A Scalable, Commodity Data Center Network Architecture

Mohammad Al-Fares
malfares@cs.ucsd.edu

Alexander Loukissas
aloukiss@cs.ucsd.edu

Amin Vahdat
vahdat@cs.ucsd.edu

Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0404

ABSTRACT

Today's data centers may contain tens of thousands of computers with significant aggregate bandwidth requirements. The network architecture typically consists of a tree of routing and switching elements with progressively more specialized and expensive equipment moving up the network hierarchy. Unfortunately, even when deploying the highest-end IP switches/routers, resulting topologies may only support 50% of the aggregate bandwidth available at the edge of the network, while still incurring tremendous cost. Non-uniform bandwidth among data center nodes complicates application design and limits overall system performance.

In this paper, we show how to leverage largely commodity Ethernet switches to support the full aggregate bandwidth of clusters consisting of tens of thousands of elements. Similar to how clusters of commodity computers have largely replaced more specialized SMPs and MPPs, we argue that appropriately architected and interconnected commodity switches may deliver more performance at less cost than available from today's higher-end solutions. Our approach requires no modifications to the end host network interface, operating system, or applications; critically, it is fully backward compatible with Ethernet, IP, and TCP.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Network topology; C.2.2 [**Network Protocols**]: Routing protocols

General Terms

Design, Performance, Management, Reliability

Keywords

Data center topology, equal-cost routing

1. INTRODUCTION

Growing expertise with clusters of commodity PCs have enabled a number of institutions to harness petaflops of computation power and petabytes of storage in a cost-efficient manner. Clusters consisting of tens of thousands of PCs are not unheard of in the largest institutions and thousand-node clusters are increasingly common in universities, research labs, and companies. Important applications classes include scientific computing, financial analysis, data analysis and warehousing, and large-scale network services.

Today, the principle bottleneck in large-scale clusters is often inter-node communication bandwidth. Many applications must exchange information with remote nodes to proceed with their local computation. For example, MapReduce [12] must perform significant data shuffling to transport the output of its map phase before proceeding with its reduce phase. Applications running on cluster-based file systems [18, 28, 13, 26] often require remote-node access before proceeding with their I/O operations. A query to a web search engine often requires parallel communication with every node in the cluster hosting the inverted index to return the most relevant results [7]. Even between logically distinct clusters, there are often significant communication requirements, e.g., when updating the inverted index for individual clusters performing search from the site responsible for building the index. Internet services increasingly employ service oriented architectures [13], where the retrieval of a single web page can require coordination and communication with literally hundreds of individual sub-services running on remote nodes. Finally, the significant communication requirements of parallel scientific applications are well known [27, 8].

There are two high-level choices for building the communication fabric for large-scale clusters. One option leverages specialized hardware and communication protocols, such as InfiniBand [2] or Myrinet [6]. While these solutions can scale to clusters of thousands of nodes with high bandwidth, they do not leverage commodity parts (and are hence more expensive) and are not natively compatible with TCP/IP applications. The second choice leverages commodity Ethernet switches and routers to interconnect cluster machines. This approach supports a familiar management infrastructure along with unmodified applications, operating systems, and hardware. Unfortunately, aggregate cluster bandwidth scales poorly with cluster size, and achieving the highest levels of bandwidth incurs non-linear cost increases with cluster size.

For compatibility and cost reasons, most cluster communication systems follow the second approach. However, communication bandwidth in large clusters may become oversubscribed by a significant factor depending on the communication patterns. That is, two nodes connected to the same physical switch may be able to communicate at full bandwidth (e.g., 1Gbps) but moving between switches, potentially across multiple levels in a hierarchy, may limit available bandwidth severely. Addressing these bottlenecks requires non-commodity solutions, e.g., large 10Gbps switches and routers. Further, typical single path routing along trees of interconnected switches means that overall cluster bandwidth is limited by the bandwidth available at the root of the communication hierarchy.

SIGCOMM'08, August 17–22, 2008, Seattle, Washington, USA.

Even as we are at a transition point where 10Gbps technology is becoming cost-competitive, the largest 10Gbps switches still incur significant cost and still limit overall available bandwidth for the largest clusters.

In this context, the goal of this paper is to design a data center communication architecture that meets the following goals:

- Scalable interconnection bandwidth: it should be possible for an arbitrary host in the data center to communicate with any other host in the network at the full bandwidth of its local network interface.
- Economies of scale: just as commodity personal computers became the basis for large-scale computing environments, we hope to leverage the same economies of scale to make cheap off-the-shelf Ethernet switches the basis for large-scale data center networks.
- Backward compatibility: the entire system should be backward compatible with hosts running Ethernet and IP. That is, existing data centers, which almost universally leverage commodity Ethernet and run IP, should be able to take advantage of the new interconnect architecture with no modifications.

We show that by interconnecting commodity switches in a fat-tree architecture, we can achieve the full bisection bandwidth of clusters consisting of tens of thousands of nodes. Specifically, one instance of our architecture employs 48-port Ethernet switches capable of providing full bandwidth to up 27,648 hosts. By leveraging strictly commodity switches, we achieve lower cost than existing solutions while simultaneously delivering more bandwidth. Our solution requires no changes to end hosts, is fully TCP/IP compatible, and imposes only moderate modifications to the forwarding functions of the switches themselves. We also expect that our approach will be the *only* way to deliver full bandwidth for large clusters once 10 GigE switches become commodity at the edge, given the current lack of any higher-speed Ethernet alternatives (at any cost). Even when higher-speed Ethernet solutions become available, they will initially have small port densities at significant cost.

2. BACKGROUND

2.1 Current Data Center Network Topologies

We conducted a study to determine the current best practices for data center communication networks. We focus here on commodity designs leveraging Ethernet and IP; we discuss the relationship of our work to alternative technologies in Section 7.

2.1.1 Topology

Typical architectures today consist of either two- or three-level trees of switches or routers. A three-tiered design (see Figure 1) has a *core* tier in the root of the tree, an *aggregation* tier in the middle and an *edge* tier at the leaves of the tree. A two-tiered design has only the core and the edge tiers. Typically, a two-tiered design can support between 5K to 8K hosts. Since we target approximately 25,000 hosts, we restrict our attention to the three-tier design.

Switches[1] at the leaves of the tree have some number of GigE ports (48–288) as well as some number of 10 GigE uplinks to one or more layers of network elements that aggregate and transfer packets between the leaf switches. In the higher levels of the hierarchy there are switches with 10 GigE ports (typically 32–128) and significant switching capacity to aggregate traffic between the edges.

[1]We use the term *switch* throughout the rest of the paper to refer to devices that perform both layer 2 switching and layer 3 routing.

We assume the use of two types of switches, which represent the current high-end in both port density and bandwidth. The first, used at the edge of the tree, is a 48-port GigE switch, with four 10 GigE uplinks. For higher levels of a communication hierarchy, we consider 128-port 10 GigE switches. Both types of switches allow all directly connected hosts to communicate with one another at the full speed of their network interface.

2.1.2 Oversubscription

Many data center designs introduce oversubscription as a means to lower the total cost of the design. We define the term *oversubscription* to be the ratio of the worst-case achievable aggregate bandwidth among the end hosts to the total bisection bandwidth of a particular communication topology. An oversubscription of 1:1 indicates that all hosts may potentially communicate with arbitrary other hosts at the full bandwidth of their network interface (e.g., 1 Gb/s for commodity Ethernet designs). An oversubscription value of 5:1 means that only 20% of available host bandwidth is available for some communication patterns. Typical designs are oversubscribed by a factor of 2.5:1 (400 Mbps) to 8:1 (125 Mbps) [1]. Although data centers with oversubscription of 1:1 are possible for 1 Gb/s Ethernet, as we discuss in Section 2.1.4, the cost for such designs is typically prohibitive, even for modest-size data centers. Achieving full bisection bandwidth for 10 Gb/s Ethernet is not currently possible when moving beyond a single switch.

2.1.3 Multi-path Routing

Delivering full bandwidth between arbitrary hosts in larger clusters requires a "multi-rooted" tree with multiple core switches (see Figure 1). This in turn requires a multi-path routing technique, such as ECMP [19]. Currently, most enterprise core switches support ECMP. Without the use of ECMP, the largest cluster that can be supported with a singly rooted core with 1:1 oversubscription would be limited to 1,280 nodes (corresponding to the bandwidth available from a single 128-port 10 GigE switch).

