspen tree network in terms of increased switch count (the difference between the curves labeled *Aspen Total* and *LSP Total*) as well as the reduced control overhead in terms of the number of switches that react to each failure (the difference between the curves labeled *Aspen React* and *LSP React*). In general, LSP involves most switches on each failure reaction[12] whereas ANP involves less than 15% of the total switches in each Aspen tree.

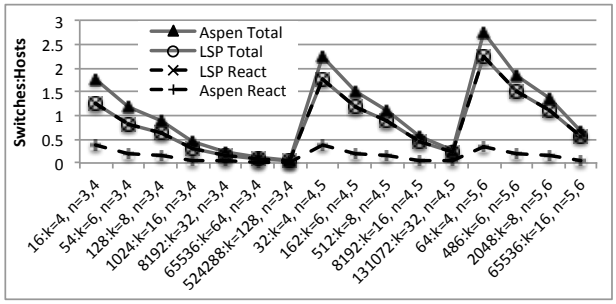
[12]Though all switches always process all LSAs, our measurements only attribute an LSA to a switch that changes its forwarding table.



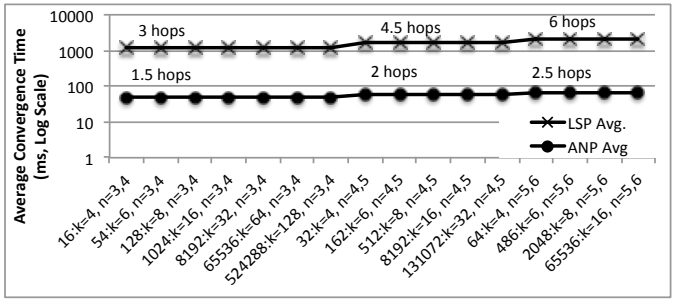
(a) Total Switches vs. Reactive Switches: Small Trees



(b) Convergence Time: Small Trees



(c) Total Switches vs. Reactive Switches: Large Trees



(d) Convergence Time: Large Trees

Figure 10: Network Size, Convergence Time, and Control Overhead for Pairs of Corresponding Fat and Aspen Trees

As Figure 10(b) shows, the convergence time for each fat tree is substantially longer than that for the corresponding Aspen tree and this difference grows as n increases. To give more realistic results, we convert hop counts into time, estimating the propagation delay between switches and the time to process ANP and LSA packets as 1μs, 20ms, and 300 ms, respectively. These estimates are conservatively tuned to favor LSP. Figure 10(b) shows convergence time on a log scale, with actual hop count labels for reference.

Since the model checker scales to at most a few hundred switches, we use additional analysis for mega data center sized networks. Figure 10(c) is similar to Figure 10(a), but shows a wider range of topologies. Here, we plot switch-to-host ratios in order to normalize across a wide range of trees. As in our simulations, a modest increase to the total number of switches (the graph's upper two curves) leads to significantly fewer switches that react to each failure (the graph's lower two curves). In fact, as the figure shows, LSP re-convergence consistently involves all switches in the tree, whereas only 10-20% of Aspen switches react to each failure. Finally, Figure 10(d) compares the average convergence times for each pair of trees, confirming the drastic reduction in convergence time that Aspen trees can provide, for a fairly small increase in network size. As the figure shows, ANP converges orders of magnitude more quickly than LSP.

10. RELATED WORK

Our work is largely motivated by direct experience with large data center operations as well as by the findings of [9], which show that link failures are common, isolated, and impactful. The network redundancy inherent to multi-rooted trees helps by only a factor of 40%, in part due to protocols not taking full advantage of redundant links (e.g. requiring global OSPF re-convergence prior to switching to backup paths). Together, Aspen trees and their corresponding notification protocol ANP better leverage network redundancy for quick failure reaction. The study in [9] also finds that links in the core of the network have the highest probability of failure and benefit most from network redundancy. This aligns well with the subset of Aspen trees highlighted in § 8.1. Finally the study shows that link failures are sporadic and short-lived, supporting our belief that such failures should not cause global re-convergence.

Related Topologies: Aspen trees derive from the initial presentations of Clos networks as non-blocking communication architectures [5], fat trees as universal networks for supercomputer communication [24], and Greenberg's and Leiserson's multi-rooted fat trees [13], also referred to as *Butterfly Fat Trees* [12]. We also take inspiration from Upfal's multi-butterfly networks [34], Leighton et al's corresponding routing algorithms [23], and Goldberg et al's splitter networks [10], which create a subset of Aspen trees by raising c_i from 1 to x>1 uniformly at all levels of the tree. These works consider topologies in the context of a priori message scheduling rather than that of running packet-switched protocols (e.g. IP) over modern switch hardware in today's data centers. More recent topologies such as DCell [15] and Bcube [14] are inherently more fault-tolerant than Aspen trees, at the cost of more complex (non-shortest path) forwarding protocols and in DCell, the possibility of forwarding loops. Finally, Jellyfish [32] uses a random graph to trade bisection bandwidth and regular graph structure for ease of expansion and reduced hardware.

