Scalable and Cost-Effective Interconnection of Data-Center Servers Using Dual Server Ports

Dan Li, Member, IEEE, Chuanxiong Guo, Haitao Wu, Member, IEEE, Kun Tan, Member, IEEE, Yongguang Zhang, Songwu Lu, and Jianping Wu, Senior Member, IEEE

Abstract—The goal of data-center networking is to interconnect a large number of server machines with low equipment cost while providing high network capacity and high bisection width. It is well understood that the current practice where servers are connected by a tree hierarchy of network switches cannot meet these requirements. In this paper, we explore a new server-interconnection structure. We observe that the commodity server machines used in today’s data centers usually come with two built-in Ethernet ports, one for network connection and the other left for backup purposes. We believe that if both ports are actively used in network connections, we can build a scalable, cost-effective interconnection structure without either the expensive higher-level large switches or any additional hardware on servers. We design such a networking structure called FiConn. Although the server node degree is only 2 in this structure, we have proven that FiConn is highly scalable to encompass hundreds of thousands of servers with low diameter and high bisection width. We have developed a low-overhead traffic-aware routing mechanism to improve effective link utilization based on dynamic traffic state. We have also proposed how to incrementally deploy FiConn.

Index Terms— Computer networks, computers and information processing, Internet.

I. INTRODUCTION

DATA-CENTER networking designs both the network structure and associated protocols to interconnect thousands of [8] or even hundreds of thousands of servers [1]–[3] at a data center, with low equipment cost, high and balanced network capacity, and robustness to link/neighbor faults. Its operation is essential to offering both numerous online applications, e.g., search, gaming, Web mail, and infrastructure services, e.g., GFS [5], Map-reduce [6], and Dryad [7]. It is well understood that tree-based solutions in current practice cannot meet the requirements [8], [9].

In this paper, we study a simple technical problem: Can we build a scalable, low-cost network infrastructure for data centers, using only the commodity servers with two ports and low-end, multiprotocol commodity switches? If we can solve the problem, the potential benefits are multifaceted. First, it costs less to build a data-center network. We do not need high-end, expensive switches, which are widely used today. Standard, off-shelf servers with two ports (one for operation in network connection, the other for backup) are also readily available. Second, the wiring becomes relatively easy since only two server ports are used for interconnection. We do not need to add additional hardware or wires on a server except the two NIC ports. Third, it may spawn more academic research into data centers. New problems and solutions in data-center networking, systems, and applications can be found, implemented, and assessed through an easy-to-build test bed at a university or institution. Today, data-center infrastructure may only be afforded by a few cash-rich companies such as Microsoft, Google, and Yahoo.

Neither current practice nor recent proposals [8]–[10] can solve our problem. The tree-based solution requires expensive, high-end switches at the top level of the tree in order to alleviate the bandwidth bottleneck. The scaling of the Fat-Tree solution [8] is limited to the number of ports at a switch, and it also needs more switches. DCell [9] and BCube [10] typically require more ports per server—e.g., four—to scale to a large server population. The fundamental problem is that, we need to design a new network structure that works for servers with node degree of only 2 in order to scale.

We propose FiConn, a scalable solution that works with servers with two ports only and low-cost commodity switches. FiConn defines a recursive network structure in levels. A high-level FiConn is constructed by many low-level FiConn. When constructing a higher-level FiConn, the lower-level FiConn use half of their available backup ports for interconnections and form a mesh. This way, the number of servers in FiConn, N, grows double-exponentially with FiConn levels. For example, if 48-port switches are used, a 2-level FiConn can support 361 200 servers. The diameter of FiConn is \( O(\log N) \), which is small and can thus support applications with real-time requirements. The bisection width of FiConn is \( O(N/\log N) \), showing that FiConn may well tolerate port/link faults. Although we use the backup port of each server, the server’s reliability is not compromised because it still uses the other port when one fails.

Some preliminary results were published at IEEE INFOCOM 2009. In this version, the following modifications are made: 1) add a section to present the incremental deployment of FiConn; 2) add a discussion section for more design issues in FiConn; 3) add the comparison with BCube and more evaluations.
Routing over FiConn is also renovated in two aspects. First, our routing solution balances the usage of different levels of links. Second, FiConn uses traffic-aware routing to improve effective link utilization based on dynamic traffic state. In the traffic-aware routing, considering the large server population, we use no central server(s) for traffic scheduling and do not exchange traffic state information among even neighboring servers. Instead, the traffic-aware path is computed hop-by-hop by each intermediate server based on the available bandwidth of its two outgoing links.

We have also considered how to incrementally deploy FiConn, which is important for building mega data centers. By adding shortcut links in incomplete FiConn, we guarantee the high bisection width of incomplete FiConn. The shortcut links added can also be fully utilized during the process of incremental deployment and are easy to rewire. In addition, the shortcut links do not break the routing scheme in FiConn.

Evaluation of FiConn is conducted on two sides. First, we conduct simulations to study the effectiveness of traffic-aware routing. The results show that our traffic-aware routing achieves much higher throughput for burst traffic between two subsets of FiConn servers, which is common for data-center applications such as Map-Reduce. Second, we compare FiConn with other recently proposed data-center networking structures in detail, disclosing that FiConn is advantageous over all other in terms of deploying cost, but holds high scalability with constant number of server ports and switch ports.

In summary, we make three main contributions in FiConn. First, FiConn offers a novel network structure that is highly scalable with off-the-shelf servers of node degree 2 and low-end commodity switches while having low diameter and high bisection width. Second, FiConn uses traffic-aware routing that exploits the available link capacities based on traffic dynamics and balances the usage of different links to improve the overall network throughput. Third, FiConn keeps the merits of high bisection and easy wiring during the incremental deployment by adding shortcut links in incomplete FiConn. Of course, FiConn does not offer these appealing features with no cost. The wiring cost is higher compared to the current practice of tree. Besides, servers consume more CPU resources in packet forwarding in FiConn. However, this overhead is not an issue over time as more servers use multicore CPUs.

The rest of this paper is organized as follows. Section II introduces the related work. Section III describes the physical structure of FiConn and the basic routing on top of it. Section IV presents the traffic-aware routing protocol in FiConn. Section V tackles the problem of incremental deployment in FiConn. Section VI conducts simulations to evaluate the routing in FiConn and compares FiConn to other structures in detail. Section VII discusses more issues in FiConn design. Finally, Section VIII concludes this paper.

II. RELATED WORK

A. Interconnection Structure for Data Centers

We now discuss four interconnection structures proposed for data centers, the current practice of the tree-based structure, and two recent proposals of Fat-Tree [8], DCell [9], and BCube [10].

Tree: In current practice, servers are connected by a tree hierarchy of network switches, with commodity switches at the first level and increasingly larger and more expensive switches at the higher levels. It is well known that this kind of tree structure has many limitations [8], [9]. The top-level switches are the bandwidth bottleneck, and high-end high-speed switches have to be used. Moreover, a high-level switch shows as a single-point failure spot for its subtree branch. Using redundant switches does not fundamentally solve the problem, but incurs even higher cost.

