# FiConn: Using Backup Port for Server Interconnection in Data Centers

Dan Li*, Chuanxiong Guo*, Haitao Wu*, Kun Tan*, Yongguang Zhang*, Songwu Lu†

*Microsoft Research, Asia, †University of California, Los Angeles

***Abstract*— The goal of data center networking is to interconnect a large number of server machines with low equipment cost, high and balanced network capacity, and robustness to link/server faults. It is well understood that, the current practice where servers are connected by a tree hierarchy of network switches cannot meet these requirements [8], [9].**

**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 purpose. We believe that, if both ports are actively used in network connections, we can build a low-cost interconnection structure without the expensive higher-level large switches. Our new network design, called FiConn, utilizes both ports and only the low-end commodity switches to form a scalable and highly effective structure.**

**Although the server node degree is only two 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. The routing mechanism in FiConn balances different levels of links. We have further developed a low-overhead traffic-aware routing mechanism to improve effective link utilization based on dynamic traffic state. Simulation results have demonstrated that the routing mechanisms indeed achieve high networking throughput.**

## 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], [2], [3] at a data center, with low equipment cost, high and balanced network capacity, and robustness to link/sever 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-base solution 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, multi-port commodity switches? If we can solve the problem, the potential benefits are multi-faceted. First, building a data center network becomes relatively easy. 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, 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 testbed 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. Finally, data center technology may become pervasive in campus, small- to medium-sized enterprise, and big companies.

Neither current practice nor recent proposals [8], [9] 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] typically requires more ports per server, e.g., 4, 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 two in order to scale.

In this paper, 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 FiConns. When constructing a higher-level FiConn, the lower-level FiConns 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 $O(logN)$, which is small and can thus support applications with real-time requirements. The bisection width of FiConn is $O(N/logN)$, 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.

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. Simulation results show that our traffic-aware routing achieves much higher throughput for burst traffic between two subsets of FiConn severs, which is common for data center applications such as Map-Reduce.

In summary, we make two main contributions in FiConn. First, FiConn offers a novel network structure that is scalable with servers of node-degree two and has low diameter and high bisection width. FiConn places more intelligence into end-servers while minimizing the complexity at the switch. Therefore, FiConn is able to scale with commodity switches and off-the-shelf servers. 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. Of course, FiConn offers these appealing features at additional overhead compared with the tree structure in current practice. Wiring cost is higher since each server has only two Ethernet ports. Besides, servers consume more CPU resources in packet forwarding in FiConn. This overhead is not an issue over time as more servers use multi-core 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 uses simulations to evaluate the routing in FiConn. Finally, Section VI concludes this paper.

## II. Related Work

### A. Interconnection Structure for Data Centers

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

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.

Figure 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 uses 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. There are three levels of switches in Fat-Tree, but only one lowest-level in FiConn. Hence, the number of used switches are 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 with Fat-Tree. Second, the number of servers Fat-Tree supports is restricted by the number of switch ports. FiConn does not have this limitation and extends to a very large number of servers, each of which has a node degree of two. Third, Fat-Tree depends on central server(s) for traffic scheduling, but traffic-aware routing in FiConn computes the routing path in a distributed manner with little control overhead.

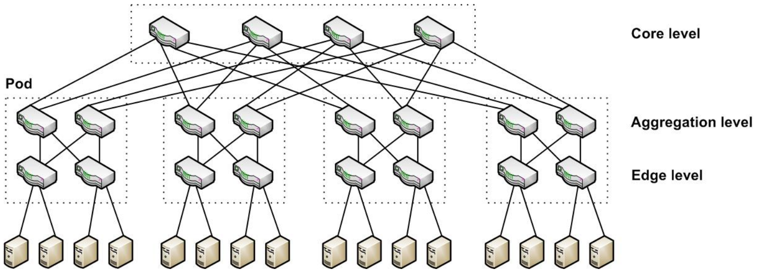

Fig. 1. A Fat-Tree structure with $n = 4$. It has three levels of switches.