Alternative Routing Techniques: Alternative routing techniques can provide fault tolerance in a network without the need for added hardware. We discussed bounce routing, DDC and F10 in § 2; here we consider DDC and F10 more closely.

DDC [26] and Aspen trees share the motivation that it is unacceptable to disrupt communication for tens of seconds while waiting for control plane re-convergence. DDC's approach is to bounce a packet that encounters a failure back along its path until it reaches a node with an alternate path to the destination, repeating as necessary. This essentially performs a depth-first search (DFS) rooted at the sender, in which the leaf of the first searched path is the switch incident on the failure. This can lead to long paths but has the benefit of working with arbitrary graphs. Unfortunately, DFS-style routing performs particularly poorly over fat trees, as the decisions that affect whether a packet will ultimately be bounced are made as early as the path's first hop. This means that many bounced DDC packets return all the way to the sender before trying an alternate path. The authors' evaluation of DDC's effectiveness over various topologies hints at this fact, noting that fat trees lack "resilient nodes" with multiple output ports to a destination. In fact, fat trees contain such resilient nodes only on the upward segment of a packet's path, whereas Aspen trees contain resilient nodes on the downward segment as well. Finally, because of forwarding state size constraints, DDC supports only exact-match forwarding, as opposed to longest prefix-matching forwarding, a common data center requirement.

F10 [27] introduces a modified striping policy, coupled with a set of cascading reaction protocols that bounce packets through a number of limited hops around transient failures. For many cases, a two-hop detour is sufficient though in some cases $\frac{k}{4}$ targeted failures bounce packets all the way back to the sender's level. This leads to more significant path inflation and increased network state. On the other hand, F10 does not require any additional hardware other than that of a traditional fat tree and provides zero convergence time on failure. Overall, F10 and Aspen trees take different points in the design space, trading software complexity for hardware cost.

Failure carrying packets (FCP) [22] eliminate the need for network re-convergence by encoding failure information in data packets, at the expense of the implementation and deployment of a new data plane and the use of long paths. In general, the difficulty of (temporary) path inflation is inherent to many alternative routing techniques; Aspen trees leverage the underlying topology's regularity to render this a non-issue.

Multi-path TCP (MPTCP) [31] breaks individual flows into *subflows*, each of which may be sent via a different path based on current congestion conditions in the network. A path that includes a failed link will appear to be congested since a portion of it offers no bandwidth, and MPTCP will move any corresponding subflows to another path. MPTCP relies on the ability to modify end host software; this is not always possible in the data center. There has been some work towards sub-second IGP convergence [8] but the topologies considered are orders of magnitude smaller than modern data centers.

Another way to improve the fault tolerance of a network is to establish backup paths for use when a primary path (or link along the path) fails. This can be done on flow entry [16, 33] or dynamically on failure [2]. These works differ from Aspen trees in their use of source routing. Additionally, a limitation of techniques based on backup paths in general is that it may take a sender a full round-trip delay to determine that a primary path has failed.

Finally, the idea behind this work is derived from fast failure recovery [21] techniques in WANs. Our approach is to engineer data center topologies so as to enable FFR for link failures.

11. CONCLUSION

We have considered the issue of improving failure recovery in the data center by modifying fat tree topologies to enable local failure reactions. A single link failure in a fat tree can disconnect a portion of the network's hosts for a substantial period of time while updated routing information propagates to every switch in the tree. This is unacceptable in the data center, where the highest levels of avail-

ability are required. To this end, we introduce the Aspen tree — a multi-rooted tree topology with the ability to react to failures locally — and its corresponding failure notification protocol, ANP. Aspen trees provide decreased convergence times to improve a data center's availability, at the expense of scalability (e.g. reduced host count) or financial cost (e.g. increased network size). We provide a taxonomy for discussing the range of Aspen trees available given a set of input constraints and perform a thorough exploration of the tradeoffs between fault tolerance, scalability, and network cost in these trees.

12. ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd, Michael Schapira, for their valuable suggestions and feedback. We also thank John R. Douceur for his feedback and his insight into the complexities of increasing an Aspen tree's depth.

13. REFERENCES

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. ACM SIGCOMM 2008.

[2] A. Banerjea. Fault Recovery for Guaranteed Performance Communications Connections. *ToN*, 7(5):653–668, Oct 1999.

[3] Cisco Systems, Inc. OSPF Design Guide. `https://learningnetwork.cisco.com/docs/DOC-3046`, 2005.

[4] Cisco Systems, Inc. Cisco Data Center Infrastructure 2.5 Design Guide. `www.cisco.com/univercd/cc/td/doc/solution/`, 2008.

