

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—It is challenging to interconnect the large and rapidly growing number of servers in data centers with low cost. In current practice, tree is used and expensive high-speed switches are required in top levels of the tree. As an alternative solution, interconnection intelligence can be put on servers.

Commodity servers in data centers often have two NIC ports, but only one is used for operation in current practice and the other is for backup. In this paper, we design FiConn, a structure for server interconnection in data centers which exploits the backup port on servers and eliminates the requirement of any switches other than the lowest-level commodity ones. Although with the server node degree of only two, FiConn is highly scalable to encompass hundreds of thousands of servers, while at the same time has low diameter and high bisection width. In addition, routing in FiConn makes a balancing use of different levels of links.

To further utilize the link capacities according to traffic states, we design *traffic-aware routing* in FiConn with little control overhead. The traffic-aware path is computed hop-by-hop by each intermediate server based on the available bandwidths of its two outgoing links. Simulation results demonstrate that the traffic-aware routing indeed achieves high networking throughput.

I. INTRODUCTION

The number of servers in today’s data centers is very large. It is reported that companies such as Google, Yahoo! and Microsoft have built data centers composed of hundreds of thousands of servers, and the number is still increasing [1], [2], [3]. It is also not unusual that some universities or institutions have thousands of servers or even more [7] in their data centers. These servers need to be interconnected because of large volumes of data exchange among them, from running on-line applications such as search, gaming, web mail, to providing infrastructure services such as GFS [4], map-reduce [5] and Dryad [6]. Considering that the cost of interconnection structure is among the major budgets for building a data center, it is desirable to design a networking structure with low cost.

In current practice, tree is used for interconnection. Each server uses one NIC port to connect to a lowest-level commodity switch, and higher-speed switches are placed on higher levels to interconnect the lower-level ones. It is well understood that the tree structure has many disadvantages [7], [8]. First, the top-level switches are the bandwidth bottlenecks. As a result, expensive high-end switches are demanded for encompassing tens of thousands or even hundreds of thousands of servers. Second, a high-level switch is the single-fault point for the sub-tree rooted from it. Using redundant switches does not intrinsically solve the problem, but brings even higher cost.

As an alternative solution other than high-end switches, we can put interconnection intelligence on servers. Commodity

servers in data centers often have two on-board NIC ports [9], but only one is used for operation in current practice and the other is for backup. It gives us hint that we can exploit the backup port for server interconnection. In this way, we do not bring much deploying cost on servers. However, there is no structure in the literature with the server node degree of two that can scale to tens of thousands or even hundreds of thousands of servers with acceptable diameter and bisection width. In this paper, we design such an interconnection structure for data centers, which we call FiConn. Each server in FiConn uses two NIC ports, one connecting to a commodity switch (the original operation port) and one connecting to another FiConn server (the original backup port). Hence, switches other than the lowest-level commodity ones are unnecessary.

FiConn is recursively defined. 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 interconnection and they form a mesh. In this way, the total 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 361200 servers. The diameter of FiConn is $O(\log N)$, which is small and can thus support applications with real-time requirement. The bisection width of FiConn is $O(N/\log N)$, indicating that FiConn can well tolerate port/link faults. Interestingly, the basic routing algorithm in FiConn can make a balancing use of different levels of links. Note that although we use the backup port of each server, the server’s reliability is not sacrificed because when one port fails, it can still work using the other port.

To further utilize the link capacities according to traffic states, we design *traffic-aware routing* in FiConn based on the basic routing algorithm. Because of the large number of servers, we do not depend on any central server(s) for traffic scheduling nor exchange traffic states among FiConn servers. Instead, the traffic-aware path is computed hop-by-hop by each intermediate server based on the available bandwidths of its two outgoing links. Simulation results demonstrate that the traffic-aware routing performs similar to the basic routing algorithm for random traffic, while achieves much higher throughput for burst traffic between two subsets of FiConn servers, which is common in data centers produced by computation models such as map-reduce. In other words, the traffic-aware routing can indeed well exploit the link capacities of FiConn to improve the networking throughput.

In summary, the primary contribution of this paper lies in two-folds. First, we propose FiConn, a novel structure using

backup port for server interconnection in data centers. FiConn is highly scalable with the server node degree of two, while at the same time has low diameter and high bisection width. Second, we design traffic-aware routing in FiConn that well exploits the link capacities based on traffic states so as to improve the networking throughput.

