JOURNAL OF
Optical Communications
and Networking

Flexspander: augmenting expander networks in high-performance systems with optical bandwidth steering

Min Yee Teh,* Zhenguo Wu, and Keren Bergman

Department of Electrical Engineering, Columbia University, New York, New York 10025, USA
**Corresponding author: mt3126@columbia.edu*

Received 2 October 2019; revised 16 December 2019; accepted 22 January 2020; published 28 February 2020 (Doc. ID 379487)

Communication efficiency is one of the deciding factors in determining many of today's high-performance computing (HPC) applications. Traditionally, HPC systems have been on static network topologies, making them inflexible to the variety of skewed traffic demands that may arise due to the spatial locality inherent in many applications. To handle traffic locality, researchers have proposed integrating optical circuit switches (OCSs) into the network architecture, which reconfigures the network topology to alter and dynamically adapt to the predicted traffic. In this paper, we present a novel reconfigurable network topology called Flexspander. Beyond offering a flexible interconnect, Flexspander also offers full flexibility in terms of construction and can be built with any arbitrary combination of commercial electrical packet switches and OCSs. We evaluate Flexspander performance through extensive simulations with multiple network traces, and our results show improved performance for Flexspander over currently proposed static and reconfigurable topologies in terms of the flow completion time.

https://doi.org/10.1364/JOCN.379487

1. INTRODUCTION

Many state-of-the-art high-performance computing (HPC) systems and Cloud data centers (DCs) are typically interconnected through either a hierarchical Clos [1,2] or Dragonfly [3] topology. In the case of the former, Clos (or fat tree) topology can provide multiple paths between compute nodes for efficient load balancing. The latter promises a low network diameter to guarantee low-latency pathways between arbitrary pairs of compute nodes. In the meantime, researchers in the DC networking community have proposed building networks based on expander graphs [4,5]. These networks ensure good network connectivity at scale, even when built with packet switches of modest radices. More importantly, expander networks outperform most other topology classes at the same cost [6], which makes them cheaper alternatives to build. Regardless of the interconnect topology, statically wired networks are inherently inflexible and can suffer performance degradation when running workloads that generate highly skewed traffic.

In order to deal with skewed traffic workloads, researchers have begun looking into employing optical circuit switches (OCSs) to introduce reconfigurability to the network topology. Early works in reconfigurable networks were predominantly focused on the DC front, in which researchers proposed augmenting the network with high switching latency OCSs based on microelectromechanical system (MEMS) technology [7,8]. Limited by the high switching latency of earlier MEMS OCSs, these architectures generally operate by serving traffic with large flows that are bandwidth-bound using high-capacity circuits that go through the OCSs, while sending the latency-bound traffic with small flows via a conventional static network. Meanwhile, Flexfly represents the first instance of a reconfigurable network aimed for HPC applications. Unlike earlier works on reconfigurable networks, Flexfly employs silicon photonics (SiP) switches, which have nanosecond-level switching latency.

Regardless of the technology behind the OCS, prior works have been rather limited in that they do very little in addressing how the physical wiring of the network should be done. Specifically, very little work has been done on elaborating the design of flexible interconnects built with any arbitrary combination of electrical packet switch (EPS) and OCS radices. In practice, it is often not possible to purchase a specific combination of hardware radices, especially when system operators are building systems with commercially available switches.

In this paper, we propose a novel reconfigurable topology built upon the foundations of expander networks, called Flexspander. In the Flexspander topology, different pods of EPSs are interconnected via a layer of OCSs. By configuring the switching states of OCSs, different logical connectivities

between network pods can be realized, thus allowing the topology to handle skewed traffic workloads. Within each pod, the EPSs are wired to form an expander network, allowing the intra-pod topology to be resilient to bottlenecks even with a very sparse number of links. Since the intra-pod expander topology can be generated at random (see Jellyfish [4]) and can be formed with an arbitrary number of links per switch, Flexspander gives system operators total freedom when it comes to physical wiring of the network.

Using network traces obtained from real applications, we evaluate the performance of Flexspander and compare it to many other classes of network topologies. Our simulations show that Flexspander is able to outperform the current state-of-the-art reconfigurable topology called Flexfly in terms of flow completion time (FCT). Specifically, when compared to Flexfly, Flexspander is capable of reducing FCT by as much as 93% on average for moderate to low network loads.

The rest of this paper is organized as follows: we begin by discussing the high-level properties and the limitations of expander networks in Section 2. Next, we introduce the Flexspander network topology and discuss its construction in Section 3. Details of the bandwidth steering algorithm are fleshed out in Section 4. The performance of Flexspander, alongside other baseline topologies, is simulated and analyzed in Section 5.

2. EXPANDER NETWORKS

Expander networks are built based on the mathematical concept of expander graphs, which falls into a class of graphs with high edge expansion at any given degree. The edge expansion metric essentially measures the weakest bisection bandwidth of the graph, making it a good measure of how difficult it is to bottleneck the said graph. The high edge expansion property of expander graphs implies that every cut in the graph is traversed by many links. In other words, in an expander graph, the total number of links connecting any set of nodes to the rest of the network is large with respect to the size of that set [5]. This property renders static expander networks a big advantage: diversity of paths for traffic flow. Since there are many paths between any two nodes in an expander graph, traffic can be distributed across the network and is therefore hardly bottlenecked anywhere.

A few expander-graph-based network topology designs have been proposed in the past. For example, random connected graphs such as Jellyfish [4] are shown to exhibit a high degree of edge expansion assuming uniform traffic. Xpander [5], another instance of expander-based network topology that relies on lifting an already existing expander graph, also shows near optimal edge expansion. Using this technique, Xpander DCs can match the performance of today's DCs with about 80%–85% of the switches and are significantly more robust to network changes than today's DCs built with fat trees.

However, neither Jellyfish nor Xpander can accommodate the inherent heterogeneity of network components and account for the traffic skewness that exists in DCs and HPCs [9–11]. In HPC, traffic locality tends to arise from the nature of the underlying computation pattern. Similarly, DC traffic tends to exhibit high degrees of spatial locality (albeit the locality is much less-predictable), rendering these topologies sub-optimal for non-uniform traffic characteristics [12].

3. FLEXSPANDER ARCHITECTURE

A. Topological Structure

Although prior works have shown that expander networks (e.g., Xpander [13], Jellyfish [4]) achieve good network throughput, their evaluations have not considered the performance under highly skewed traffic. Even though expander networks have high edge expansion, which means that they are difficult to bottleneck, the construction of expanders implicitly assumes uniform traffic, which is rarely the case in most production DCs with highly skewed traffic [14–16]. Under highly skewed network traffic, it is likely that expanders would suffer from poor performance due to uneven link utilization.

We therefore propose Flexspander (see Fig. 1), a network topology built upon the foundations of expanders, but also capable of dynamically adapting to skewed traffic. The Flexspander topology is built by inter-connecting a collection of pods via an OCS layer. By reconfiguring the OCS states, different pod-level topologies between pods can be realized.

