DPillar: Scalable Dual-Port Server Interconnection for Data Center Networks

Yong Liao
ECE Department
University of Massachusetts
Amherst, MA 01003
yliao@engin.umass.edu

Dong Yin
ECE Department
University of Massachusetts
Amherst, MA 01003
dyin@engin.umass.edu

Lixin Gao
ECE Department
University of Massachusetts
Amherst, MA 01003
lgao@ecs.umass.edu

***Abstract*—Data centers are becoming increasingly important infrastructures for many essential applications such as data intensive computing and large-scale network services. A typical future data center can consist of up to hundreds of thousands of servers. Yet, the conventional data center networks are not able to keep up with the network bandwidth requirement for efficiently connecting that huge number of servers in a cost-efficient manner. It is hard to scale the hierarchical tree-based interconnection network used in conventional data centers to support the massive amount of data communication in future data centers. In this paper, we present DPillar, a highly scalable data center interconnection architecture, which uses only low-end off-the-shelf commodity PC servers and Ethernet switches. DPillar has minimal requirements for the equipment. The Ethernet switches can be low-cost plug-and-play layer-2 devices and the servers can be typical dual-port commodity PCs. The salient feature of DPillar is that it expands to any number of servers without requiring to physically upgrading the existing servers. We present a simple yet efficient routing scheme that maintains short path length in server communications. We also propose fault-tolerant and traffic-aware routing schemes to ensure balanced load in the data center network.**

I. Introduction

Data centers with a cluster of commodity servers become common places for data storage, data analysis, and large-scale network services [1,16]. Such a data center infrastructure is driven by the demand of petabyte of data storage and high computation power required for processing the data. More importantly, it is projected that the demand for data storage and processing will grow rapidly as more data are available for applications such as web searching, medical image processing, social network mining, and scientific computing. To meet the demand of the growth, one of the essential requirements for data center infrastructure is that it must scale to hundreds of thousands or millions of servers.

While inexpensive commodity PCs make it possible to expand a data center to millions of servers, interconnecting these servers in a scalable and cost-efficient fashion can be challenging. With a data center of increasing server number and storage size, the communication bandwidth has to scale more than linearly (or squarely) to meet the bandwidth demand of frequent data accessing and shuffling in distributed data processing and storage [6,8,9]. In order to keep the interconnection cost low, one natural choice for interconnecting these servers is to leverage commodity hardware such as inexpensive Ethernet switches and the existing network cards in commodity PC servers. So far, there are two approaches for interconnecting servers with commodity switches. The first approach is *switch centric* where the switch functionality is extended to accommodate the need of the interconnection, while requiring no modification to the servers (including network interface, operating system, and applications) [3,4,15]. The second approach is *server centric* where each server acts as both data processing/storage and data relay node while requiring no change to the switches [11,12,14].

In this paper we take a server centric approach and propose a server interconnection structure called DPillar. Each server in a DPillar network is a computation workstation as well as an intermediate node relaying the data between other servers. This server centric design offers many unbeatable advantages. First, it avoids the configuration effort required by a separate switching fabric and simplifies the management of the data center network. Second, shifting the networking functionalities from a separate switching fabric to the servers provides much higher degree of programming capability, which facilitates the design of efficient fault-tolerate routing scheme and traffic-aware routing scheme. Third, the server-based design is much more cost-efficient because it uses only low-end layer-2 dummy switches.

More specifically, DPillar aims to leverage plug-and-play commodity Ethernet switches with only layer-2 switching capability. Ethernet switches with moderate number of ports (e.g., 24 or 48 ports) and with the ability to switch layer-2 frames at line speed are widely available and relatively inexpensive. Layer-2 Ethernet switches also have the advantage of requiring minimal configuration effort as they are basically plug-and-play devices. Further, DPillar requires only two network interfaces for each server. As most off-the-shelf PC servers offer two high-speed Gbit Ethernet ports, one primary port and one backup port, there is no need to physically upgrade the servers. More importantly, when expanding an existing DPillar network with additional servers, it is not required to upgrade existing servers in the data center. Therefore, such an interconnection structure can scale to any number of servers with minimal deployment overhead.

Despite the fact that each server has only two network interfaces, the DPillar interconnection offers rich connections between servers, so the aggregate bandwidth can facilitate data-intensive applications. The structure of the DPillar network is totally symmetric so that it removes any network

bottleneck at the architecture level. We have designed a simple yet highly efficient routing scheme for DPillar network. One salient feature of the proposed routing scheme is that it eliminates the need of doing table lookup when servers are relaying packets. Our prototyping implementation using commodity PC shows that the PC servers can perform data forwarding in line speed without consuming significant resources at the servers. Therefore, such an interconnection structure is feasible for the server centric approach. Furthermore, we propose routing schemes that can efficiently handle a wide range of failures and perform load balancing in DPillar network.

The rest of this paper is organized as follows. Section II presents the network topological structure of the DPillar network. Section III and section IV are devoted to the discussion of routing in DPillar network. Section V presents the prototyping implementation and performance evaluation of DPillar. Section VI provides the background of interconnection networks and related work on data center networks. Section VII concludes this paper.

II. Interconnection of DPillar

In this section, we first present the interconnection structure of DPillar. Then we discuss some of the topological properties of DPillar and the cost of building such a network.

A. Network Structure of DPillar

A DPillar network is built of two kinds of devices, dual-port servers and n-port switches. The servers are arranged into k columns; the switches are arranged into k columns too. We use $H_0 \sim H_{k-1}$ to represent the k server columns and $S_0 \sim S_{k-1}$ to represent the k switch columns. The k server columns and k switch columns are alternately placed along a cycle, as shown in Figure 1. Visually, it looks like the $2k$ columns of servers and switches are attached to the cylindrical surface of a pillar. Using its two ports, a server in each server column is connected to the two switches in its two neighboring switch columns. In other words, for a server in column H_i, one of its ports is connected to a switch in column S_i and the other port of the server is connected to a switch in column $S_{(i+k-1)\%k}$. For a switch in column S_i, half of its n ports are connected to $n/2$ servers in H_i and the other half are connected to $n/2$ servers in $H_{(i+1)\%k}$. Deciding which n servers are connected to the same n-port switch is important and we will discuss it soon later. For easy description, in the rest of this paper we call server column $H_{(i+1)\%k}$ a *clockwise neighboring column* of H_i and $H_{(i+k-1)\%k}$ a *counter-clockwise neighboring column* of H_i.

In a DPillar network with k columns of servers, each server column has $(n/2)^k$ servers; each switch column has $(n/2)^{k-1}$ switches, where n is the number of ports of the switches. For the $(n/2)^k$ servers in any server column H_i, each of them is assigned with a unique k-symbol label $(\nu^{k-1}...\nu^0)$, where a symbol ν^i $(0 \le i \le k-1)$ is an integer number between 0 and $(n/2-1)$. Under this naming scheme, one server in DPillar can be uniquely identified as $(C, \nu^{k-1}...\nu^0)$, which means a server with label $(\nu^{k-1}...\nu^0)$ in server column H_C. We call $(C, \nu^{k-1}...\nu^0)$ the ID or the address of the server.