Compared with tree structure, FiConn introduces some additional overheads. First, servers in FiConn participates in packet forwarding thus more CPU resource is consumed. Second, the wiring effort is higher since each server uses two NIC ports. However, these overheads are well justified because multi-core is getting popular, and the use of two NIC ports can provide higher end-to-end throughput than using only one port.

The rest of this paper is organized as follows. Section II introduces the related work. Section III describes the interconnection structure of FiConn and the basic properties. Section IV designs the traffic-aware routing in FiConn. Section V conducts simulations to evaluate the traffic-aware routing. Finally, Section VI concludes the whole paper.

II. RELATED WORK

A. Interconnection Structure for Data Centers

Due to the well understood limitations of tree structure, new structures are recently proposed for server interconnection in data centers, such as Fat-Tree [7] and DCell [8].

In the Fat-Tree structure, commodity GigE switches are used to interconnect the servers and there is no requirement of 10GigE or faster switches. Fig.1 illustrates the topology, which has three levels of switches. There are n pods (in this example, $n = 4$), each containing two levels of $n/2$ switches, i.e., edge level and aggregation level. Every n -port switch in 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 the Fat-Tree structure supports is $n^3/4$. Given a typical value $n = 48$, the number of servers there is 27648.

FiConn differs from Fat-Tree in several aspects. First, FiConn puts the interconnection intelligence on servers instead of on switches, so the number of switches used are much less than in Fat-Tree. Given the total number of servers is N and the n -port switches are used, the number of switches needed in Fat-Tree is $5N/n$, while that required in FiConn is N/n . In other words, 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 the limitation and can extend to very large number of servers with the server node degree of two. Third, Fat-Tree depends on central server(s) for traffic scheduling in the present design, but the traffic-aware routing in FiConn distributedly computes the routing path with little control overhead.

As we show in Fig.2, DCell is a level-based structure. In DCell_0 , n servers are connected to a commodity n -port switch. Given there are totally t servers in a DCell_k , $t + 1$ DCell_k s are used to build a DCell_{k+1} and the t servers in a DCell_k

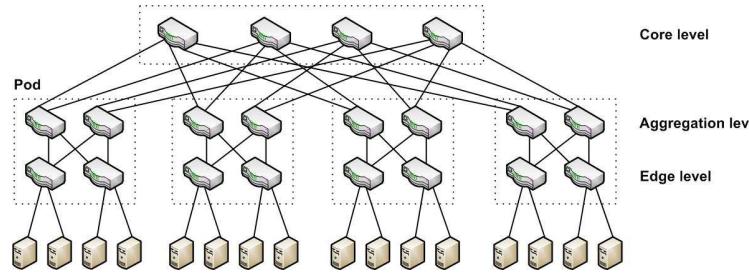


Fig. 1. A Fat-Tree structure with $n = 4$. It has three levels of switches.

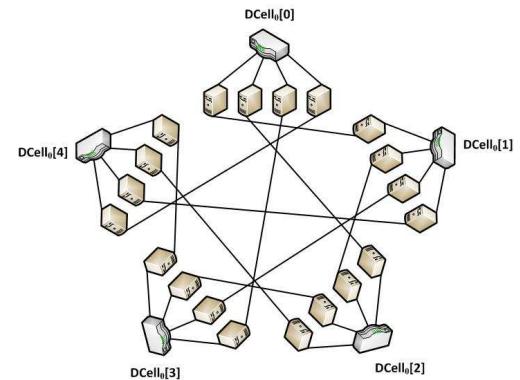


Fig. 2. A DCell₁ structure with $n = 4$. It is composed of 5 DCell₀s.

connect to the other t DCell_k s respectively. In this way, DCell achieves high scalability and high bisection width.

FiConn shares the same principle as DCell to put the interconnection intelligence on servers. However, they are quite different. First, the server node degree in a DCell_k is $k+1$, but that of FiConn is always two. As a result, FiConn just needs to take use of the existing backup port on each server for interconnection, and no other hardware cost or deploying issue on servers is introduced. Second, the wiring effort in FiConn is less than that of DCell because each server uses only two ports. Third, routing in FiConn makes a balancing use of links in different levels. Traffic-aware routing is further designed to exploit the link capacities according to traffic states. On the contrary, DCell does not have the balancing property and there is no traffic-aware routing in it.