Meanwhile, each pod comprises a group of EPSs, interconnected "statically" such that the intra-pod topology forms a good expander network. Note that there is no strict definition of what an expander is. A full mesh is defined as the perfect expander, but another regular graph with an identical number of nodes but with sparser edges is also an expander, as long as its edge expansion is high given its degree. Flexspander places no requirements on what the intra-pod level topology should be, as long as it is a well-connected sub-network. In this sense, Flexspander can be viewed as a generalization of the Flexfly topology [10]; in fact, specific instances of Flexspander designs are indeed Flexfly instances.

A Flexspander topology can be uniquely identified with the following parameters:

- r_o—the OCS radix,
- r_e—the EPS network-facing radix,

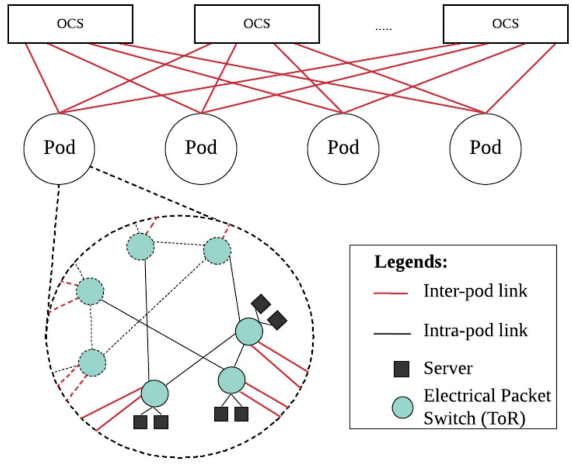

Fig. 1. Illustration of the Flexspander topology. All pods in the network are interconnected via a flexible layer comprising multiple OCSs. Within each pod, EPSs are connected statically as an expander network.

- $[s_i]$—the array of the number of EPSs in pod i,
- σ—the ratio of inter-pod to intra-pod bandwidth.

These parameters are determined at design time by the network operator, based on various design choices including, but not limited to, the target system size (i.e., the number of compute nodes), the port count of commercial packet switches, and the radices of OCSs. Imagine a network operator has decided on using a 32-port Mellanox switch [17] and has decided that each switch will dedicate 6 ports for server-facing connections. This leaves 26 ports for network-facing connections, which we shall denote as r_e. Then, the network designer decides on σ, which controls the ratio of network-facing ports used for inter-pod connection to intra-pod connection. This parameter acts as a design knob, which allows the network operator to adjust for the desired amount of reconfigurable bandwidth (i.e., the number of links wired to OCSs that can be steered to different pods based on traffic demands). The relationship between σ and the number of inter-pod connection ports, $r_{e,\text{inter}}$, and number of intra-pod connection ports, $r_{e,\text{intra}}$, is given as

$$r_{e,\text{inter}} = \max\left(1, r_e * \frac{\sigma}{1+\sigma}\right),$$

$$r_{e,\text{intra}} = \max(2, r_e - r_{e,\text{inter}}).$$

We enforce a minimum number of ports that should be allocated for inter-pod and intra-pod connections. For the inter-pod ports, at least one port per EPS should be dedicated for OCS connection, which is a minimum requirement for reconfigurable network topologies. For intra-pod connections, each switch must dedicate at least two ports, since this is the minimum number of ports needed to form a ring topology, which can be seen as a minimally connected intra-pod topology. Note that as σ gets larger, more of the EPSs' network-facing ports are wired to the OCS layer, so more OCSs may be needed to fully connect the inter-pod links. The tradeoff here is that the intra-pod topology becomes more sparse. Evaluating the opportunity-cost of this tradeoff is up to the network operator to decide.

1. Inter-pod Wiring

The first step to building a Flexspander network is wiring each pod to the OCS layer. The manner in which the OCSs and pod switches are physically wired sets up the physical topology. (We use the terms wiring and striping interchangeably to denote the pattern of physical wiring between network components.) Even though the network performance is determined largely by the logical topology as a result of a specific set of optical switch configurations, the physical topology construction determines the range of achievable logical topologies.

Here, we detail the inter-pod wiring algorithm. The pseudocode is shown in Algorithm 1.

When wiring pods to the OCS layer, we assume that each pods is fully connected. This means that there has to be at least one logical topology in which any arbitrary pod pair is interconnected. We assert later on in the intra-pod topology wiring that every pod must indeed be a strongly connected component. As we will see in Section 3.A.2, building an expander intra-pod topology implicitly guarantees that our assumption here holds.

Algorithm 1. Pseudocode of the algorithm for wiring pods to the OCS layer

```
1:  procedure InterPodWiring
2:    Input:
3:    1) r_o - OCS radix
4:    2) r_e - EPS network-facing radix
5:    3) [s_i] - Number of EPSs in pod i
6:    4) σ - Ratio of inter-pod to intra-pod bandwidth
7:    Output:
8:    1) [y_i^k] - Physical wiring plan
9:    BEGIN
10:     L = [(i, l_i)] ← [(i, 0)]   ∀ i ∈ {1, ..., N}
11:     r'_e = max(1, ⌈r_e * σ/(1+σ)⌉)
12:     for each OCS k do
13:       Sort L in ascending order
14:       for each port in current OCS do
15:         (i, l_i) = L.pop( )
16:         if l_i < s_i * r'_e then
17:           y_i^k ← y_i^k + 1
18:           L.insert((i, l_i + 1))
19:     return [y_i^k]
20:     END
```

When wiring the pods to the OCS layer, we want to ensure that each pod is striped to as many OCSs as possible. The rationale here is that the number of logical links two pods can form is at most the number of OCSs that these two pods are both physically wired to. Another way to understand this is that two pods cannot form logical forms if they are not physically connected to at least one OCS in common. Based on this intuition, we want every pod to be connected to as many different OCSs as possible, and no OCS should ever be wired to the same pod more than once, if possible. The pseudocodes in between lines 14 and 18 essentially aim to allocate the most under-connected pods to the current OCS. That said, we leave the formalized proof of the validity of this argument as an exercise for future work.

2. Intra-pod Wiring

Within each pod, the EPS network is interconnected statically. At design time, the network operator gets to specify σ, which denotes the ratio of bandwidth each pod allocates for inter-pod connections to intra-pod connections. There are two possible cases to consider when wiring intra-pod topology: 1) when the number of intra-pod links per EPS is sparse relative to the pod size and 2) when the number of intra-pod links per EPS is dense relative to the pod size.

When the number of intra-pod links per switch, $r_{e,\text{intra}}$, is sparse relative to the pod size, s_i (i.e., $r_{e,\text{intra}} < s_i - 1$), we cannot form a full mesh. Even so, we would still want the intra-pod network topology to be well connected and robust to bottlenecks. To achieve this, we borrow the powerful concept of expander graphs from graph theory. (Expander graphs are commonly interpreted as sparse approximations of full-mesh

networks [18].) Expanders, due to their high edge expansion, are very resilient to bottlenecks. Constructing good expanders is still an active research thread in graph theory and is outside of the scope of this paper. We refer interested readers to [18–20] for more details on expander construction. It turns out, however, that random graphs tend to be very good expanders, which is the core idea behind the conception of the Jellyfish topology [4]. Flexspander constructs intra-pod EPS networks as expanders, using a wiring technique similar to [4], by generating several random graphs and selecting the graph with the highest edge expansion.