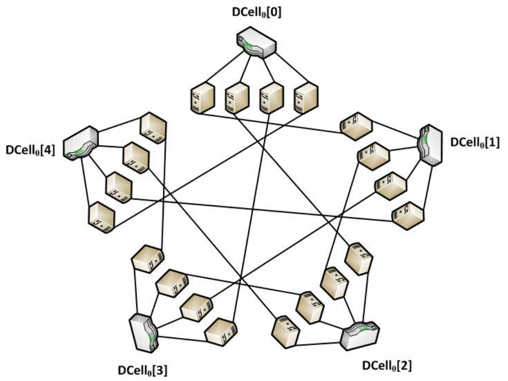

Fig. 2. A $\text{DCell}_1$ structure with $n = 4$. It is composed of 5 $\text{DCell}_0$s.

DCell is a new, level-based structure [8] as illustrated in Figure 2. In $\text{DCell}_0$, $n$ servers are connected to a $n$-port commodity switch. Given $t$ servers in a $\text{DCell}_k$, $t+1$ $\text{DCell}_k$s are used to build a $\text{DCell}_{k+1}$. The $t$ servers in a $\text{DCell}_k$ connect to the other $t$ $\text{DCell}_k$s, respectively. This way, DCell achieves high scalability and high bisection width.

FiConn and DCell share the same design principle to place the interconnection intelligence onto servers. They are different in several aspects. First, the server node degree in a $\text{DCell}_k$ is $k + 1$, but that of FiConn is always two. As a result, FiConn just needs to use the existing backup port on each server for interconnection, and no other hardware cost is introduced on a server. Second, the wiring cost in FiConn is less than that of DCell because each server uses only two ports. Third, routing in FiConn makes a balanced use of links at different levels, which DCell cannot. Finally, traffic-aware routing in FiConn is further designed to exploit the link capacities according to current traffic state.

One downside of FiConn compared with Fat-tree and DCell is that FiConn has lower aggregate networking capacity. Fat-Tree achieves non-block communication between any pair of servers [8], and DCell has more ports on a server for routing selection. In fact, the lower networking capacity of FiConn results from the less number of links, which is the tradeoff of easy wiring. Moreover, routing in FiConn makes a balanced use of different levels of links, and is traffic-aware to better utilize the link capacities.

### B. Interconnection Structures in Other Areas

Besides in data centers, interconnection structures are widely studied in various areas such as parallel computing [14], [15], [16], on-chip network [13], and switching fabric [17]. Proposed structures include Ring [16], HyperCube [11], [12], Butterfly [15], Torus [16], De Bruijin [18], Flattened Butterfly [19] and DragonFly [20].

Among these structures, only Ring has the server node degree of two, 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 with 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. Physical Structure

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, $u_k$, to identify a server $s$ in FiConn$_k$. Assume the total number of servers in a FiConn$_k$ is $N_k$, there is $0 \le u_k < N_k$. Equivalently, $s$ can be identified by a $(k+1)$-tuple, $[a_k, ..., a_1, a_0]$, where $a_0$ identifies $s$ in its FiConn$_0$, and $a_l$ ($1 \le l \le k$) identifies the FiConn$_{l-1}$ comprising $s$ in its FiConn$_l$. Obviously, there is $u_k = a_0 + \sum_{l=1}^{k}(a_l * N_{l-1})$. For ease of expression, $s$ can also be identified by $[a_k, u_{k-1}]$, $[a_k, a_{k-1}, u_{k-2}]$, 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 $(u_{k-1} - 2^{k-1} + 1)\ mod\ 2^k == 0$ are selected as level-$k$ servers (Line 3), and they are interconnected as Lines 4-6 instruct.

```
01 FiConnConstruct($k$){
02   for($i_1 = 0; i_1 < g_k; i_1 + +$)
03     for($j_1 = i_1 * 2^k + 2^{k-1} - 1; j_1 < N_{k-1}; j_1 = j_1 + 2^k$)
04       $i_2 = (j_1 - 2^{k-1} + 1)/2^k + 1$
05       $j_2 = i_1 * 2^k + 2^{k-1} - 1$
06       connect servers $[i_1, j_1]$ with $[i_2, j_2]$
07   return
08 }
```

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

We take Fig.3 as an example to illustrate the FiConn interconnection rule, in which $n = 4$ and $k = 2$. FiConn$_0$ is composed of 4 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 6 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, $[0,0,1]$ with $[1,0,1]$, $[0,1,1]$ with $[2,0,1]$, $[0,2,1]$ with $[3,0,1]$, $[1,1,1]$ with $[2,1,1]$, $[1,2,1]$ with $[3,1,1]$, and $[2,2,1]$ with $[3,2,1]$.