B. Interconnection Structure in Other Areas

Besides in data centers, interconnection structures are widely studied in various areas, such as parallel computing [13], [14], [15], on-chip network [12], and switching fabric [16]. Proposed structures include Ring [15], HyperCube [10], [11], Butterfly [14], Torus [15], De Bruijin [17], Flattened Butterfly [18] and DragonFly [19]. However, these structures target at other scenarios, and are not specifically designed to meet the requirements of data centers, such as high scalability for supporting large number of nodes, small node degree for low wiring cost and high bisection width for tolerating link/node faults.

Among these structures, only Ring [15] has the constant node degree of two, which is similar to FiConn. However, the diameter of Ring is 2 and the bisection width is $N/2$,

where N is the total number of nodes. On the contrary, FiConn has the diameter of $O(\log N)$ and the bisection width of $O(N/\log N)$. Undoubtedly, FiConn is more desirable for server interconnection in data centers where N is very large.

III. FiCONN: A NOVEL INTERCONNECTION STRUCTURE FOR DATA CENTERS

In this section we design the interconnection rule of FiConn and a basic routing algorithm in the structure.

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 them. Typically, n is an even number such as 16, 32, or 48. The speed of the switch ports is unnecessary faster than that of the server ports (usually 1G). 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 with 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 backup ports of servers in $FiConn_{k-1}$ s. 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 $0 \leq u_k < N_k$. In addition, s can be identified by a $(k+1)$ -tuple, $[a_k, \dots, a_1, a_0]$, where a_0 identifies s in its $FiConn_0$, and a_l ($1 \leq l \leq k$) 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 $(u_{k-1} - 2^{k-1} + 1) \bmod 2^k == 0$ are selected as *level- k servers* (Line 3), and they are interconnected as Lines 4-6 instruct.

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*

```

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  $[i_1, j_1]$  with  $[i_2, j_2]$ 
07   return
08 }
```

Algorithm 1: Constructing $FiConn_k$ upon g_k $FiConn_{k-1}$ s.

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 $[3, 2, 1]$.

Based on the interconnection rule of FiConn, we discuss some of its properties.

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

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

We validate the correctness of Theorem 1.

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

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 361200. If we choose $n = 16$ and $k = 3$, the number becomes 3553776.

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 have a level- i link ($1 \leq i \leq k$). As a result, the average server node degree in $FiConn_k$ is $(N_k + \sum_{i=1}^k (N_k/2^i))/N_k = 2 - 1/2^k$. \blacksquare

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.

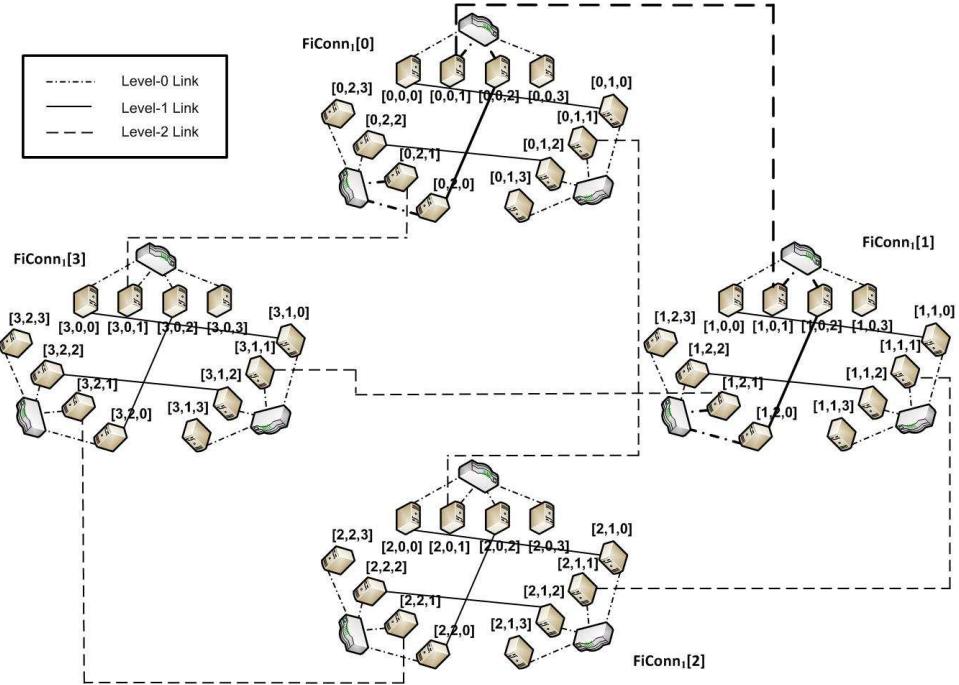


Fig. 3. A FiConn₂ with $n = 4$. The FiConn₂ is composed of 4 FiConn₁s, and each FiConn₁ is composed of 3 FiConn₀s. A level-0 link connects one server port (the original operation port) to a switch, denoted by dot-dashed line. A Level-1 or Level-2 link connects the other port (the original backup port) of two servers, denoted by solid line and dashed line respectively. The path from [0,2,1] to [1,2,1] using level-recursive routing 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]).