Next, we consider the wiring for the dense case (i.e., $r_{e,\text{intra}} \geq s_i - 1$). The intra-pod wiring can be done by uniformly connecting each switch together, until one or more switches run out of spare links to connect to other switches. This process is then repeated for all other switches within the pod until all switches run out of spare ports for intra-pod connections.

B. Network Controller

In terms of hardware, the Flexspander network can be built with off-the-shelf OCSs [21,22] and does not require specialized OCS hardware to function. However, a Flexspander network built with high-speed OCSs will no doubt be more responsive in its reconfigurability. Most commercial OCSs in today's market have switching speeds on the order of tens of milliseconds. For instance, an Agiltron MEMS Matrix 48×48 Fiber Optical Switch boasts a switching speed of 20 ms. In addition to switching latency, reprogramming the routing tables in response to path changes from a topology reconfiguration may take seconds, as is the case in Google's B4 [23]. This delay could be delayed in smaller scale networks, but we conservatively estimate the overall reconfiguration latency to be on the order of seconds.

The responsibility of deciding when, where, and how the topology reconfiguration should be triggered falls on the network controller, which subsumes the bandwidth steering algorithm. The network controller should also contain means for traffic estimation, which can take the form of a centralized database containing the traffic patterns of applications that have come before. While the network controller does not explicitly need to be integrated with the scheduler, it should be aware of how the new application is physically mapped onto the network in order to estimate a traffic matrix. Using a combination of historical data on applications and current traffic measurements from switches (e.g., using sFlow [24]), an estimated traffic matrix can be computed. Based on the estimated traffic matrix, the network controller can then verify if the new estimate is sufficiently different from the traffic matrix the topology is currently configured for. If the controller deems that there is sufficient difference between the traffic loads, then the bandwidth steering algorithm can be triggered. This overall logic is shown in Fig. 2.

A recent study in [11] reveals that on average, NERSC's Cori supercomputer receives a new job on the order of every 17 s. So, on first-order approximation, the network controller could trigger a reconfiguration every 17 s to set up the network for the incoming application. For DC oriented workloads, a more data-driven approach can be used to determine the suitable frequency to trigger bandwidth steering.

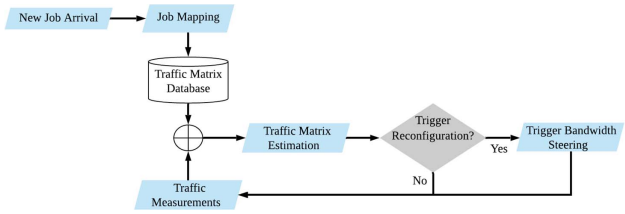

Fig. 2. Proposed network controller for the Flexspander topology. The network controller contains the bandwidth steering algorithm and is responsible for deciding when topology reconfiguration should be triggered.

4. BANDWIDTH STEERING IN FLEXSPANDER

Next, we formalize the bandwidth steering algorithm in a Flexspander network. We first give a naive formulation of the bandwidth steering problem under a known traffic load, which is conceptually straightforward, but has a runtime complexity that scales exponentially with the network size. We then flesh out the details of our bandwidth steering heuristic that has a polynomial time complexity.

Note that the wiring between the EPSs of the same pod are static, as they do not go through a layer of OCSs. Only the inter-pod links are reconfigurable, since only the inter-pod fibers are physically wired to the optical switches. We refer the readers to Table 1 for the collection of recurring mathematical notations used in this section.

A. Naive Approach

Given an inter-pod traffic matrix, $T = [t_{ij}]$, we need to determine how the inter-pod links should be allocated such that we

Table 1. Collection of Mathematical Notations and Their Respective Descriptions

Notation	Description
N	Number of pods in Flexspander
M	Number of OCSs in Flexspander
$T = [t_{ij}] \in \mathbb{R}^{N \times N}$	Traffic matrix, where t_{ij} denotes the traffic rate (Gbps) sent from s_i to s_j
$D = [d_{ij}] \in \mathbb{R}^{N \times N}$	Target logical topology, where d_{ij} denotes the number of s_i egress links connected to ingress links of s_j
$X = [x_{ij}^k] \in \mathbb{Z}^{N \times N \times M}$	Integer number of logical links connecting the egress port of pod i to the ingress port of pod j, via OCS k
$Y = [y_i^k] \in \mathbb{Z}^{N \times M}$	Number of physical links connecting pod i to OCS k
$S = [s_i] \in \mathbb{R}^N$	Number of EPSs in pod i
$A = [a_{ij}] \in \mathbb{R}^{N \times N}$	Cost of connecting an egress port of pod i to an ingress port of pod j
r_o	OCS radix
r_e	EPS radix (Note that this term includes only the network-facing ports.)
σ	Ratio of inter-pod to intra-pod bandwidth

can maximize network throughput. Based on this rationale, we would expect the ideal topology to be one that best "matches" T. This means that the logical topology allocates links between pods proportional to the volume of traffic exchanged between them. In essence, bandwidth steering allows us to trade path-diversity for more short paths between the network hotspots. To compute the logical topology overlay, we can use the following integer linear program (ILP) formulation:

$$\begin{aligned}
&\min_{\mathbf{x}} \sum_{i=1}^{N}\sum_{j=1}^{N}\left(\sum_{k=1}^{M} x_{ij}^{k} - d_{ij}\right)^2 \\
&\text{s.t.}\quad 1)\ \sum_{i=1}^{N} x_{ij}^{k} \le y_{j}^{k} \quad \forall\, 1 \le j \le N,\ 1 \le k \le M, \\
&\qquad 2)\ \sum_{j=1}^{N} x_{ij}^{k} \le y_{i}^{k} \quad \forall\, 1 \le i \le N,\ 1 \le k \le M, \\
&\qquad 3)\ x_{ij}^{k} \text{ is an integer.}
\end{aligned} \tag{1}$$

Note that the objective function in Eq. (1) is a least-squares fit between the logical topology and the target logical topology. Meanwhile, constraints 1 and 2 enforce the physical constraints, in that they ensure that the number of logical links connecting pod i to pod j through OCS k cannot exceed the number of physical wires that connect OCS k to both pods. The optimal solution to Eq. (1) is the one with the shortest euclidean distance to our target. We use a quadratic objective function due to its convexity, which helps the optimization algorithm converge. Unfortunately, computing an optimal logical topology to Eq. (1) is an NP-complete problem [25,26]. Therefore, we need to design a good heuristic to relax the complexity associated with solving Eq. (1); we detail the heuristic next.