Fat-Tree: Fig. 1 illustrates the topology of Fat-Tree solution, which has three levels of switches. There are \( n \) pods \((n = 4\) in the example), each containing two levels of \( n/2 \) switches, i.e., the edge level and the aggregation level. Each \( n \)-port switch at the edge level uses \( n/2 \) ports to connect the \( n/2 \) servers while using the remaining \( n/2 \) ports to connect the \( n/2 \) aggregation-level switches in the pod. At the core level, there are \((n/2)^2\) \( n \)-port switches, and each switch has one port connecting to one pod. Therefore, the total number of servers supported by the Fat-Tree structure is \( n^3/4 \). Given a typical \( n = 48 \) switch, the number of servers supported is 27,648.

FiConn differs from Fat-Tree in several aspects. First, FiConn puts the interconnection intelligence on servers, rather than on switches as in Fat-Tree. Second, there are three levels of switches in Fat-Tree, but only one lowest level in FiConn. Hence, the number of used switches is much smaller in FiConn. Consider the total number of servers as \( N \) and \( n \)-port switches being used. The number of switches needed in Fat-Tree is \( 5N/n \), while the number in FiConn is \( N/n \). Therefore, FiConn reduces the cost on switches by 80% compared to Fat-Tree. Third, the number of servers Fat-Tree supports is restricted by the number of switch ports, given the three layers of switches. FiConn does not have this limitation and extends to a very large number of servers, each of which has a node degree of 2.
Fat-Tree achieves nonblock communication between any pair of servers, while DCell and BCube have more ports on a server for routing selection. In fact, the lower networking capacity of FiConn results from the lesser number of links and switches, which is the tradeoff for low cost and easy wiring. However, the routing in FiConn we design makes a balanced use of different levels of links and is traffic-aware to better utilize the link capacities. We will compare these metrics in detail in Section VI.

B. Interconnection Structures in Other Areas

Besides in data centers, interconnection structures are widely studied in various areas such as parallel computing [15]–[17], on-chip network [14], and switching fabric [18]. Proposed structures include Ring [17], HyperCube [12], [13], Butterfly [16], Torus [17], De Bruijin [19], Flattened Butterfly [20], and DragonFly [21].

Among these structures, only Ring has the server node degree of 2, which is similar to FiConn. However, the diameter of Ring is $N/2$ and the bisection width is 2, where $N$ is the total number of nodes. Undoubtedly, Ring is not viable for server interconnection in data centers even when $N$ is very small, e.g., less than 100. As for the other structures, they are much more expensive to build a data center, and the wiring effort is also much higher compared to FiConn.

III. FiConn: A Novel Interconnection Structure for Data Centers

In this section, we present our FiConn physical structure and design the basic routing algorithm on top of FiConn.

A. Interconnection Rule

FiConn is a recursively defined structure. A high-level FiConn is constructed by many low-level FiConns. We denote a level-$k$ FiConn as FiConn$_k$. FiConn$_0$ is the basic construction unit, which is composed of $n$ servers and an $n$-port commodity switch connecting the $n$ servers. Typically, $n$ is an even number such as 16, 32, or 48. Every server in FiConn has one port connected to the switch in FiConn$_0$, and we call this port level-0 port. The link connecting a level-0 port and the switch is called level-0 link. Level-0 port can be regarded as the original operation port on servers in current practice. If the backup port of a server is not connected to another server, we call it an available backup port. For instance, there are initially $n$ servers each with an available backup port in a FiConn$_0$.

Now, we focus on how to construct FiConn$_k$ ($k > 0$) upon FiConn$_{k-1}$’s by interconnecting the server backup ports. If there are totally $b$ servers with available backup ports in a FiConn$_{k-1}$, the number of FiConn$_{k-1}$’s in a FiConn$_k$, $g_k$, is equal to $b/2 + 1$. In each FiConn$_{k-1}$, $b/2$ servers out of the $b$ servers with available backup ports are selected to connect the other $b/2$ FiConn$_{k-1}$’s using their backup ports, each for one FiConn$_{k-1}$. The $b/2$ selected servers are called level-$k$ servers, the backup ports of the level-$k$ servers are called level-$k$ ports, and the links connecting two level-$k$ ports are called level-$k$ links. If we take FiConn$_{k-1}$ as a virtual server, FiConn$_k$ is in fact a mesh over FiConn$_{k-1}$’s connected by level-$k$ links.

We can use a sequential number $t_k$ to identify a server $s$ in FiConn$_k$. Assume the total number of servers in a FiConn$_k$ is $N_k$.
and there is $0 \leq u_k < N_k$. Equivalently, $s$ can be identified by a $(k+1)$-tuple, $[q_k,...,q_1,q_0]$, where $q_0$ identifies $s$ in its FiConn$_0$, and $q_l$ ($1 \leq l \leq k$) identifies the FiConn$_{l-1}$ comprising $s$ in its FiConn$_l$. Obviously, there is $u_k \equiv a_0 + \sum_{l=1}^{k} (a_l \cdot N_{l-1})$. For ease of expression, $s$ can also be identified by $[a_k,u_{k-1},a_{k-1},u_{k-2},...]$, and etc.

Algorithm 1 shows the construction of a FiConn$_k$ $(k > 0)$ upon $g_k$ FiConn$_{k-1}$s. In each FiConn$_{k-1}$ (Line 2), the servers satisfying $(u_k-1 = 2^{k-1} + 1) \mod 2^k \equiv 0$ are selected as level-$k$ servers (Line 3), and they are interconnected as Lines 4–6 instruct.

**Algorithm 1: Constructing FiConn$_k$ upon $g_k$ FiConn$_{k-1}$s**

```java
01 FiConnConstruct($k$)
02 for ($i_2 = 0; i_2 < g_k; i_2++$)
03    ($j_1 = i_2 \cdot 2^k + 2^{k-1} - 1; j_1 < N_{k-1}; j_1 = j_1 + 2^k$)
04    ($j_2 = (j_1 - 2^{k-1} + 1)/2^k + 1$)
05    ($j_2 = i_2 \cdot 2^k + 2^{k-1} - 1$)
06    connect servers ($i_2$,$j_1$) with ($i_2$,$j_2$)
07 return
```

We take Fig. 4 as an example to illustrate the FiConn interconnection rule, in which $n = 4$ and $k = 2$. FiConn$_0$ is composed of four servers and a 4-port switch. The number of FiConn$_0$s to construct FiConn$_1$ is $4/2+1 = 3$. The servers [0,0], [0,2], [1,0], [1,2], [2,0], and [2,2] are selected as level-1 servers, and we connect [0,0] with [1,0], [0,2] with [2,0], and [1,2] with [2,2].

In each FiConn$_1$, there are six servers with available backup ports, so the number of FiConn$_1$s in a FiConn$_2$ is $6/2+1 = 4$. We connect the selected level-2 servers as follows: [0,0] with [1,0], [0,1] with [2,0], [0,2] with [2,0], [0,2] with [3,0], [1,1] with [2,1], [1,2,1] with [3,1,1], and [2,2,1] with [3,2,1].

**B. Basic Properties**

FiConn has several nice properties that we discuss here.

**Theorem 1:** If we denote the total number of servers in a FiConn$_k$ as $N_k$, there is $N_k \geq 2^{k+2} \cdot (n/4)^{2^k}$ (for $n > 4$), where $n$ is the number of servers in FiConn$_0$.