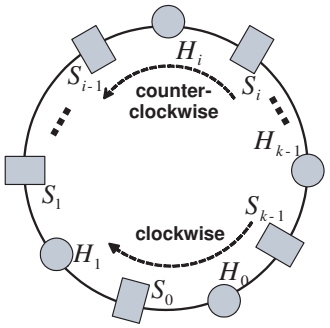

Fig. 1. The vertical view of a DPillar network.

Given the IDs of the servers in a DPillar network, the interconnection between the servers and the switches is as follows. For all the $2(n/2)^k$ servers in any server column H_C and its clockwise neighboring server column $H_{(C+1)\%k}$, they can be divided into $(n/2)^{k-1}$ groups, with each group having n servers. The labels of the n servers in the same group have the following property. That is, their labels are the same if the Cth symbol (i.e., symbol ν^C) is removed. It is easy to see that among the n servers within the same group, half of them are from H_C and the other half are from $H_{(C+1)\%k}$. The n servers in the same group are connected to the same switch in switch column S_C. In other words, given any label $(\nu^{k-1}...\nu^C...\nu^0)$, there are $n/2$ servers in H_C whose labels are $(\nu^{k-1}...\nu^C_*...\nu^0)$ where $0 \le \nu^C_* \le n/2-1$; there are $n/2$ such servers in $H_{(C+1)\%k}$ too. Those n servers are connected to the same n-port switch in S_C.

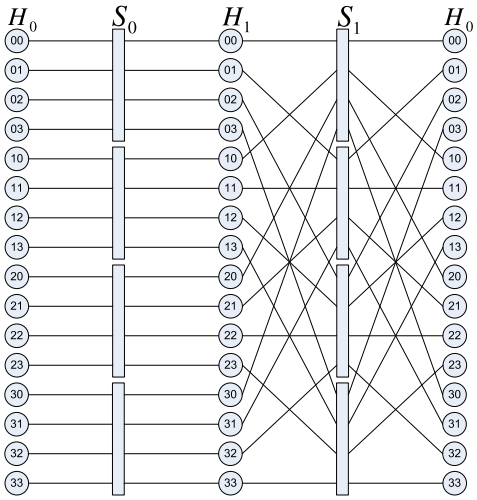

Fig. 2. A DPillar network in two-dimension. The number in each circle is the label of that server.

Figure 2 shows a DPillar network built from 8-port switches. There are two server columns in this network. We duplicate server column H_0, cut the cylindrical surface of the pillar along column H_0, and spread that cylindrical surface into a two-dimension area. As each switch has eight ports and there are two columns of servers $(n = 8, k = 2)$, a server column has $(\frac{8}{2})^2 = 16$ servers. The label of each server has two symbols. The first row of servers in Figure 2 have label (00) and the last row of servers have label (33). If we select a label $(\nu^1\nu^0) = (00)$, there are four servers in H_1 whose labels are (ν^1_*0) with $0 \le \nu^1_* \le 3$, i.e., (00), (10), (20), and (30). There

are four servers in H_0 whose labels are (00), (10), (20), and (30) too. Those eight servers are connected to the same switch in switch column S_1.

B. Topological Properties of DPillar

After presenting the interconnection of DPillar, we proceed to study the basic topological properties of the DPillar network. As we can see from section II-A, a DPillar network is uniquely defined by two parameters, k, the number of server columns, and n, the number of ports of a switch. We call such a DPillar network (n, k) DPillar network for short.

1) Number of servers DPillar accommodates: Clearly, since each server column has $(n/2)^k$ servers, there are $k(n/2)^k$ servers in a (n, k) DPillar network. In the rest of this paper, we use N to represent the total number of servers in a (n, k) DPillar network, i.e., $N = k(n/2)^k$.

Proposition II.1 *A (n, k) DPillar network can accommodate $k(n/2)^k$ servers.*

Given Proposition II.1, let's provide some examples on how many servers a (n, k) DPillar network can support. Considering that 48-port Gbit Ethernet switches are widely available now and relatively inexpensive, a $(48, 3)$ DPillar network has 41472 servers. The number of servers will be about 1.3 million for a $(48, 4)$ DPillar network. If we build a $(48, 5)$ DPillar network, it has about 40 million servers.

2) Number of switches used by DPillar: Next we consider the number of n-port switches used in a DPillar network. This number is important because the switches are the major "networking devices" we will invest in building a DPillar network. As we have mentioned, each switch column has $(n/2)^{k-1}$ switches in a (n, k) DPillar network. As there are k switch columns, the total number of switches is $k(n/2)^{k-1}$. Actually, there is an explanation why there are $(n/2)^{k-1}$ switches in each switch column. If we change all other symbols in label $(\nu^{k-1}...\nu^C...\nu^0)$ except symbol ν^C, there are $(n/2)^{k-1}$ different combinations. Each of those $(n/2)^{k-1}$ combinations requires one switch to connect n servers whose labels are $(\nu^{k-1}...\nu^C_*...\nu^0)$ where $0 \leq \nu^C_* \leq n/2 - 1$. Therefore, the number of switches in each switch column is $(n/2)^{k-1}$ and the total number of switches in a (n, k) DPillar network is $k(n/2)^{k-1}$

Proposition II.2 *A (n, k) DPillar network uses $k(n/2)^{k-1}$ switches.*

3) Bisection width of DPillar: Bisection width is an important factor to quantify the performance of an inter-connection network. It is defined as the smallest number of edges removal of which divides the nodes in the network into two parts of equal size. Larger bisection width means the network can sustain more communications between nodes in the network. Because servers in a data center usually have lots of interactions among them, such as running MapReduce [8] applications, it is desirable that a data center network has large bisection width. The bisection width of a (n, k) DPillar network is $(n/2)^k$, as stated in Proposition II.3. The proof of Proposition II.3 is presented in Appendix A.

Proposition II.3 *The bisection width of a (n, k) DPillar network is close to $(n/2)^k$.*

We can see that the bisection width or a (n, k) DPillar network is equal to the number of servers in each server column. The bisection width of *wrapped butterfly network* [13] is also equal to the number of nodes in each level[1]. However, wrapped butterfly network requires each node to have degree 4 to achieve that bisection width. DPillar network requires each node (the PC server) to have degree 2 (two NICs) and achieves the same bisection width.

C. Cost of Building DPillar

DPillar network is cost-efficient as it uses only inexpensive commodity hardware. Here we provide some "example budgets" of building DPillar networks. We ignore the cost of servers and focus on the networking devices of DPillar, including the switches and the Ethernet cables. As most off-the-shelf servers already integrate dual-port interfaces, there is no need to invest on NICs.

The unit prices we get from an online retailing store (www.newegg.com) are \$150 for a 16-port Gbit Ethernet switch (TRENDnet TEG-S16R) and \$1 for an Ethernet cable. We expect the wholesale price of the switches and cables would be even lower. For a $(16, 4)$ DPillar network, there are $16,384$ servers. The cost of the switches is $4 \times (\frac{16}{2})^3 \times 150 = \$307,200$. The cost of the cables is $16,384 \times 2 = \$32,768$ as we need two cables for each server. The total for the networking devices is about 0.34 million dollars, which means on average it costs about \$20 to connect one server in this $(16, 4)$ DPillar network. Table I shows the total cost and the per-server cost of building DPillar networks with four columns of servers, when different types of switches are used. In general, letting U_s be the unit price of a n-port switch and U_c be the unit price of an Ethernet cable, the average cost of connecting one server in the DPillar network is $2(U_s/n + U_c)$. If we ignore the cost of the cables, the average cost of connecting a server in a DPillar network is two times the per-port cost of the switches used in this DPillar network.