B. Bandwidth Steering Heuristic

Our heuristic is split into multiple sub-steps. Note that the key source of the bandwidth steering algorithm's complexity lies in configuring each switch individually such that the overall logical topology best matches our target logical connectivity. To relax this complexity, we solve the overall bandwidth steering problem using two sequential, smaller steps:

1. Compute an ideal target logical topology to strive for by ignoring physical constraints that would otherwise make this computation very challenging.
2. Configure all OCSs such that the overall logical topology is close to the target logical topology.

1. Sub-step 1: Computing the Target Logical Topology

The first step to bandwidth steering is to derive the best "target" logical topology. A target logical topology does not necessarily have to be a realizable logical topology, but merely one that is deemed to be best suited to a given traffic load. This target logical topology then acts as a reference when we configure the individual OCSs such that all the OCSs are configured in a way that the overall logical topology is as close to the target logical topology as possible.

In order to compute the target logical topology, we need to consider the routing as well, since the performance of a topology is intimately tied to how network traffic is routed over the network. In this paper, we assume that the topology is designed based on shortest-path routing. The underlying assumption here is that the optimal target logical topology is the one that is proportionally closest to the traffic matrix, since such a logical connectivity offers the most bandwidth between the hotspots. Given a traffic matrix, T, we then compute the target logical topology, D, as follows:

$$\begin{aligned}
&\min_{D} (d_{ij} - t_{ij})^2 \\
&\text{s.t.}\quad 1)\ \sum_{j=1}^{N} d_{ij} \le \sum_{k=1}^{M} y_{i}^{k} \quad \forall\, 1 \le i \le N, \\
&\qquad 2)\ \sum_{i=1}^{N} d_{ij} \le \sum_{k=1}^{M} y_{j}^{k} \quad \forall\, 1 \le j \le N, \\
&\qquad 3) d_{ij} \in \mathbb{R} \quad \forall\, i,\ j \in \{1, \ldots, N\}.
\end{aligned} \tag{2}$$

Note that constraints 1 and 2 of Eq. (2) merely constrain the ingress and egress degree of each pod, such that the target logical topology does not violate the number of ingress and egress ports that are connected to the OCS layer. While the target logical topology should be defined as a matrix of integer numbers, enforcing this makes the optimization problem hard to solve. We found that there is no reason for the target logical topology to take on integer values, so we drop the integer constraint to reduce complexity.

2. Sub-step 2: Configure the OCSs to Match the Target Logical Topology

Once a target logical topology has been computed, we proceed to configure the OCSs such that the overall inter-pod logical topology is as close to the target logical topology, D, as possible. To do this, we iteratively configure the switch state of each OCS, placing more priority in forming links between pod pairs that are still very far from reaching their target logical connectivity. Modeling each OCS as a bipartite flow graph, as shown in Fig. 3, allows us to configure the switch state of an OCS by solving a min-cost flow problem. Based on the physical wiring pattern of an OCS, Algorithm 2 generates the corresponding bipartite representation.

Algorithm 2. Generate a bipartite flow graph for an OCS

```
1:  procedure GenerateOCSBipartite(o_k)
2:      Initialize empty bipartite graph, G_k
3:      Add dummy source and sink nodes
4:      for each ingress port do
5:          Add node, mark with source pod ID
6:          Add edge from dummy source to node
7:      for each egress port do
8:          Add node, mark with destination pod ID
9:          Add edge from node to dummy sink
10:     for each ingress node do
11:         for each egress node do
12:             Add edge between ingress and egress node
13:     return G_k
```

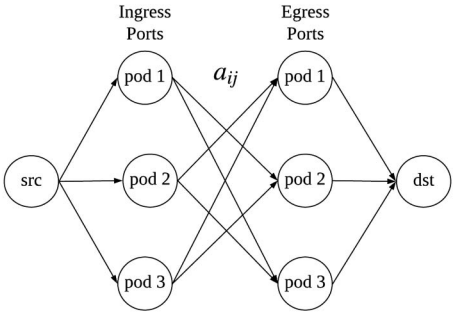

Fig. 3. Each optical circuit switch can be modeled as a bipartite network flow graph, with each ingress/egress port labeled with the source/destination pods.

Next, using the "distance" between the current logical topology and the target logical topology to "score" the flow costs, we can obtain the best switch configuration of each switch. The optimal configuration of each OCS can be solved using Edmonds–Karp, which can be done in polynomial time complexity.

C. Algorithmic Complexity

Solving for an optimal logical topology under a specific traffic matrix has no known polynomial-time algorithm; our heuristic, however, is a polynomial-time algorithm. To bootstrap the system, we need to generate a bipartite representation of each OCS in the form shown in Fig. 3, using Algorithm 2. This needs to be done only once at design time; the bipartite graphs can be stored to disk for reuse. The time complexity for generating all bipartite representations is $O(r * M)$, where r denotes the OCS radix, and M denotes the total number of OCSs in Flexspander.

The bandwidth steering algorithm is triggered every time the network controller decides to reconfigure the logical topology in anticipation of a given traffic matrix. Therefore, the algorithmic complexity must be low. Our bandwidth steering approach comprises two steps (detailed in Sections 4.B.1 and 4.B.2, respectively). The first step involves solving a quadratic program, which can be solved efficiently using commercial optimization tools such as Gurobi [27] and CPLEX [28]. Once the target logical topology has been computed, configuring all of the OCS switches using Algorithm 3 has a runtime of $O(M * r^4)$.

Algorithm 3. Configuring OCS states

```
procedure ConfigOCS({G_k}, D)
    x_ij^k ← 0  ∀ i, j ∈ {1, ..., N}, k ∈ {1, ..., M}   ▷ Initialization
    for k ∈ {1, 2, ..., M} do   ▷ iterate over each OCS
        Solve matching for G_k with a_ij   ∀i, j ∈ {1, 2, ..., N}
    a_ij ← Σ_{k=1}^{M} x_ij^k − d_ij   ▷ Update the matching scores
    return x_ij^k   ∀ i, j ∈ {1, ..., N}, k ∈ {1, ..., M}
```

5. PERFORMANCE SIMULATION

A. Evaluation Methodology

We use Netbench [29], an open-source packet level simulator, to evaluate network performance.

1. Flow Distribution

We incrementally vary the flow arrival rate (also referred to as load), λ, from 9000 to 30,000 arrivals per second for all of our simulations. At each (Poisson) flow arrival, a source and destination for the flow are picked from a traffic probability distribution for each pair of switches. The distribution of flow sizes used in our simulation is based on pFabric's web search [36], as shown in Fig. 4.

2. Benchmark

Our simulations are based on application traces derived in real HPC applications. The types of applications that we simulate are MiniDFT, AMG, and MILC. We also use network traces from a Facebook Hadoop cluster in order to simulate performance for more DC-oriented workloads [12]. In Fig. 5, the switch-to-switch traffic matrices are collected from these traces. We can immediately see a high degree of spatial locality, especially for the AMG and MILC traces. Meanwhile, MiniDFT2 exhibits very little traffic altogether, as seen in the predominantly dark traffic matrix heatmap.

