Explicit Path Control in Commodity Data Centers: Design and Applications

Shuihai Hu, Kai Chen, Haitao Wu, Wei Bai, Chang Lan, Hao Wang, Hongze Zhao, and Chuanxiong Guo

***Abstract*—Many data center network (DCN) applications require explicit routing path control over the underlying topologies. In this paper, we present XPath, a simple, practical and readily-deployable way to implement explicit path control, using existing commodity switches. At its core, XPath explicitly identifies an end-to-end path with a path ID and leverages a two-step compression algorithm to pre-install all the desired paths into IP TCAM tables of commodity switches. Our evaluation and implementation show that XPath scales to large DCNs and is readily-deployable. Furthermore, on our testbed, we integrate XPath into four applications to showcase its utility.**

***Index Terms*—Data center networks, explicit path control, commodity switches.**

I. Introduction

DRIVEN by modern Internet applications and cloud computing, data centers are being built around the world. To obtain high bandwidth and achieve fault tolerance, data center networks (DCNs) are often designed with multiple paths between any two nodes [4], [5], [15], [19], [20], [33]. Equal Cost Multi-Path routing (ECMP) [25] is the state-of-the-art for multipath routing and load-balancing in DCNs [6], [19], [33].

In ECMP, a switch locally decides the next hop from multiple equal cost paths by calculating a hash value, typically from the source and destination IP addresses and transport port numbers. Applications therefore cannot explicitly control the routing path in DCNs.

However, many emerging DCN applications such as provisioned IOPS (input/output operations per second), fine-grained flow scheduling, bandwidth guarantee, etc. [6], [8], [9], [21], [23], [24], [27], [28], [41], [47], all require explicit routing path control over the underlying topologies (Section II).

Many approaches such as source routing [38], MPLS [37], and OpenFlow [31] can enforce explicit path control. However, source routing is not supported in the hardware of the data center switches, which typically only support destination IP based routing. MPLS needs a signaling protocol, i.e., Label Distribution Protocol, to establish the path, which is typically used only for traffic engineering in core networks instead of application-level or flow-level path control. OpenFlow in theory can establish fine-grained routing paths by installing flow entries in the OpenFlow switches via the controller. But in practice, there are practical challenges such as limited flow table size and dynamic flow path setup that need to be addressed (see Section VI for more details).

In order to address the scalability and deployment challenges faced by the above mentioned approaches, this paper presents XPath for flow-level explicit path control. XPath addresses the dynamic path setup challenge by giving a positive answer to the following question: can we pre-install all desired routing paths between any two nodes? Further, XPath shows that we can pre-install all these paths using the destination IP based forwarding TCAM tables of commodity switches.[1]

One cannot enumerate all possible paths in a DCN as the number can be extremely large. However, we observe that DCNs (e.g., [3]–[5], [19], [20], [22]) do not intend to use *all* possible paths but a set of *desired* paths that are sufficient to exploit the topology redundancy (Section II-B). Based on this observation, XPath focuses on pre-installing these desired paths in this paper. Even though, the challenge is that the sheer number of desired paths in large DCNs is still large, e.g., a Fattree ($k = 64$) has over 2^{32} paths among ToRs (Top-of-Rack switches), exceeding the size of IP table with 144 K entries, by many magnitudes.

To tackle the above challenge, we introduce a two-step compression algorithm, i.e., paths to path sets aggregation and path ID assignment for prefix aggregation, which is capable of compressing a large number of paths to a practical number of routing entries for commodity switches (Section III).

To show XPath's scalability, we evaluate it on various well-known DCNs (Section III-C). Our results suggest that XPath effectively expresses tens of billions of paths using only tens of thousands of routing entries. For example, for Fattree(64), we pre-install 4 billion paths using $\sim$ 64 K entries[2]; for HyperX(4,16,100), we pre-install 17 billion paths using $\sim$ 36 K entries. With such algorithm, XPath easily pre-installs all desired paths into IP LPM tables with 144 K entries, while still reserving space to accommodate more paths.

Manuscript received May 23, 2015; accepted September 16, 2015; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor S. Sen. This work was supported by the Hong Kong RGC ECS 26200014, GRF 16203715, and China 973 Program under Grant No. 2014CB340303.