FiConn has several nice properties which we discuss as follows.

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

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

We validate the correctness of Theorem 1.

i) If $k = 0$, there is $N_0 = 4 * (n/4) = n$.

ii) If $N_{k-1} \ge 2^{k+1} * (n/4)^{2^{k-1}}$, then we have $N_k = N_{k-1} * (N_{k-1}/2^k + 1) \ge N_{k-1}^2/2^k \ge 2^{2k+2} * (n/4)^{2^k}/2^k = 2^{k+2} * (n/4)^{2^k}$. ■

Fig.4 illustrates the total number of servers in FiConn versus the level $k$. We use $log_{10}(log_{10}N_k)$ 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$_k$ is $2 - 1/2^k$.

*Proof:* Assume there are totally $N_k$ servers in FiConn$_k$. All servers have one level-0 link. In addition, $N_k/2^i$ servers

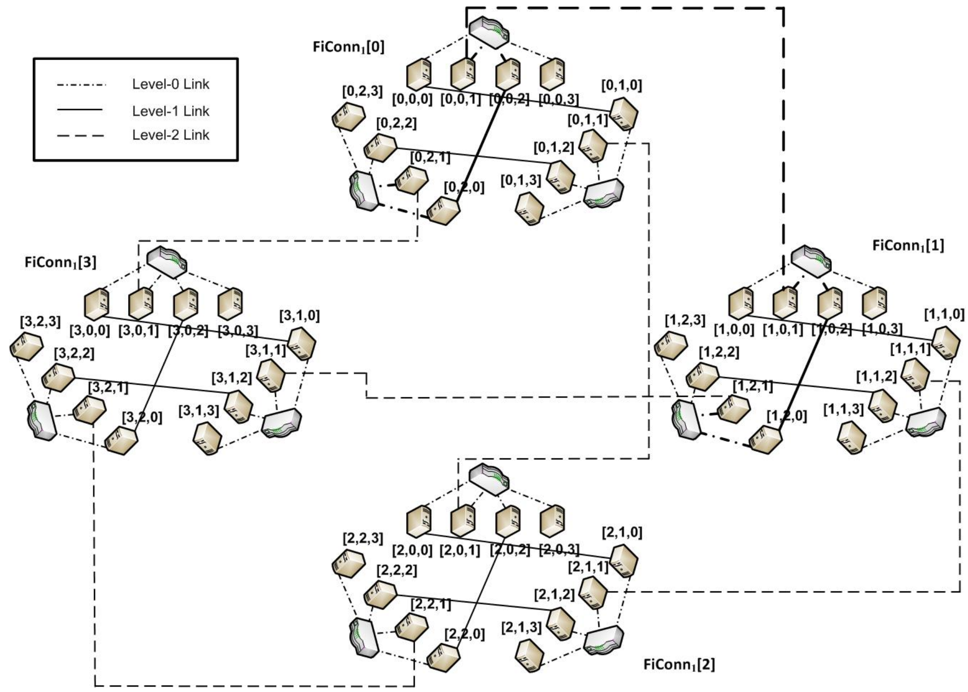

Fig. 3. A FiConn$_2$ with $n = 4$. The FiConn$_2$ is composed of 4 FiConn$_1$s, and each FiConn$_1$ is composed of 3 FiConn$_0$s. A level-0 link connects one server port (the original operation port) to a switch, denoted by dot-dashed line. A Level-1 or Level-2 link connects the other port (the original backup port) of two servers, denoted by solid line and dashed line respectively. The path from [0,2,1] to [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])

have a level-$i$ link ($1 \leq i \leq k$). As a result, the average server node degree in FiConn$_k$ 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 to 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 two.