3. Topology

All EPSs have 32 ports (i.e., $r = 32$), and OCSs have 16 (bidirectional) ports. The Flexspander instance used in this paper is formed using eight pods, with each pod being formed using eight EPSs. We compare the performance of Flexspander against Flexfly [10], which represents the state-of-the-art reconfigurable topology that we are aware of. Further, we compare the performance of Flexspander against some other baseline topologies such as Dragonfly [30], High-dimensional Torus (HD Torus), and HyperX [31]. Note that the Flexfly, Dragonfly, HD Torus, and HyperX topology instances are similarly sized, using 8 groups of 8 switches each, and built using 32-port EPSs and 16-port OCSs. Although there are many variants of Dragonfly [32], we use the canonically defined Dragonfly topology, as proposed in the seminal work of Kim *et al.* [3].

4. Routing

We used all three routing algorithms, namely, Yen's K-shortest paths (KSP) algorithm [33], equal-cost multi-path (ECMP) [34], and valiant load balancing (VLB) [35], for our simulations with Flexspander, Flexfly, Dragonfly, HD Torus, HyperX, and Xpander. Specifically, we use $k = 10$ for KSP between all switch pairs in order to take advantage of the path diversity of expander networks [6,13]. However, the performance results in Figs. 6–8 show only the routing that yields the best performance for each topology.

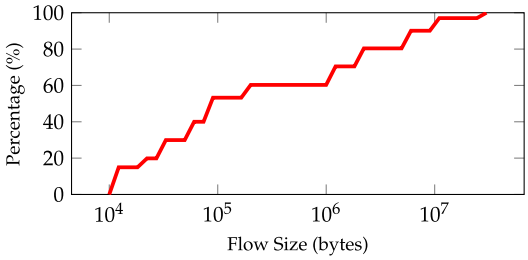

Fig. 4. Flow size distribution used in our simulations. Distribution is drawn from [29].

5. Metrics

In this paper, we evaluate the performance of all topologies using FCT. A higher flow completion time indicates poorer network performance. Based on the simulated flow completion of all the network flows, we analyze the worst-case and average-case performances using the 100th percentile and 50th percentile FCTs, respectively.

B. Simulation Results

In this section, we use the aforementioned evaluation methodology to evaluate Flexspander performance relative to Flexfly, Dragonfly, HyperX, and HD Torus. We assume a 10 Gbps bandwidth for both intra-pod and inter-pod links and an average link latency of 20 ns. The worst-case and median FCTs for all simulated applications using ECMP routing are shown in Fig. 6 and Fig. 7, respectively. We similarly analyze the fraction of flows completed within the simulated time frame; the results are shown in Fig. 8.

We first look at the worst-case FCTs for different topologies. Figure 6 shows that in general, Flexspander performs the best at low to medium load levels, though at high traffic loads, all topology performances converge. This is because at high load levels, the bottlenecked links move towards the injection/ejection links. The superior performance of Flexspander can be accredited to its topological reconfigurability, allowing the topology to dynamically allocate more network capacity to alleviate network hotspots. In other words, Flexspander is able to trade path diversity for more direct capacity between pods that are expected to exchange more traffic, which allows

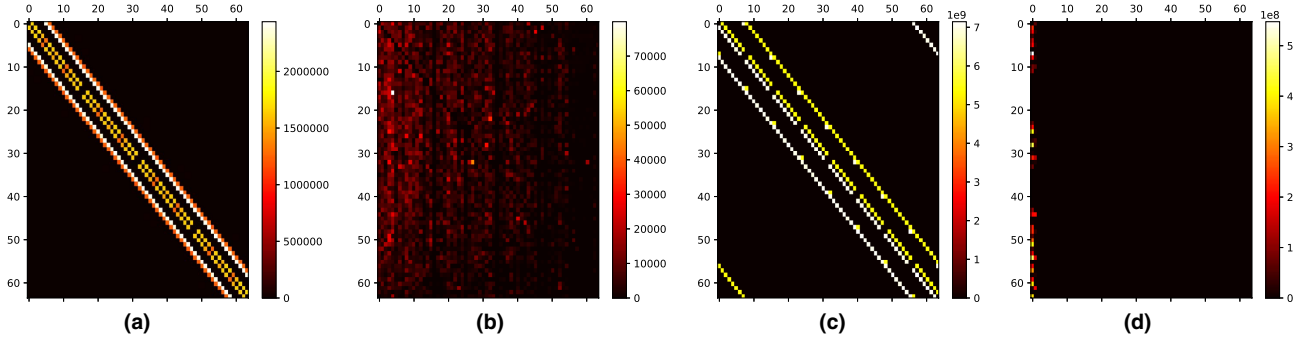

Fig. 5. Switch-to-switch traffic matrices of application traces used in our simulations, represented as 2D heatmaps. (a) AMG, (b) Facebook Hadoop, (c) MILC, and (d) MiniDFT2. The traffic matrices represent an aggregated traffic exchanged over the course of each application. The brighter colors denote heavier traffic.

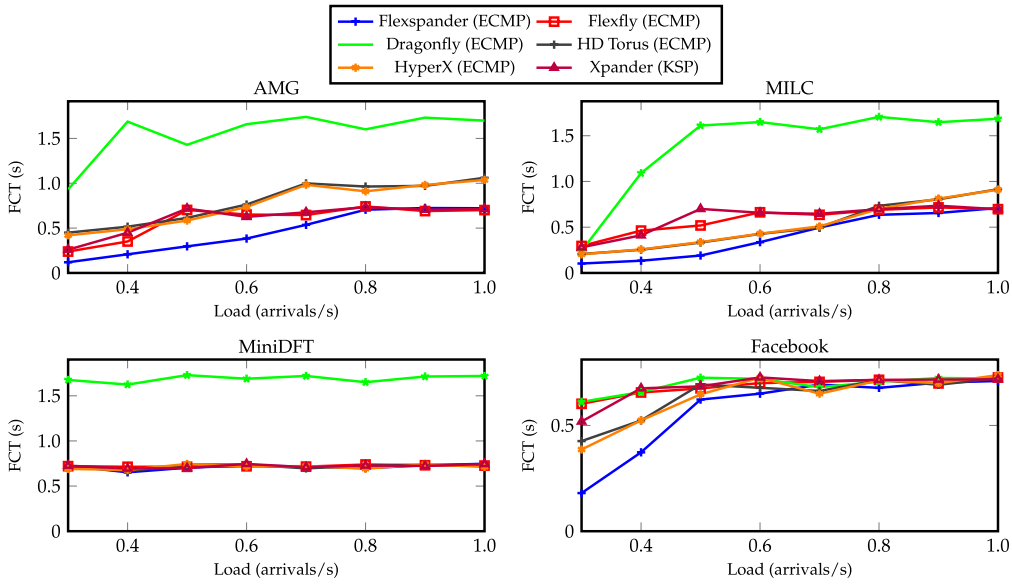

Fig. 6. 100th percentile (i.e., worst-case) FCT simulations for all application-generated traces, comparing Flexspander to similarly sized Dragonfly, Flexfly, Xpander, HD Torus, and HyperX.