Theorem 3: Suppose L_l denote the number of level- l links in FiConn _{k} , there is

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

Proof: First we prove $L_0 = 4 * L_1$, and we only need to prove it holds in a FiConn₁. Each server in a FiConn₁ has one level-0 link, so there is $L_0 = N_1$. Half of the servers in FiConn₁ 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 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 balancing use of FiConn links. It will be further explained in the following subsection.

B. Level-Recursive Routing

We design a basic routing 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

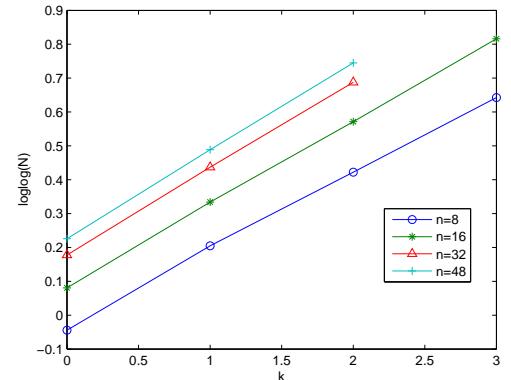


Fig. 4. The relationship between the total number of servers N and the FiConn level k . The y-axis is $\log_{10}(\log_{10}N_k)$.

FiConn _{$l-1$} s. Hence, the routing path between two servers can be recursively calculated. We call this basic routing algorithm *level-recursive routing*.