**Proof:** Based on the interconnection rule, a FiConn$_{k-1}$ has $N_{k-1}/2^{k-1}$ servers with available backup ports. When it is used to construct FiConn$_k$, half of the servers with available backup ports are selected as level-$k$ servers to connect other FiConn$_{k-1}$s. Hence, there is $g_k = N_{k-1}/2^k + 1$. We have

$$N_k = \begin{cases} n, & \text{if } k = 0 \\ N_{k-1} \cdot g_k = N_{k-1} \cdot (N_{k-1}/2^k + 1), & \text{if } k > 0, \end{cases}$$

We validate the correctness of Theorem 1.

1) If $k = 0$, there is $N_0 = 4 \cdot (n/4) = n$.
2) If $N_{k-1} \geq 2^{k+1} \cdot (n/4)^{2^k}$, then we have $N_k = N_{k-1} \cdot (N_{k-1}/2^k + 1) \geq N_{k-1}^2/2^k \geq 2^{2k+2} \cdot (n/4)^{2^k}/2^k = 2^{k+2} \cdot (n/4)^{2^k}$.

Fig. 5 illustrates the total number of servers in FiConn versus the level $k$. We use $\log_{10} N_k$ in y-axis. The figure shows clearly the linear relationship between $\log_{10} (\log_{10} N_k)$ and $k$, which implies that $N_k$ grows double-exponentially with $k$. For a typical value of $n = 48$ and $k = 2$, the number of servers in
FiConn is 361 200. If we choose \( n = 16 \) and \( k = 3 \), the number becomes 3 553 776.

**Theorem 2**: The average server node degree in FiConn is \( 2 - 1/2^k \).

**Proof**: Assume there are totally \( N_k \) servers in FiConn. All servers have one level-0 link. In addition, \( N_k/2^i \) servers have a level-\( i \) link (\( 1 \leq i \leq k \)). As a result, the average server node degree in FiConn is \( (N_k + \sum_{i=1}^{k} (N_k/2^i)) / N_k = 2 - 1/2^k \).

Theorem 2 tells that the average server node degree of FiConn approaches 2 when \( k \) grows, but never reaches 2. In other words, FiConn is always incomplete in the sense that there are always servers with available backup ports in it. In fact, it is just the incompleteness characteristic of FiConn that makes it highly scalable with the server node degree of 2.

**Theorem 3**: Suppose the number of FiConn’s in a FiConn is \( m \). The number of level-(\( k + 1 \)) links in the FiConn is \( (m \times (m - 1)) / 2 \).

**Proof**: This theorem naturally holds since there is a level-(\( k + 1 \)) link between each pair of FiConn’s based on the interconnection rule of FiConn.

**Theorem 4**: Suppose \( L_k \) denotes the number of level-\( i \) links in FiConn, there is

\[
L_k = \begin{cases} 
4 \times L_{k+1}, & \text{if } l = 0 \\
2 \times L_{k+1}, & \text{if } 0 < l < k. 
\end{cases}
\]

**Proof**: First, we prove \( L_0 = 4 \times L_1 \), and we only need to prove that it holds in a FiConn. Each server in a FiConn has one level-0 link, so there is \( L_0 = N_1 \). Half of the servers in FiConn are selected as level-1 servers, and every two level-1 servers share one level-1 link. Hence, we have \( L_1 = N_1 / 4 \). As a result, there is \( L_0 = 4 \times L_1 \).

Then, we prove for any \( 0 < l < k \), \( L_l = 2 \times L_{l+1} \). Again, we only need to prove that it holds in a FiConn. In a FiConn, the number of level-\( l \) servers is \( N_l / 2^l \), and the number of level-\( l \) links is thus \( N_l/2^{l+1} \). Hence, in FiConn, \( L_l = g_{l+1} \times N_l/2^{l+1} \). Similarly, the number of level-(\( l + 1 \)) links in FiConn is \( L_{l+1} = N_{l+1}/2^{l+2} \). Note that \( N_{l+1} = g_{l+1} \times N_l \), so we have \( L_l = 2 \times L_{l+1} \).

The relationship among the numbers of links in different levels disclosed in Theorem 4 matches the basic routing described in FiConn, which is in favor of making a balanced use of FiConn links. It will be further explained in Section IV.

**C. Traffic-Oblivious Routing**

We design a basic routing algorithm in FiConn that leverages the level-based characteristic of FiConn. To differentiate it with the Traffic-Aware Routing (TAR), which we design in Section IV, we call it Traffic-Oblivious Routing (TOR). The routing principle of TOR is that, for any pair of servers, if the lowest common level of FiConn they belong to is FiConn, the routing path between them is constrained to the two FiConn’s comprising the two servers, respectively, and the level-\( l \) link connecting the two FiConn’s is used. Hence, the routing path between two servers can be recursively calculated.

Algorithm 2 shows how TOR works on a server \( s \) to route a packet destined to \( dst \). The function TORouting() returns the next-hop server. First of all, the lowest common FiConn level of \( s \) and \( dst \) is found based on their identifiers, say, \( l \) (Line 2). If \( l \) is zero (Line 3), it means the destination server is within the same FiConn as \( s \), and the function returns \( dst \) (Line 4). Next, we get the level-\( l \) link connecting the two FiConn’s comprising \( s \) and \( dst \), respectively, say \( (i_1, i_2) \) (Line 5). If \( i_1 \) is itself (Line 6), then \( i_2 \) is returned (Line 7). Otherwise, we recursively compute and return the next-hop server from \( s \) toward \( i_1 \) (Line 8).

**Algorithm 2**: Traffic-oblivious routing (TOR) in FiConn

```plaintext
/* s: current server. dst: destination server of the packet to be routed. */
01 TORouting(s, dst)
02 l = lowestCommonLevel(s, dst)
03 if(l == 0)
04 return dst
05 (i_1, i_2) = getLink(s, dst, l)
06 if(i_1 == s)
07 return i_2
08 return TORoute(s, i_1)
09 }
```

Take Fig. 4 as an example. The path from source server [0,2,1] to destination server [1,2,1] using TOR is [(0,2,1), [0,2,0], [0,0,2], [0,0,1], [1,0,1], [1,0,2], [1,2,0], [1,2,1]), which takes seven hops.

From TOR, the number of level-\( l \) links (\( 0 < l < k \)) in a typical routing path in FiConn is twice that of level-(\( l + 1 \) ) links, and the number of level-0 links is four times that of level-1 links (note that one hop in FiConn includes two links since it crosses the switch). Meanwhile, Theorem 3 tells that in FiConn, the total number of level-\( l \) links (\( 0 < l < k \)) is twice that of level-(\( l + 1 \) ) links, and the number of level-0 links is four times that of level-1 links. Therefore, TOR makes a balanced use of different levels of FiConn links, which helps improve the aggregate throughput, especially in random traffic pattern.

Leveraging TOR, we can calculate the diameter and bisection width of FiConn.