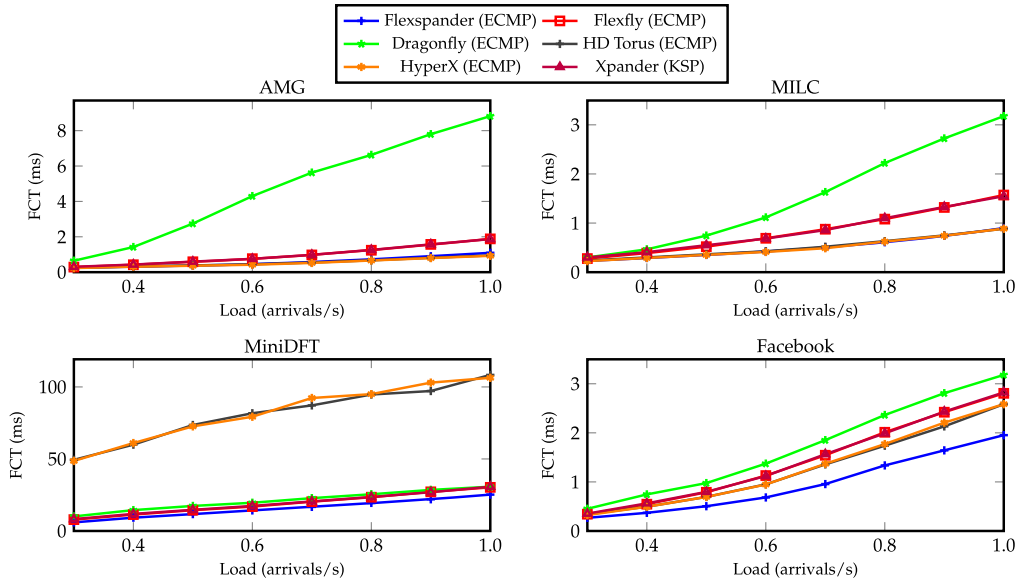

Fig. 7. 50th percentile (i.e., median) FCT simulations for all application-generated traces. The median flow completion time is a good indicator of the average case performance.

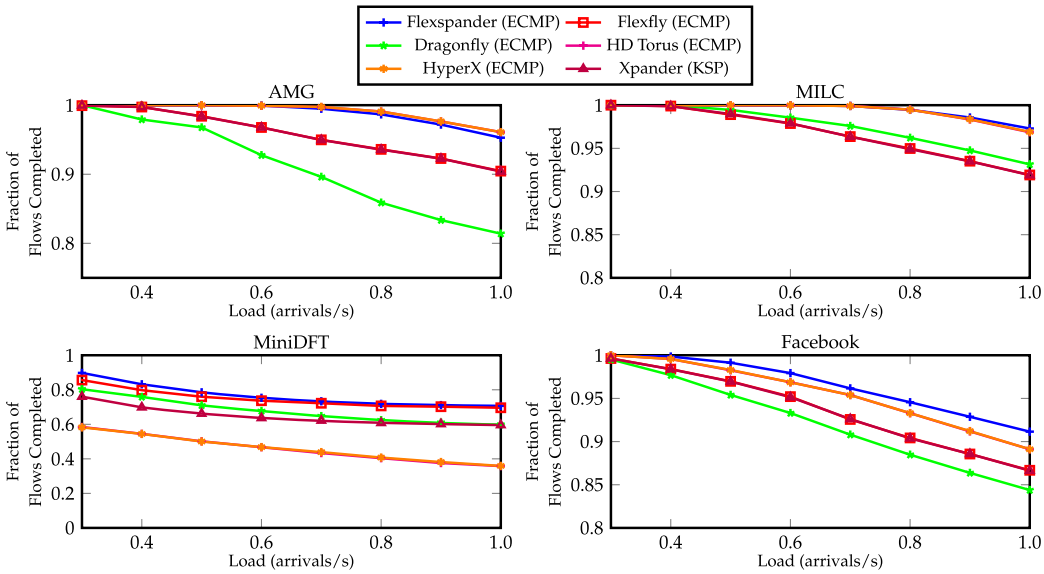

Fig. 8. Fraction of flows completed within a simulated time frame of 1 s. Only flows that are created between 0.25 s and 0.75 s are considered, in order to rule out flows that arrive either 1) too late into the network to complete or 2) too early into a network that has just initialized and uncongested.

these pods to send traffic via shortest paths without the risk of congesting the direct paths. In terms of percentage of flows completed, routing traffic with KSP on an Xpander topology matches the performance of Flexspander, which is routed with ECMP. KSP allows traffic to take more path options to reach its destination and exploit the impressive path diversity of Xpander topologies. That said, Xpander's FCT performance suffers as a result of KSP sending most traffic via non-shortest paths. This may result in an increased routing latency for small flows and an increased overall congestion level, as traffic stays in the network for longer periods of time.

It is worth noting that Flexspander consistently outperforms Flexfly, even though Flexfly possesses a richer intra-pod bandwidth due to its fully connected intra-pod topology. Compared to the Flexfly, Flexspander still offers a plethora of short paths between any two switches within the same pod. Both Figs. 6 and 7 show that Dragonfly generally performs the worst, due to its relatively poor inter-group bandwidth. Since there is only a single link connecting any two Dragonfly groups, the inter-group bandwidth may quickly become overwhelmed when all traffic between two groups is sent along this bottlenecked link.

C. Routing in Flexspander

Next, we evaluate the effects of different routing schemes on the Flexspander topology and investigate the type of routing that would work best alongside a flexible interconnect. Three routing strategies are compared in this experiment: 1) ECMP, 2) KSP, and 3) VLB. Figure 9 shows the effects of different routings on worst-case FCT. Since the trends for the median FCT are similar to the worst-case FCT, we present only the worst-case FCT performance for a detailed discussion.

We can see here that the ECMP performance is comparable to those of KSP and VLB in terms of worst-case FCT. This finding goes against those in [6], in which the authors found that KSP can more effectively utilize the path diversity in expander-based networks to load-balance traffic, therefore resulting in superior performance. Our results show that with a traffic-aware topology, the routing algorithm should prioritize sending traffic along the shortest paths instead of using ECMP. This makes sense, since bandwidth steering allocates more shortest paths between network pods that are expected to exchange more traffic, at the expense of reducing the number of paths between pods that exchange little traffic. Thus, KSP does not fully take advantage of the augmented capacity between pods when it sends more traffic along non-shortest paths, thus nullifying the advantages of bandwidth steering.

This argument similarly applies to VLB. VLB aims to more evenly distribute traffic across the network through the use of a randomly chosen first step, which helps load-balance links better in uniform network topologies. As a result, VLB would not perform as well in Flexspander, as the logical topology already has more capacity between the hotspots, which the router should take advantage of.