III. Routing in DPillar Network

Because of the symmetric structure of the DPillar network, routing and packet forwarding in DPillar network can be simple and efficient. In this section, we present a routing scheme which has constant running time in computing routes in DPillar.

A. The Routing Algorithm

The packet routing and forwarding process in DPillar can be divided into two phases. In the first phase, the packet is forwarded from the source server to an intermediate server whose label is the same as the destination server's label. In the second phase, the packet is sent from that intermediate server to the destination.

We consider a scenario where a server A sends a packet to destination server B. The addresses of those two servers are

[1]For a wrapped butterfly network with k levels, each level has 2^k nodes.

TABLE I
THE COST OF THE NETWORKING DEVICES, INCLUDING SWITCHES AND CABLES, WHEN USING SWITCHES WITH DIFFERENT NUMBER OF PORTS TO BUILD DPILLAR NETWORKS WITH FOUR COLUMNS OF SERVERS.

switch type	**8-port**	**16-port**	**24-port**	**48-port**
switch unit price	\$50	\$150	\$180	\$600
number of servers	1,024	16,384	82,944	1,327,104
total networking cost	\$14,848	\$339,968	\$1,410,048	\$35,831,808
per-server cost	\$14.5	\$20.75	\$17	\$27

(C_A, L_A) and (C_B, L_B), where L_A and L_B are the k-symbol labels of server A and B. Let the labels of those two servers be $L_A = (\nu_A^{k-1}...\nu_A^0)$, $L_B = (\nu_B^{k-1}...\nu_B^0)$, and $L_A \neq L_B$.

From server A in column H_{C_A}, the packet can be sent to a server A_1 in column $H_{(C_A+1)\%k}$. The label of A_1 is the same as the label of A except the C_Ath symbol in A_1's label can be any number from 0 to $(n/2-1)$. If server A_1 sends the packet to server A_2 in column $H_{(C_A+2)\%k}$, A_2's label is the same as A_1's except that the $((C_A+1)\%k)$th symbol of A_2's label can be any number from 0 to $(n/2-1)$. We see that when a packet is forwarded for one hop, we can "change" one symbol in the label of the server which receives that packet. When a packet is always forwarded from one server column to its clockwise neighboring server column (see Figure 1), within k hops, the packet can reach a server with any given label. For example, in a (n, k) DPillar network, the trace of a packet forwarded from $(0, 0...0)$ to $(k-1, 1...1)$ can be $(0, 0...0) \rightarrow (1, 0...1) \rightarrow (2, 0...11) \rightarrow (k-1, 1...1)$. As this path resembles a helix, we call the first phase of the packet forwarding process the *helix phase*.

Note that in the helix phase, we can always send the packet to either a server in the clockwise neighboring column or a server in the counter-clockwise neighboring column. However, the direction of forwarding a packet should not be changed back and forth in order to avoid loops. Some field in the packet header can be used to record the forwarding direction information of this packet. *In DPillar routing, the default direction of forwarding a packet in the helix phase is the clockwise direction.*

After the packet is forwarded to an intermediate server B^* whose label is the same as the label of destination server B, one can forward the packet to B by always sending it to the server in the clockwise neighboring column whose label is L_B too, or sending along the counter-clockwise direction. We select the shorter one among those two paths in our DPillar routing. Obviously, the shorter one is also the shortest path between B and B^*. In other words, suppose server B^* is in column $H_{C_{B^*}}$, B^* sends the packet to a server in column $H_{(C_{B^*}+1)\%k}$ if $(C_B + k - C_{B^*})\%k \leq \lfloor \frac{k}{2} \rfloor$; otherwise B^* sends the packet to a server in column $H_{(C_{B^*}+k-1)\%k}$. In either case, the label of the nexthop server should be L_B. We see that in this phase of packet forwarding, the trace of the packet is like a segment in a ring. We call the second phase the *ring phase* in packet forwarding.

Algorithm 1 shows the pseudocode of the routing algorithm each server in DPillar runs to decide the nexthop server of forwarding a packet. This algorithm takes the address of the server running this algorithm (C_A, L_A), the destination server's address (C_B, L_B), and the forwarding direction D as input parameters. D=1 means the direction is clockwise; D=−1 indicates the counter-clockwise direction. The default value of D should be 1. The output is the address of the nexthop server (C_P, L_P).

In Algorithm 1, first server A checks whether it can directly reach server B. There are two cases. (i) If L_A and L_B are the same after removing the C_Ath symbol, and B is in the same column of A or B is in the clockwise neighboring column of A, A can directly reach B. (ii) If L_A and L_B are the same after removing the C_Bth symbol, and B is in the same column of A or B is in the counter-clockwise neighboring column of A, A can directly reach B as well.

If server A cannot directly reach server B, server A first checks whether its label L_A is the same as L_B. (i) If $L_A \neq L_B$, the forwarding should be in the *helix phase*. Server A always forwards the packet to a server in its clockwise neighboring server column, i.e., column $H_{(C_A+1)\%k}$. The label of the nexthop server is the same as L_A except that the C_Ath symbol of L_A is changed to the C_Ath symbol in L_B. (ii) If $L_A = L_B$, the label of the nexthop server is fixed to L_A and the packet forwarding is in the *ring phase*. We choose the one which is closer to column H_{C_B}, among the two neighboring server column of H_{C_A}, as the column of the nexthop server.

Algorithm 1: $SRoute(C_A, L_A, C_B, L_B, D)$

input : (C_A, L_A) is the address of the server running this algorithm. (C_B, L_B) is the destination. D is the forwarding direction recorded in the packet header (either 1 or −1). L_A=$(\nu_A^{k-1}...\nu_A^0)$, L_B=$(\nu_B^{k-1}...\nu_B^0)$.
output : Address of the nexthop server (C_P, L_P).

```
/* L_A-ν_A^{C_A} means removing the C_Ath symbol
   from label L_A                                   */
if {(C_B+k-C_A)%k≤1 and L_A-ν_A^{C_A} == L_B-ν_B^{C_A}} or
{(C_A+k-C_B)%k≤1 and L_B-ν_B^{C_B} ==L_A-ν_A^{C_B}} then   /* A
can directly reach B */
    C_P ← C_B;
    L_P ← L_B;
else                    /* A cannot directly reach B */
    if L_A == L_B then                  /* the ring phase */
        L_P ← L_B;
        if (C_B + k − C_A)%k ≤ ⌊k/2⌋ then
        C_P ← (C_A + 1)%k;
        else C_P ← (C_A + k − 1)%k;
    else                                /* the helix phase */
        L_P ← (ν_A^{k-1}...ν_B^A...ν_A^0);
        C_P ← (C_A + D + k)%k;
return (C_P, L_P);
```

B. Overhead of Routing and Forwarding

With the aforementioned DPillar routing, there is no need to maintain any routing table in the servers. Each server can determine the nexthop of forwarding a packet in constant time, no matter how many servers are there in a DPillar network.