**Theorem 5**: The upper bound of the diameter of FiConn is \( 2^{k+1} - 1 \).
Proof: Using the TOR, the longest routing path between any two servers in FiConn$_k$ takes one level-$k$ hop, two level-$(k-1)$ hops, ..., $2^{k-1}$ level-$l$ hops, ..., and $2^k$ level-0 hops. Hence, the upper bound of the diameter of FiConn$_k$ is $1 + 2 + \ldots + 2^k = 2^{k+1} - 1$.

In combination with Theorem 1, the diameter of FiConn is $O(\log N_k)$, where $N_k$ is the total number of servers in FiConn$_k$. Obviously, the diameter of FiConn is small considering the total number of servers, benefitting applications with real-time requirements.

We can also compute the bisection width of FiConn. In all-to-all communication, the number of flows on the FiConn$_k$ link that carries the most flows is about $2^k \times N_k$ times of that in its embedding complete graph. Based on [16], the lower bound of the bisection width of FiConn$_k$ is $1/(2^k \times N_k)$ times of that of the complete graph, that is, $(1/(2^k \times N_k)) \times (N_k^2/4) = N_k/(4 \times 2^k)$.

Considering Theorem 1, the bisection width of FiConn$_k$ is also $O(N_k/\log N_k)$. The high bisection width of FiConn implies that there are many possible paths between a pair of servers. FiConn is therefore intrinsically fault-tolerant, and it provides the possibility to design multipath routing on top of it.

IV. TRAFFIC-AWARE ROUTING IN FiConn

TOR balances the use of different levels of FiConn links and serves as the basis for FiConn routing. However, it has two limitations. First, a pair of servers cannot leverage the two ports on each to improve their end-to-end throughput in TOR. Second, TOR cannot further utilize the available link capacities according to dynamic traffic states to improve the networking throughput. To overcome these limitations, we design TAR in FiConn.

A. Basic Design and Challenges

Because of the large server population in data centers, we do not rely on central server(s) for traffic scheduling or exchange traffic states among all the FiConn servers. Instead, we seek to compute the routing path in a distributed manner with little control overhead.

We take a greedy approach to hop-by-hop setup of the traffic-aware path on each intermediate server. Each server seeks to balance the traffic volume between its two outgoing links. Specifically, the source server always selects the outgoing link with higher available bandwidth to forward the traffic. For a level-$l$ ($l > 0$) intermediate server, if the outgoing link using TOR is its level-$l$ link and the available bandwidth of its level-0 link is higher, its level-$l$ link is bypassed via randomly selecting a third FiConn$_{l-1}$ in the FiConn$_{l-1}$ to relay the traffic; otherwise, the traffic is routed by TOR.

When the level-$l$ server $s$ selects a third FiConn$_{l-1}$ for relay, a possible choice beyond the random selection is to exchange traffic states among all the level-$l$ servers within each FiConn$_{l-1}$, and $s$ can then choose the third FiConn$_{l-1}$ to which the level-$l$ link has the highest available bandwidth. However, we do not adopt this method because when $l$ is high, the number of level-$l$ servers in a FiConn$_{l-1}$ may be too large. It incurs considerable overhead to exchange traffic states with each other. One may argue that traffic states can be exchanged within a smaller range, such as FiConn$_k$ or FiConn$_{l-1}$. However, there may be few or no level-$l$ servers in such a range if $l$ is high, and the candidate third FiConn$_{l-1}$’s are consequently very limited. As a result, in our present design we let server $s$ randomly select a third FiConn$_{l-1}$ in the FiConn$_l$ for relay, which avoids traffic state exchange and retains a large candidate set of third FiConn$_{l-1}$’s.

Note that our idea of TAR can be readily extended to handle port/link faults, which may be common in large data centers. When a port or a link fails, it is treated the same as that the available bandwidth of the link becomes zero. The traffic will always be routed via the other link of the server. In this sense, port/link fault management is just an extreme case for TAR. The only modification is that, when a level-$l$ server $s$ receives traffic from its level-$l$ link but its level-0 link fails, $s$ routes the traffic back to its level-$l$ neighboring server to bypass the level-$l$ link as if the level-$l$ link fails.

To limit the control overhead, we do not compute the traffic-aware path on a per-packet basis. Instead, we target on a per-flow basis and dynamically setup the traffic-aware path for a flow using a special path-probing packet. When a flow is initiated on the source server, it is intercepted by the FiConn routing module of the source server, and a path-probing packet for the flow is sent out toward the destination server. Each intermediate server routes the path-probing packet based on local traffic states as stated and establishes the routing entry for the flow, which includes the previous hop and the next hop. When the destination server receives the path-probing packet, it responds by sending another path-probing packet back toward the source server, in which the source and destination fields are exchanged, and the return path is accordingly setup. After the source server receives the replied path-probing packet, it sends out the corresponding intercepted flow. Intermediate servers forward the flow based on established routing entries. During the session time of a flow, path-probing packets for the flow are periodically sent out to update the routing path based on dynamic traffic states.

We illustrate the basic design of TAR via the example of Fig. 6. There is already one flow in the level-1 link from [2,0] to [0,2], and all other links carry no traffic. Server [2,1] now initiates a flow toward server [0,1]. The path using TOR is [(2,1), [2,0], [0,2], [0,1]]. In TAR, when [2,0] receives the path-probing packet from [2,1], it discovers that its level-1 outgoing link to [0,2] has less available bandwidth than its level-0 outgoing link. It then randomly selects a third FiConn$_0$ in the FiConn$_1$ for relay. In this case, FiConn$_{0}[1]$ is selected. Finally, the packet is routed to [0,1] by the relay of FiConn$_0[1]$.

To make the above idea work, we need to address several challenges in TAR.

Routing Back: When an intermediate server chooses to bypass its level-$l$ ($l > 0$) link and routes the path-probing packet to a next-hop server in the same FiConn$_l$, the next-hop server may route the packet back using TOR. In the example of Fig. 6, when [2,2] receives the path-probing packet from [2,0], it routes the packet back to [2,0] using TOR unless otherwise specified.

Multiple Bypassing: When one level-$l$ ($l > 0$) link is bypassed, a third FiConn$_{l-1}$ is chosen as the relay, and two other level-$l$ links in the current FiConn$_l$ will be passed through. However, the two level-$l$ links may need to be bypassed again according to the basic design. It may iteratively occur, and routing
in the FiConn$_i$ thus takes too long a path or even falls into a loop. In the example of Fig. 6, assume the level-1 link from [2,2] to [1,2] should also be bypassed because there is a flow in it. Routing then gets trapped in a loop between [2,0] and [2,2]. Solution is needed to limit the bypassing times and avoid path loops.

Path Redundancy: A redundant path implies that there are intermediate servers to be removed from the path without reducing the throughput of the path. In the example of Fig. 6, [2,0] can be removed from the traffic-aware path, and thus [2,1] sends the packet to [2,2] directly.

Imbalance Trap: Assume that a level-$i$ server $s$ routes a flow via its level-$i$ outgoing link and there is no traffic in its level-0 outgoing link. All subsequent flows that arrive from its level-0 incoming link will bypass its level-$i$ link because the available bandwidth of its level-0 outgoing link is always higher. In this case, the outgoing bandwidth of its level-$i$ link cannot be well utilized even though the other level-$i$ links in the FiConn$_i$ are heavily loaded. In the example of Fig. 6, all subsequent flows from FiConn$_{[2]}$ to FiConn$_{[0]}$ will bypass the level-$i$ link of [2,0]. In fact, the problem results from the idea that TAR seeks to balance the local outgoing links of a server, not links among servers. We call it an imbalance trap problem, and corresponding mechanism is demanded.