To take advantage of multiple paths, ECMP performs static *load splitting* among flows. This does not account for flow bandwidth in making allocation decisions, which can lead to oversubscription even for simple communication patterns. Further, current ECMP implementations limit the multiplicity of paths to 8–16, which is often less diversity than required to deliver high bisection bandwidth for larger data centers. In addition, the number of routing table entries grows multiplicatively with the number of paths considered, which increases cost and can also increase lookup latency.

2.1.4 Cost

The cost for building a network interconnect for a large cluster greatly affects design decisions. As we discussed above, oversubscription is typically introduced to lower the total cost. Here we give the rough cost of various configurations for different number of hosts and oversubscription using current best practices. We assume a cost of $7,000 for each 48-port GigE switch at the edge and $700,000 for 128-port 10 GigE switches in the aggregation and core layers. We do not consider cabling costs in these calculations.

Figure 2 plots the cost in millions of US dollars as a function of the total number of end hosts on the x axis. Each curve represents a target oversubscription ratio. For instance, the switching hardware to interconnect 20,000 hosts with full bandwidth among all hosts comes to approximately $37M. The curve corresponding to an oversubscription of 3:1 plots the cost to interconnect end hosts where the maximum available bandwidth for arbitrary end host communication would be limited to approximately 330 Mbps.

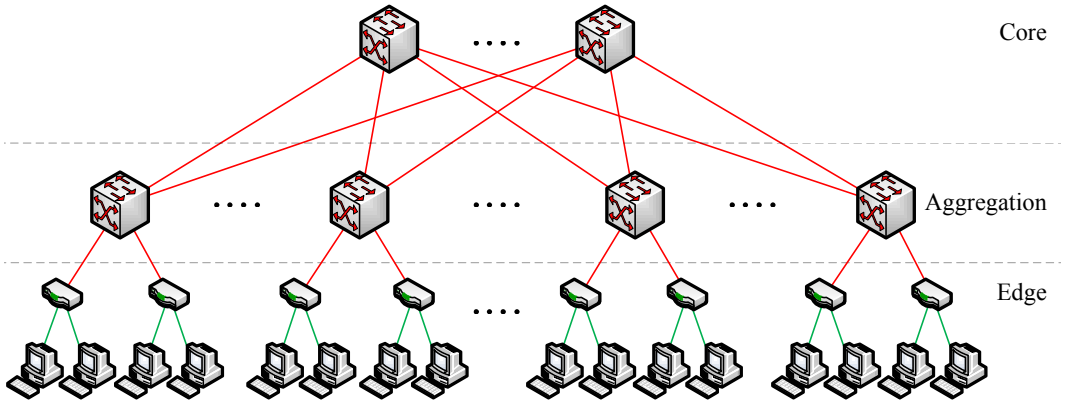

Figure 1: Common data center interconnect topology. Host to switch links are GigE and links between switches are 10 GigE.

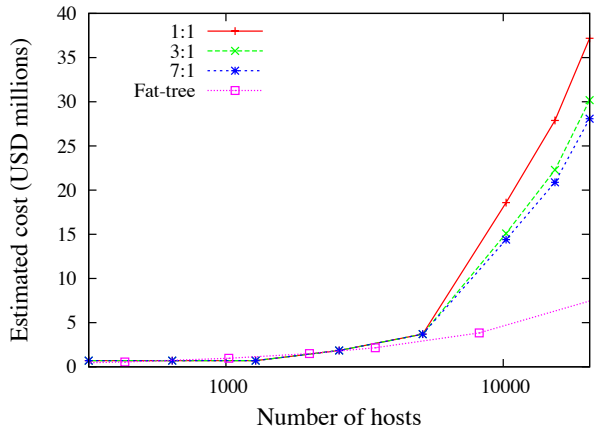

Figure 2: Current cost estimate vs. maximum possible number of hosts for different oversubscription ratios.

We also include the cost to deliver an oversubscription of 1:1 using our proposed fat-tree architecture for comparison.

Overall, we find that existing techniques for delivering high levels of bandwidth in large clusters incur significant cost and that fat-tree based cluster interconnects hold significant promise for delivering scalable bandwidth at moderate cost. However, in some sense, Figure 2 understates the difficulty and expense of employing the highest-end components in building data center architectures. In 2008, 10 GigE switches are on the verge of becoming commodity parts; there is roughly a factor of 5 differential in price per port per bit/sec when comparing GigE to 10 GigE switches, and this differential continues to shrink. To explore the historical trend, we show in Table 1 the cost of the largest cluster configuration that could be supported using the highest-end switches available in a particular year. We based these values on a historical study of product announcements from various vendors of high-end 10 GigE switches in 2002, 2004, 2006, and 2008.

We use our findings to build the largest cluster configuration that technology in that year could support while maintaining an oversubscription of 1:1. Table 1 shows the largest 10 GigE switch available in a particular year; we employ these switches in the core and aggregation layers for the hierarchical design. Tables 1 also shows the largest commodity GigE switch available in that year; we employ these switches at all layers of the fat-tree and at the edge layer for the hierarchical design.

	Hierarchical design			Fat-tree		
Year	**10 GigE**	**Hosts**	**Cost/ GigE**	**GigE**	**Hosts**	**Cost/ GigE**
2002	28-port	4,480	$25.3K	28-port	5,488	$4.5K
2004	32-port	7,680	$4.4K	48-port	27,648	$1.6K
2006	64-port	10,240	$2.1K	48-port	27,648	$1.2K
2008	128-port	20,480	$1.8K	48-port	27,648	$0.3K

Table 1: The maximum possible cluster size with an oversubscription ratio of 1:1 for different years.

The maximum cluster size supported by traditional techniques employing high-end switches has been limited by available port density until recently. Further, the high-end switches incurred prohibitive costs when 10 GigE switches were initially available. Note that we are being somewhat generous with our calculations for traditional hierarchies since commodity GigE switches at the aggregation layer did not have the necessary 10 GigE uplinks until quite recently. Clusters based on fat-tree topologies on the other hand scale well, with the total cost dropping more rapidly and earlier (as a result of following commodity pricing trends earlier). Also, there is no requirement for higher-speed uplinks in the fat-tree topology.

Finally, it is interesting to note that, today, it is technically infeasible to build a 27,648-node cluster with 10 Gbps bandwidth potentially available among all nodes. On the other hand, a fat-tree switch architecture would leverage near-commodity 48-port 10 GigE switches and incur a cost of over $690 million. While likely cost-prohibitive in most settings, the bottom line is that it is not even possible to build such a configuration using traditional aggregation with high-end switches because today there is no product or even Ethernet standard for switches faster than 10 GigE.

2.2 Clos Networks/Fat-Trees

Today, the price differential between commodity and non-commodity switches provides a strong incentive to build large-scale communication networks from many small commodity switches rather than fewer larger and more expensive ones. More than fifty years ago, similar trends in telephone switches led Charles Clos to design a network topology that delivers high levels of bandwidth for many end devices by appropriately interconnecting smaller commodity switches [11].

We adopt a special instance of a Clos topology called a *fat-tree* [23] to interconnect commodity Ethernet switches. We organize a k-ary fat-tree as shown in Figure 3. There are k pods, each containing two layers of $k/2$ switches. Each k-port switch in the lower layer is directly connected to $k/2$ hosts. Each of the remaining $k/2$ ports is connected to $k/2$ of the k ports in the aggregation layer of the hierarchy.

There are $(k/2)^2$ k-port core switches. Each core switch has one port connected to each of k pods. The i^{th} port of any core switch is connected to pod i such that consecutive ports in the aggregation layer of each pod switch are connected to core switches on $(k/2)$ strides. In general, a fat-tree built with k-port switches supports $k^3/4$ hosts. In this paper, we focus on designs up to $k = 48$. Our approach generalizes to arbitrary values for k.

An advantage of the fat-tree topology is that all switching elements are identical, enabling us to leverage cheap commodity parts for all of the switches in the communication architecture.[2] Further, fat-trees are *rearrangeably non-blocking*, meaning that for arbitrary communication patterns, there is some set of paths that will saturate all the bandwidth available to the end hosts in the topology. Achieving an oversubscription ratio of 1:1 in practice may be difficult because of the need to prevent packet reordering for TCP flows.

Figure 3 shows the simplest non-trivial instance of the fat-tree with $k = 4$. All hosts connected to the same edge switch form their own subnet. Therefore, all traffic to a host connected to the same lower-layer switch is switched, whereas all other traffic is routed.

