Scalability vs. Fault Tolerance in Aspen Trees

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

Keith Marzullo
UCSD
marzullo@cs.ucsd.edu

Amin Vahdat
Google, UCSD
vahdat@cs.ucsd.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 propagates to every network element 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 the control overhead of failure recovery. We find that for a given network size, decreasing the convergence time for a topology results in a decrease to the topology's scalability (e.g. the number of hosts supported). On the other hand, for a fixed host count, a reduction in convergence time comes at the cost of additional network elements and links. We explore the tradeoffs between fault tolerance, scalability and network size, and we propose a range of modified multi-rooted tree topologies that provide significantly reduced convergence time while retaining much of the traditionally-defined fat tree's scalability.

1. INTRODUCTION

Modern data centers often operate over hierarchically structured network fabrics. In fact, one of the most common topologies for interconnects of network elements —switches— in the data center is a fat tree, or Clos network [1, 5, 19, 24]. This topology's popularity is in part due to its support for full bisection bandwidth. In our experience, 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 end 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 end 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 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 avoid this loss could be expensive. Moreover, 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, the time required for updating network elements to work around failures and to use alternate paths can be substantial. 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 [4, 23] can be tens of seconds [20, 21]. 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 [21, 22] and in practice, settings such as protocol timers can further compound these delays [17]. The processing time at each switch along the path from a failure to the farthest switches adds up quickly. Packets continue to be dropped 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 topology 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 time for each individual failure, by modifying fat tree topologies to enable *local fail-*

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

ure 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 on 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 re-convergence 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 for years after the failure of 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 topologies 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 can react to failures locally. We first give an algorithm to determine the set of Aspen trees that can be created, given constraints such as the number of available switches or the requirements for host support (§ 3). Next, to precisely specify the fault tolerance properties of Aspen trees, we introduce a *Fault Tolerance Vector (FTV)*. An FTV quantifies failure reactivity by indicating the quality and locations of added fault tolerance throughout a tree (§ 4).

Engineering topologies to support local failure reactions comes with a cost, namely, the tree supports fewer hosts and accommodates less hierarchical aggregation. We formalize a tree's scalability properties in terms of its FTV, and find that the introduction of redundant links *at a single level of the tree* results in a multiplicative decrease to the maximum number of hosts that can be supported by the tree. That is, we reduce the total number of hosts in the tree by 50% for each level at which we increase from 0 to 1 the number of link failures tolerable without host disconnection. Interestingly, improving fault tolerance by increasing the number of switches and links in a topology (while keeping host count fixed) has the potential to do more harm than good by introducing substantially more points of failure. However, we show (§ 7.2)that the decreased convergence time enabled by local reaction more than makes up for the added opportunity for link failures.

In § 5, we offer a failure reaction protocol that leverages an Aspen tree's redundant links; this lends intuition to our discussion (§ 6) of the options for interconnection patterns among Aspen tree switches. Finally, in § 8 we provide a thorough analysis of the tradeoffs between fault tolerance, scalability, and network size for a variety of trees.

2. MOTIVATION AND CONTEXT

In a traditional fat tree, 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 node in the topology*. For 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.

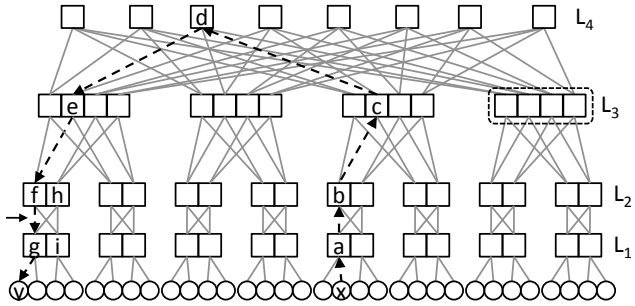


Figure 1: Packet Travel in a 4-Level, 4-Port Fat Tree

This means that in the worst case, information about 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. The time for this information to propagate grows with the depth of the tree, and the time for recalculating routing state and updating forwarding tables can be substantial. Isolated link failures are both common and impactful in the data center [9], so timely reaction is critical.

There are alternative routing techniques that avoid packet loss. For instance, bounce routing techniques work around failures by temporarily sending packets away from a destination. In Figure 1, a bounce routing-based 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. However, bounce routing based on local information introduces additional software complexity to support the calculation and activation of extra, non-shortest path entries and to avoid forwarding loops. Additionally, combining bounce routing with flow control protocols can lead to deadlock [7, 14].

It is also possible to construct a protocol that sends a packet back along its path to the nearest switch that can re-route around a failed link, similar to the technique employed by data-driven connectivity (DDC) [21]. In DDC, a packet sent along the path in Figure 1 would need to travel from f back up to the top of the tree and then down three levels to a before it could be re-routed towards h. DDC provides the 'ideal connectivity" property, in which packets are not dropped unless the destination is physically unreachable. However, this comes at the cost of (temporarily) introducing long paths. We more closely compare Aspen trees to DDC in § 9.

Our approach is to offer an alternative to bouncing packets in either direction. We modify the fat tree by introducing redundancy at one or more levels; this allows switches to handle a failure locally without requiring global re-convergence of topology information, but at a cost in the topology's scale. We coin the resulting modified fat trees Aspen trees.

Before describing Aspen trees in detail, we first define several key terms and conventions. An n-level, k-port Aspen tree consists of switches at levels L_1 through L_n (as marked in Figure 1) and hosts at 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*. 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 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. For now, we do not consider multi-homed hosts, given the associated addressing complications.

3. DESIGNING ASPEN TREES

In this section, we describe our method for generating trees with varying fault tolerance properties. 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.

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 decide to provide redundant connections from s to pods q and r. We first need to free some upward facing ports from q and r, and we chose 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 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. In general, we use L_i fault tolerance to refer to L_i-to-L_{i-1} links.