**Theorem 3:** Suppose $L_l$ denote the number of level-$l$ links in FiConn$_k$, there is

$$L_l = \begin{cases} 4 * L_{l+1}, & \text{if} \quad l = 0 \\ 2 * L_{l+1}, & \text{if} \quad 0 < l < k \end{cases}$$

*Proof:* First we prove $L_0 = 4 * L_1$, and we only need to prove that it holds in a FiConn$_1$. Each server in a FiConn$_1$ has one level-0 link, so there is $L_0 = N_1$. Half of the servers in FiConn$_1$ 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 $L_0 = 4 * L_1$.

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

The relationship among the numbers of links in different levels disclosed in Theorem 3 matches the basic routing designed below in FiConn, which is in favor of making a balanced use of FiConn links. It will be further explained in the following subsection.

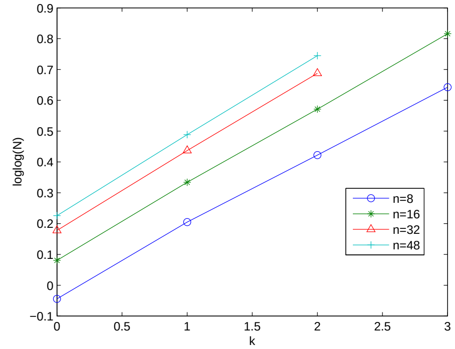

Fig. 4. The relationship between the total number of servers $N$ and the FiConn level $k$. The y-axis is $log_{10}(log_{10}N_k)$.

## B. *Traffic-Oblivious Routing*

We design a Traffic-Oblivious Routing (TOR) algorithm in FiConn which takes advantage of the level-based characteristic of FiConn. For any pair of servers, if the lowest common level of FiConn they belong to is FiConn$_l$, the routing path between them is constrained to the two FiConn$_{l-1}$s comprising the two servers respectively, and the level-$l$ link connecting the two FiConn$_{l-1}$s. 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$_0$ as $s$, and the function returns $dst$ (Line 4). Next, we get the level-$l$ link connecting the two FiConn$_{l-1}$s comprising $s$ and $dst$ respectively, say, $(i_1, i_2)$ (Line 5). If $i_1$ is $s$ itself (Line 6), then $i_2$ is returned (Line 7). Otherwise, we recursively compute and return the next-hop server from $s$ towards $i_1$ (Line 8).

```
/*$s$: current server.
$dst$: destination server of the packet to be routed.
*/
01 TORoute($s, dst$){
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 }
```

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

Take Fig. 3 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 7 hops.

From TOR, the number of level-$l$ links ($0 < l < k$) in a typical routing path in FiConn$_k$ 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$_0$ includes two links since it crosses the switch). Meanwhile, Theorem 3 tells that in FiConn$_k$, 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 4:** The upper bound of the diameter of FiConn$_k$ is $2^{k+1} - 1$.

*Proof:* Using the TOR, the longest routing path between any two serves in FiConn$_k$ takes 1 level-$k$ hop, 2 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 $1 + 2 + ... + 2^k = 2^{k+1} - 1$. ■

In combination with Theorem 1, the diameter of FiConn is $O(logN_k)$, 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, benefiting applications with real-time requirement.

**Theorem 5:** The lower bound of the bisection width of FiConn$_k$ is $N_k/(4 * 2^k)$, where $N_k$ is the total number of servers in FiConn$_k$.

*Proof:* In all-to-all communication, the number of flows on the FiConn$_k$ link that carries the most flows is about $2^k * N_k$ times of that in its embedding complete graph. Based on [15], the lower bound of the bisection width of FiConn$_k$ is $1/(2^k * N_k)$ times of that of complete graph, that is, $(1/(2^k * N_k)) * (N_k^2/4) = N_k/(4 * 2^k)$. ■

Considering Theorem 1, the bisection width of FiConn$_k$ is also $O(N_k/logN_k)$. 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 multi-path 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 Traffic-Aware Routing (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, nor 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$_l$ 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$_0$ or FiConn$_1$. 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,