The only table each server needs to maintain is an ARP table which maps the network layer address (such as IP address) to Ethernet MAC address. As each server connects with two n-port switches, a server is directly connected with $2(n-1)$ servers. The number of entries in the ARP table is $2(n-1)$, which is a constant too. Therefore, once we have selected the type of switches to build a DPillar, the overhead of mapping network layer address to Ethernet MAC address is also a constant which does not depend on the number of servers in a DPillar network. We will present a prototyping implementation of the routing algorithm and the performance evaluation of our implementation in section V-A.

C. Longest Path in DPillar

The DPillar routing presented in Algorithm 1 does not compute the shortest path between two servers. However, the paths computed by Algorithm 1 have bounded length. Here we study what is the length of the longest path in a DPillar network.

Because we use Ethernet switches as dummy layer-2 connection media, the switches should switch packets at line speed. In counting the path length in a DPillar network, we treat the distance between two servers connecting to the same switch as *one* hop, although a server needs to go through two server-to-switch links to reach another server connecting to the same switch.

We consider the worst case scenario where the labels of two servers have no common symbols. If the label of server A and the label of server B have no common symbols, a packet needs to be forwarded k times in order to reach a server B^* whose label is the same as server B's label. After that, from server B^*, the packet still needs to be forwarded $\lfloor k/2 \rfloor$ hops in order to reach server B in the worst case. The longest path computed by Algorithm 1 in a (n,k) DPillar network is $k + \lfloor k/2 \rfloor$.

Proposition III.1 *Using Algorithm 1, any two servers in a (n,k) DPillar network can reach each other in at most $k + \lfloor k/2 \rfloor$ hops.*

PROOF: For a (n,k) DPillar network, the server label has k symbols. In the worst case, a packet needs to be forwarded k hops in order to reach a server whose label is the same as the destination's label. At this point, the server having the packet and the destination server are located along a ring with k nodes (all those nodes have the same label). In the worst case, the packet needs to be forwarded $\lfloor k/2 \rfloor$ more hops to reach the destination server. Therefore, the longest path in a (n,k) DPillar network is $k + \lfloor k/2 \rfloor$. ■

One thing worthy of mention is that in a (n,k) DPillar network, the shortest path from servers $(0,000)$ to server $(\lfloor \frac{k}{2} \rfloor, 1...1)$ is also $k + \lfloor k/2 \rfloor$. Therefore, the physical diameter of a (n,k) DPillar network is also $k + \lfloor k/2 \rfloor$.

D. Traffic Distribution in All-to-All Communication

For all-to-all communication in a (n,k) DPillar network, each server has $N-1$ flows to other $N-1$ servers, where $N = k(n/2)^k$ is the total number of servers in the network. The total number of flows is $N(N-1)$. As each flow traverses at most $3k/2$ hops[2] and each hop consists of 2 server-to-switch links, the total number of links those $N(N-1)$ flows traverse is at most $3kN(N-1)$. The total number of server-to-switch links is $2k(n/2)^k = 2N$ and all links in a DPillar network are identical. Therefore, each link carries about $3k(N-1)/2$ flows.

Proposition III.2 *In all-to-all communication where each server in a (n,k) DPillar network has one flow to each of the other servers, a server-to-switch link carries at most $3k(N-1)/2$ flows.*

IV. Handling Failure and Balancing Load in DPillar

The rich connections in DPillar facilitate the design of simple yet efficient fault-tolerant routing scheme. In this section, we present the design of a routing scheme which can bypass a wide range of failures in DPillar. We also discuss how to use the insights gotten from designing the fault-tolerant routing scheme to balance the traffic load in DPillar.

A. Discovering Failures

To bypass failures in DPillar network, the first step would be detecting the failures. Each server in DPillar runs a lightweight *Hello* protocol to report the reachability to other servers connected to the same switches. If a server A does not hear *Hello* message from server B for a certain period of time, server A assumes B is not directly reachable. Servers should not forward any *Hello* messages.

We can set the period of sending *Hello* messages to be slightly smaller than the time for ARP cache to timeout. Therefore, the server can always have "fresh" ARP cache so it does not need to query for the MAC address of a neighbor connecting to the same switch before sending out any data packets to that neighbor.

B. Bypassing Failures

We consider a scenario where a server A needs to use P as the next hop server to reach destination server B, according to the DPillar routing presented in Algorithm 1. However, server P is not reachable from A, so A should try to bypass this failure by sending packets to another nexthop $P^{'}$. Suppose the IDs of those four servers are (C_A, L_A), (C_B, L_B), (C_P, L_P), and $(C_{P'}, L_{P'})$. We discuss how to bypass the failure in the *ring phase* and the *helix phase* separately.

[2]We ignore the floor operator for simplicity.

1) The ring phase: Bypassing a failure in the ring phase is relatively straightforward. As there are two ways to forward a packet to the destination in the ring phase, either along the clockwise direction or the counter-clockwise direction, a packet should be forwarded along the other direction, if it cannot be forwarded along one direction. However, to avoid forwarding loops, the forwarding direction of a packet should not be changed more than once in the ring phase. If a packet has changed its forwarding direction in the ring phase and it encounters a failed server again, that packet should be dropped.

2) The helix phase: When a failure is detected in the helix phase, i.e., nexthop P is unreachable, A first tries to bypass the failed server by sending the packet to a reachable server in the clockwise neighboring column. If none of the servers in the clockwise neighboring column is reachable, the packet will be forwarded to a server in the counter-clockwise neighboring column of server A.

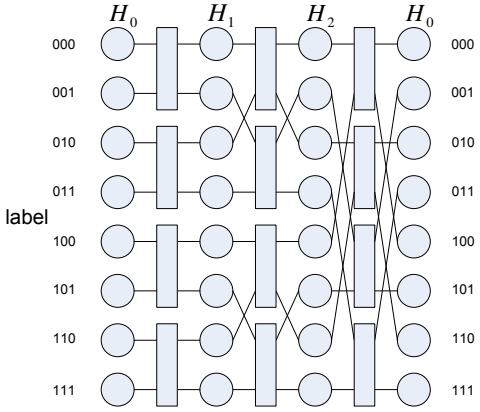

Fig. 3. A $(4, 3)$ DPillar network example.

Select a clockwise nexthop. How to select the label of the new nexthop server is important for avoiding any forwarding loops. We use the example shown in Figure 3 to explain the reason. Let's consider the scenario that server A, whose address is (0,000), sends a packet to server B, whose address is (2,001). According to Algorithm 1, the ID of the nexthop server P should be (1,001) but server (1,001) has failed. Suppose server A tries to detour the failed server (1,001) by sending the packet to a new nexthop server (1,000), the path taken by the packet will be (0,000) $\rightarrow$ (1,000) $\rightarrow$ (2,000) $\rightarrow$ (0,000), which is a loop.

We see that the reason for the aforementioned loop to occur is that, although A detours the packet and sends it to server (1,000), the packet will reach A again because A is an intermediate node from server (1,000) to destination B, according to Algorithm 1. To avoid this loop, we can let server (1,000) not forward the packet to (2,000). For example, if (1,000) forwards the packet to (2,010), the path taken by the packet will be (0,000) $\rightarrow$ (1,000) $\rightarrow$ (2,010) $\rightarrow$ (0,010) $\rightarrow$ (1,011) $\rightarrow$ (2,001). The insights here are two-fold: (i) Forwarding from (0,000) to (1,000) will detour the failed server (1,001); (ii) Forwarding from (1,000) to (2,010) will ensure (0,000) is not in the path from (2,010) to (2,001), because when the packet reaches a server in column 0 again, the address of that server must be $(0, x1y)$, where $x, y \in \{0, 1\}$.