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.

3.1 Preliminary Assumptions

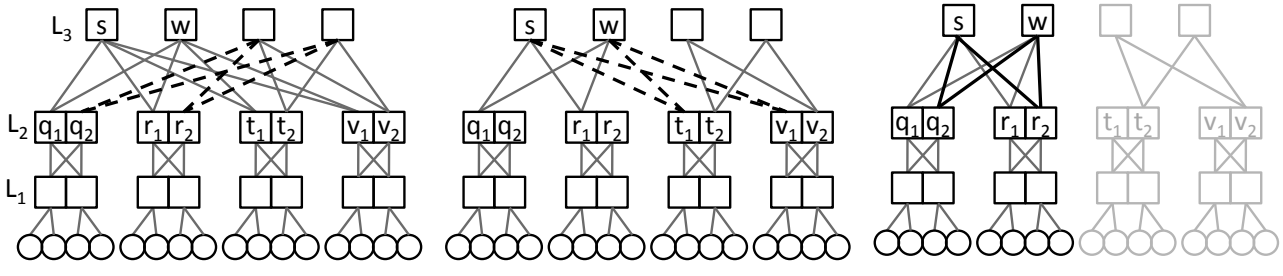
In order to limit our attention to a tractable set of options, we introduce a few restrictions on the trees that we wish to generate. First, we consider only trees in which switches at each level are divided into pods of uniform size. That is, all pods at L_i must be of equal size, though this size may differ from that of the pods at $L_{f:f\neq i}$. Similarly, within a single level, all switches have equal fault tolerance to neighboring pods in the level below, but the fault tolerance of switches at L_i need not equal that of switches at $L_{f:f\neq i}$.

3.2 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 work together to form a coherent topology.

3.2.1 Variables and Constraints

Before presenting the technical details of our algorithm, we first introduce several helpful variables and the relationships amongst them. Recall that an Aspen tree has n levels



(a) Freeing Uplinks from L_2 (b) Selecting L_2-L_3 Links for Repurposing (c) Reconnecting Redundant Links

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

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 allow for full bisection bandwidth, the number of switches at all levels of the tree except L_n must be the same. 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.

At each level L_i, our algorithm first groups switches into pods and then 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 variables 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. Combining these variables with the number of switches at each level, we have the constraint:

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

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 (§ 3.1) 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:

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

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 § 6.

3.2.2 Aspen Tree Generation Algorithm

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

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 defer calculation of m_n until the value of S is determined.

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.[2] 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.