As an example instance of this topology, a fat-tree built from 48-port GigE switches would consist of 48 pods, each containing an edge layer and an aggregation layer with 24 switches each. The edge switches in every pod are assigned 24 hosts each. The network supports 27,648 hosts, made up of 1,152 subnets with 24 hosts each. There are 576 equal-cost paths between any given pair of hosts in different pods. The cost of deploying such a network architecture would be $\$8.64M$, compared to $\$37M$ for the traditional techniques described earlier.

2.3 Summary

Given our target network architecture, in the rest of this paper we address two principal issues with adopting this topology in Ethernet deployments. First, IP/Ethernet networks typically build a single routing path between each source and destination. For even simple communication patterns, such single-path routing will quickly lead to bottlenecks up and down the fat-tree, significantly limiting overall performance. We describe simple extensions to IP forwarding to effectively utilize the high fan-out available from fat-trees. Second, fat-tree topologies can impose significant wiring complexity in large networks. To some extent, this overhead is inherent in fat-tree topologies, but in Section 6 we present packaging and placement techniques to ameliorate this overhead. Finally, we have built a prototype of our architecture in Click [21] as described in Section 3. An initial performance evaluation presented in Section 5 confirms the potential performance benefits of our approach in a small-scale deployment.

3. ARCHITECTURE

In this section, we describe an architecture to interconnect commodity switches in a fat-tree topology. We first motivate the need for a slight modification in the routing table structure. We then describe how we assign IP addresses to hosts in the cluster. Next, we introduce the concept of two-level route lookups to assist with multi-path routing across the fat-tree. We then present the algorithms we employ to populate the forwarding table in each switch. We also describe flow classification and flow scheduling techniques as alternate multi-path routing methods. And finally, we present a simple fault-tolerance scheme, as well as describe the heat and power characteristics of our approach.

[2]Note that switch homogeneity is not required, as bigger switches could be used at the core (e.g. for multiplexing). While these likely have a longer mean time to failure (MTTF), this defeats the cost benefits, and maintains the same cabling overhead.

3.1 Motivation

Achieving maximum bisection bandwidth in this network requires spreading outgoing traffic from any given pod as evenly as possible among the core switches. Routing protocols such as OSPF2 [25] usually take the hop-count as their metric of "shortest-path," and in the k-ary fat-tree topology (see Section 2.2), there are $(k/2)^2$ such shortest-paths between any two hosts on different pods, but only one is chosen. Switches, therefore, concentrate traffic going to a given subnet to a single port even though other choices exist that give the same cost. Furthermore, depending on the interleaving of the arrival times of OSPF messages, it is possible for a small subset of core switches, perhaps only one, to be chosen as the intermediate links between pods. This will cause severe congestion at those points and does not take advantage of path redundancy in the fat-tree.

Extensions such as OSPF-ECMP [30], in addition to being unavailable in the class of switches under consideration, cause an explosion in the number of required prefixes. A lower-level pod switch would need $(k/2)$ prefixes for *every* other subnet; a total of $k * (k/2)^2$ prefixes.

We therefore need a simple, fine-grained method of traffic diffusion between pods that takes advantage of the structure of the topology. The switches must be able to recognize, and give special treatment to, the class of traffic that needs to be evenly spread. To achieve this, we propose using two-level routing tables that spread outgoing traffic based on the low-order bits of the destination IP address (see Section 3.3).

3.2 Addressing

We allocate all the IP addresses in the network within the private $10.0.0.0/8$ block. We follow the familiar quad-dotted form with the following conditions: The pod switches are given addresses of the form $10.pod.switch.1$, where pod denotes the pod number (in $[0, k-1]$), and $switch$ denotes the position of that switch in the pod (in $[0, k-1]$, starting from left to right, bottom to top). We give core switches addresses of the form $10.k.j.i$, where j and i denote that switch's coordinates in the $(k/2)^2$ core switch grid (each in $[1, (k/2)]$, starting from top-left).

The address of a host follows from the pod switch it is connected to; hosts have addresses of the form: $10.pod.switch.ID$, where ID is the host's position in that subnet (in $[2, k/2+1]$, starting from left to right). Therefore, each lower-level switch is responsible for a $/24$ subnet of $k/2$ hosts (for $k < 256$). Figure 3 shows examples of this addressing scheme for a fat-tree corresponding to $k = 4$. Even though this is relatively wasteful use of the available address space, it simplifies building the routing tables, as seen below. Nonetheless, this scheme scales up to 4.2M hosts.

3.3 Two-Level Routing Table

To provide the even-distribution mechanism motivated in Section 3.1, we modify routing tables to allow two-level prefix lookup. Each entry in the main routing table will potentially have an additional pointer to a small secondary table of *(suffix, port)* entries. A first-level prefix is *terminating* if it does not contain any second-level suffixes, and a secondary table may be pointed to by more

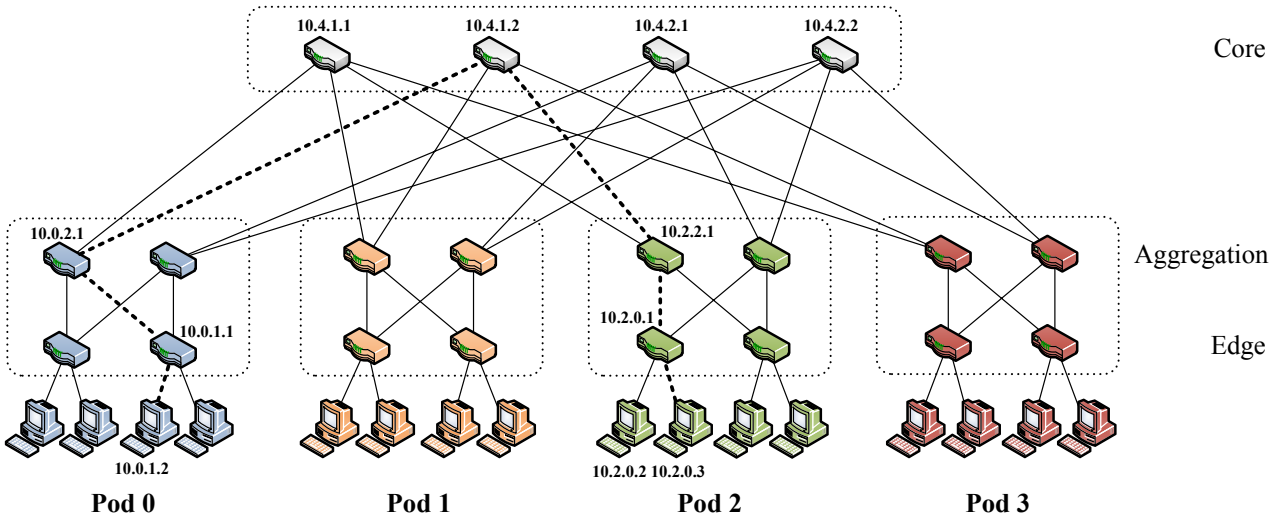

Figure 3: Simple fat-tree topology. Using the two-level routing tables described in Section 3.3, packets from source 10.0.1.2 **to destination** 10.2.0.3 **would take the dashed path.**

Prefix	Output port
10.2.0.0/24	0
10.2.1.0/24	1
0.0.0.0/0	

Suffix	Output port
0.0.0.2/8	2
0.0.0.3/8	3

Figure 4: Two-level table example. This is the table at switch 10.2.2.1**. An incoming packet with destination IP address** 10.2.1.2 **is forwarded on port 1, whereas a packet with destination IP address** 10.3.0.3 **is forwarded on port 3.**

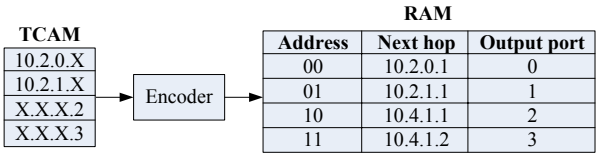

Figure 5: TCAM two-level routing table implementation.

than one first-level prefix. Whereas entries in the primary table are left-handed (i.e., */m prefix* masks of the form 1^m0^{32-m}), entries in the secondary tables are right-handed (i.e. */m suffix* masks of the form $0^{32-m}1^m$). If the longest-matching prefix search yields a non-terminating prefix, then the longest-matching suffix in the secondary table is found and used.

This two-level structure will slightly increase the routing table lookup latency, but the parallel nature of prefix search in hardware should ensure only a marginal penalty (see below). This is helped by the fact that these tables are meant to be very small. As shown below, the routing table of any pod switch will contain no more than $k/2$ prefixes and $k/2$ suffixes.