In the above example, to make sure the packet is forwarded from (0,000) to (2,010), we use *tunneling*. That is, server (0,000) encapsulate the packet with another outer header, whose destination is set to (2,010). Tunneling the packet between (0,000) and (2,010) will make sure the path of the encapsulated packet is (0,000) $\rightarrow$ (1,000) $\rightarrow$ (2,010). Therefore, the failed server (1,001) is bypassed.

We conclude the discussion into the following Proposition IV.1 regarding how to bypass a failed servers in the helix phase. Note that we omit the $\%k$ in presenting the indexes for clarity.

Proposition IV.1 *When sending a packet to destination* $(C_B, \nu_B^{k-1}...\nu_B^0)$*, server* $(C_A, \nu_A^{k-1}...\nu_A^0)$ *can bypass server* $(C_A + 1, \nu_A^{k-1}...\nu_B^{C_A}...\nu_A^0)$ *by tunneling the packet to* $(C_A + 2, \nu_A^{k-1}...\nu_{P'}^{C_A+1}\nu_{P'}^{C_A}...\nu_A^0)$*, if* $\nu_{P'}^{C_A} \neq \nu_B^{C_A}$ *and* $\nu_{P'}^{C_A+1} \neq \nu_A^{C_A+1}$.

PROOF: Because of the symmetric structure of DPillar, without losing generality, we consider server A, with address $(0, \nu_A^{k-1}...\nu_A^0)$, bypasses server $(1, \nu_A^{k-1}...\nu_A^1\nu_B^0)$ in reaching destination $(C_B, \nu_B^{k-1}...\nu_B^0)$. Server A tunnels the packet to server $P^{'}$, whose address is $(2, \nu_A^{k-1}...\nu_{P'}^1\nu_{P'}^0)$, and we have

$$\nu_{P'}^1 \neq \nu_A^1 \qquad \nu_{P'}^0 \neq \nu_B^0$$

To tunnel the packet from A to $P^{'}$, the nexthop is $(1, \nu_A^{k-1}...\nu_A^1\nu_{P'}^0)$. Because $\nu_{P'}^0 \neq \nu_B^0$, that nexthop is not the failed server $(1, \nu_A^{k-1}...\nu_A^1\nu_B^0)$. After the packet is forwarded from $P^{'}$ to some server A_0 in column H_0, the address of A_0 is $(0, \nu_B^{k-1}...\nu_{P'}^1\nu_{P'}^0)$. Because $\nu_{P'}^1 \neq \nu_A^1$, A_0 is not A. A_0 forwards the packet to a server A_1 in column H_1 whose address is $(0, \nu_B^{k-1}...\nu_{P'}^1\nu_B^0)$. Because $\nu_{P'}^1 \neq \nu_A^1$, A_1 will not be the failed server $(1, \nu_A^{k-1}...\nu_A^1\nu_B^0)$. After A_1 forwards the packet, it reaches some server A_2 in column H_2 whose label is the same as the destination. After that, it is the ring phase and we can always bypass a failure in ring phase by changing the packet forwarding direction. ■

Make a "packet u-turn". Proposition IV.1 assumes a server can always forward a packet to another server in its clockwise neighboring column in the helix phase. However, it is possible that a server A cannot send the packet to any servers in its clockwise neighboring column, e.g., the server (0,000) in Figure 3 has a link failure so it is disconnected from the top switch between H_0 and H_1, or the top switch between H_0 and H_1 has failed. If that is the case, server A needs to change the packet forwarding direction (make a "u-turn") to bypass a failure in the helix phase, i.e., if a packet cannot be forwarded to a server in the clockwise neighboring column, it can be forwarded to a server in the counter-clockwise neighboring column. The forwarding direction information should be recorded in the packet header, so that other servers will forward this packet along the new direction. Similar to the rationale behind Proposition IV.1, we can prove the following Proposition IV.2. Again, the $\%k$ is omitted in presenting the indexes for clarity.

Proposition IV.2 *When sending a packet to destination* $(C_B, \nu_B^{k-1}...\nu_B^0)$*, if server* $(C_A, \nu_A^{k-1}...\nu_A^0)$ *cannot reach any server in* H_{C_A+1}*, it can bypass the failure by changing the*

packet forwarding direction and sending the packet to server $(C_A-1,\ \nu_A^{k-1}...\nu_{P'}^{C_A-1}...\nu_A^0)$*, where* $\nu_{P'}^{C_A-1} \neq \nu_A^{C_A-1}$.

PROOF: We consider the scenario where server A, whose address is $(0, \nu_A^{k-1}...\nu_A^0)$, sends a packet to destination B $(C_B, \nu_B^{k-1}...\nu_B^0)$ but A cannot send the packet to any server in H_1 during the helix phase. To bypass the failure, server A changes the forwarding direction of the packet and sends it to nexthop $P^{'}$ whose address is $(k-1, \nu_{P'}^{k-1}...\nu_A^0)$, where

$$\nu_{P'}^{k-1} \neq \nu_A^{k-1}$$

After the packet is forwarded along the counter-clockwise direction from $P^{'}$ to some server A_1 in H_1, the address of A_1 should be $(1, \nu_{P'}^{k-1}...\nu_B^2\nu_B^1\nu_A^0)$. Because $\nu_{P'}^{k-1} \neq \nu_A^{k-1}$, A_1 is not a server connected to server A by the same switch and A_1 can directly reach some servers in H_0. After A_1 forwards the packet to some server A_0 in H_0, A_0's address is $(0, \nu_{P'}^{k-1}...\nu_B^0)$. Because $\nu_{P'}^{k-1} \neq \nu_A^{k-1}$, A_0 is not A and there will be no loops. After A_0 forwards the packet to a server A_{k-1} in column H_{k-1}, the packet reaches a server whose label is same as the label of B. ■

Selecting a counter-clockwise nexthop. After the forwarding direction of a packet is changed, it should be recorded in the packet header so that all DPillar servers will follow the new direction in forwarding that packet. To avoid potential forwarding loops, when the forwarding direction of a packet was changed in the helix phase, that packet should not change direction again. In other words, for a packet whose direction was changed before from clockwise to counter-clockwise and it cannot be forwarded along the new direction (no reachable servers in the counter-clockwise neighboring column), the packet should be dropped. If server A needs to forward packet to server P in the counter-clockwise neighboring column but P is unreachable, A can bypass the failure according to the following Proposition IV.3, when there are reachable servers in the counter-clockwise neighboring column. The proof of Proposition IV.3 is similar to the proof of Proposition IV.1. We omit it here to save space.

Proposition IV.3 *When sending a packet to destination* $(C_B, \nu_B^{k-1}...\nu_B^0)$*, server* $(C_A, \nu_A^{k-1}...\nu_A^0)$ *can bypass server* $(C_A-1, \nu_A^{k-1}...\nu_B^{C_A-1}...\nu_A^0)$ *by tunneling the packet to* $(C_A-2, \nu_A^{k-1}...\nu_{P'}^{C_A-1}\nu_{P'}^{C_A-2}...\nu_A^0)$*, if* $\nu_{P'}^{(C_A-1)} \neq \nu_B^{C_A-1}$ *and* $\nu_{P'}^{(C_A-2)} \neq \nu_A^{(C_A-2)}$.