Algorithm 2 shows how the level-recursive routing works on a server s to route a packet destined to dst . The function LRRoute() 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 FiConn₀ 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 LRRoute(s, dst){
02   l = lowestCommonLevel(s, dst)
03   if(l == 0)
04     return dst
05   (i1, i2) = getLink(s, dst, l)
06   if(i1 == s)
07     return i2
08   return LRRoute(s, i1)
09 }
```

Algorithm 2: Level-recursive routing in FiConn.

Take Fig. 3 as an example. The path from source server [0,2,1] to destination server [1,2,1] using level-recursive routing 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 the level-recursive routing, 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, the level-recursive routing makes a balancing use of different levels of FiConn links, which helps improve the aggregate throughput, especially in random traffic pattern.

Leveraging the level-recursive routing, 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 level-recursive routing, the longest routing path between any two servers 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 $1 + 2 + \dots + 2^k = 2^{k+1} - 1$. ■

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, 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 [14], 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 $O(N_k/\log N_k)$. The high bisection width of FiConn implies that there are many possible paths between a pair of servers, resulting from which FiConn can well tolerate port/link faults and it is possible for multi-path routing.

IV. TRAFFIC-AWARE ROUTING IN FiCONN

Level-recursive routing makes a balancing use of different levels of FiConn links and we take it as the basis for FiConn routing. However, it has some limitations. Firstly, a pair of servers cannot take advantage of two ports on each to improve their end-to-end throughput in level-recursive routing. Secondly, level-recursive routing cannot further utilize the link capacities according to traffic states to improve the networking throughput. To overcome these limitations, we design traffic-aware routing in FiConn.

A. Basic Design and Challenges

Because of the large number of servers in data centers, we do not depend on central server(s) for traffic scheduling, nor exchange traffic states among all the FiConn servers. Instead, we target at distributedly computing the routing path with little control overhead.

We use a greedy approach to hop-by-hop setup the traffic-aware path on each intermediate server, that is, each server tries to balance the traffic between its two outgoing links. Specifically, the source server always chooses the outgoing link with higher available bandwidth to send the traffic. For a level- l ($l > 0$) intermediate server, if the outgoing link based on level-recursive routing is its level- l link and the available bandwidth of its level-0 link is higher, its level- l link is bypassed by *randomly* selecting a third FiConn_{l-1} in the FiConn_l to relay the traffic; otherwise, the traffic is routed by level-recursive routing.

When the level- l server s selecting a third FiConn_{l-1} for relay, a possible choice except random selection is to exchange traffic states among all the level- l servers within each FiConn_{l-1} , and so s can 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 and it brings 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 . But there may be few or even no level- l servers in such a range if l is high, and so the candidate third FiConn_{l-1} s are very limited. As a result, in the 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 wide candidate set of third FiConn_{l-1} s.

In fact, our idea of traffic-aware routing can be easily extended to handle port/link faults, which is common in data centers. When a port or link fails, it can be regarded as that the available bandwidth of the link becomes zero, so the traffic will always be routed by the other link of the server. In this sense, port/link fault handling is just an extreme case of traffic-aware routing. The only extension 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 packet basis. Instead, we target at flow basis and dynamically setup the traffic-aware path for a flow using

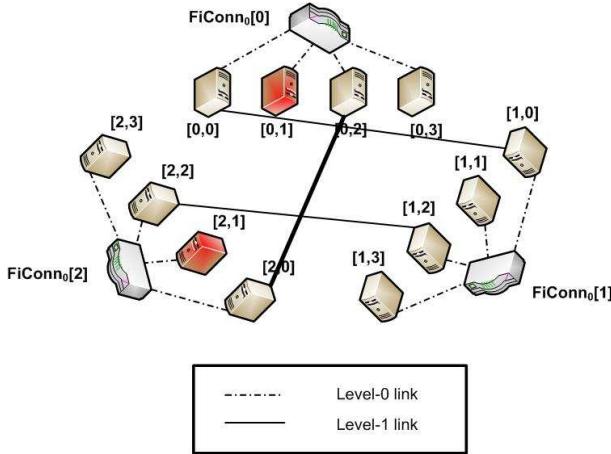


Fig. 5. Illustration for traffic-aware path setup. There is already one flow in the level-1 link from [2,0] to [0,2] and all other links carry no traffic. Now [2,1] initiates a flow towards [0,1]. The path using level-recursive routing is ([2,1], [2,0], [0,2], [0,1]). The path using traffic-aware routing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

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 replies the packet 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 traffic-aware routing in the example of Fig.5. There is already one flow in the level-1 link from [2,0] to [0,2] and all other links carry no traffic. Now server [2,1] initiates a flow towards server [0,1]. Obviously, the path using level-recursive routing is ([2,1], [2,0], [0,2], [0,1]). Under traffic-aware routing, when [2,0] receives the path-probing packet from [2,1], it finds that its level-1 outgoing link to [0,2] has less available bandwidth than its level-0 outgoing link, and then randomly selects a third FiConn₀ in the FiConn₁ for relay. In this case, FiConn₀[1] is selected. Finally the packet is routed to [0,1] by the relay of FiConn₀[1].

However, we should address some challenges in the basic design of traffic-aware routing.

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 FiConn₀, the next-hop server may route the packet back using level-recursive routing. For example in Fig.5, when [2,2] receives the path-probing packet from [2,0], it routes the packet back to [2,0] using level-recursive routing unless otherwise specified.

Multiple bypassing: When one level- l ($l > 0$) link is bypassed, a third FiConn_{l-1} is chosen for relay and two other level- l links in the current FiConn_l will be passed through. But the two level- l links may be needed to bypass again according to the basic design. It can iteratively occur and thus routing in the FiConn_l takes too long a path, or even falls into a loop. For example in Fig.5, supposing the level-1 link from [2,2] to [1,2] should also be bypassed because there is a flow in it, the routing falls into 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 means there are intermediate servers that can be removed from the path without reducing the throughput of the path. For example in Fig.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 a level- l server s routes a flow using 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. For example in Fig.5, all subsequent flows from FiConn₀[2] to FiConn₀[0] will bypass the level-1 link of [2,0]. In fact, the problem results from the basic idea that our traffic-aware routing tries 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 balancing 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,