3.4 Two-Level Lookup Implementation

We now describe how the two-level lookup can be implemented in hardware using Content-Addressable Memory (CAM) [9]. CAMs are used in search-intensive applications and are faster than algorithmic approaches [15, 29] for finding a match against a bit pattern. A CAM can perform parallel searches among all its entries in a single clock cycle. Lookup engines use a special kind of CAM, called Ternary CAM (TCAM). A TCAM can store *don't care* bits in addition to matching 0's and 1's in particular positions, making it suitable for storing variable length prefixes, such as the ones found in routing tables. On the downside, CAMs have rather low storage density, they are very power hungry, and expensive per bit. However, in our architecture, routing tables can be implemented in a TCAM of a relatively modest size (k entries each 32 bits wide).

Figure 5 shows our proposed implementation of the two-level lookup engine. A TCAM stores address prefixes and suffixes, which in turn indexes a RAM that stores the IP address of the next hop and the output port. We store left-handed (prefix) entries in numerically smaller addresses and right-handed (suffix) entries in larger addresses. We encode the output of the CAM so that the entry with the numerically smallest matching address is output. This satisfies the semantics of our specific application of two-level lookup: when the destination IP address of a packet matches both a left-handed and a right-handed entry, then the left-handed entry is chosen. For example, using the routing table in Figure 5, a packet with destination IP address 10.2.0.3 matches the left-handed entry 10.2.0.X and the right-handed entry $X.X.X.3$. The packet is correctly forwarded on port 0. However, a packet with destination IP address 10.3.1.2 matches only the right-handed entry $X.X.X.2$ and is forwarded on port 2.

3.5 Routing Algorithm

The first two levels of switches in a fat-tree act as filtering traffic diffusers; the lower- and upper-layer switches in any given pod have terminating prefixes to the subnets in that pod. Hence, if a host sends a packet to another host in the same pod but on a different subnet, then all upper-level switches in that pod will have a terminating prefix pointing to the destination subnet's switch.

For all other outgoing inter-pod traffic, the pod switches have a default /0 prefix with a secondary table matching host IDs (the

least-significant byte of the destination IP address). We employ the host IDs as a source of deterministic entropy; they will cause traffic to be evenly spread upward among the outgoing links to the core switches[3]. This will also cause subsequent packets to the same host to follow the same path, and therefore avoid packet reordering.

In the core switches, we assign terminating first-level prefixes for all network IDs, each pointing to the appropriate pod containing that network. Once a packet reaches a core switch, there is exactly one link to its destination pod, and that switch will include a terminating $/16$ prefix for the pod of that packet $(10.pod.0.0/16, port)$. Once a packet reaches its destination pod, the receiving upper-level pod switch will also include a $(10.pod.switch.0/24, port)$ prefix to direct that packet to its destination subnet switch, where it is finally switched to its destination host. Hence, traffic diffusion occurs only in the first half of a packet's journey.

It is possible to design distributed protocols to build the necessary forwarding state incrementally in each switch. For simplicity however, we assume a central entity with full knowledge of cluster interconnect topology. This central route control is responsible for statically generating all routing tables and loading the tables into the switches at the network setup phase. Dynamic routing protocols would also be responsible for detecting failures of individual switches and performing path fail-over (see Section 3.8). Below, we summarize the steps for generating forwarding tables at both the pods and core switches.

Pod Switches.

In each pod switch, we assign terminating prefixes for subnets contained in the same pod. For inter-pod traffic, we add a $/0$ prefix with a secondary table matching host IDs. Algorithm 1 shows the pseudo-code for generating the routing tables for the upper pod switches. The reason for the modulo shift in the outgoing port is to avoid traffic from different lower-layer switches addressed to a host with the same host ID going to the same upper-layer switch.

For the lower pod switches, we simply omit the $/24$ subnet prefix step, in line 3, since that subnet's own traffic is switched, and intra- and inter-pod traffic should be evenly split among the upper switches.

Core Switches.

Since each core switch is connected to every pod (port i is connected to pod i), the core switches contains only terminating $/16$ prefixes pointing to their destination pods, as shown in Algorithm 2. This algorithm generates tables whose size is linear in k. No switch in the network contains a table with more than k first-level prefixes or $k/2$ second-level suffixes.

Routing Example.

To illustrate network operation using the two-level tables, we give an example for the routing decisions taken for a packet from source $10.0.1.2$ to destination $10.2.0.3$, as shown in Figure 3. First, the gateway switch of the source host ($10.0.1.1$) will only match the packet with the $/0$ first-level prefix, and therefore will forward the packet based on the host ID byte according to the secondary table for that prefix. In that table, the packet matches the $0.0.0.3/8$ suffix, which points to port 2 and switch $10.0.2.1$. Switch $10.0.2.1$ also follows the same steps and forwards on port 3, connected to core switch $10.4.1.1$. The core switch matches the packet to a terminating $10.2.0.0/16$ prefix, which points to the destination pod 2

[3]Since the tables are static, it is possible to fall short of perfect distribution. We examine worst-case communication patterns in Section 5

```
1  foreach pod x in [0, k − 1] do
2      foreach switch z in [(k/2), k − 1] do
3          foreach subnet i in [0, (k/2) − 1] do
4              addPrefix(10.x.z.1, 10.x.i.0/24, i);
5          end
6          addPrefix(10.x.z.1, 0.0.0.0/0, 0);
7          foreach host ID i in [2, (k/2) + 1] do
8              addSuffix(10.x.z.1, 0.0.0.i/8,
               (i − 2 + z)mod(k/2) + (k/2));
9          end
10     end
11 end
```

Algorithm 1: Generating aggregation switch routing tables. Assume Function signatures *addPrefix(switch, prefix, port)*, *addSuffix(switch, suffix, port)* and $addSuffix$ adds a second-level suffix to the last-added first-level prefix.

```
1  foreach j in [1, (k/2)] do
2      foreach i in [1, (k/2)] do
3          foreach destination pod x in [0, (k/2) − 1] do
4              addPrefix(10.k.j.i,10.x.0.0/16, x);
5          end
6      end
7  end
```

Algorithm 2: Generating core switch routing tables.

on port 2, and switch $10.2.2.1$. This switch belongs to the same pod as the destination subnet, and therefore has a terminating prefix, $10.2.0.0/24$, which points to the switch responsible for that subnet, $10.2.0.1$ on port 0. From there, standard switching techniques deliver the packet to the destination host $10.2.0.3$.

Note that for simultaneous communication from $10.0.1.3$ to another host $10.2.0.2$, traditional single-path IP routing would follow the same path as the flow above because both destinations are on the same subnet. Unfortunately, this would eliminate all of the fan-out benefits of the fat-tree topology. Instead, our two-level table lookup allows switch $10.0.1.1$ to forward the second flow to $10.0.3.1$ based on right-handed matching in the two-level table.

3.6 Flow Classification

In addition to the two-level routing technique described above, we also consider two optional dynamic routing techniques, as they are currently available in several commercial routers [10, 3]. Our goal is to quantify the potential benefits of these techniques but acknowledge that they will incur additional per-packet overhead. Importantly, any maintained state in these schemes is soft and individual switches can fall back to two-level routing in case the state is lost.

As an alternate method of traffic diffusion to the core switches, we perform flow classification with dynamic port-reassignment in pod switches to overcome cases of avoidable local congestion (e.g. when two flows compete for the same output port, even though another port that has the same cost to the destination is underused). We define a *flow* as a sequence of packets with the same entries for a subset of fields of the packet headers (typically source and destination IP addresses, destination transport port). In particular, pod switches:

1. Recognize subsequent packets of the same flow, and forward them on the same outgoing port.

2. Periodically reassign a minimal number of flow output ports to minimize any disparity between the aggregate flow capacity of different ports.

Step 1 is a measure against packet reordering, while step 2 aims to ensure fair distribution on flows on upward-pointing ports in the face of dynamically changing flow sizes. Section 4.2 describes our implementation and flow distribution heuristic of the flow classifier in more detail.

3.7 Flow Scheduling

Several studies have indicated that the distribution of transfer times and burst lengths of Internet traffic is long-tailed [14], and characterized by few large long-lived flows (responsible for most of the bandwidth) and many small short-lived ones [16]. We argue that routing large flows plays the most important role in determining the achievable bisection bandwidth of a network and therefore merits special handling. In this alternative approach to flow management, we schedule large flows to minimize overlap with one another. A central scheduler makes this choice, with global knowledge of all active large flows in the network. In this initial design, we only consider the case of a single large flow originating from each host at a time.