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
```

3.3 Aspen Trees with Fixed Host Counts

The algorithm in Listing 1 demonstrates a method for creating Aspen trees given a fixed switch size (k), number of tree levels (n), and desired fault tolerance values ($c_2...c_n$). The number of hosts that the topology supports is an output 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 the same as 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

[2] Alternatively, we could accept as an input, desired per-level fault tolerance values. In this case, we would set each c_i value by adding 1 to the desired fault tolerance for L_i.

values according to the desired fault tolerance.[3] A concern with this alternate generation algorithm is that the addition of more network elements (and their interconnecting links) introduces more points of failure. We more carefully consider this concern in § 7.2 and show that the ability to react to failures locally outweighs the increased likelihood of a packet encountering a failure.

4. ASPEN TREE PROPERTIES

An Aspen tree generated by the algorithm of § 3 is defined by the set of per-level values selected for p_i, m_i, r_i, and c_i; these values in turn determine the per-level fault tolerance, the number of switches needed and hosts supported, and the amount of hierarchical aggregation from one level to the next.

4.1 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 fault tolerance of a tree as a whole, 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. $<c_n - 1,...c_2 - 1>$.[4] 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 Aspen trees made up of 6-port switches, each with different FTVs. Figure 3(a) lists all possible n=4, k = 6 Aspen trees, omitting trees that have 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 this tree, 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.

4.2 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:

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

We use Equation 2 and the fact that S is equal to the number of pods at L_1 to express S as a function 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, adds one to 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.[5] For instance, the DCC of an Aspen tree with FTV <1,2,3> is 2 × 3 × 4=24. We rewrite the equation for S as:

$$S = \frac{k^{n-1}}{2^{n-2}} \times \frac{1}{DCC} \tag{4}$$

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) gives the number of switches in each example tree, given that $n - \frac{1}{2}$=3.5.

4.3 Number of Hosts Supported

The most apparent cost of adding fault tolerance to an Aspen tree 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 minimal fat tree, the number of hosts in 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 of L_1 switches multiplied by the number of downward facing ports per L_1 switch. That is,

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

As Equation 5 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

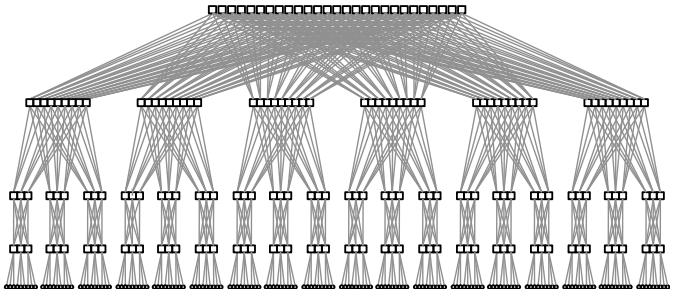
[3] In this case, the desired fault tolerance values must be known a priori in order to determine the number of levels that must be added to the corresponding fat tree.

[4] One could instead consider a *Connection Vector*, $CV = <c_n,...c_2>$. While a CV would fit more easily into our derivations, we chose the FTV because it expresses the topology's fault tolerance at a glance.

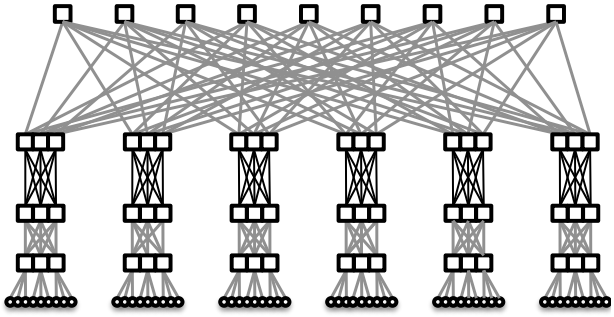
[5] The DCC expresses the number of distinct paths from an L_n switch s to an L_1 switch t.

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**

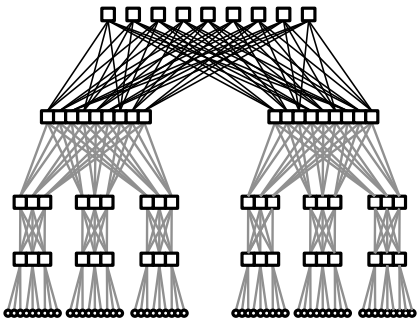
(a) All Possible 4-Level, 6-Port Aspen Trees
(Bold rows correspond to topologies pictured.)



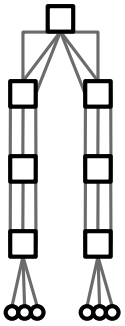
(b) Unmodified 4-Level 6-Port Fat Tree: FTV=< 0,0,0 >



(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

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 § 7.1.

4.4 Hierarchical Aggregation

Another property of interest is hierarchical aggregation, that is, the number of L_{i-1} pods that are folded into each L_i pod. While hierarchical aggregation may be less of a concern than the number of hosts supported, it plays a role in determining the efficiency of certain communication schemes. For hierarchical topologies, a labeling scheme such as those in [24, 28] can be used to enable compact forwarding state. In this type of labeling scheme, descendant switches below a given L_i pod share the same label prefix, and therefore it is desirable to group as many L_{i-1} switches together as possible under a single L_i switch. The hierarchical aggregation at L_i of an Aspen tree expresses the number of L_{i-1} pods to which each L_i switch connects, and can be written as $\frac{m_i}{m_{i-1}}$.

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. So if the c_i value for a switch s is increased, the extra links must come from other downward neighbors of s. This necessarily reduces the number of pods to which s connects below.

It is difficult to provide an equation that directly relates fault tolerance and hierarchical aggregation to one another at a single level, because hierarchical aggregation is not a single-level concept. To increase the hierarchical aggregation at L_i ($\frac{m_i}{m_{i-1}}$) we must either increase m_i or decrease m_{i-1}. However, this in turn reduces hierarchical aggregation at either L_{i+1} or L_{i-1}. Because of this, we consider the hierarchical aggregation across the entire tree. While this does not provide a complete picture, it does give intuition about the tradeoff between fault tolerance and hierarchical aggregation. We express an Aspen tree's overall hierarchical aggregation as the product of its per-level hierarchical aggregation values:

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

Therefore, hierarchical aggregation relates to an Aspen's FTV in an identical manner to that of host count; an additive increase to a level's c_i value results in a multiplicative reduction in hierarchical aggregation by the same factor. Figure 3(b) has the maximal possible hierarchical aggregation at each level (in this case, 3) while Figure 3(e) has no hierar-