In the following three subsections, we address the first two problems by Progressive Route (PR), the third problem by Source ReRoute (SRR), and the last problem by Virtual Flow (VF).

B. Progressive Route

Progressive Route (PR) solves both the routing back problem and the multiple bypassing problem by making the intermediate servers aware of the routing context. When the source server sends the path-probing packet, it adds a PR field in the packet header, and the PR field can be modified by intermediate servers. PR field has $m$ entries, where $m$ is the lowest common level of the source and destination servers. We use $\text{PR}_l (1 \leq l \leq m)$ to denote the $l$th entry of PR field. Each $\text{PR}_l$ plays two roles. First, when bypassing a level-$l$ link, the level-$l$ server in the selected third FiConn$_{-l}$ is chosen as the proxy server and is set in PR$_l$. Intermediate servers check the PR field and route the packet to the lowest-level proxy server. Hence, the path-probing packet will not be routed back. Second, PR$_l$ can carry information about the bypassing times in the current FiConn$_l$. If the number of bypassing times exceeds a threshold, the packet jumps out of the current FiConn$_l$ and chooses a third FiConn$_l$ for relay. One can see that the higher the threshold of bypassing times is, the more likely that the path-probing packet finds a balanced path. However, the tradeoff is the path length and probing time. In the present design, we set the threshold as 1, which means only one level-$l$ link can be bypassed in a FiConn$_l$.

Since the threshold of bypassing times is 1, we design two special identifiers different from server identifiers for a PR$_l$, BYZERO and BYONE. BYZERO indicates no level-$l$ link is bypassed in the current FiConn$_l$, so it is set in PR$_l$ when the packet is initialized or after crossing a level-$i$ link if $i > l$. BYONE means there is already one level-$l$ link bypassed in the current FiConn$_l$, and it is set in PR$_l$ after traversing the level-$l$ proxy server in the current FiConn$_l$. PR$_l$ is set as the identifier of the level-$l$ proxy server between the selection of the proxy server and the arrival to the proxy server.

Take Fig. 6 as the instance. The source server [2,1] initializes PR entries (in this case, $m = 1$) as BYZERO. When [2,0] selects [1,2] as the level-1 proxy server, it modifies PR$_1$ as [1,2] and sends the packet to [2,2]. [2,2] checks the PR field, finds [1,2] is the lowest-level proxy server, and sends the packet toward [1,2] (in this case, [1,2] is just its neighboring server). [1,2] receives the packet and finds PR$_1$ is the identifier of its own, so it modifies PR$_1$ as BYONE before sending it to the next hop [1,0]. Therefore, using PR, the traffic-aware path in this example is ([2,1], [2,0], [2,2], [1,2], [1,0], [0,0], [0,1]).

C. Source ReRoute

As aforementioned, the server [2,0] can be removed from the path using PR in the example above. We use Source ReRoute (SRR) to achieve this. When a server $s$ decides to bypass its level-$i$ ($i > 0$) link and chooses a proxy server, it modifies the PR field and then routes the path-probing packet back to the previous hop from which it received the packet. Then, the original intermediate servers from the source server to $s$ will all receive the path-probing packet from the next hop for the flow in the routing table, and they just send the packet to the previous hop for the flow in the routing table and clear the corresponding routing entry. After the source server receives the packet, it also clears the routing entry for the flow and reroutes the packet toward the lowest-level proxy server in PR field.

In the example above, when [2,0] selects [1,2] as the level-1 proxy server, it modifies PR$_1$ as [1,2] and sends the path-probing packet to the previous hop of this packet, [2,1]. [2,1] checks the routing table, finding that it receives the packet from the next hop of the flow it once routed to, which is an indication of SRR processing, but the previous hop of the flow is NULL, which implies that it is the source server. Therefore, [2,1] clears the corresponding routing entry, checks that PR$_1$ is [1,2], and then selects [2,2] as the next hop. In this way, [2,0]
is removed from the path, and the traffic-aware path becomes
\{(2,1), [2,2], [1,2], [1,0], [0,0], [0,1]\).

\section{Virtual Flow}

To alleviate the imbalance trap problem, we use Virtual Flow (VF) to compare the available bandwidth between two outgoing links. Virtual flows for a server \(s\) indicate the flows that once arrive at \(s\) from its level-0 link but are not routed by \(s\) because of bypassing (\(s\) is removed from the path by SRR). Each server initiates a virtual flow counter (VFC) as zero. When a flow bypasses its level-1 link, VFC is added by one. When a flow is routed by its level-0 outgoing link, VFC is reduced by one given it is a positive value. When evaluating the available bandwidth of an outgoing link, not only the current routed flows are counted, but the virtual flows for the level-0 link are also considered. The traffic volume of a virtual flow is set as the average traffic volume of routed flows. In this way, the imbalance trap problem is overcome.

\section{Algorithm}

Taking the solutions above together, we design the algorithm of TAR in FiConn, as illustrated in Algorithm 3. The function TARoute() returns the next-hop server when a level-1 server \(s\) routes the path-probing packet \(pkt\).

\begin{algorithm}[h]
\caption{Traffic-aware routing (TAR) in FiConn}
\begin{algorithmic}[1]
\Statex / * \(s\): current server,
\(l\): the level of \(s\). (\(l > 0\))
\(RT\)table: the routing table of \(s\), maintaining the previous hop \((prev\)hop\) and next hop \((next\)hop\) for a flow.
\(hb\): the available bandwidth of the level-1 link of \(s\).
\(zb\): the available bandwidth of the level-0 link of \(s\).
\(hn\): the level-1 neighboring server of \(s\).
\(vfc\): virtual flow counter of \(s\).
\(pkt\): the path-probing packet to be routed, including flow id \((flow)\), source \((src)\), destination \((dst)\), previous hop \((phop)\), and PR field \((pr)\).
\Statex */
\Function{TARoute}{(s, pkt)}
\If{\((\text{pkt}.\text{dst} == s)\)}%This the destination/
\State return \(\text{NULL} / \text{Deliver pkt to upper layer}/*
\EndIf
\If{\((\text{pkt}.\text{phop} == \text{RT}\text{table}[\text{pkt}.\text{flow}].\text{nexthop})\)}%SRR of pkt to upper layer/
\State \(\text{hnop} = \text{RT}\text{table}[\text{pkt}.\text{flow}].\text{prevhop}\)
\State \(\text{RT}\text{table}[\text{pkt}.\text{flow}] = \text{NULL}\)
\If{\((\text{hnop} == \text{NULL})\)}%This is not source server/
\State return \(\text{hnop}\)
\EndIf
\If{\((s == \text{pkt}.\text{pr}[l])\)}%This is the proxy server/
\State \(\text{hnop} = \text{BY ONE}\)
\EndIf
\State \(\text{ldst} = \text{getPRDest}(\text{pkt})%\text{Check PR for proxy servers}/\)
\State \(\text{hnop} = \text{TARoute}(\text{s}, \text{ldst})\)
\If{\((s == \text{pkt}.\text{src} \text{and} \text{hnop} \neq \text{hn} \text{and} hb > zb)\)}
\State \(\text{hnop} = \text{hn}\)
\EndIf
\If{\((\text{pkt}.\text{phop} == \text{hn} \text{and} \text{hnop} \neq \text{hn})\)}%or \((\text{pkt}.\text{phop} \neq \text{hn} \text{and} hb \geq zb)\)
\State return \(\text{hnop}\)
\EndIf
\State \(\text{RT}\text{table}[\text{pkt}.\text{flow}] = (\text{pkt}.\text{phop}, \text{hnop})\)
\EndFunction
\end{algorithmic}
\end{algorithm}

Lines 2–3 handle the case when the path-probing packet arrives at the destination server \(s\). The packet is delivered to the upper layer.