```

/*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 != 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 = LRRoute(s, ldst)
13   if(s == pkt.src and nhop != hn and hb > zb)
14     nhop = hn
15   if(pkt.phop == hn and nhop != hn)
16     or (pkt.phop != hn and hb ≥ zb)
17     resetPR(pkt.pr, l)
18     RTable[pkt.flow] = (pkt.phop, nhop)
19     if(nhop ≠ hn and vfc > 0)
20       vfc = vfc - 1 /*VF*/
21     return nhop
22   fwdhop = nhop
23   while(fwdhop == nhop)
24     fwdhop = bypassLink(s, pkt, l)/*Try to bypass*/
25   if(fwdhop == NULL) /*Cannot find a bypassing path*/
26     resetPR(pkt.pr, l)
27     RTable[pkt.flow] = (pkt.phop, nhop)
28     vfc = vfc + 1 /*VF*/
29   return pkt.phop /*Proxy found, SRR*/
30 }

```

Algorithm 3: Traffic-aware routing in $FiConn$.

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.

E. Algorithm

Taking the solutions above together, we design the algorithm of traffic-aware routing 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 level-recursive routing. 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 level-recursive routing (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 level-recursive routing 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 level-recursive routing 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 ($i < l$) 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 traffic-aware routing designed above, we can calculate the maximum length of routing path in traffic-aware routing.

Theorem 6: In traffic-aware routing, 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 traffic-aware routing is M_k , the longest traffic-aware routing 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 $M_k = 2 * 3^k - 1$. ■

V. EVALUATION

We have analyzed the basic properties of FiConn in Section III, such as the high scalability, low diameter, high bisection

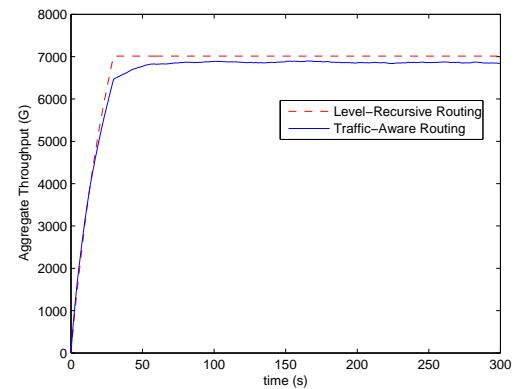


Fig. 6. Aggregate throughput in FiConn for random traffic.

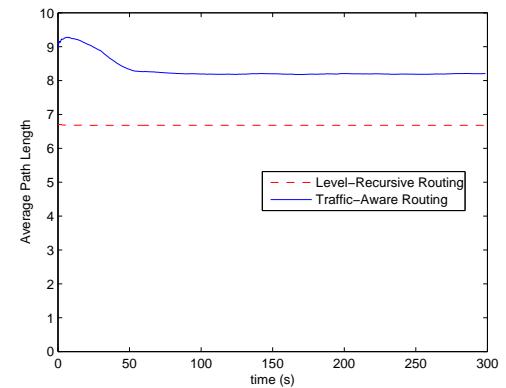


Fig. 7. Average path length in FiConn for random traffic.

width, as well as the balancing use of different levels of links in routing. In this section, we conduct simulations to evaluate the traffic-aware routing in FiConn .

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. We run the simulation on a FiConn_2 in which $n = 32$, thus there are in total $N = 74528$ servers. The speed of all the NIC ports and links are 1G. 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 37264 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 295936 flows in the network. All the flows are initiated sequentially in the first 30 seconds, and the path-probing packet in traffic-aware routing is sent every 30 seconds for a flow. We compute the aggregate throughput and average path length of traffic-aware routing and level-recursive routing 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 traffic-aware routing and level-recursive routing are very close. At the end of the first 30 seconds, the throughput of level-recursive

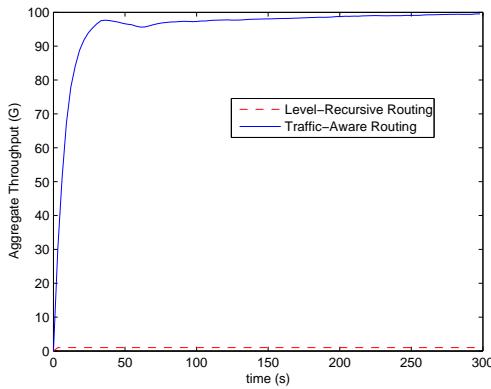


Fig. 8. Aggregate throughput in FiConn for burst traffic.

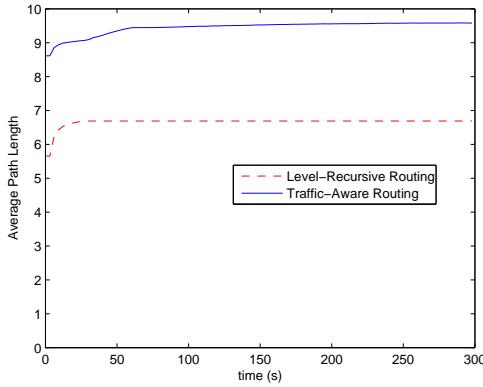


Fig. 9. Average path length in FiConn for burst traffic.

routing is about 8.5% higher than that of traffic-aware routing. However, after several rounds of dynamical adjustment, the difference between them is within 2.5%. The slight advance of level-recursive routing comes from its shorter routing path, which benefits improving the aggregate throughput when traffic is randomly distributed.

Fig.7 shows that the average path length of traffic-aware routing is always more than that of level-recursive routing, but within 1.5 hops in steady state. In combination with Fig.6, we also find that the traffic-aware routing 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 level-recursive routing is only 1G, resulting from the bottleneck level-2 link that connects the two selected FiConn₁s. However, by exploiting the links beyond the two FiConn₁s and the bottleneck level-2 link, traffic-aware routing achieves an aggregate throughput of 99.5G, which shows a tremendous improvement over level-recursive routing.