chical aggregation at all. The additional fault tolerance at a single level of each of Figures 3(d) and 3(c) costs these trees a corresponding factor of 3 in overall aggregation. The values related to hierarchical aggregation for all possible n=4, k=6 Aspen trees are given in Figure 3(a).

5. LEVERAGING FAULT TOLERANCE

Recall from the example of § 2 that a minimal fat tree has no choice but to drop a packet arriving at a switch incident on a failed link. In fact, in Figure 1, a packet sent from host x to host y would be doomed to be lost the instant that x's ingress switch a selected b as the packet's next hop. Extra fault tolerance links push this "dooming" decision farther along the packet's path; this reduces the chances that a packet will be dropped due to a failure that occurs while the packet is in flight. Moreover, keeping the set of switches that need to react close to a failure limits both the convergence time and overhead of the reaction.

Figure 4 shows an n=4, k=4 Aspen tree, modified from the 4-level, 4-port fat tree of Figure 1 to have an FTV of <0,1,0>, that is, it has additional fault tolerance links between L_3 and L_2. As described in § 4, this comes at the cost of half of the hosts in the tree. The additional links between L_3 and L_2 give a packet sent from x to y an alternate path through h, as indicated by the darkened arrows. If switch e knows about the failure of link f-g, it can route packets towards h rather than f. Thus the switch that needs to know about the failure (e) is relatively far along the packet's path, whereas in the traditional fat tree of Figure 1, knowledge of the failure needs to propagate all the way back to the sender's ingress switch.

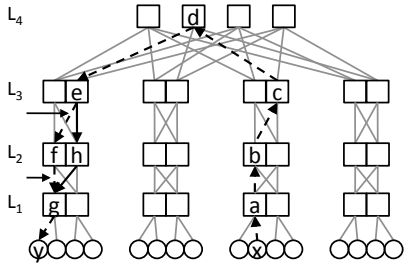


Figure 4: 4-Level, 4-Port Aspen Tree with FTV=<0,1,0>

5.1 Failure Notification Protocol Overview

A key reason for the slow convergence of broadcast-based protocols (e.g.OSPF and IS-IS) in the data center is the need to disseminate topology information to every possible sender after a single link failure. Each switch performs expensive calculations that grow with the size of the topology, and routing updates propagate through a number of hops proportional to the depth of the tree. In Aspen trees, added fault tolerance links limit the set of switches that react to a link failure to the ancestors of a switch incident on the failure.

We leverage these fault tolerance links in Aspen trees by considering an insight similar to that of failure-carrying packets [17]: 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 notification protocol to react to transient link failures and recoveries. With this protocol, notifications are sent upwards to ancestors located near a failure, rather than being broadcast throughout the entire tree. 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.[6]

5.2 Propagating Failure Notifications

To determine the set of ancestors that should receive a failure notification, we 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 path simply changes on the fly. This is because each of a switch's uplinks leads to some subset of L_n switches. In § 3, 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. Therefore, no failure notifications are necessary to support re-routing of upward-moving packets.

The case in which a link fails along the downward segment of a packet's intended path is somewhat more complicated. Consider a failure that occurs 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 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:

1. c_i>1: The failed link is at a level with added fault tolerance.
2. c_i=1, c_{i+1}>1: The closest added fault tolerance is at the level immediately above the failure.
3. c_i=1, c_f>1, for some f>i+1: The nearest level with additional links is more than one hop above.

Case 1: 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. Since e is incident on the failed link, it does not need to propagate any notifications.

Case 2: Case (2) 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

[6]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.

be dropped. But if switch e learns of the failure of $f-g$ before the packet's arrival, it chooses 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: Finally, Figure 5 shows an example of case (3), in which L_2 link $f-g$ fails and the closest added fault tolerance is at L_4. Here, the nearest ancestor of f that can route around the failure is d. Upon a packet's arrival, d can select 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) with a downward path to another member of f's pod.

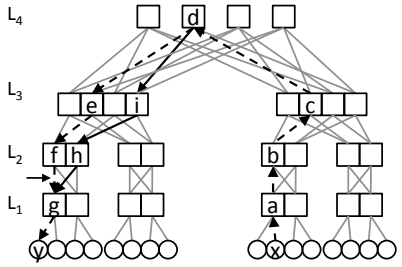


Figure 5: 4-Level, 4-Port Aspen Tree with FTV=<1,0,0>

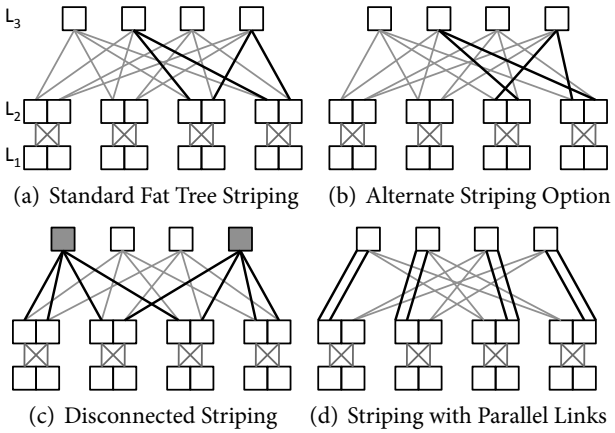
To generalize, when a link from L_i switch s to L_{i-1} neighbor t fails, s first determines whether it has non-zero fault tolerance. If so, it subsequently routes all packets intended for t to an alternate member of t's pod. Otherwise, s passes a failure notification (indicating the hosts that it no longer reaches) upwards. If an ancestor that receives this notification has alternate paths to these hosts, via an alternate member of s's pod, it adjusts its local state accordingly. Otherwise it forwards the notification upwards. As we show in § 8, these notifications introduce little complexity.

6. WIRING THE TREE: STRIPING

In § 3, 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 organization of connections between switches, a process we refer to as *striping*. We have deferred this discussion until now because of the topic's dependence on the techniques described in § 5 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 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 darkened lines.

Striping can affect connectivity, over-subscription ratios, and the effectiveness of redundant links in Aspen trees. Some striping schemes even disconnect switches at one level from



(a) Standard Fat Tree Striping (b) Alternate Striping Option

(c) Disconnected Striping (d) Striping with Parallel Links

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