C. The Fault-tolerant Routing Algorithm

Based on the discussion in section IV-B, we design the DPillar fault tolerant routing algorithm as shown in Algorithm 2. This fault-tolerant algorithm first calls Algorithm 1 to compute a nexthop (C_P, L_P). Then Algorithm 2 tests whether (C_P, L_P) is reachable. If it is, Algorithm 2 does nothing. Otherwise, Algorithm 2 tries to find a new nexthop according to the basic ideas specified in Proposition IV.1 $\sim$ Proposition IV.3 and returns a new nexthop. If the new nexthop is null, it means the packet should be dropped to prevent forwarding loops.

Algorithm 2: $FTRoute(C_A, L_A, C_B, L_B, D)$

input : (C_A, L_A) is the address of the server running this algorithm. (C_B, L_B) is the address of the destination. D is the forwarding direction, either 1 or -1.
output : Address of the nexthop server (C_P, L_P).

```
/* call SRoute to compute a nexthop                    */
(C_P, L_P) = SRoute(C_A, L_A, C_B, L_B, D);
if (C_P,L_P) is reachable then                  /* no failure */
    return ((C_P,L_P));
/* try to bypass the failure                           */
if L_A==L_B then                          /* the ring phase */
    if the packet direction was changed then
        C_P ← null; L_P ← null;
    else
        if (C_A + k − C_B)%k ≤ ⌊k/2⌋ then
            C_P ← (C_A + 1)%k;
        else C_P ← (C_A + k − 1)%k;
else                                      /* the helix phase */
    if the packet direction was changed then
        apply Proposition IV.3 to get new nexthop;
        if nexthop exists then set C_P and L_P;
        else C_P ← null; L_P ← null;
    else
        apply Proposition IV.1 to get new nexthop;
        if nexthop exists then set C_P and L_P;
        else
            change packet direction according to
            Proposition IV.2;
            if nexthop exists then set C_P and L_P;
            else C_P ← null; L_P ← null;
return (C_P, L_P);
```

D. Traffic-aware Routing in DPillar Networks

It is worthy to highlight that because DPillar has totally symmetrical structure, traffic in a DPillar network should be evenly distributed in the long term, if communications happen evenly between any source and destination servers, and the volume of each communication is also evenly distributed. However, in the short term, traffic bursts may overload some servers and we focus on how to alleviate that situation.

We make the traffic balancing decision in a local manner to avoid complicating the DPillar routing too much. Basically, each server monitors the load of its directly connected neighboring servers (servers one hop away). The forwarding engine of each server maintains some statistics regarding the history of packet forwarding during the last few seconds or minutes, such as the packet dropping rate during the last few minutes, the CPU share used in packet forwarding. We can extend the *Hello* protocol discussed in section IV-A to also report the server load information.

When a server A forwards packets, it should take into account the load of those neighbors in selecting the nexthop. If a nexthop server P is overloaded, A should avoid using P and select another nexthop. The exact same rationales we have discussed in presenting the fault-tolerant routing scheme can be adopted to *bypass* the overload server P.

V. Evaluations

We evaluate DPillar from two different aspects. First, using a small testbed, we study the microscopic behavior of DPillar by measuring the overhead of one DPillar server in computing nexthop and in forwarding traffic for other servers. Second, we study the macroscopic behavior of DPillar by simulating the packet routing and forwarding in a large scale DPillar network, using a simulation tool we developed.

A. Implementing DPillar Routing & Measuring the Performance

1) Implementation: We have implemented the static routing algorithm presented in section III as an element in the Click software router [2]. In our implementation, we use IP address to encode the column and label information of each server, so as to provide backward comparability to the upper layer applications running in the data center and those applications can still use TCP/IP. We use the most significant 8 bits in the IP header to represent the column number of a server. As each host has two NICs, the least significant bit in the IP header is used to represent the NICs. The k-symbol label consumes $k\lceil \log_2(n) - 1 \rceil$ bits. In our implementation, the 32 bits in the IPv4 address is allocated as shown in Figure 4.

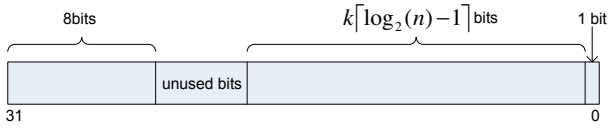

Fig. 4. Configure the IP addresses of each host according to the host column and label information.

Suppose we use 48-port switches to build a DPillar network with 4 columns of hosts. The total number of hosts in this DPillar network is about 1.3 million. We need to use 2 bits in the most significant 8 bits in the IP header to represent the column. In the rest 24 bits, we use $4\lceil \log_2(48) - 1 \rceil + 1 = 21$ bits to represent the label of each server and the interface.

2) Performance measurement: **The testbed network.** We use a PC with 2.4GHz dual-core CPU and 1GB memory to run the DPillar routing and forwarding element implemented in Click. A testbed network consisting of three servers, as shown in Figure 5, are used in our experiment. In Figure 5, server P is a DPillar server which forwards packets between server A and server B.

Overhead of processing one packet. For each packet forwarded by server P, we record the time it takes to compute the nexthop server. Our measurement results show that the static routing algorithm of DPillar consumes about 90 CPU cycles to compute the nexthop server for one packet. The fault-tolerant routing algorithm of DPillar consumes about 250 CPU cycles.

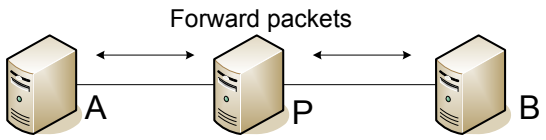

Fig. 5. The testbed network. Server P forwards packets between A and B.

For comparison purpose, we also measure the time it takes to do table lookup in the conventional IP forwarding implemented in Click (the RadixIPLookup element), when the table size varies. Figure 6 plots the results of our experiments. One table lookup takes more than 600 CPU cycles when the table has 128 entries. As the number of entries in the table increases, it takes longer time to do lookup.

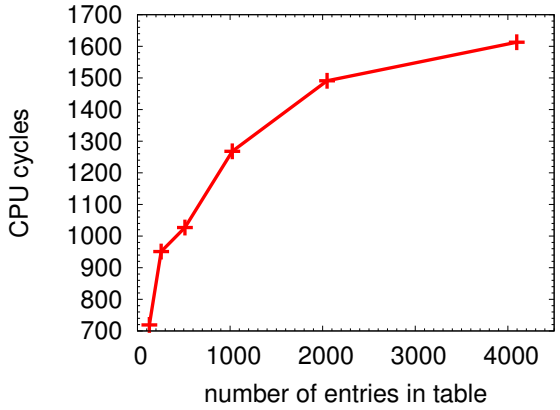

Fig. 6. The number of CPU cycles one table lookup operation consumes in conventional IP forwarding.

Overhead of forwarding traffic. We also test the overhead in terms of CPU usage when a DPillar server forwards traffic for other servers. In this experiment, server A and B in Figure 5 use iperf to send out UDP traffic to each other at various speed and server P forwards the traffic between A and B. We record the CPU usage of the Click kernel thread and plot the result in Figure 7.