The result of Fig.9 also tells that the average path length of traffic-aware routing is longer than that of level-recursive routing, but the difference is within three hops.

Taking the two groups of simulations together, we draw the following conclusions. First, our traffic-aware routing can

adapt to dynamical networking conditions to improve the throughput as well as reduce the routing path length. Second, the average path length in traffic-aware routing is always more than that in level-recursive routing, but the difference is no more than 1-3 hops in the FiConn₂. Third, the aggregate throughput of traffic-aware routing is quite similar to level-recursive routing in uniform traffic, but much higher than level-recursive routing in burst traffic that is common in data centers. In other words, the traffic-aware routing can indeed well exploit the link capacities of FiConn to improve the networking throughput. Considering the little control overhead, our traffic-aware routing is especially suitable for FiConn.

VI. CONCLUSION

In this paper we propose FiConn, a novel interconnection structure for data centers, and design traffic-aware routing in FiConn. FiConn is desirable for server interconnections in data centers in the following aspects. First, FiConn eliminates the use of expensive high-end switches. Second, FiConn can take use of the existing backup port on each server for interconnection because the server node degree is only two. Third, FiConn is highly scalable to encompass very large number of servers, while at the same time has low diameter and high bisection width. Fourth, traffic-aware routing in FiConn is designed to utilize the link capacities according to traffic states, which helps improve the aggregate throughput of data centers.

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] S. Ghemawat, H. Gobio, and S. Leungm, "The Google File System", In *Proceedings of ACM SOSP'03*, 2003
- [5] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", In *Proceedings of OSDI'04*, 2004
- [6] M. Isard, M. Budiu, Y. Yu and etc., "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks", In *Proceedings of ACM EuroSys'07*, 2007.
- [7] M. Al-Fares, A. Loukissas and A. Vahdat, "A Scalable, Commodity Data Center Network Architecture", In *Proceedings of ACM SIGCOMM'08*, Aug 2008
- [8] 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
- [9] Dell Poweredge Servers. <http://www.dell.com/content/products/category.aspx/servers>
- [10] H. Sullivan and T. R. Bashkow, "A large scale, homogeneous, fully distributed parallel machine I", In *Proceedings of ISCA'77*, Mar 1977
- [11] 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
- [12] W. Dally and B. Towles, "Route Packets, Not Wires: On-Chip Interconnection Networks", In *Proceedings of DAC'01*, Jun 2001
- [13] L. Bhuyan and D. Agrawal, "A general class of processor interconnection strategies", In *Proceedings of ISCA'82*, Apr 1982
- [14] F. Leighton, "Introduction to Parallel Algorithms and Architectures: Arrays. Trees. Hypercubes", *Morgan Kaufmann*, 1992
- [15] B. Parhami, "Introduction to Parallel Processing: Algorithms and Architectures", *Kluwer Academic*, 2002
- [16] W. Dally, P. Carvey and L. Dennison, "The Avici Terabit Switch/Router", In *Proceedings of Hot Interconnects'98*, Aug 1998
- [17] 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

- [18] J Kim, W. Dally and D. Abts, "Flattened Butterfly: A Cost-Efficient Topology for High-Radix Networks", In *Proceedings of ISCA'07*, Jun 2007
- [19] J Kim, W. Dally, S. Scott and D. Abts, "Technology-Driven, Highly-Scalable Dragonfly Topology", In *Proceedings of ISCA'08*, Jun 2008