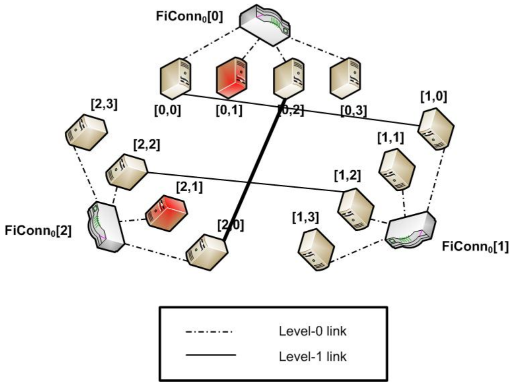

Fig. 5. Illustration for routing path setup. There is already one flow in the level-1 link from [2,0] to [0,2] and all other links carry no traffic. Now [2,1] initiates a flow towards [0,1]. The path using TOR is ([2,1], [2,0], [0,2], [0,1]). The path using TAR is ([2,1], [2,2], [1,2], [1,0], [0,0], [0,1]).

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-$s$ 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 towards the destination server. Each intermediate server routes the path-probing packet based on local traffic states as stated above, 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 towards 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 Figure 5. 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 towards 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$_0$, the next-hop server may route the packet back using TOR. In the example of Figure 5, 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. But 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$_l$ thus takes too long a path or even falls into a loop. In the example of Figure 5, 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 Figure 5, [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-$l$ server $s$ routes a flow via its level-$l$ 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-$l$ link because the available bandwidth of its level-0 outgoing link is always higher. In this case, the outgoing bandwidth of its level-$l$ link cannot be well utilized even though the other level-$l$ links in the FiConn$_l$ are heavily-loaded. In the example of Figure 5, all subsequent flows from FiConn$_0$[2] to FiConn$_0$[0] will bypass the level-1 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 PR$_l$ ($1 \leq l \leq m$) to denote the $l^{th}$ entry of PR field. Each PR$_l$ plays two roles. First, when bypassing a level-$l$ link, the level-$l$ server in the selected third FiConn$_{l-1}$ 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. But 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.5 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 towards [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-$l$ ($l > 0$) 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 towards 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]).

### *D. 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-$l$ 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.

```
/*$s$: current server.
$l$: the level of $s$. ($l > 0$)
$RTable$: the routing table of $s$, maintaining the previous hop
($.prevhop$) and next hop ($.nexthop$) for a flow.
$hb$: the available bandwidth of the level-$l$ link of $s$.
$zb$: the available bandwidth of the level-0 link of $s$.
$hn$: the level-$l$ 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 ($.pr$).
*/
01 TARoute($s$, $pkt$){
02    if($pkt.dst == s$) /*This the destination*/
03       return NULL /*Deliver pkt to upper layer*/
04    if($pkt.phop == RTable[pkt.flow].nexthop$) /*SRR*/
05       $nhop = RTable[pkt.flow].prevhop$
06       $RTable[pkt.flow] =$ NULL
07       if($nhop \neq$ NULL) /*This is not source server*/
08          return $nhop$
09    if($s == pkt.pr[l]$) /*This is the proxy server*/
10       $pkt.pr[l] = BYONE$
11    $ldst =$ getPRDest($pkt$) /*Check PR for proxy server*/
12    $nhop =$ TORoute($s, ldst$)
13    if($s == pkt.src$ and $nhop \neq hn$ and $hb > zb$)
14       $nhop = hn$
15    if($pkt.phop == hn$ and $nhop \neq hn$)
        or ($pkt.phop \neq hn$ and $hb \geq zb$)
16       resetPR($pkt.pr, l$)
17       $RTable[pkt.flow] = (pkt.phop, nhop)$
18       if($nhop \neq hn$ and $vfc > 0$)
19          $vfc = vfc - 1$ /*VF*/
20       return $nhop$
21    $fwdhop = nhop$
22    while($fwdhop == nhop$)
23       $fwdhop =$ bypassLink($s, pkt, l$)/*Try to bypass*/
24    if($fwdhop ==$ NULL) /*Cannot find a bypassing path*/
25       resetPR($pkt.pr, l$)
26       $RTable[pkt.flow] = (pkt.phop, nhop)$
27       return $nhop$
28    $vfc = vfc + 1$ /*VF*/
29    return $pkt.phop$ /*Proxy found, SRR*/
30 }
```

**Algorithm 3**: Traffic-aware routing (TAR) in FiConn.

### *E. 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-$l$ server $s$ routes the path-probing packet $pkt$.

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-$l$ proxy server in the current FiConn$_l$ (line 9). It modifies PR$_l$ as *BYONE*.