Lines 4–8 are the SRR processing. If \(s\) once routed the path-probing packet and now receives the packet from the next hop of the flow in the routing table (Line 4), it is an indication that this is the SRR processing. \(s\) then gets the original previous hop of the flow (Line 5) and erases the routing entry (Line 6). If \(s\) is not the source server for the flow (Line 7), it just routes the path-probing packet to the original previous hop (Line 8).

Lines 9–10 are for the case when \(s\) is the level-1 proxy server in the current FiConn (Line 9). It modifies PRs as \(\text{BY ONE}\).

Lines 11–12 get the next hop by TOR. First, we find the next destination server (Line 11). The function getPRDest() returns the lowest-level proxy server in PR field of the packet; if there is no proxy server, it returns the destination server of the packet. Then, we compute the next hop toward the next destination server using TOR (Line 12).

Lines 13–14 process the special case for source server to compute the next hop. The difference for a source server from other intermediate servers is that if the next hop using TOR is within the same FiConn but the available bandwidth of its level-1 link is higher than that of its level-0 link (Line 13), its level-1 neighboring server is selected as the next hop (Line 14). Note that virtual flows are considered to compute the available bandwidth.

Lines 15–20 are responsible for the cases that do not need to bypass the level-1 link. The first case is that the previous hop is the level-1 neighboring server and the next hop is not the same. Note that the next hop based on TOR may be the same as the previous hop if the previous hop is the source server. The second case is that the previous hop is from the same FiConn but the available bandwidth of its level-1 link is not less than that of the level-0 link. Line 15 makes the judgement. Lines 16–17 reduce \(vfc\) by one if this flow is to be routed by level-0 link. Before returning the next hop (line 20), \(s\) resets the PR field (line 21) and updates the routing table. The function resetPR() resets all PRs \((i < l)\) as \(\text{BY ZERO}\).

Lines 21–29 deal with how to bypass the level-1 link. The function bypassLink() in Line 23 finds a proxy server to bypass the level-1 link of \(s\), updates the PR field, and returns the next hop toward the proxy server. However, if it cannot find a proxy server, it returns NULL. Therefore, if bypassLink() returns NULL (Line 24), level-1 link is not bypassed (Line 25–27);
otherwise, the level-$l$ link is bypassed, and the packet is sent to the previous hop of the flow for SRR processing (Line 29), before which $ufc$ is added by one.

Based on the algorithm of TAR described, we can compute the maximum length of routing path in TAR.

**Theorem 7:** In TAR, the maximum length of routing path between any two servers in FiConn$_k$ is $2 \times 3^k - 1$.

**Proof:** Assume the maximum length of a routing path between two servers in a FiConn$_k$ based on TAR as $M_k$. The longest TAR path between two servers in a FiConn$_{k+1}$ traverses three FiConn$_{k+1}$’s and two level-$k$ links between them. Hence, there is $M_{k+1} = 3 \times M_k + 2$, and $M_0 = 1$. As a result, we have $M_k = 2 \times 3^k - 1$.

V. INCREMENTAL DEPLOYMENT OF FiCONN

In practice, it is much likely that the total number of servers we need in FiConn does not exactly meet the number of servers in a certain FiConn$_k$. Instead, the number is between FiConn$_k$ and FiConn$_{k+1}$, we call such a structure an incomplete FiConn.

For incremental deployment, we need to address the interconnection in incomplete FiConn. Our principle is that the interconnection should not only retain high bisection width in incomplete FiConn, but also incur low rewiring cost.

In the construction of complete FiConn, we use a bottom-up approach as in Algorithm 1. In this way, we first deploy a complete FiConn$_1$, then a complete FiConn$_2$, and so forth. One problem of this approach is that it may generate incomplete FiConn with low bisection width. For example, if there are only two FiConn$_{k-1}$’s in an incomplete FiConn$_k$, the two FiConn$_{k-1}$’s will be connected by a single level-$k$ link. The bisection width of this structure becomes 1, and the single level-$k$ link is the communication backbone in the structure.

In fact, the construction of DCell faces the same problem. Dcell uses a top-down approach to solve it [9]. When incrementally constructing a DCell$_k$, Dcell starts from building many incomplete DCell$_{k-1}$ and makes them fully connected. In other words, DCell first connects higher-level links, and then lower-level links. The basic construction unit is DCell$_1$. In this way, an incomplete DCell$_k$ still has high bisection width. However, this method used in DCell is not readily applicable in FiConn. One difference of FiConn from DCell is that the higher-level links in FiConn are fewer than lower-level ones. Hence, the connectivity in top-down approach is sparser than bottom-up approach. For instance, assume there are $m$ level-2 servers in a FiConn$_1$ and we have $m + 1$ FiConn$_1$’s to connect. If we use the bottom-up approach and put these FiConn$_1$’s in a complete FiConn$_2$, the links connecting the FiConn$_1$’s are level-2 links, and each FiConn$_1$ has $m$ direct links to other FiConn$_1$’s. However, if we use the top-down approach and put these FiConn$_1$’s in $m + 1$ different FiConn$_2$’s, the links connecting the FiConn$_1$’s are level-3 links. According to Theorem 4, the number of level-3 links in a FiConn$_1$ is $m/2$. Hence, each FiConn$_1$’s has at most $m/2$ direct links to other FiConn$_1$’s, which is half that of the bottom-up approach. If the levels of FiConn we plan to deploy are higher, the connectivity between these FiConn$_1$’s will be even sparser.

**Our Approach:** Because of the reason presented, we do not use the top-down approach to solve the incremental deployment problem in FiConn. Instead, we still build FiConn in a bottom-up way, but add level-$k$ shortcut links to increase the bisection width in an incomplete FiConn$_k$. Assume there are $m$ deployed FiConn$_{k-1}$’s in an incomplete FiConn$_k$, and the number of FiConn$_{k-1}$’s in a complete FiConn$_k$ is $m'$. Thus, the number of undeployed FiConn$_{k-1}$’s in the complete FiConn$_k$ is $m' - m$. For a certain undeployed FiConn$_{k-1}$, each of the $m$ deployed FiConn$_{k-1}$’s has a level-$k$ server connecting toward it in the complete FiConn$_k$. For $m$ such level-$k$ servers, we add a level-$k$ shortcut link between a pair of them (if $m$ is an odd number, there will be one server left). In this way, we add $(m' - m) \times \lceil m/2 \rceil$ level-$k$ shortcut links. The total number of level-$k$ links in the incomplete FiConn$_k$ is $(m \times (m - 1))/2 + (m' - m) \times \lceil m/2 \rceil$. By adding shortcut links, we greatly increase the bisection width of an incomplete FiConn$_k$.