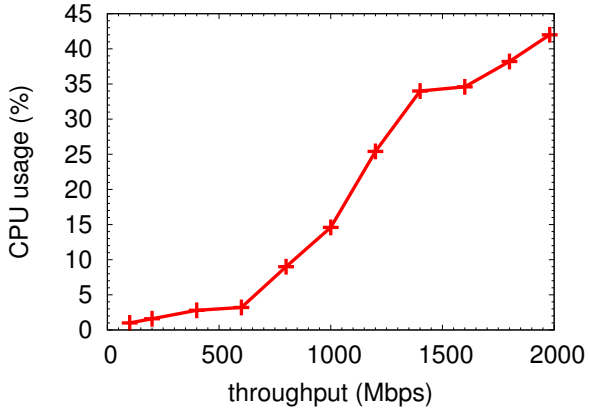

Fig. 7. The server CPU usage under various traffic load. The packet size is 1024 bytes.

The results in Figure 7 show that even the DPillar server forwards traffic at full load, i.e., 1 Gbps each direction, 2 Gbps in total, the CPU usage of the DPillar server is less than 50%. Our server is a dual-core machine and the less than 50% CPU usage is for one CPU core. The other core is almost 100% idle. As commodity PCs with multi-core CPUs (dual-core or quad-core) are becoming common, we expect the traffic forwarding overhead of the DPillar server can be amortized.

Another way to decrease the CPU load but still maintain the throughput is to use larger packets [11]. As *Jumbo Frame packet transfer* is commonly supported in commodity Ethernet switches[3], we can take advantage of this feature to reduce the CPU load of DPillar servers in relaying traffic for other servers.

B. Simulation Evaluations

We also developed a simulation tool to simulate the packet routing and forwarding in a large scale DPillar network. Given the number of server columns k and the switch port number n, our simulation tool builds a (n, k) DPillar network topology and simulates how each server routes packets.

1) Average Path Length: In our simulations, we focus on a scenario where the DPillar network is built from 12-port switches, i.e., $n = 12$, and the number of server columns k varies. For each network, we randomly select $100,000$ source-destination pairs and simulate packet routing and forwarding in the network. We also simulate the shortest paths between those source-destination pairs. Figure 8 plots the results of the average path lengths.

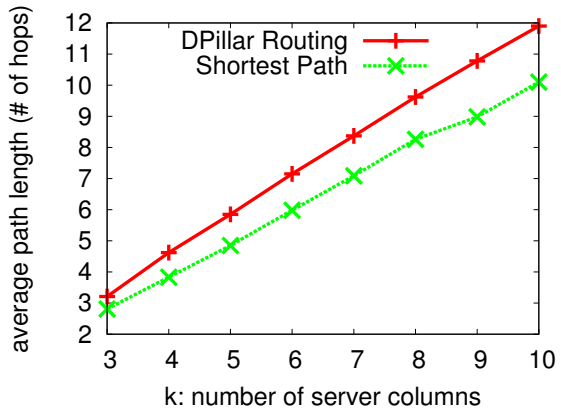

Fig. 8. Comparison of DPillar routing and shortest path routing.

The results show that the path lengths are proportional to the number of server columns in a DPillar network. Using the simple routing algorithm does not inflate the path length too much, especially when the number of server columns k is relatively small.

2) Average Path Length in Failure-tolerant Routing: We also study the fault-tolerant behavior of DPillar routing. In this simulation, we use a $(12, 4)$ DPillar network to conduct our experiments. In each simulation instance, we first randomly fail a certain number of servers in the network. Then we randomly select 100,000 source-destination pairs and simulate the fault-tolerant routing scheme presented in Algorithm 2. We range the number of failed servers from 1 to 300. The average path length, as a function of the number of failed hosts, is plotted in Figure 9

Our simulation results show that the average path length increases as there are more failed servers in the DPillar network.

[3]The $150 16-port Gbit switch TRENDnet TEG-S16R mentioned in section II-C supports up to 12.2k bytes jumbo frame.

Fig. 9. Path length in fault-tolerant routing.

However, no packet drop occurs in our simulations, even when 300 servers in the DPillar network have failed. As a $(12, 4)$ DPillar network has about 5,000 servers, 300 server failures mean 6% of the servers are failed.

VI. Background and Related Work

A. Interconnection Networks in Data Centers

There are two categories of interconnection networks used in building data centers. The first one has a clear boundary between the network and the end hosts. Usually, multiple levels of switches are interconnected into a switching fabric and the servers are attached as the "leafs" of the switching fabric [3]. The servers are pure end-hosts, which perform computation task only. Having one interface is enough for each server to be connected with other servers.

In the second category of interconnection network, the servers are not only the computation workstations but also intermediate nodes relaying traffic for other servers. Servers are connected with each other by point-to-point links or hubs to construct certain topologies. Classic interconnection topologies include full mesh, hypercube, butterfly, de Bruijn, etc [7,13,17]. Compared with the interconnection network with a switching fabric, using servers as relay nodes and placing more intelligence on servers is usually more flexible, because the servers are much easier to program than the switching devices in a switching fabric.

B. Related Work in Data Center Networks

A thread of recent research activities on data center networks have proposed several interconnection architectures. The Monsoon network presented in [10] uses a hierarchical switching fabric where the top-of-rack switches are connected to a high-bandwidth core switch. There is a *Directory Service* in Monsoon network to provide the mapping between network addresses and MAC addresses. The fat-tree network presented in [4] is also a switching fabric-based interconnection network. The fat-tree network uses identical switches to build the switching fabric. Therefore, there is no need to install the high-bandwidth and expensive core switches. The switches used in [4] should have layer-3 switching capability and need to be

TABLE II

COMPARISON BETWEEN DIFFERENT DATA CENTER INTERCONNECTION NETWORKS. PARAMETER n IS THE NUMBER OF PORTS EACH SWITCH HAS; N REPRESENTS THE TOTAL NUMBER OF SERVERS; U_s IS THE UNIT PRICE OF A n-PORT SWITCH; D IS THE NETWORK DIAMETER. FOR DCELL, FICONN, AND BCUBE, l IS THE NUMBER OF RECURSIVELY CONSTRUCTION LEVEL. FOR DPILLAR, k IS THE NUMBER OF SERVER COLUMNS.

	DCell	**FiConn**	**BCube**	**FatTree**	**DPillar**
server degree	$l+1$	2	$l+1$	1	2
diameter D	2^l	$\approx 2^{l+1}$	$l+1$	constant	$\leq \frac{3k}{2}$
bisection width	$\frac{N}{4\log_n N}$	$\frac{N}{4\times 2^l}$	$\frac{N}{2}$	$\frac{N}{2}$	$\frac{N}{k}$
number of servers N	$(n+1)^{2^l}$	$2^{l+2}(\frac{n}{4})^{2^l}$	n^{l+1}	$\frac{n^3}{4}$	$k(\frac{n}{2})^k$
number of servers as a function of diameter D	$(n+1)^D$	$2D(\frac{n}{4})^{\frac{D}{2}}$	n^D	N/A	$\frac{2}{3}D(\frac{n}{2})^{\frac{2}{3}D}$
number of switches	$\frac{N}{n}$	$\frac{N}{n}$	$(l+1)\frac{N}{n}$	$6\frac{N}{n}$	$k\frac{N}{n}$
cost of connecting one server†	$\frac{U_s}{n}$	$\frac{U_s}{n}$	$(l+1)\frac{U_s}{n}$	$5\frac{U_s}{n}$	$2\frac{U_s}{n}$
switch upgrade	no	no	no	yes	no
traffic balance	no	no	yes	yes	yes

†Not including the cost of NICs and cables.

slightly upgraded in order to make full use of the underlying topology. The PortLand network is proposed in [15], which is also based on fat-tree. PortLand uses hierarchical pseudo MAC addresses, so as to support efficient layer-2 routing and forwarding, as well as virtual machine migration. A centralized *fabric manager* is used to maintain soft state about the network topology and assist ARP resolution.