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 towards 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$_0$ but the available bandwidth of its level-$l$ link is higher than that of its level-0 link (Line 13), its level-$l$ 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-$l$ link. The first case is that the previous hop is the level-$l$ 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$_0$ and the available bandwidth of the level-$l$ link is not less than that of the level-0 link. Line 15 makes the judgement. Lines 16-17 reduces $vfc$ 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 PR$_i$s ($i < l$) as *BYZERO*.

Lines 21-29 deal with how to bypass the level-$l$ link. The function bypassLink() in Line 23 finds a proxy server to bypass the level-$l$ link of $s$, updates the PR field and returns the next hop towards the proxy server; but if it cannot find a proxy server, it returns NULL. Therefore, if bypassLink() returns NULL (Line 24), level-$l$ 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 $vfc$ is added by one.

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

**Theorem 6:** In TAR, the maximum length of routing path between any two servers in FiConn$_k$ is $2 * 3^k - 1$.

*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$_k$s and two level-$k$ links between them. Hence, there is $M_{k+1} = 3 * M_k + 2$, and $M_0 = 1$. As a result, we have $M_k = 2 * 3^k - 1$. ∎

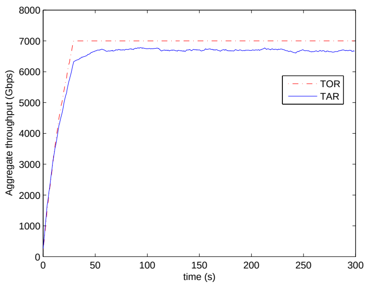

Fig. 6. Aggregate throughput in FiConn for random traffic.

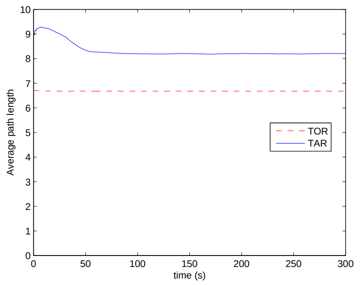

Fig. 7. Average path length in FiConn for random traffic.

## V. Evaluation

We have analyzed the basic properties of FiConn in Section III, such as the high scalability, low diameter, high bisection width, as well as the balanced use of different levels of links in routing. In this section, we conduct simulations to evaluate the routing algorithms we design for FiConn.

We run the simulation on a FiConn$_2$ in which $n = 32$, thus there are in total $N = 74,528$ servers. The speed of all the Ethernet ports and links are 1Gbps. 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. So 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 seconds, and the path-probing packet in TAR is sent every 30 seconds for a flow. We compute the aggregate throughput and average path length of TAR and TOR respectively.

**Random Traffic:** Fig.6 and Fig.7 illustrate the aggregate throughput and the average path length respectively for random traffic.

From Fig.6, we see that the aggregate throughputs of TAR and TOR are very close. At the end of the first 30 seconds, 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

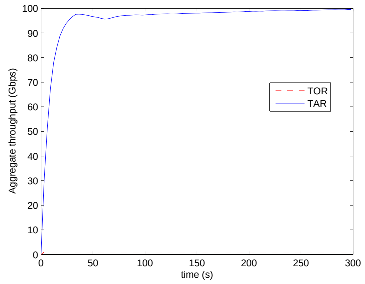

Fig. 8. Aggregate throughput in FiConn for burst traffic.