6. RELATED WORKS

A. Folded Clos Networks

Folded Clos networks, more commonly known as fat trees [2], have been used widely as interconnects in large-scale HPC systems [37,38]. Fat tree topology benefits from having a regular structure, making routing and load balancing across the network very manageable. Further, fat trees possess a full-bisection bandwidth, making them very resilient under heavy traffic. However, these topologies can be very expensive to deploy, due to the high number of EPSs and long cable length required to build them. Therefore, practical compute systems that employ fat tree networks typically build oversubscribed fat trees instead to save cost, though oversubscribing fat trees may ultimately congest the core layer links when traffic demand soars [39]. Unlike the folded Clos topology, Flexspander is not tiered, and the network resembles expander networks due to its flat hierarchy.

B. Expander Networks

The class of expanders in graph theory has always received much attention from researchers in the field of mathematics and computer science. Some recent works in the networks community have thus proposed building DC topologies as expanders to save cost and improve network performance. Both Jellyfish [4] and Xpander [5] propose interconnecting top-of-rack (ToR) switches in DC networks based on foundations in expander graphs and are shown to have improved performance over today's DC topologies. However, these topologies optimized for uniform traffic demand would demonstrate sub-optimal performance when the inherent heterogeneity of network components and the traffic skewness that exists in DC networks are taken into account. Flexspander attempts to mitigate this by incorporating OCSs into the network, allowing dynamic topology reconfiguration to better serve non-uniform traffic loads.

C. Reconfigurable Networks

To handle skewed traffic patterns that may bottleneck a uniform network topology, some prior research has proposed using OCSs to dynamically change the logical topology. For

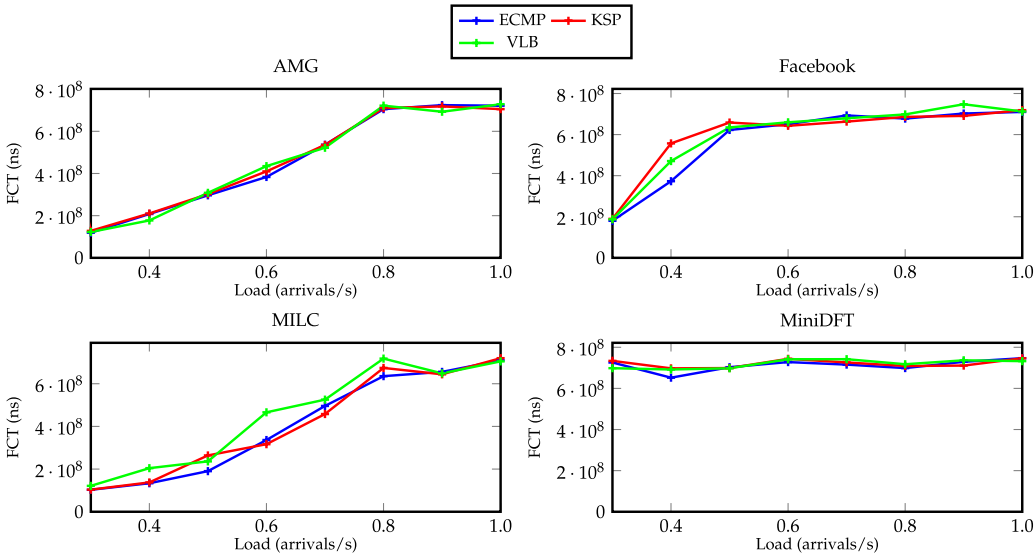

Fig. 9. 100th percentile FCT for Flexspander using different routing schemes.

large-scale networks (DCs or supercomputers), many proposed approaches employ OCSs for "elephant" flows to mitigate the load on EPSs [7,8,40,41]. The Flexfly architecture exploits optical switching to reallocate bandwidth between groups according to application needs [10]. With transparent optical switching, Flexfly reconfigures the inter-group topology based on the traffic pattern, stealing additional direct bandwidth for communication-intensive group pairs. Compared to prior literature, Flexspander allows network operators to explore different relative proportions of reconfigurable capacity with static capacity and determine the best design for each use case.

More recently, researchers have also proposed integrating optical switching with conventional fat tree topologies for HPC systems [11]. Specifically, the authors proposed adding a layer of OCSs between the ToR and aggregation layers to enable packets to traverse shorter paths to their destinations and prevent heavy traffic from congesting the core links.

More recently, researchers have also proposed reconfigurable network architectures built with wireless technologies, based on either free-space optics (FSO) [42,43] or radio waves [44]. Compared to OCS-based approaches, these wireless architectures offer much higher connection flexibility, since any arbitrary pair of tranceivers can form links. However, steering wireless "links" between tranceivers presents a significant challenge in practice, as it requires highly sophisticated hardware and software controllers to ensure proper steering. Further, wireless link quality can be affected easily by a variety of environmental factors that may arise in either a DC or a supercomputer facility, causing reliability issues that may ultimately lead to performance deterioration.

7. CONCLUDING REMARKS

Many state-of-the-art network topology architectures currently employed for DCs and HPCs assume a uniform traffic pattern when being designed, and most of them employ static topologies that can match only a limited number of traffic patterns. Thus, when these networks encounter highly skewed and localized traffic, their performance may become degraded. In this work, we proposed Flexspander network architecture, which integrates OCSs into expander networks to achieve a flexible reconfiguration. We demonstrated through simulations that Flexspander topology would offer valuable advantages over current static and reconfigurable topological designs such as Dragonfly and Flexfly in terms of its speed and flexibility [45].

Funding. Advanced Research Projects Agency - Energy (DE-AR00000843); Hewlett-Packard Development Company; U.S. Department of Energy (B621301); National Security Agency (FA8075-14-D-0002-0007, TAT 15-1158).

Acknowledgment. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied by the sponsoring agencies.