[5] C. Clos. A Study of Non-Blocking Switching Networks. *BSTF*, 32(2):406–424, Mar 1953.

[6] W. J. Dally and H. Aoki. Deadlock-Free Adaptive Routing in Multicomputer Networks Using Virtual Channels. *ToPaDS*, 4(4):466–475, Apr 1993.

[7] D. Dao, J. Albrecht, C. Killian, and A. Vahdat. Live Debugging of Distributed Systems. Springer-Verlag CC 2009.

[8] P. Francois. Achieving sub-second IGP convergence in large IP networks. *SIGCOMM CCR*, 35:2005, 2005.

[9] P. Gill, N. Jain, and N. Nagappan. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. ACM SIGCOMM 2011.

[10] A. V. Goldberg, B. M. Maggs, and S. A. Plotkin. A Parallel Algorithm for Reconfiguring a Multibutterfly Network with Faulty Switches. *ToC*, 43(3):321–326, Mar 1994.

[11] 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. ACM SIGCOMM 2009.

[12] R. I. Greenberg. *Efficient Interconnection Schemes for VLSI and Parallel Computation*. PhD thesis, MIT, 1989.

[13] R. I. Greenberg and C. E. Leiserson. Randomized Routing on Fat-Trees. In *Advances in Computing Research*, pages 345–374. JAI Press, 1996.

[14] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. BCube: A High Performance, Server-Centric Network Architecture for Modular Data Centers. ACM SIGCOMM 2009.

[15] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. DCell: A Scalable and Fault-Tolerant Network Structure for Data Centers. ACM SIGCOMM 2008.

[16] S. Han and K. G. Shin. Fast Restoration of Real-time Communication Service from Component Failures in Multi-hop Networks. ACM SIGCOMM 1997.

[17] M. Karol, S. J. Golestani, and D. Lee. Prevention of Deadlocks and Livelocks in Lossless Backpressured Packet Networks. *ToN*, 11(6):923–934, Dec 2003.

[18] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. USENIX NSDI 2007.

[19] C. Killian, J. W. Anderson, R. B. R. Jhala, and A. Vahdat. Mace: Language Support for Building Distributed Systems. ACM PLDI 2007.

[20] C. Killian, K. Nagarak, S. Pervez, R. Braud, J. W. Anderson, and R. Jhala. Finding Latent Performance Bugs in Systems Implementations. ACM FSE 2010.

[21] A. Kvalbein, A. F. Hansen, T. Cicic, S. Gjessing, and O. Lysne. Fast IP Network Recovery Using Multiple Routing Configurations. IEEE INFOCOM 2006.

[22] K. Lakshminarayanan, M. Caesar, M. Rangan, T. Anderson, S. Shenker, and I. Stoica. Achieving Convergence-Free Routing using Failure-Carrying Packets. ACM SIGCOMM 2007.

[23] F. T. Leighton and B. M. Maggs. Fast Algorithms for Routing Around Faults in Multibutterflies and Randomly-Wired Splitter Networks. *ToC*, 41(5):578–587, May 1992.

[24] C. E. Leiserson. Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing. *ToC*, 34(10):892–901, Oct 1985.

[25] K. Levchenko, G. M. Voelker, R. Paturi, and S. Savage. XL: An Efficient Network Routing Algorithm. ACM SIGCOMM 2008.

[26] J. Liu, A. Panda, A. Singla, B. Godfrey, M. Schapira, and S. Shenker. Ensuring Connectivity via Data Plane Mechanisms. USENIX NSDI 2013.

[27] V. Liu, D. Halperin, A. Krishnamurthy, and T. Anderson. F10: A Fault-Tolerant Engineered Network. USENIX NSDI 2013.

[28] J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, A. R. Curtis, and S. Banerjee. DevoFlow: Cost-Effective Flow Management for High Performance Enterprise Networks. ACM HotNets 2010.

[29] J. Moy. OSPF version 2. RFC 2328, IETF, 1998.

[30] 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. ACM SIGCOMM 2009.

[31] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving Datacenter Performance and Robustness with Multipath TCP. ACM SIGCOMM 2011.

[32] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey. Jellyfish: Networking Data Centers Randomly. USENIX NSDI 2012.

[33] L. Song and B. Mukherjee. On the Study of Multiple Backups and Primary-Backup Link Sharing for Dynamic Service Provisioning in Survivable WDM Mesh Networks. *JSAC*, 26(6):84 –91, Aug 2008.

[34] E. Upfal. An O(logN) Deterministic Packet Routing Scheme. ACM STOC 1989.

[35] M. Walraed-Sullivan, R. N. Mysore, M. Tewari, Y. Zhang, K. Marzullo, and A. Vahdat. ALIAS: Scalable, Decentralized Label Assignment for Data Centers. ACM SOCC 2011.