**Fig. 7:** Shows how to connect two FiConn$_1$’s in an incomplete FiConn$_2$. The incomplete FiConn$_2$ is the one in Fig. 4. Based on the bottom-up construction rule, there is only one level-2 link between the two FiConn$_1$’s, connecting server [0,0,1] and server [1,0,1]. For undeployed FiConn$_1$’s, server [0,1,1] and server [1,1,1] each has a level-2 link connecting it in the complete so the two servers are connected via a level-2 shortcut link. Similarly, for undeployed FiConn$_1$’s, server [0,2,1] and server [1,2,1] each has a level-2 link toward it, and they are also connected by a shortcut link.

**Wiring Cost During Incremental Deployment:** Still assume there are $m$ deployed FiConn$_{k-1}$’s in an incomplete FiConn$_k$, and the number of FiConn$_{k-1}$’s in a complete FiConn$_k$ is $m'$. Now, we add a new FiConn$_{k-1}$ into the incomplete FiConn$_k$. The wiring is as follows. First, for each of the $\lceil m/2 \rceil$ level-$k$ shortcut links that connect the level-$k$ servers that should connect the newly added FiConn$_{k-1}$, we unplug one end and connect it to the new FiConn$_{k-1}$. Second, $\lceil m/2 \rceil$ level-$k$ links are added to connect the new FiConn$_{k-1}$ and each remaining de-
ployed FiConn$_{k=1}$. Third, if $m$ is an odd number, $m' = m - 1$ level-$k$ shortcut links are added, each connecting the new FiConn$_{k=1}$ and a corresponding deployed FiConn$_{k=1}$, based on the shortcut link addition rule.

Still take Fig. 7 as an example. We have $m = 2$ and $m' = 4$. Now, we will add FiConn$_{1}$ into the incomplete FiConn$_{2}$. First, we unplug the level-2 shortcut link on server [1,1,1] and plug it on server [2,0,1]. Second, we add a new level-2 link connecting server [2,1,1] and server [1,1,1]. Third, since $m$ is an even number, we do not need to add any new shortcut links.

We find that the shortcut links we add in an incomplete FiConn not only increase the bisection width of the incomplete FiConn, but are also fully utilized during the process of incremental deployment and easy to rewire.

Routing: We briefly discuss the routing in an incomplete FiConn. TOR is the same as that in complete FiConn since the basic interconnection rule is not broken.

In TAR, the shortcut links are used just for bypassing. If a level-$k$ server with a shortcut link in FiConn$_{k}$ decides to bypass the level-0 link by high-level link based on Algorithm 3, it simply forwards the packet via the shortcut link. However, there are also cases that the level-$k$ server has no level-$k$ link, in which case the level-$k$ server simply does not bypass the level-0 link.

VI. EVALUATION

We evaluate FiConn on two sides. First, we do simulations to study the effectiveness of the routing algorithm we design. Second, we make a detailed comparison of FiConn with other data-center networking structures in different metrics.

A. TAR versus TOR

We run the simulation on a FiConn$_{2}$ with $n = 32$, thus there are in total $N = 74,528$ servers. All servers have two Gigabit NIC ports. The switch hardware can support full-speed Gigabit switching among all ports. We make a simple assumption that each server hops has the same propagation delay and all flows are TCP-like elastic flows.

Two types of traffic patterns are considered. One is random traffic, and the other is burst traffic between two subsets of FiConn servers produced by computation models such as map-reduce. For the random traffic, we randomly choose $N/2$ pairs of servers from all the servers, and there is one flow between each pair. Thus, there are altogether 37,264 flows in the network. For the burst traffic, we randomly choose two FiConn$_{1}$’s. For every server in one FiConn$_{1}$, there is a flow from it to every server in the other FiConn$_{1}$. Hence, there are totally 295,936 flows in the network. All the flows are initiated sequentially in the first 30 s, and the path-probing packet in TAR is sent every 30 s for a flow. We compute the aggregate throughput and average path length of TAR and TOR, respectively.

Random Traffic: Figs. 8 and 9 illustrate the aggregate throughput and the average path length, respectively, for random traffic.

From Fig. 8, we see that the aggregate throughputs of TAR and TOR are very close. At the end of the first 30 s, the throughput of TOR is about 8.5% higher than that of TAR. However, after several rounds of dynamic adjustment, the difference between them is within 2.5%. The slight advance of TOR comes from its shorter routing path, which benefits improving the aggregate throughput when traffic is randomly distributed.

Fig. 9 shows that the average path length of TAR is always more than that of TOR, but within 1.5 hops in steady state. In combination with Fig. 8, we also find that TAR can dynamically adapt to traffic states and improve the throughput as well as reduce the path length.

Burst Traffic: Figs. 10 and 11 show the aggregate throughput and the average path length, respectively, for burst traffic.

From Fig. 10, we find that the aggregate throughput of TOR is only 1 Gb/s, resulting from the bottleneck level-2 link that connects the two selected FiConn$_{1}$’s. However, by exploiting the links beyond the two FiConn$_{1}$’s and the bottleneck level-2 link, TAR achieves an aggregate throughput of 99.5 Gb/s, which shows a tremendous improvement over TOR.

The result of Fig. 11 also tells that the average path length of TAR is longer than that of TOR, but the difference is within three hops.

Taking the two groups of simulations together, we draw the following conclusions. First, our TAR can adapt to dynamical networking conditions to improve the throughput as well as reduce the routing path length. Second, the average path length in TAR is always more than that in TOR, but the difference is
no more than one to three hops in the FiConn. Third, the aggregate throughput of TAR is quite similar to TOR in uniform traffic, but much higher than TOR in burst traffic that is common in data centers. In other words, the TAR can indeed well exploit the link capacities of FiConn to improve the networking throughput. Considering the little control overhead, our TAR is especially suitable for FiConn.

### B. FiConn versus Other Structures

There are recently several other novel interconnection structures for data centers proposed, as we presented in Section II. Here, we make a more detailed comparison of FiConn with DCell, Fat-Tree, and BCube, as illustrated in Table I.

For fair comparison, we assume the four structures support the same number of servers ($N$) in data centers and use the same type of switches ($n$ ports). FiConn, DCell, and BCube are all recursively defined structures, and we denote the levels of them as $k$, $k'$, and $k''$, respectively. Typically, there is $k'' \geq k \geq k'$. First, we study the scalability of the four structures. All these structures can scale to large server population. However, only FiConn can scale without adding server ports or switch ports. On the other hand, DCell and BCube scale with more server ports, while the scalability of Fat-Tree is limited by the number of switch ports given three layers of switches. Hence, FiConn can scale to large data center with less deploying efforts than the other three structures.

Second, we consider the number of switches and links required to support a certain number of servers. These numbers not only determine the cost of data-center networking, but also reflect the wiring efforts in deploying data center. We can find that FiConn and DCell both use the same number of switches, only one-fifth the number in Fat-Tree. The number of switches used in BCube depends on the number of levels, but typically more than FiConn and DCell, while less than Fat-Tree. In addition, FiConn beats all other three structures in the number of links. As a result, FiConn has the lowest deploying cost and wiring effort among all these structures.