The DCell [12] interconnection network is a server-centric network where the servers are not only the computation workstations but also the intermediate interconnection nodes. A higher level DCell network can be recursively constructed from lower level DCell networks, so that the number of servers in a DCell network grows double exponentially as the level increases. As the levels in a DCell network increases, the servers need to install more interfaces to do interconnection. The links in DCell network are not evenly loaded. Those links connecting lower level DCells are usually more loaded than the links connecting higher level DCells. The FiConn network proposed in [14] uses similar recursive construction scheme as DCell. However, each server in FiConn can have only two interfaces. FiConn also has the issue of unevenly loaded links. The BCube [11] is another server-based network. Servers in BCube have multiple interfaces and multiple layers of commodity switches are used to connect the servers. BCube has rich connections so as to support bandwidth intensive applications running in data centers. Table II summarizes some key features of different data center interconnection networks.

VII. CONCLUSION

In this paper, we present DPillar, a scalable data center network architecture which uses only commodity off-the-shelf hardware. DPillar can easily scale to huge number of servers without imposing any additional requirements to the devices, such as installing additional NICs in the servers. The topology of DPillar is totally symmetric and a DPillar network has balanced network capacity. DPillar is a server centric and the networking intelligence is placed in the servers, so that the switches used in DPillar are merely dummy layer-2 devices connecting the servers. We designed simple yet efficient routing schemes for DPillar. Our prototyping implementation and simulation studies show that the routing schemes are lightweight, high-performance, and efficient in bypassing failures in the network.

REFERENCES

[1] Amazon elastic compute cloud. http://aws.amazon.com/ec2/.

[2] The click modular router project. http://read.cs.ucla.edu/click/.

[3] Cisco data center infrastructure 2.5 design guide, Dec. 2007. www.cisco.com/application/pdf/en/us/guest/netsol/ns107/c649/ccmigration_09186a008073377d.pdf.

[4] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of SIGCOMM' 08*, 2008.

[5] C. Bornstein, A. Litman, B. Maggs, R. Sitaraman, and T. Yatzkar. On the bisection width and expansion of butterfly networks. Technical Report CMU-CS-97-132, School of Computer Science, Carnegie Mellon University, May 1997.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 205–218, Berkeley, CA, USA, 2006. USENIX Association.

[7] W. J. Dally and B. P. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2004.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI '04: Proceedings of the 5th symposium on Operating systems design and implementation*, pages 137–150, 2004.

[9] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, New York, NY, USA, 2003. ACM Press.

[10] A. Greenberg, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Towards a next generation data center architecture: scalability and commoditization. In *PRESTO '08: Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, pages 57–62, New York, NY, USA, 2008. ACM.

[11] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, T. Chen, Y. Zhang, and S. Lu. BCube: A high performance, server-centric network architecture for modular data centers. In *Proceedings of SIGCOMM' 09*, 2009.

[12] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. Dcell: a scalable and fault-tolerant network structure for data centers. In *Proceedings of SIGCOMM' 08*, 2008.

[13] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays. Trees. Hypercubes*. Morgan Kaufmann, 1992.

[14] D. Li, C. Guo, H. Wu, K. Tan, Y. Zhang, and S. Lu. FiConn: Using backup port for server interconnection in data centers. In *Proceedings of INFOCOM' 09*, 2009.

[15] 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. In *Proceedings of SIGCOMM' 09*, 2009.

[16] L. Rabbe. Powering the Yahoo! network, Nov. 2006. http://ycorpblog.com/2006/11/27/powering-the-yahoo-network.

[17] M. R. Samatham and D. K. Pradhan. The de Bruijn Multiprocessor Network: A Versatile Parallel Processing and Sorting Network for VLSI. *IEEE Trans. Comput.*, 38(4):567–581, 1989.

APPENDIX

We prove that the bisection width of a (n, k) DPillar network is $(n/2)^k$.

PROOF: Our proof is inspired by previous work [5] in studying the bisection width of butterfly networks.

Clearly, if we cut a (n, k) DPillar network horizontally, i.e., each server column is cut into halves, we can always cut a DPillar network into a top half and a bottom half by cutting the connections among H_{k-1}, S_{k-1}, and H_0. For example, we can divide the DPillar network shown in Figure 2 into top and bottom halves by cutting the links cross a "virtual" line between row (13) and row (20). Only some of the connections among H_{k-1}, S_{k-1}, and H_0 will cross that virtual line. Because each switch in S_{k-1} has $n/2$ links crossing the virtual line, the total number of links is $(n/2)^{k-1} \times (n/2) = (n/2)^k$. Hence, we have an upper bound, $(n/2)^k$, for the bisection width of a (n, k) DPillar network.

Next we find the lower bound of the bisection width. Let G denote the number of servers in each column of a (n, k) DPillar network. We consider bisecting the $2G$ servers in server columns S_0 and S_{k-1} by embedding a complete bipartite graph $K_{G,G}$ into a (n, k) DPillar network, so that the left side nodes and right side nodes of $K_{G,G}$ are mapped to the servers in S_0 and servers in S_{k-1} of the (n, k) DPillar network, respectively. If each of the G servers in S_0 has a path to every server in S_{k-1}, because DPillar network is symmetry, there are at most $G/2$ paths uses the same server-to-switch link. Also because the bisection width of a complete bipartite graph $K_{G,G}$ is $G^2/2$, the size of the cut that bisects the $2G$ servers in S_0 and S_{k-1} should be at least G.

Now we consider a minimal cut C that bisects all servers in a (n, k) DPillar network into Set_1 and Set_2. If there exist two neighboring server columns, e.g., S_i and $S_{(i+1)\%k}$, where the $2G$ servers are bisected by cut C, we know that the size of cut C is at least G. Otherwise, we find two neighbors server columns S_j and $S_{(j+1)\%k}$ so that among the $2G$ servers in those two server columns, more are in Set_1 than in Set_2. Then we move some servers (among those $2G$ servers in server columns S_j and $S_{(j+1)\%k}$) from Set_1 to Set_2 so that half of those $2G$ servers are in Set_1. Note that moving the servers from Set_1 to Set_2 does not increase the size of cut C. We already know that bisecting the $2G$ servers in S_j and $S_{(j+1)\%k}$ requires cutting at least G links. Hence, the size of cut C has lower bound $G = (n/2)^k$. Also because the upper bound is $(n/2)^k$, the bisection width of a (n, k) DPillar network is $(n/2)^k$. ■