pods below. In fact, we made a striping assumption in § 3 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 single 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. Since the techniques of § 5 rely on the existence of ancestors common to a switch s incident on a failed link and alternate members of s's pod, a reasonable striping policy must yield such common ancestors. This means, for instance, that striping patterns should not consist entirely of duplicate, parallel links; In the example of Figure 4, had switch e simply had duplicate parallel connections to f, it would not be able to route around the failure of link $f-g$. In general, the following striping policy is necessary to enable re-routing: *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.*

7. DISCUSSION

In this section, we consider instances of Aspen trees that provide a significant reduction in convergence time at a moderate scalability cost (e.g. 80% faster convergence with 50% host loss). We also more closely examine the tradeoff between decreased convergence time and newly introduced points of failure when generating Aspen trees with fixed host counts.

7.1 Practical Aspen Trees

We showed in § 5 that the most useful and efficient fault tolerance is (1) above failures and (2) as close to failures as possible. We formalize this in terms of the FTV. The most fault-tolerant tree has an FTV with all maximal (and non-

zero) entries. However, such an FTV may come at too high of a scalability cost. To enable usable and efficient fault tolerance, in FTVs with non maximal entries it is best to cluster non-zero values to the left while simultaneously minimizing the lengths of series of contiguous zeros. For instance, if we can put only two non-zero entries in an FTV of length 6, the ideal placement would be <1,0,0,1,0,0>. There are at most two contiguous zeros, so updates propagate a maximum of two hops, and each 0 has a corresponding 1 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 (and towards the left of an FTV), the largest value-add with minimal scalability 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 has an FTV of <1,0,0,...> and a DCC of 2, and therefore supports half as many hosts as does a traditional fat tree. The average convergence propagation distance for this tree is less than half of that for a traditional fat tree, and more importantly, all updates only travel upward rather than fanning out to all switches in the tree. For instance, for an Aspen tree with n=4, k=16 and FTV=<1,0,0,...>, host count is reduced by only 50% while convergence time is reduced by 80% from that of the corresponding fat tree. In fact, the topology used for VL2 [11] is an instance of an Aspen tree with an FTV of <1,0,0,...>.

7.2 Aspen Trees with Fixed Host Counts

Thus far we have approached the design of Aspen trees from the viewpoint of first selecting the network size and desired fault tolerance, and then determining the amount of host support as compared to that of a traditionally-defined fat tree. The algorithm of Listing 1 accepts as inputs the values for k (switch size) and n (tree depth) and the desired fault tolerance, and these values in turn determine the number of hosts supported (Equation 5). If we instead fix both the number of hosts supported and the desired fault tolerance, network size becomes the dependent variable.

A concern with creating Aspen trees in this manner relates to the fact that an Aspen tree with non-zero fault tolerance needs more levels of switches to support the same number of hosts than does a traditionally-defined 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 failure resulting from the addition of switches and links. To evaluate this, we first calculate the number of links added if we fix the number of hosts while turning a given fat tree into an Aspen tree with non-zero fault tolerance. We then compute the average convergence times across all links for both trees. Finally, we multiply the number of links in each tree by its average per-link convergence time and compare the two results.

We present this comparison for Aspen trees with FTVs of <1,0,0,...> (as suggested in § 7.1); we omit a full derivation across all Aspen tree types for brevity. For a fixed switch size, the ratio of failure convergence cost of a fat tree of depth n to that of an Aspen tree with the same host count is:

$$(n-\frac{1}{2})(3n-4) : (n+\frac{1}{2})(n-1) \quad (6)$$

Figure 7 depicts this ratio varied across the tree depths that we expect to see in practice. This confirms that the overall convergence cost in a fat tree is always larger than that in an Aspen tree with the same host count and an FTV of <1,0,0,...>, and this benefit increases with the depth of the tree.

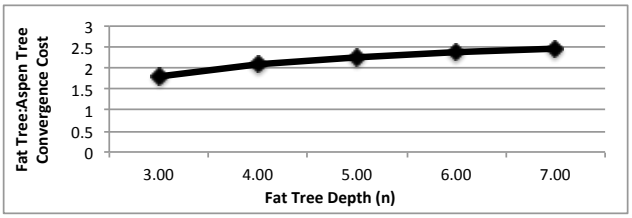


Figure 7: Fat Tree versus Aspen Tree Convergence Costs

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.

It is important to note that by adding more levels of switches and links to an Aspen tree, we increase the path length for the common case packet that does not encounter a failure. Because this occurs in the data plane, this amounts to a latency increase on the order of nano or microseconds. However, this may not be acceptable in all scenarios. As such, it is crucial that a data center operator use only the minimal added fault tolerance that is absolutely necessary for correct operation when building an Aspen tree with fixed host count.

8. 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 given network sizes, from the perspective of the discussion and algorithm in § 3.2. That is, we consider the scalability cost of adding redundant links to a traditionally-defined fat tree of fixed depth. Next, we consider Aspen tree tradeoffs from the point of view introduced in § 3.3. For a fixed host count, we show the increase in network size necessary to decrease re-convergence time by varying amounts.

8.1 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 eliminates the need for global re-convergence of broadcast-based routing protocols on failure, and instead relies on a simple failure notification protocol. These failure notifications require less processing time, travel shorter distances, and are sent to fewer switches, significantly reducing convergence time and control overhead.

If the fault tolerance at 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 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 overall scalability, in terms of both host support and hierarchical aggregation. Each FTV entry $x > 0$ reduces the maximum possible number of hosts as well as the tree's hierarchical aggregation; this reduction is by a factor of $x+1$.