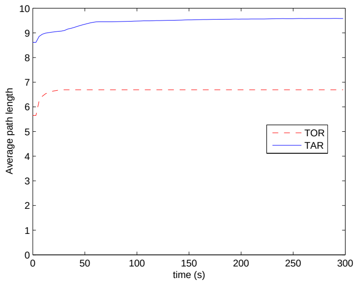

Fig. 9. Average path length in FiConn for burst traffic.

distributed.

Fig.7 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.6, we also find that TAR can dynamically adapt to traffic states and improve the throughput as well as reduce the path length.

**Burst Traffic:** Fig.8 and Fig.9 show the aggregate throughput and the average path length respectively for burst traffic.

From Fig.8 we find that the aggregate throughput of TOR is only 1Gbps, 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 99.5G, which shows a tremendous improvement over TOR.

The result of Fig.9 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 1-3 hops in the FiConn$_2$. 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.

## VI. 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 low-cost structure because it eliminates the use of expensive high-end switches and puts no additional hardware cost on servers. It is highly scalable to encompass hundreds of thousands of servers with low diameter and high bisection width. The routing mechanisms in FiConn balance different levels of links and are traffic-aware to better utilize the link capacities according to traffic states.

## References

[1] T. Hoff, "Google Architecture", *http://highscalability.com/google-architecture*, Jul 2007

[2] L. Rabbe, "Powering the Yahoo! network", *http://yodel.yahoo.com /2006/11/27/powering-the-yahoo-network*, Nov 2006

[3] A. Carter, "Do It Green: Media Interview with Michael Manos", *http://edge.technet.com/Media/Doing-IT-Green*, Dec 2007

[4] J. Snyder, "Microsoft: Datacenter Growth Defies Moore's Law", *http://www.pcworld.com/article/id,130921/article.html*, 2007

[5] S. Ghemawat, H. Gobio, and S. Leungm, "The Google File System", In *Proceedings of ACM SOSP'03*, 2003

[6] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", In *Proceedings of OSDI'04*, 2004

[7] M. Isard, M. Budiu, Y. Yu and etc., "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks", In *Proceedings of ACM EuroSys'07*, 2007.

[8] M. Al-Fares, A. Loukissas and A. Vahdat, "A Scalable, Commodity Data Center Network Architecture", In *Proceedings of ACM SIGCOMM'08*, Aug 2008

[9] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang and Songwu Lu, "DCell: A Scalable and Fault-Tolerant Network Structure for Data Centers", In *Proceedings of ACM SIGCOMM'08*, Aug 2008

[10] Dell Powerage Servers. *http://www.dell.com/content/products/category.aspx /servers*

[11] H. Sullivan and T. R. Bashkow, "A large scale, homogeneous, fully distributed parallel machine I", In *Proceediings of ISCA'77*, Mar 1977

[12] L. Bhuyan and D. Agrawal, "Generalized Hypercube and Hyperbus Structures for a Computer Network", In *IEEE TRANSACTIONS ON COMPUTERS*, 33(4):323-333, Apr 1984

[13] W. Dally and B. Towles, "Route Packets, Not Wires: On-Chip Interconnection Networks", In *Proceedings of DAC'01*, Jun 2001

[14] L. Bhuyan and D. Agrawal, "A general class of processor interconnection strategies", In *Proceedings of ISCA'82*, Apr 1982

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

[16] B. Parhami, "Introduction to Parallel Processing: Algorithms and Architectures", *Kluwer Academic*, 2002

[17] W. Dally, P. Carvey and L. Dennison, "The Avici Terabit Switch/Router", In Proceedings of Hot Interconnects'98, Aug 1998

[18] D. Loguinov, A. Kumar, V. Rai and S. Ganesh, "Graph-Theoretic Analysis of Structured Peer-to-Peer Systems: Routing Distances and Fault Resilience", In *Proceedings of ACM SIGCOMM'03*, Aug 2003

[19] J Kim, W. Dally and D. Abts, "Flattened Butterfly: A Cost-Efficient Topology for High-Radix Networks", In *Proceedings of ISCA'07*, Jun 2007

[20] J Kim, W. Dally, S. Scott and D. Abts, "Technology-Driven, Highly-Scalable Dragonfly Topology", In *Proceedings of ISCA'08*, Jun 2008