3.7.1 Edge Switches

As before, edge switches locally assign a new flow to the least-loaded port initially. However, edge switches additionally detect any outgoing flow whose size grows above a predefined threshold, and periodically send notifications to a central scheduler specifying the source and destination for all active large flows. This represents a request by the edge switch for placement of that flow in an uncontended path.

Note that unlike Section 3.6, this scheme does not allow edge switches to independently reassign a flow's port, regardless of size. The central scheduler is the only entity with the authority to order a re-assignment.

3.7.2 Central Scheduler

A central scheduler, possibly replicated, tracks all active large flows and tries to assign them non-conflicting paths if possible. The scheduler maintains boolean state for all links in the network signifying their availability to carry large flows.

For inter-pod traffic, recall that there are $(k/2)^2$ possible paths between any given pair of hosts in the network, and each of these paths corresponds to a core switch. When the scheduler receives a notification of a new flow, it linearly searches through the core switches to find one whose corresponding path components do not include a reserved link.[4] Upon finding such a path, the scheduler marks those links as reserved, and notifies the relevant lower- and upper-layer switches in the source pod with the correct outgoing port that corresponds to that flow's chosen path. A similar search is performed for intra-pod large flows; this time for an uncontended path through an upper-layer pod switch. The scheduler garbage collects flows whose last update is older than a given time, clearing their reservations. Note that the edge switches do not block and wait for the scheduler to perform this computation, but initially treat a large flow like any other.

[4]Finding the optimal placement for all large flows requires either knowing the source and destination of all flows ahead of time or path reassignment of existing flows; however, this greedy heuristic gives a good approximation and achieves in simulations 94% efficiency for randomly destined flows among 27k hosts.

3.8 Fault-Tolerance

The redundancy of available paths between any pair of hosts makes the fat-tree topology attractive for fault-tolerance. We propose a simple failure broadcast protocol that allows switches to route around link- or switch-failures one or two hops downstream.

In this scheme, each switch in the network maintains a *Bidirectional Forwarding Detection* session (BFD [20]) with each of its neighbors to determine when a link or neighboring switch fails. From a fault-tolerance perspective, two classes of failure can be weathered: (a) between lower- and upper-layer switches inside a pod, and (b) between core and a upper-level switches. Clearly, the failure of a lower-level switch will cause disconnection for the directly connected hosts; redundant switch elements at the leaves are the only way to tolerate such failures. We describe link failures here because switch failures trigger the same BFD alerts and elicit the same responses.

3.8.1 Lower- to Upper-layer Switches

A link failure between lower- and upper-level switches affects three classes of traffic:

1. Outgoing inter- and intra-pod traffic originating from the lower-layer switch. In this case the local flow classifier sets the 'cost' of that link to infinity and does not assign it any new flows, and chooses another available upper-layer switch.

2. Intra-pod traffic using the upper-layer switch as an intermediary. In response, this switch broadcasts a tag notifying all other lower-layer switches in the same pod of the link failure. These switches would check when assigning new flows whether the intended output port corresponds to one of those tags and avoid it if possible.[5]

3. Inter-pod traffic coming into the upper-layer switch. The core switch connected to the upper-layer switch has it as its only access to that pod, therefore the upper-layer switch broadcasts this tag to all its core switches signifying its inability to carry traffic to the lower-layer switch's subnet. These core switches in turn mirror this tag to all upper-layer switches they are connected to in other pods. Finally, the upper-layer switches avoid the single affected core switch when assigning new flows to that subnet.

3.8.2 Upper-layer to Core Switches

A failure of a link from an upper-layer switch to a core affects two classes of traffic:

1. Outgoing inter-pod traffic, in which case the local routing table marks the affected link as unavailable and locally chooses another core switch.

2. Incoming inter-pod traffic. In this case the core switch broadcasts a tag to all other upper-layer switches it is directly connected to signifying its inability to carry traffic to that entire *pod*. As before, these upper-layer switches would avoid that core switch when assigning flows destined to that pod.

Naturally, when failed links and switches come back up and reestablish their BFD sessions, the previous steps are reversed to cancel their effect. In addition, adapting the scheme of Section 3.7 to accommodate link- and switch-failures is relatively simple. The scheduler marks any link reported to be down as busy or unavailable, thereby disqualifying any path that includes it from consideration, in effect routing large flows around the fault.

[5]We rely on end-to-end mechanisms to restart interrupted flows

3.9 Power and Heat Issues

Besides performance and cost, another major issue that arises in data center design is power consumption. The switches that make up the higher tiers of the interconnect in data centers typically consume thousands of Watts, and in a large-scale data center the power requirements of the interconnect can be hundreds of kilowatts. Almost equally important is the issue of heat dissipation from the switches. Enterprise-grade switches generate considerable amounts of heat and thus require dedicated cooling systems.

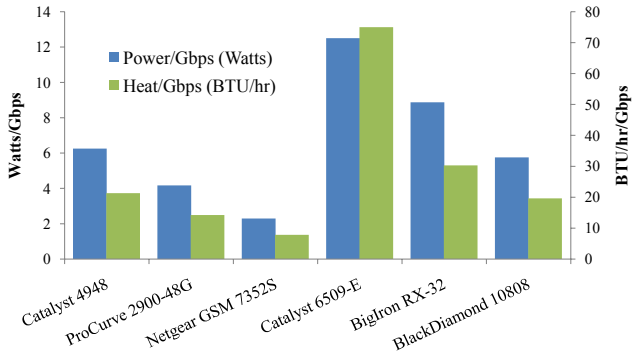

Figure 6: Comparison of power and heat dissipation.

In this section we analyze the power requirements and heat dissipation in our architecture and compare it with other typical approaches. We base our analysis on numbers reported in the switch data sheets, though we acknowledge that these reported values are measured in different ways by different vendors and hence may not always reflect system characteristics in deployment.

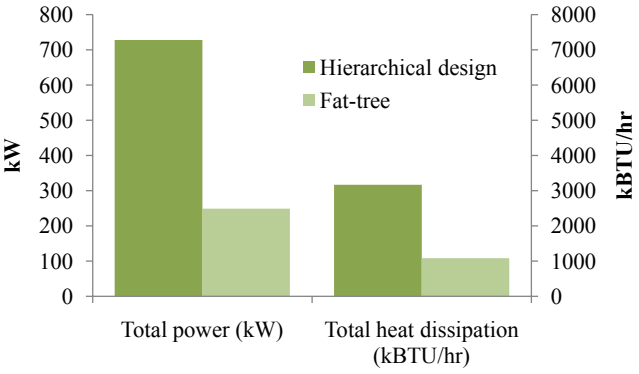

Figure 7: Comparison of total power consumption and heat dissipation.

To compare the power requirement for each class of switch, we normalize the total power consumption and heat dissipation by the switch over the total aggregate bandwidth that a switch can support in Gbps. Figure 6 plots the average over three different switch models. As we can see, 10 GigE switches (the last three on the x-axis) consume roughly double the Watts per Gbps and dissipate roughly three times the heat of commodity GigE switches when normalized for bandwidth.

Finally, we also calculated the estimated total power consumption and heat dissipation for an interconnect that can support roughly 27k hosts. For the hierarchical design, we employ 576 ProCurve 2900 edge switches and 54 BigIron RX-32 switches (36 in the aggregation and 18 in the core layer). The fat-tree architecture employs 2,880 Netgear GSM 7252S switches. We are able to use the cheaper NetGear switch because we do not require 10 GigE uplinks (present in the ProCurve) in the fat-tree interconnect. Figure 7 shows that while our architecture employs more individual switches, the power consumption and heat dissipation is superior to those incurred by current data center designs, with 56.6% less power consumption and 56.5% less heat dissipation. Of course, the actual power consumption and heat dissipation must be measured in deployment; we leave such a study to our ongoing work.

4. IMPLEMENTATION

To validate the communication architecture described in this paper, we built a simple prototype of the forwarding algorithms described in the previous section. We have completed a prototype using NetFPGAs [24]. The NetFPGA contains an IPv4 router implementation that leverages TCAMs. We appropriately modified the routing table lookup routine, as described in Section 3.4. Our modifications totaled less than 100 lines of additional code and introduced no measureable additional lookup latency, supporting our belief that our proposed modifications can be incorporated into existing switches.