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. 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. We average this propagation distance across failures at all levels of the tree[7] to give a metric for expressing average convergence time for a tree.

Alongside convergence time, 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 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.[8] To normalize, values are shown as percentages of the worst case. The graphs for hierarchical aggregation show identical trends; we omit them for brevity.

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.

We observe that there are often several ways to generate trees with the same host count but with differing convergence times. This is shown in the second, third and fourth entries

[7] We exclude 1^{st} hop failures as neither technique mitigates these.

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

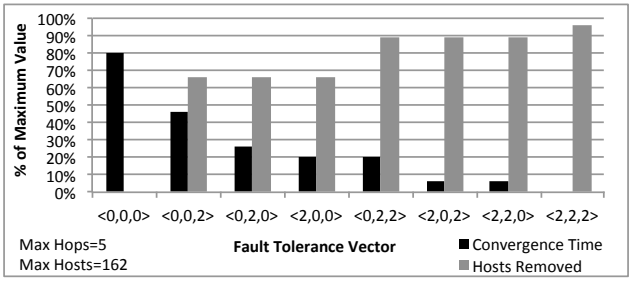


Figure 8: Convergence vs. Scalability: n=4, k=6 Aspen Trees

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.

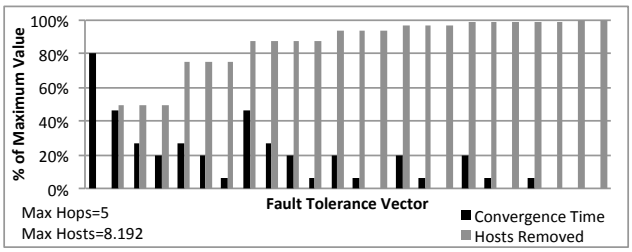
We now examine more realistically sized Aspen trees. In practice, we expect trees with $3 \leq n \leq 7$ levels and $16 \leq k \leq 128$ ports per switch, in support of tens of thousands of hosts. Figures 9(a) and 9(b) show graphs similar to that of Figure 8, for 16-port trees of depths 4 and 5, respectively. Because of the large number of configuration options for these values of n and k, we often find that numerous 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 resulting graphs.

Figures 9(a) and 9(b) show the same trend as does Figure 8, but since there are more options for generating trees, the results are perhaps more apparent. As we move from left to right in the bar graphs, we remove more hosts. However, the host removal bars in the graph are grouped into steps; each individual number of hosts removed corresponds to several different values for average convergence time. We mark one such step in Figure 9(b) with arrows. In this case, if we are constrained by the number of hosts to support, we would select the rightmost entry in the corresponding step, i.e. that with the smallest convergence time.

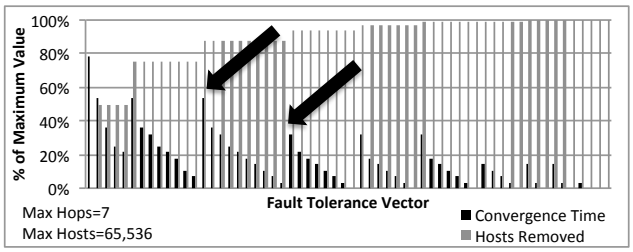
Figure 10 shows trees with larger switches (k=32 and 64) but with smaller values for the depth of the tree (n=3) so as to keep our results in line with the topology sizes we expect to see in practice. For these graphs, we again collapsed duplicates and thus omitted FTV labels, but since the small number of levels limits the number of possible trees, there are fewer entries than in the graphs of Figure 9. These results again show that with only modest reductions to host count, the reaction time of a tree can be significantly improved.

8.2 Convergence versus Network Size

We next consider the scenario in which a data center operator wishes to add fault tolerance to an existing topology by increasing the network size (i.e. by adding switches and links)

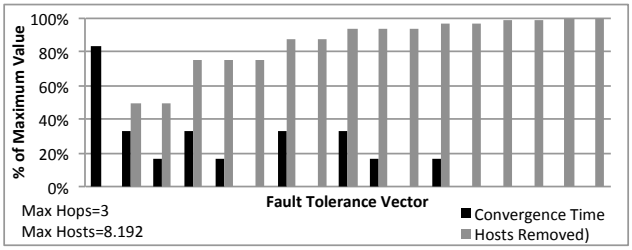


(a) Convergence Time vs. Hosts Removed: n=4, k=16 Aspen Trees

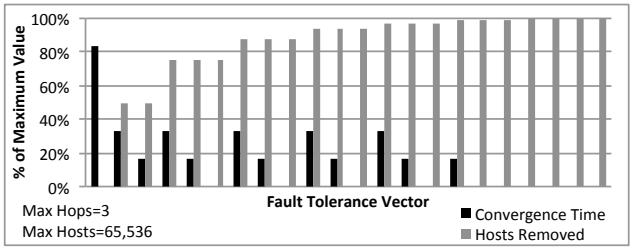


(b) Convergence Time vs. Hosts Removed: n=5, k=16 Aspen Trees
Arrows signal varying convergence time with single host count value

Figure 9: Convergence versus. Scalability Tradeoff for 4- and 5-Level, 16-Port Aspen Trees



(a) Convergence Time vs. Hosts Removed: n=3, k=32 Aspen Trees



(b) Convergence Time vs. Hosts Removed: n=4, k=64 Aspen Trees

Figure 10: Convergence vs. Scalability Tradeoff for 3-Level, 32- and 64-Port Aspen trees

while keeping host count constant. We examine the various Aspen tree options for a given host count, measuring for each tree the number of additional switches needed, the number of switches involved in a failure reaction, and the tree's re-convergence time. These values correspond to the added financial cost, reduced control overhead, and decreased convergence time, respectively, of the expanded network.

8.2.1 Implementation and Simulation

We implemented our failure notification protocol in Mace, a language for distributed systems development. We selected Mace because of the language's accompanying model checker and simulator that allow us to test our failure notification protocol over a variety of different Aspen trees and sets of random failures. We used safety and liveness properties in the Mace model checker to verify the correctness of our protocol and we used the Mace simulator to compare the failure reaction time and overhead of our protocol to those of a reference implementation of a link-state protocol based on OSPF.

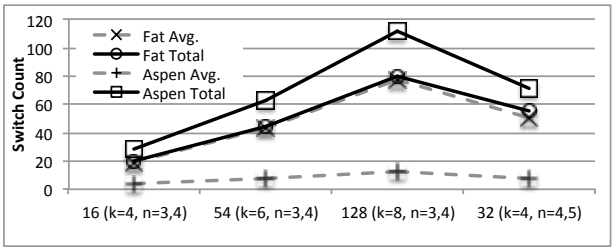
We built a topology generator that takes as inputs the tree depth (n), switch size (k), and FTV, and creates an n-level Aspen tree of k-port switches matching the input FTV. We used this generator to create input topologies for the Mace model checker as follows: for varying values of k and n, we created an n-level, k- port fat tree and a corresponding (n+1)-level, k-port Aspen tree with FTV <x,0,0,...>, where x is a value that leads to identical host counts in both trees. We highlight this particular FTV for the reasons introduced in § 7.1.

For each pair of trees, we initially ran our link-state protocol to set up routes for the base deployed topology and verified the accuracy of these routes using the model checker. We then failed each link in each tree several times and allowed the corresponding recovery protocol to react and restore the switches' forwarding tables. For fat trees, we used standard link-state advertisements (LSAs) and for Aspen trees, we used the protocol described in § 5. We recorded the minimum, maximum, and average re-convergence time across all failures for each tree, as well as the minimum, maximum and average number of switches involved in each failure and the number of switches needed to build each tree.

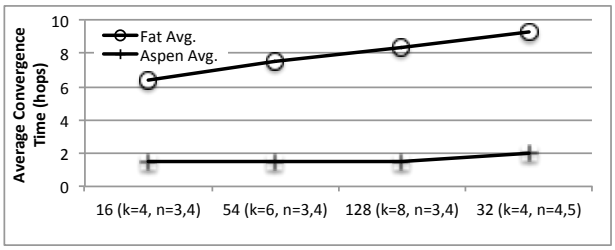