Third, we analyze the one-to-one throughput and all-to-all throughput, respectively, both of which are important performance metrics in data-center networking. Assume the link bandwidth is $1$. The maximum one-to-one throughput is just the number of server ports. FiConn is twice that of Fat-Tree, but less than DCell and BCube (note that $k'$ and $k''$ are typically larger than 2). For all-to-all throughput, Fat-Tree and BCube perform best, and FiConn is a little worse than DCell. This is not strange because FiConn uses less switches, less links, or less ports than the other three structures. However, in a large-scale data center, it is unlikely that all servers participate in all-to-all communication. Therefore, we argue that the merit of FiConn on low cost and easy deployment outweighs the downside on all-to-all throughput.

### VII. Discussions

In this section, we make more discussions on the design of FiConn.

**Packaging and Wiring:** The packaging of servers and switches in FiConn depends on the number of switch ports, $n$. Take a typical value of $n = 48$ as an example. We can put the 48 servers under the same switch as well as the switch itself into a rack. The intrack wiring is easy to complete. The FiConn has 25 racks, which can be put into two columns. The level-1 links between servers are paced on the top of racks. It is relatively challenging for wiring when deploying FiConn since a complete FiConn with $n = 48$ contains 361 200 servers. Long-distance fibers can be used in this case to solve the problem. Also note that the number of links required in FiConn is much less than DCell, Fat-Tree, and BCube, as elaborated in Section VI.

**Extension to More Server Ports:** In FiConn, we assume that all servers have two built-in NIC ports. However, the innovation on server hardware is fast, and servers with four embedded NIC ports are typically used in data centers and use the same type of switches ($n$ ports). FiConn, DCell, and BCube are all recursively defined structures, and we denote the levels of them as $k$, $k'$, and $k''$, respectively. Typically, there is $k'' \geq k \geq k'$.
ports are emerging. In fact, the basic idea used in FiConn can be easily extended to embrace any constant number of server ports. The principle is that when we conduct level-$L$ interconnection, we do not let every server in the level-$L$ contribute a port, but reserve some for future higher-level interconnection. There can be many specific ways for interconnecting servers with constant degree of more than 2, and we put the investigation as our future work.

**Locality-Aware Task Placement:** In common data centers, a server is likely to communicate with a small subset of other servers. Typical applications include group communication in distributed database, VM migration across a certain set of physical servers, as well as file chunk replication among a few number of servers. We can use a locality-aware mechanism when placing these tasks onto data-center servers in FiConn. More specifically, several or tens of servers with intensive data exchange can be placed onto servers from a FiConn$_0$, which are connected to the same switch. There is only one server hop between these servers. A FiConn$_1$ is usually sufficient to contain hundreds of servers (with $n=16$ as an example), where the number of server hops is at most three. This kind of locality-aware task placement can largely reduce communication latency within the task and also save network bandwidth.

**VIII. Conclusion**

In this paper, we propose FiConn, a novel server-interconnection network structure that utilizes the dual-port configuration existing in most commodity data-center server machines. It is a highly scalable structure because the total number of servers it can support is not limited by the number of server ports or switch ports. It is cost-effective because it requires less number of switches and links than other recently proposed structures for data centers. We have designed traffic-aware routing in FiConn to make better utilization of the link capacities according to traffic states. We also have proposed solutions to increase the bisection width in incomplete FiConn$_1$s during incremental deployment.

**REFERENCES**


**Dan Li** (M’10) received the Ph.D. degree in computer science from Tsinghua University, Beijing, China, in 2007.

He is now an Assistant Professor with the Computer Science Department, Tsinghua University. Before joining the faculty of Tsinghua University, he spent two years as an Associate Researcher with the Wireless and Networking Group, Microsoft Research Asia, Beijing, China. His research interest spans from Internet architecture and protocols, P2P networks, to cloud computing networks.

**Chuanxiong Guo** received the Ph.D. degree in communications and information systems from Nanjing Institute of Communications Engineering, Nanjing, China, in 2000.

He is now a Lead Researcher with the Wireless and Networking Group, Microsoft Research Asia, Beijing, China. His research interests lie in the field of networking, encompassing network algorithm design and analysis, data center networking (DCN), novel network applications, and networking support in operating systems.

**Haitao Wu** (M’03) received the Bachelor’s degree in telecommunications engineering and the Ph.D. degree in telecommunication and information systems from Beijing University of Posts and Telecommunications (BUPT), Beijing, China, in 1998 and 2003, respectively.

He joined the Wireless and Networking Group, Microsoft Research Asia (MSRA), Beijing, China, in 2003. His research interests span from Internet architecture and protocols, TCP/IP, P2P, and wireless networks.
Kun Tan (M’03) received the B.E., M.E., and Ph.D. degrees in computer science and engineering from Tsinghua University, Beijing, China, in 1997, 1999, and 2002, respectively.

He joined Microsoft Research Asia, Beijing, China, after his graduation, and he is now a Researcher in the Wireless and Networking Group. He has filed over 30 pending patents and seven granted patents after he joined Microsoft. His research interests include transport protocols, congestion control, delay-tolerant networking, and wireless networks and systems.

Dr. Tan is a Member of the Association for Computing Machinery (ACM).

Yongguang Zhang received the Ph.D. degree in computer science from Purdue University, West Lafayette, IN, in 1994.

He is a Senior Researcher and Research Manager with the Wireless and Networking Research Group, Microsoft Research Asia, Beijing, China. Previously from 1994 to 2006, he was a Senior Research Scientist with HRL Labs, Malibu, CA. He was also an Adjunct Assistant Professor of computer science with the University of Texas at Austin from 2001 to 2003. He has published over 50 technical papers and one book, including top conferences and journals of his fields (like ACM SIGCOMM, MobiCom, MobiSys, the IEEE/ACM TRANSACTIONS ON NETWORKING).

Dr. Zhang recently won the Best Paper Award at NSDI 2009 and four best demo awards in a row: at MobiSys 2007, at SenSys 2007, again at MobiSys 2008, and at NSDI 2009. He is an Associate Editor for the IEEE TRANSACTIONS ON MOBILE COMPUTING, and an Area Editor for Mobile Computing and Communications Review. He was a Guest Editor for Mobile Networks and Applications. He has organized and chaired/co-chaired several international conferences, workshops, and an IETF working group. He was a General Co-Chair for ACM MobiCom 2009.

Songwu Lu received the Ph.D. degree in electrical engineering from the University of Illinois at Urbana-Champaign in 1999.

He is now a Full Professor with the Computer Science Department, University of California, Los Angeles. His research interests include wireless networks, data-center networking, mobile systems, and wireless network and Internet security.

Jianping Wu (SM’05) received the M.S. and Ph.D. degrees in computer science from Tsinghua University, Beijing, China, in 1997.

He is now a Full Professor with the Computer Science Department, Tsinghua University. In the research areas of the network architecture, high-performance routing and switching, protocol testing, and formal methods, he has published more than 200 technical papers in academic journals and proceedings of international conferences.