To carry out larger-scale evaluations, we also built a prototype using Click, the focus of our evaluation in this paper. Click [21] is a modular software router architecture that supports implementation of experimental router designs. A Click router is a graph of packet processing modules called *elements* that perform tasks such as routing table lookup or decrementing a packet's TTL. When chained together, Click elements can carry out complex router functionality and protocols in software.

4.1 TwoLevelTable

We build a new Click element, *TwoLevelTable*, which implements the idea of a two-level routing table described in Section 3.3. This element has one input, and two or more outputs. The routing table's contents are initialized using an input file that gives all the prefixes and suffixes. For every packet, the TwoLevelTable element looks up the longest-matching first-level prefix. If that prefix is terminating, it will immediately forward the packet on that prefix's port. Otherwise, it will perform a right-handed longest-matching suffix search on the secondary table and forward on the corresponding port.

This element can replace the central routing table element of the standards-compliant IP router configuration example provided in [21]. We generate an analogous 4-port version of the IP router with the added modification of bandwidth-limiting elements on all ports to emulate link saturation capacity.

4.2 FlowClassifier

To provide the flow classification functionality described in Section 3.6, we describe our implementation of the Click element *FlowClassifier* that has one input and two or more outputs. It performs simple flow classification based on the source and destination IP addresses of the incoming packets, such that subsequent packets with the same source and destination exit the same port (to avoid packet reordering). The element has the added goal of minimizing the difference between the aggregate flow capacity of its highest- and lowest-loaded output ports.

Even if the individual flow sizes are known in advance, this problem is a variant of the NP-hard Bin Packing optimization problem [17]. However, the flow sizes are in fact not known *a priori*, making the problem more difficult. We follow the greedy heuris-

tic outlined in Algorithm 3. Every few seconds, the heuristic attempts to switch, if needed, the output port of at most three flows to minimize the difference between the aggregate flow capacity of its output ports.

```
   // Call on every incoming packet
 1 IncomingPacket(packet)
 2 begin
 3     Hash source and destination IP fields of packet;
       // Have we seen this flow before?
 4     if seen(hash) then
 5         Lookup previously assigned port x;
 6         Send packet on port x;
 7     else
 8         Record the new flow f;
 9         Assign f to the least-loaded upward port x;
10         Send the packet on port x;
11     end
12 end
   // Call every t seconds
13 RearrangeFlows()
14 begin
15     for i=0 to 2 do
16         Find upward ports p_max and p_min with the largest and
           smallest aggregate outgoing traffic, respectively;
17         Calculate D, the difference between p_max and p_min;
18         Find the largest flow f assigned to port p_max whose size
           is smaller than D;
19         if such a flow exists then
20             Switch the output port of flow f to p_min;
21         end
22     end
23 end
```

Algorithm 3: The flow classifier heuristic. For the experiments in Section 5, t is 1 second.

Recall that the FlowClassifier element is an alternative to the two-level table for traffic diffusion. Networks using these elements would employ ordinary routing tables. For example, the routing table of an upper pod switch contains all the subnet prefixes assigned to that pod like before. However, in addition, we add a /0 prefix to match all remaining inter-pod traffic that needs to be evenly spread upwards to the core layer. All packets that match only that prefix are directed to the input of the FlowClassifier. The classifier tries to evenly distribute outgoing inter-pod flows among its outputs according to the described heuristic, and its outputs are connected directly to the core switches. The core switches do not need a classifier, and their routing tables are unchanged.

Note that this solution has soft state that is not needed for correctness, but only used as a performance optimization. This classifier is occasionally disruptive, as a minimal number of flows may be re-arranged periodically, potentially resulting in packet reordering. However, it is also adaptive to dynamically changing flow sizes and 'fair' in the long-term.[6]

4.3 FlowScheduler

As described in Section 3.7, we implemented the element *FlowReporter*, which resides in all edge switches, and detects outgoing flows whose size is larger than a given threshold. It sends regular notifications to the central scheduler about these active large flows.

The *FlowScheduler* element receives notifications regarding active large flows from edge switches and tries to find uncontended paths for them. To this end, it keeps the binary status of all the links in the network, as well as a list of previously placed flows. For any new large flow, the scheduler performs a linear search among all equal-cost paths between the source and destination hosts to find one whose path components are all unreserved. Upon finding such a path, the flow scheduler marks all the component links as reserved and sends notifications regarding this flow's path to the concerned pod switches. We also modify the pod switches to process these port re-assignment messages from the scheduler.

The scheduler maintains two main data structures: a binary array of all the links in the network (a total of $4 * k * (k/2)^2$ links), and a hashtable of previously placed flows and their assigned paths. The linear search for new flow placement requires on average $2 * (k/2)^2$ memory accesses, making the computational complexity of the scheduler to be $O(k^3)$ for space and $O(k^2)$ for time. A typical value for k (the number of ports per switch) is 48, making both these values manageable, as quantified in Section 5.3.

5. EVALUATION

To measure the total bisection bandwidth of our design, we generate a benchmark suite of communication mappings to evaluate the performance of the 4-port fat-tree using the TwoLevelTable switches, the FlowClassifier and the FlowScheduler. We compare these methods to a standard hierarchical tree with a $3.6:1$ oversubscription ratio, similar to ones found in current data center designs.

5.1 Experiment Description

In the 4-port fat-tree, there are 16 hosts, four pods (each with four switches), and four core switches. Thus, there is a total of 20 switches and 16 end hosts (for larger clusters, the number of switches will be smaller than the number of hosts). We multiplex these 36 elements onto ten physical machines, interconnected by a 48-port ProCurve 2900 switch with 1 Gigabit Ethernet links. These machines have dual-core Intel Xeon CPUs at 2.33GHz, with 4096KB cache and 4GB of RAM, running Debian GNU/Linux 2.6.17.3. Each pod of switches is hosted on one machine; each pod's hosts are hosted on one machine; and the two remaining machines run two core switches each. Both the switches and the hosts are Click configurations, running in user level. All virtual links between the Click elements in the network are bandwidth-limited to 96Mbit/s to ensure that the configuration is not CPU limited.

For the comparison case of the hierarchical tree network, we have four machines running four hosts each, and four machines each running four pod switches with one additional uplink. The four pod switches are connected to a 4-port core switch running on a dedicated machine. To enforce the 3.6:1 oversubscription on the uplinks from the pod switches to the core switch, these links are bandwidth-limited to 106.67Mbit/s, and all other links are limited to 96Mbit/s.

Each host generates a constant 96Mbit/s of outgoing traffic. We measure the rate of its incoming traffic. The minimum aggregate incoming traffic of all the hosts for all bijective communication mappings is the effective bisection bandwidth of the network.

5.2 Benchmark Suite

We generate the communicating pairs according to the following strategies, with the added restriction that any host receives traffic from exactly one host (i.e. the mapping is 1-to-1):

- Random: A host sends to any other host in the network with uniform probability.
- Stride(i): A host with index x will send to the host with index $(x + i)\ mod\ 16$.

[6]Fair in the sense that initial placement decisions are constantly being corrected since all flows' sizes are continually tracked to approximate the optimal distribution of flows to ports.

Test	Tree	Two-Level Table	Flow Classification	Flow Scheduling
Random	53.4%	75.0%	76.3%	93.5%
Stride (1)	100.0%	100.0%	100.0%	100.0%
Stride (2)	78.1%	100.0%	100.0%	99.5%
Stride (4)	27.9%	100.0%	100.0%	100.0%
Stride (8)	28.0%	100.0%	100.0%	99.9%
Staggered Prob (1.0, 0.0)	100.0%	100.0%	100.0%	100.0%
Staggered Prob (0.5, 0.3)	83.6%	82.0%	86.2%	93.4%
Staggered Prob (0.2, 0.3)	64.9%	75.6%	80.2%	88.5%
Worst cases:				
Inter-pod Incoming	28.0%	50.6%	75.1%	99.9%
Same-ID Outgoing	27.8%	38.5%	75.4%	87.4%

Table 2: Aggregate Bandwidth of the network, as a percentage of ideal bisection bandwidth for the Tree, Two-Level Table, Flow Classification, and Flow Scheduling methods. The ideal bisection bandwidth for the fat-tree network is 1.536Gbps.