Figure 11(a) shows the total number of switches in each fat tree and corresponding Aspen tree, along with the average number of switches involved in each failure reaction.[9] The x-axis gives the number of hosts in the tree, the switch size (k), and the depth (n) of the fat tree and Aspen tree, respectively. 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 increases of 40%, 29% and 22%, for 3, 4 and 5-level fat trees, respectively.

Figure 11(a) expresses the cost of the Aspen tree network in terms of increased switch count as well as the benefit in terms of the reduced number of switches that react to each failure. In general, the link-state protocol involves most of its switches on each failure reaction. In fact, we would expect that all switches always process all LSAs, but our measurements are conservative and do not attribute an LSA to a switch unless the switch's forwarding table must be recalcu-

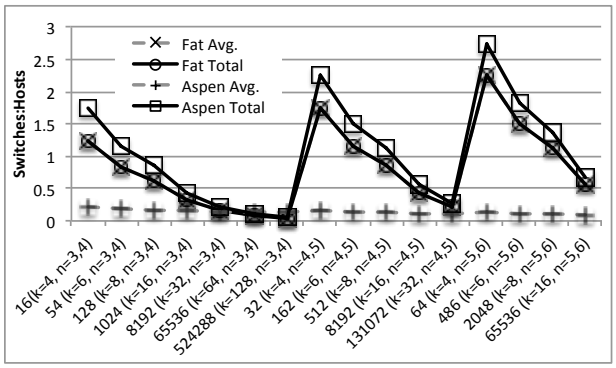
[9]We verified that the minimums and maximums matched the ranges calculated in § 8.2.2, and graph only the averages for clarity.



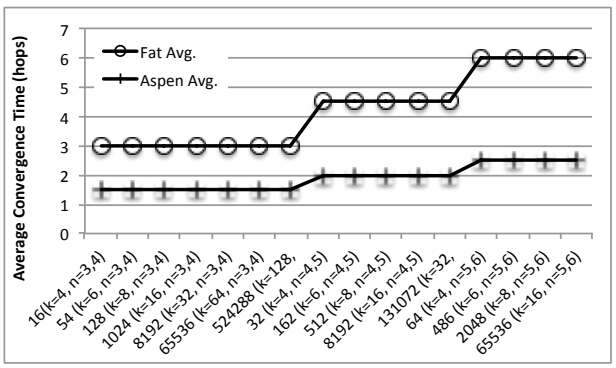
(a) Total Switches vs. Reactive Switches: Small Trees



(b) Convergence Time: Small Trees



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



(d) Convergence Time: Large Trees

Figure 11: Network Size, Convergence Time, and Control Overhead for Pairs of Corresponding Fat and Aspen Trees

lated as a result of receiving the LSA. On the other hand, less than 15% of the total switches in an Aspen tree are involved in processing each failure.

Finally, as Figure 11(b) shows, the convergence time for each fat tree is substantially longer than that for the corresponding Aspen tree and this difference grows with increasing values of n. Our measurement is somewhat conservative, as it displays convergence time in numbers of hops. For a traditional OSPF deployment, the compute time at each switch would be in the hundreds of milliseconds, whereas the time to process the corresponding Aspen notifications would be one or even two orders of magnitude less. Thus, the difference between the two plots would be even more pronounced in reality.

8.2.2 Large Tree Analysis

Since the model checker operates on instances with at most a few hundred nodes, we use a separate analysis for networks comparable in size to today's mega data centers. Figure 11(c) is similar to Figure 11(a), but shows a wider range of topologies. The two graphs differ in that in Figure 11(c), we plot switch-to-host ratios in order to normalize our results across a wide range of trees. As in our simulations, link-state reconvergence involves all switches in the tree, whereas only 10 – 20% of the switches in an Aspen tree react to a failure.

Figure 11(d) compares the average convergence times for each pair of trees, confirming the drastic reduction in convergence time that Aspen trees can provide. These results are again a conservative estimate, as they do not take into account the significant difference in computation time for processing an Aspen notification versus an LSA. Our analytical results align well with our simulations (§ 8.2.1), giving us confidence in the accuracy of our measurements.

9. RELATED WORK

Our work is largely motivated by direct experience with large data center operations as well as by the findings of [9], which shows that link failures are common, isolated, and impactful. The network redundancy inherent to multi-rooted tree topologies 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). With the design of Aspen trees and our corresponding failure notification protocol, we better leverage network redundancy for quick failure reaction. The study in [9] also find that links in the core of the network have the highest probability of failure and benefit most from network redundancy. This aligns well with our recommended Aspen trees of § 7.1. Finally the study shows that link failures are sporadic and short-lived, supporting our belief that such failures should not cause global OSPF re-convergence.

9.1 Alternative Routing Techniques

Modifying topologies to provide inherent fault tolerance in the form of redundant links has a significant cost in terms of a topology's scalability or network size. Alternative routing techniques provide another means of addressing fault tolerance in a network. We discussed bounce routing and data-driven connectivity [21] in § 2; here we consider more closely DDC and other alternative routing techniques.

DDC and Aspen trees come from the similar motivation that it is unacceptable to disrupt communication for tens of seconds while waiting for control plane re-convergence. DDC approaches the problem by bouncing a packet that encounters a failure back along its path until it reaches a node with an alternate possible 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 failure that causes the packet to bounce backwards. 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 reach its destination are made as early as the first hop along a packet's path. The authors of [21] hint at this in their evaluation of DDC's effectiveness over various topologies. They note that fat trees lack "resilient nodes" that provide 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. Additionally, in order to keep forwarding state manageable, DDC supports only exact-match forwarding, as opposed to the longest-prefix-matching (LPM) style of forwarding often required in the data center. With Aspen trees, we leverage the regularity of the underlying topology to provide fast failure reaction far along a packet's path, without introducing long paths or sacrificing LPM-style forwarding.

Failure carrying packets (FCP) [17] eliminate the convergence process after a failure by having data packets carry failure information. FCPs leverage the fact that an intradomain ISP network has a set of relatively stable links, in terms of existence, if not availability. Therefore, if all routers know the full physical network topology, they simply need to learn the set of links that are unavailable for a given packet. FCP provides guaranteed eventual delivery of packets if the graph does not become disconnected. However, the implementation and deployment cost of introducing a new data plane may hinder the adoption of FCP in the data center, and the paths ultimately taken by packets can be long. This (temporary) introduction of long paths is a difficulty inherent to many alternative routing techniques; the fact that Aspen trees have fixed path lengths renders this a non-issue.