REFERENCES

1. Oak Ridge National Laboratory Computing Facility, https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/.
2. C. E. Leiserson, "Fat-trees: universal networks for hardware-efficient supercomputing," IEEE Trans. Comput. **C-34**, 892–901 (1985).
3. J. Kim, W. J. Dally, S. Scott, and D. Abts, "Technology-driven, highly-scalable dragonfly topology," in *35th International Symposium on Computer Architecture (ISCA)* (IEEE, 2008), pp. 77–88.
4. A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey, "Jellyfish: networking data centers randomly," in *Networked Systems Design and Implementation (NSDI)* (2012).
5. A. Valadarsky, G. Shahaf, M. Dinitz, and M. Schapira, "Xpander: towards optimal-performance datacenters," in *12th International on Conference on Emerging Networking EXperiments and Technologies* (ACM, 2016), pp. 205–219.
6. S. Kassing, A. Valadarsky, G. Shahaf, M. Schapira, and A. Singla, "Beyond fat-trees without antennae, mirrors, and disco-balls," in *SIGCOMM* (ACM, New York, USA, 2017), pp. 281–294.
7. N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat, "Helios: a hybrid electrical/optical switch architecture for modular data centers," in *SIGCOMM* (2011).
8. G. Wang, D. G. Andersen, M. Kaminsky, K. Papagiannaki, T. Ng, M. Kozuch, and M. Ryan, "C-through: part-time optics in data centers," in *SIGCOMM* (2011).
9. S. Kamil, A. Pinar, D. Gunter, M. Lijewski, L. Oliker, and J. Shalf, "Reconfigurable hybrid interconnection for static and dynamic scientific applications," in *4th international Conference on Computing Frontiers* (ACM, 2007), pp. 183–194.
10. K. Wen, P. Samadi, S. Rumley, C. P. Chen, Y. Shen, M. Bahadori, K. Bergman, and J. Wilke, "Flexfly: enabling a reconfigurable dragonfly through silicon photonics," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (IEEE, 2016), pp. 166–177.
11. G. Michelogiannakis, Y. Shen, M. Y. Teh, X. Meng, B. Aivazi, T. Groves, J. Shalf, M. Glick, M. Ghobadi, L. Dennison, and K. Bergman, "Bandwidth steering in HPC using silicon nanophotonics," in *International Conference for High Performance Computing, Networking, Storage and Analysis* (ACM, 2019), article 41.
12. A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," ACM SIGCOMM Comput. Commun. Rev. **45**, 123–137 (2015).
13. A. Valadarsky, M. Dinitz, and M. Schapira, "Xpander: unveiling the secrets of high-performance datacenters," in *14th ACM Workshop on Hot Topics in Networks* (2015).
14. T. Benson, A. Anand, A. Akella, and M. Zhang, "Understanding data center traffic characteristics," in *1st ACM Workshop on Research on Enterprise Networking* (2009).
15. S. Kandula, J. Padhye, and P. Bahl, "Flyways to de-congest data center networks," in *Hot Topics in Networks (HotNets)* (2009).
16. S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: measurements & analysis," in *9th ACM SIGCOMM Conference on Internet Measurement* (2009).
17. Mellanox, "Sn2700," https://store.mellanox.com/categories/switches/ethernet-switches/sn2000.html.
18. D. Spielman, "Spectral graph theory," in *Combinatorial Scientific Computing* (Yale University, 2009), pp. 740–776.
19. O. Reingold, S. Vadhan, and A. Wigderson, "Entropy waves, the zig-zag graph product, and new constant-degree expanders," Ann. Math. **155**, 157–187 (2002).
20. A. Lubotzky, R. Phillips, and P. Sarnak, "Ramanujan graphs," Combinatorica **8**, 261–277 (1988).
21. G. Margulis, "Explicit construction of expanders," Probl. Peredachi Inf. **9**, 71–80 (1973).
22. CALIENT Technologies, Inc., https://www.calient.net/.
23. Agiltron, https://agiltron.com/.
24. S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: experience with a globally-deployed software

defined wan," ACM SIGCOMM Comput. Commun. Rev. **43**, 3–14 (2013).
25. M. Wang, B. Li, and Z. Li, "sFlow: towards resource-efficient and agile service federation in service overlay networks," in *24th International Conference on Distributed Computing Systems* (IEEE, 2004), pp. 628–635.
26. R. W. Irving and M. R. Jerrum, "Three-dimensional statistical data security problems," SIAM J. Comput. **23**, 170–184 (1994).
27. K.-T. Foerster, M. Ghobadi, and S. Schmid, "Characterizing the algorithmic complexity of reconfigurable data center architectures," in *Symposium on Architectures for Networking and Communications Systems* (ACM, 2018), pp. 89–96.
28. Gurobi Optimization, LLC, "Gurobi optimizer reference manual," 2019, http://www.gurobi.com.
29. "IBM ILOG CPLEX Version 12.1 User's Manual" (IBM, 2009), p. 157.
30. Netbench, https://github.com/ndal-eth/netbench.
31. G. Faanes, A. Bataineh, D. Roweth, T. Court, E. Froese, B. Alverson, T. Johnson, J. Kopnick, M. Higgins, and J. Reinhard, "Cray Cascade: a scalable HPC system based on a Dragonfly network," in *International Conference on High Performance Computing, Networking, Storage and Analysis* (IEEE, 2012), pp. 1–9.
32. J. H. Ahn, N. Binkert, A. Davis, M. McLaren, and R. S. Schreiber, "HyperX: topology, routing, and packaging of efficient large-scale networks," in *Conference on High Performance Computing, Networking, Storage and Analysis* (2009).
33. M. Y. Teh, J. J. Wilke, K. Bergman, and S. Rumley, "Design space exploration of the Dragonfly topology," in *International Conference on High Performance Computing* (2017).
34. J. Y. Yen, "Finding the *K* shortest loopless paths in a network," Manage. Sci. **17**, 712–716 (1971).
35. S. Mithyantha, "Systems and methods for using ECMP routes for traffic distribution," US patent 8,972,602 (March 3, 2015).
36. R. Zhang-Shen, "Valiant load-balancing: building networks that can support all traffic matrices," in *Algorithms for Next Generation Networks* (Springer, 2010), pp. 19–30.
37. M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pFabric: minimal near-optimal datacenter transport," ACM SIGCOMM Comput. Commun. Rev. **43**, 435–446 (2013).
38. A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Vl2: a scalable and flexible data center network," in *SIGCOMM* (2009).
39. M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," ACM SIGCOMM Comput. Commun. Rev. **38**, 63–74 (2008).
40. T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *10th ACM SIGCOMM Conference on Internet Measurement* (ACM, 2010), pp. 267–280.
41. A. Chatzieleftheriou, S. Legtchenko, H. Williams, and A. Rowstron, "Larry: practical network reconfigurability in the data center," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, Renton, Washington, USA, 2018, pp. 141–156.
42. G. Porter, R. Strong, N. Farrington, A. Forencich, C.-S. Pang, T. Rosing, Y. Fainman, G. Papen, and A. Vahdat, "Integrating microsecond circuit switching into the data center," ACM SIGCOMM Comput. Commun. Rev. **43**, 447–458 (2013).
43. M. Ghobadi, R. Mahajan, A. Phanishayee, N. Devanur, J. Kulkarni, G. Ranade, P.-A. Blanche, H. Rastegarfar, M. Glick, and D. Kilper, "Projector: agile reconfigurable data center interconnect," in *SIGCOMM* (2016).
44. N. Hamedazimi, Z. Qazi, H. Gupta, V. Sekar, S. R. Das, J. P. Longtin, H. Shah, and A. Tanwer, "Firefly: a reconfigurable wireless data center fabric using free-space optics," in *SIGCOMM* (2014), pp. 319–330.
45. X. Zhou, Z. Zhang, Y. Zhu, Y. Li, S. Kumar, A. Vahdat, B. Y. Zhao, and H. Zheng, "Mirror mirror on the ceiling: flexible wireless links for data centers," in *SIGCOMM* (2012).