- Staggered Prob ($SubnetP, PodP$): Where a host will send to another host in its subnet with probability $SubnetP$, and to its pod with probability $PodP$, and to anyone else with probability $1 - SubnetP - PodP$.
- Inter-pod Incoming: Multiple pods send to different hosts in the same pod, and all happen to choose the same core switch. That core switch's link to the destination pod will be oversubscribed. The worst-case *local* oversubscription ratio for this case is $(k-1) : 1$.
- Same-ID Outgoing: Hosts in the same subnet send to different hosts elsewhere in the network such that the destination hosts have the same host ID byte. Static routing techniques force them to take the same outgoing upward port. The worst-case ratio for this case is $(k/2) : 1$. This is the case where the FlowClassifier is expected to improve performance the most.

5.3 Results

Table 2 shows the results of the above described experiments. These results are averages across 5 runs/permutations of the benchmark tests, over 1 minute each. As expected, for any all-inter-pod communication pattern, the traditional tree saturates the links to the core switch, and thus achieves around 28% of the ideal bandwidth for all hosts in that case. The tree performs significantly better the closer the communicating pairs are to each other.

The two-level table switches achieve approximately 75% of the ideal bisection bandwidth for random communication patterns. This can be explained by the static nature of the tables; two hosts on any given subnet have a 50% chance of sending to hosts with the same host ID, in which case their combined throughput is halved since they are forwarded on the same output port. This makes the expectation of both to be 75%. We expect the performance for the two-level table to improve for random communication with increasing k as there will be less likelihood of multiple flows colliding on a single link with higher k. The inter-pod incoming case for the two-level table gives a 50% bisection bandwidth; however, the same-ID outgoing effect is compounded further by congestion in the core router.

Because of its dynamic flow assignment and re-allocation, the flow classifier outperforms both the traditional tree and the two-level table in all cases, with a worst-case bisection bandwidth of approximately 75%. However, it remains imperfect because the type of congestion it avoids is entirely local; it is possible to cause congestion at a core switch because of routing decisions made one or two hops upstream. This type of sub-optimal routing occurs because the switches only have local knowledge available.

The FlowScheduler, on the other hand, acts on global knowledge and tries to assign large flows to disjoint paths, thereby achieving 93% of the ideal bisection bandwidth for random communication mappings, and outperforming all other methods in all the benchmark tests. The use of a centralized scheduler with knowledge of all active large flows and the status of all links may be infeasible for large arbitrary networks, but the regularity of the fat-tree topology greatly simplifies the search for uncontended paths.

In a separate test, Table 3 shows the time and space requirements for the central scheduler when run on a modestly-provisioned 2.33GHz commodity PC. For varying k, we generated fake placement requests (one per host) to measure the average time to process a placement request, and the total memory required for the maintained link-state and flow-state data structures. For a network of 27k hosts, the scheduler requires a modest 5.6MB of memory and could place a flow in under 0.8ms.

k	Hosts	Avg Time/ Req (μs)	Link-state Memory	Flow-state Memory
4	16	50.9	64 B	4 KB
16	1,024	55.3	4 KB	205 KB
24	3,456	116.8	14 KB	691 KB
32	8,192	237.6	33 KB	1.64 MB
48	27,648	754.43	111 KB	5.53 MB

Table 3: The flow scheduler's time and memory requirements.

6. PACKAGING

One drawback of the fat-tree topology for cluster interconnects is the number of cables needed to interconnect all the machines. One trivial benefit of performing aggregation with 10 GigE switches is the factor of 10 reduction in the number of cables required to transfer the same amount of bandwidth up the hierarchy. In our proposed fat-tree topology, we do not leverage 10 GigE links or switches both because non-commodity pieces would inflate cost and, more importantly, because the fat-tree topology critically depends upon a large fan-out to multiple switches at each layer in the hierarchy to achieve its scaling properties.

Acknowledging that increased wiring overhead is inherent to the fat-tree topology, in this section we consider some packaging techniques to mitigate this overhead. In sum, our proposed packaging technique eliminates most of the required external wiring and reduces the overall length of required cabling, which in turn simplifies cluster management and reduces total cost. Moreover, this method allows for incremental deployment of the network.

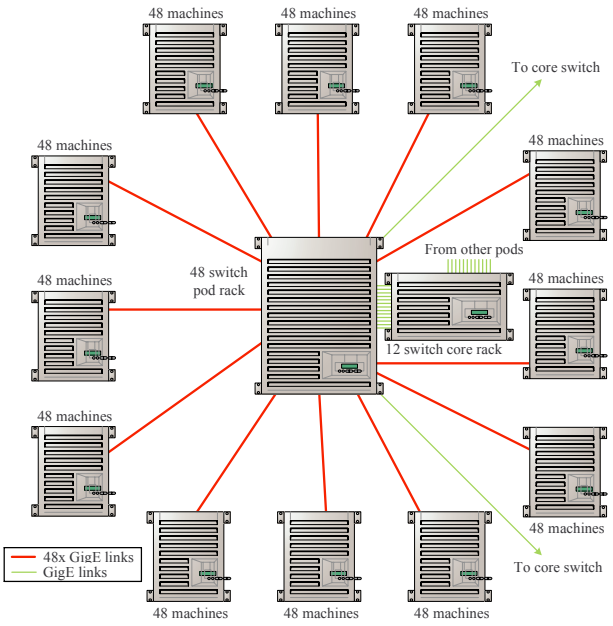

Figure 8: Proposed packaging solution. The only external cables are between the pods and the core nodes.

We present our approach in the context of a maximum-capacity 27,648-node cluster leveraging 48-port Ethernet switches as the building block of the fat-tree. This design generalizes to clusters of different sizes. We begin with the design of individual pods that make up the replication unit for the larger cluster, see Figure 8. Each pod consists of 576 machines and 48 individual 48-port GigE switches. For simplicity, we assume each end host takes up one rack unit (1RU) and that individual racks can accommodate 48 machines. Thus, each pod consists of 12 racks with 48 machines each.

We place the 48 switches that make up the first two layers of the fat-tree in each pod in a centralized rack. However, we assume the ability to package the 48 switches into a single monolithic unit with 1,152 user-facing ports. We call this the *pod switch*. Of these ports, 576 connect directly to the machines in the pod, corresponding to connectivity at the edge. Another 576 ports fan out to one port on each of the 576 switches that make up the core layer in the fat-tree. Note that the 48 switches packaged in this manner actually have 2,304 total ports ($48 * 48$). The other 1,152 ports are wired internally in the pod switch to account for the required interconnect between the edge and aggregation layers of the pod (see Figure 3).

We further spread the 576 required core switches that form the top of the fat-tree across the individual pods. Assuming a total of 48 pods, each will house 12 of the required core switches. Of the 576 cables fanning out from each pod switch to the core, 12 will connect directly to core switches placed nearby in the same pod. The remaining cables would fan out, in sets of 12, to core switches housed in remote pods. Note that the fact that cables move in sets of 12 from pod to pod and in sets of 48 from racks to pod switches opens additional opportunities for appropriate "cable packaging" to reduce wiring complexity.

Finally, minimizing total cable length is another important consideration. To do so, we place racks around the pod switch in two dimensions, as shown in Figure 8 (we do not consider three dimensional data center layouts). Doing so will reduce cable lengths relative to more "horizontal" layouts of individual racks in a pod. Similarly, we lay pods out in a 7×7 grid (with one missing spot) to accommodate all 48 pods. Once again, this grid layout will reduce inter-pod cabling distance to appropriate core switches and will support some standardization of cable lengths and packaging to support inter-pod connectivity.

We also considered an alternate design that did not collect the switches into a central rack. In this approach, two 48-port switches would be distributed to each rack. Hosts would interconnect to the switches in sets of 24. This approach has the advantage of requiring much shorter cables to connect hosts to their first hop switch and for eliminating these cables all together if the racks were appropriately internally packaged. We discarded this approach because we would lose the opportunity to eliminate the 576 cables within each pod that interconnect the edge and aggregation layers. These cables would need to crisscross the 12 racks in each pod, adding significant complexity.

7. RELATED WORK

Our work in data center network architecture necessarily builds upon work in a number of related areas. Perhaps most closely related to our efforts are various efforts in building scalable interconnects, largely coming out of the supercomputer and massively parallel processing (MPP) communities. Many MPP interconnects have been organized as fat-trees, including systems from Thinking Machines [31, 22] and SGI [33]. Thinking Machines employed pseudo-random forwarding decisions to perform load balancing among fat-tree links. While this approach achieves good load balancing, it is prone to packet reordering. Myrinet switches [6] also employ fat-tree topologies and have been popular for cluster-based supercomputers. Myrinet employs source routing based on predetermined topology knowledge, enabling cut-through low latency switch implementations. Hosts are also responsible for load balancing among available routes by measuring round-trip latencies. Relative to all of these efforts, we focus on leveraging commodity Ethernet switches to interconnect large-scale clusters, showing techniques for appropriate routing and packaging.