Multi-path TCP (MPTCP) [25] 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. A downside of MPTCP is its reliance on host modifications.

The idea behind this work is derived from fast failure recovery [16] techniques in WANs. Our approach is to engineer data center topologies so as to enable FFR for link failures.

An interesting idea for future work would be to design a hybrid approach that combines limited topology modifications with redundancy-aware alternative routing techniques.

9.2 Backup Paths

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) for a flow fails. Many works consider this topic in the context of either ad hoc networks or resource allocation for performance guarantees. Generally, such works fall into two camps. Some advocate assigning backup paths on flow entry [12, 13, 15, 26, 30], so that a flow continues to function after the failure of its primary path and even $N-1$ of its N backup paths. This comes at the cost of potentially wasting resources that are reserved for backup paths but are rarely or never used, as well as the time cost of calculating backup paths prior to flow admission. Also, for flows with strict performance requirements, it is difficult to pre-compute backup paths in the face of dynamic traffic.

Other approaches [2, 3, 29] establish a backup path on the fly at the time of a failure. The downsides of this technique are: the possibility of contention for new paths among several simultaneously affected flows, the time to calculate new paths upon failure, and the fact that recovery is not guaranteed for any given flow. However, this approach avoids the drawback of potentially wasting valuable bandwidth as well as the time cost of setting backup paths on flow entry. 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.

The authors of [2] and [3] study dynamic backup path calculation along several metrics, varying the portion of the path recalculated, the timing of recalculation, and the possibility of retrying recalculation. Their findings show that when one physical link failure affects multiple flows, local reaction is faster. This supports our belief that it is ideal to keep the failure reaction as close as possible to the failure itself.

The authors of [29] present a hybrid method, calculating backup paths prior to failure, but admitting flows once a primary path has been found without waiting for backup path calculation to complete. Backup paths are not complete paths, but rather *patches* that avoid failed links along portions of a path. While this approach differs from ours in its use of source routing, it is similar in its use of local failure reaction.

9.3 High Performance Computing Topologies

Our topologies derive from the initial presentations of fat trees as non-blocking architectures for communication in supercomputers [6, 19]. The traditionally defined fat tree of Figure 1 comes from DeHon's Butterfly Fat-Tree [8]. A number of works have extended traditional fat tree topologies by essentially raising c_i from 1 to 2 uniformly at all levels of the tree. Upfal's multi-butterfly networks [27], Leighton et al's corresponding routing algorithms [18], and Goldberg et al's splitter networks [10] all give examples of these subsets of our trees. These works consider topologies in the context a priori message scheduling rather than that of running packet-switched protocols (e.g. IP) over modern switch hardware in today's data centers.

10. 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 availability 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. 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.

11. REFERENCES

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

[2] A. Banerjea. Fault Recovery for Guaranteed Performance Communications Connections. *ToN*, Oct 1999.

[3] A. Banerjea, C. J. Parris, and D. Ferrari. Recovering Guaranteed Performance Service Connections from Single and Multiple Faults. *GLOBECOM 1994*.

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

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

[6] C. Clos. A Study of Non-Blocking Switching Networks. *BSTF*, Mar 1953.

[7] W. J. Dally and H. Aoki. Deadlock-Free Adaptive Routing in Multicomputer Networks Using Virtual Channels. *ToPaDS*, Apr 1993.

[8] A. DeHon. Compact, Multilayer Layout for Butterfly Fat-Tree. *SPAA 2008*.

[9] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. *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*, 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. *SIGCOMM 2009*.

[12] S. Guo and O. W. Yang. Performance of Backup Source Routing in Mobile Ad Hoc Networks. *WCNC 2002*.

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

[14] M. Karol, S. J. Golestani, and D. Lee. Prevention of Deadlocks and Livelocks in Lossless Backpressured Packet Networks. *ToN*, Dec 2003.

[15] S. Kim and K. G. Shin. Improving Dependability of Real-Time Communication with Preplanned Backup Routes and Spare Resource Pool. *WQOS 2003*.

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

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

[18] F. T. Leighton and B. M. Maggs. Fast Algorithms for Routing Around Faults in Multibutterflies and Randomly-Wired Splitter Networks. *ToC*, May 1992.

[19] C. E. Leiserson. Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing. *ToC*, Oct 1985.

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

[21] J. Liu, B. Yan, S. Shenker, and M. Schapira. Data-Driven Network Connectivity. *HotNets 2011*.

[22] 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. *HotNets 2010*.

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

[24] R. N. 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. *SIGCOMM 2009*.

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

[26] 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*, Aug 2008.

[27] E. Upfal. An O(logN) Deterministic Packet Routing Scheme. *STOC 1998*.

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

[29] Y.-H. Wang and C.-F. Chao. Dynamic Backup Routes Routing Protocol for Mobile Ad Hoc Networks. *InfSci*, Jan 2006.

[30] D. Xu, Y. Xiong, C. Qiao, and G. Li. Failure Protection in Layered Networks with Shared Risk Link Groups. *IEEE Net.*, May-June 2004.