InfiniBand [2] is a popular interconnect for high-performance computing environments and is currently migrating to data center environments. InfiniBand also achieves scalable bandwidth using variants of Clos topologies. For instance, Sun recently announced a 3,456-port InfiniBand switch built from 720 24-port InfiniBand switches arranged in a 5-stage fat-tree [4]. However, InfiniBand imposes its own layer 1-4 protocols, making Ethernet/IP/TCP more attractive in certain settings especially as the price of 10Gbps Ethernet continues to drop.

Another popular MPP interconnect topology is a Torus, for instance in the BlueGene/L [5] and the Cray XT3 [32]. A torus directly interconnects a processor to some number of its neighbors in a k-dimensional lattice. The number of dimensions determines the expected number of hops between source and destination. In an MPP environment, a torus has the benefit of not having any dedicated switching elements along with electrically simpler point-to-point links. In a cluster environment, the wiring complexity of a torus quickly becomes prohibitive and offloading all routing and forwarding functions to commodity hosts/operating systems is typically impractical.

Our proposed forwarding techniques are related to existing routing techniques such as OSPF2 and Equal-Cost Multipath (ECMP) [25, 30, 19]. Our proposal for multi-path leverages particular properties of a fat-tree topology to achieve good performance. Relative to our work, ECMP proposes three classes of stateless forwarding algorithms: (i) Round-robin and randomization; (ii) Region splitting where a particular prefix is split into two with a larger mask length; and (iii) A hashing technique that splits

flows among a set of output ports based on the source and destination addresses. The first approach suffers from potential packet reordering issues, especially problematic for TCP. The second approach can lead to a blowup in the number of routing prefixes. In a network with 25,000 hosts, this will require approximately 600,000 routing table entries. In addition to increasing cost, the table lookups at this scale will incur significant latency. For this reason, current enterprise-scale routers allow for a maximum of 16-way ECMP routing. The final approach does not account for flow bandwidth in making allocation decisions, which can quickly lead to oversubscription even for simple communication patterns.

8. CONCLUSIONS

Bandwidth is increasingly the scalability bottleneck in large-scale clusters. Existing solutions for addressing this bottleneck center around hierarchies of switches, with expensive, non-commodity switches at the top of the hierarchy. At any given point in time, the port density of high-end switches limits overall cluster size while at the same time incurring high cost. In this paper, we present a data center communication architecture that leverages commodity Ethernet switches to deliver scalable bandwidth for large-scale clusters. We base our topology around the fat-tree and then present techniques to perform scalable routing while remaining backward compatible with Ethernet, IP, and TCP.

Overall, we find that we are able to deliver scalable bandwidth at significantly lower cost than existing techniques. While additional work is required to fully validate our approach, we believe that larger numbers of commodity switches have the potential to displace high-end switches in data centers in the same way that clusters of commodity PCs have displaced supercomputers for high-end computing environments.

Acknowledgments

We wish to thank George Varghese as well as the anonymous referees for their valuable feedback on earlier drafts of this paper.

9. REFERENCES

[1] Cisco Data Center Infrastructure 2.5 Design Guide. `http://www.cisco.com/univercd/cc/td/doc/solution/dcidg21.pdf`.

[2] InfiniBand Architecture Specification Volume 1, Release 1.0. `http://www.infinibandta.org/specs`.

[3] Juniper J-Flow. `http://www.juniper.net/techpubs/software/erx/junose61/swconfig-routing-vol1/html/ip-jflow-stats-config2.html`.

[4] Sun Datacenter Switch 3456 Architecture White Paper. `http://www.sun.com/products/networking/datacenter/ds3456/ds3456_wp.pdf`.

[5] M. Blumrich, D. Chen, P. Coteus, A. Gara, M. Giampapa, P. Heidelberger, S. Singh, B. Steinmacher-Burow, T. Takken, and P. Vranas. Design and Analysis of the BlueGene/L Torus Interconnection Network. *IBM Research Report RC23025 (W0312-022)*, 3, 2003.

[6] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, and J. Seizovic. Myrinet: A Gigabit-per-second Local Area Network. *Micro, IEEE*, 15(1), 1995.

[7] S. Brin and L. Page. The Anatomy of a Large-scale Hypertextual Web Search Engine. *Computer Networks and ISDN Systems*, 30(1-7), 1998.

[8] R. Cheveresan, M. Ramsay, C. Feucht, and I. Sharapov. Characteristics of Workloads used in High Performance and Technical Computing. In *International Conference on Supercomputing*, 2007.

[9] L. Chisvin and R. J. Duckworth. Content-Addressable and Associative Memory: Alternatives to the Ubiquitous RAM. *Computer*, 22(7):51–64, 1989.

[10] B. Claise. Cisco Systems NetFlow Services Export Version 9. RFC 3954, Internet Engineering Task Force, 2004.

[11] C. Clos. A Study of Non-blocking Switching Networks. *Bell System Technical Journal*, 32(2), 1953.

[12] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *USENIX Symposium on Operating Systems Design and Implementation*, 2004.

[13] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store. *ACM Symposium on Operating Systems Principles*, 2007.

[14] A. B. Downey. Evidence for Long-tailed Distributions in the Internet. *ACM SIGCOMM Workshop on Internet Measurement*, 2001.

[15] W. Eatherton, G. Varghese, and Z. Dittia. Tree Bitmap : Hardware/Software IP Lookups with Incremental Updates. *SIGCOMM Computer Communications Review*, 34(2):97–122, 2004.

[16] S. B. Fred, T. Bonald, A. Proutiere, G. Régnié, and J. W. Roberts. Statistical Bandwidth Sharing: A Study of Congestion at Flow Level. *SIGCOMM Computer Communication Review*, 2001.

[17] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[18] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. *ACM SIGOPS Operating Systems Review*, 37(5), 2003.

[19] C. Hopps. Analysis of an Equal-Cost Multi-Path Algorithm. RFC 2992, Internet Engineering Task Force, 2000.

[20] D. Katz, D. Ward. BFD for IPv4 and IPv6 (Single Hop) (Draft). Technical report, Internet Engineering Task Force, 2008.

[21] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3), 2000.

[22] C. Leiserson, Z. Abuhamdeh, D. Douglas, C. Feynman, M. Ganmukhi, J. Hill, D. Hillis, B. Kuszmaul, M. Pierre, D. Wells, et al. The Network Architecture of the Connection Machine CM-5 (Extended Abstract). *ACM Symposium on Parallel Algorithms and Architectures*, 1992.

[23] C. E. Leiserson. Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing. *IEEE Transactions on Computers*, 34(10):892–901, 1985.

[24] J. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo. NetFPGA–An Open Platform for Gigabit-rate Network Switching and Routing. In *IEEE International Conference on Microelectronic Systems Education*, 2007.

[25] J. Moy. OSPF Version 2. RFC 2328, Internet Engineering Task Force, 1998.

[26] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *USENIX Conference on File and Storage Technologies*, 2002.

[27] L. R. Scott, T. Clark, and B. Bagheri. *Scientific Parallel Computing*. Princeton University Press, 2005.

[28] SGI Developer Central Open Source Linux XFS. XFS: A High-performance Journaling Filesystem. `http://oss.sgi.com/projects/xfs/`.

[29] V. Srinivasan and G. Varghese. Faster IP Lookups using Controlled Prefix Expansion. *ACM Transactions on Computer Systems (TOCS)*, 17(1):1–40, 1999.

[30] D. Thaler and C. Hopps. Multipath Issues in Unicast and Multicast Next-Hop Selection. RFC 2991, Internet Engineering Task Force, 2000.

[31] L. Tucker and G. Robertson. Architecture and Applications of the Connection Machine. *Computer*, 21(8), 1988.

[32] J. Vetter, S. Alam, J. Dunigan, T.H., M. Fahey, P. Roth, and P. Worley. Early Evaluation of the Cray XT3. In *IEEE International Parallel and Distributed Processing Symposium*, 2006.

[33] M. Woodacre, D. Robb, D. Roe, and K. Feind. The SGI Altix 3000 Global Shared-Memory Architecture. *SGI White Paper*, 2003.