S. Hu, K. Chen, and W. Bai are with Hong Kong University of Science and Technology, Hong Kong (e-mail: shuaa@cse.ust.hk; kaichen@cse.ust.hk; wbaiab@cse.ust.hk).

H. Wu, C. Guo are with Microsoft, Redmond, WA 98052 USA (e-mail: hwu@microsoft.com; chguo@microsoft.com).

C. Lan is with the University of California, Berkeley, Berkeley, CA 94710 USA (e-mail: clan@eecs.berkeley.edu).

H. Wang is with the University of Toronto, Toronto, ON M5S 3G4, Canada (e-mail: wh.sjtu@gmail.com).

H. Zhao is with Duke University, Durham, NC 27708 USA (e-mail: hongze@cs.duke.edu).

Digital Object Identifier 10.1109/TNET.2015.2482988

[1] The recent advances in switching chip technology make it ready to support 144K entries in IP LPM (Longest Prefix Match) tables of commodity switches (e.g., [2], [26]).

[2] The largest routing table size among all the switches.

To demonstrate XPath's deployability, we implement it on both Windows and Linux platforms under the umbrella of SDN, and deploy it on a 3-layer Fattree testbed with 54 servers (Section IV). Our experience shows that XPath can be readily implemented with existing commodity switches. Through basic experiments, we show that XPath handles failure smoothly.

To showcase XPath's utility, we integrate it into four applications (from provisioned IOPS [27] to Map-reduce) to enable explicit path control and show that XPath directly benefits them (Section V). For example, for provisioned IOPS application, we use XPath to arrange explicit path with necessary bandwidth to ensure the IOPS provisioned. For network update, we show that XPath easily assists networks to accomplish switch upgrades at zero traffic loss. For Map-reduce data shuffle, we use XPath to identify non-contention parallel paths in accord with the many-to-many shuffle pattern, reducing the shuffle time by over 3× compared to ECMP.

In a nutshell, the primary contribution of XPath is that it provides a practical, readily-deployable way to pre-install all the desired routing paths between any s-d pairs using existing commodity switches, so that applications only need to choose which path to use without worrying about how to set up the path, and/or the time cost or overhead of setting up the path.

To access XPath implementation scripts, please visit http://sing.cse.ust.hk/projects/XPath.

The rest of the paper is organized as follows. Section II overviews XPath. Section III elaborates XPath algorithm and evaluates its scalability. Section IV implements XPath and performs basic experiments. Section V integrates XPath into applications. Section VI discusses the related work, and Section VII concludes the paper.

II. Motivation and Overview

A. The Need for Explicit Path Control

Case #1: Provisioned IOPS: IOPS are input/output operations per second. Provisioned IOPS are designed to deliver predictable, high performance for I/O intensive workloads, such as database applications, that rely on consistent and fast response times. Amazon EBS provisioned IOPS storage was recently launched to ensure that disk resources are available whenever you need them regardless of other customer activity [27], [36]. In order to ensure provisioned IOPS, there is a need for necessary bandwidth over the network. Explicit path control is required for choosing an explicit path that can provide such necessary bandwidth (Section V-A).

Case #2: Flow scheduling: Data center networks are built with multiple paths [5], [19]. To use such multiple paths, state-of-the-art forwarding in enterprise and data center environments uses ECMP to statically stripe flows across available paths using flow hashing. Because ECMP does not account for either current network utilization or flow size, it can waste over 50% of network bisection bandwidth [6]. Thus, to fully utilize network bisection bandwidth, we need to schedule elephant flows across parallel paths to avoid contention as in [6]. Explicit path control is required to enable such fine-grained flow scheduling, which benefits data intensive applications such as Map-reduce (Section V-D).

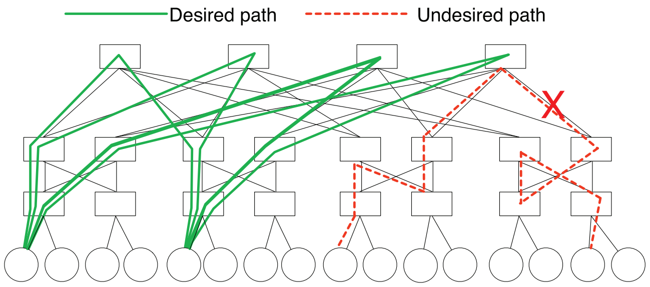


Fig. 1. Example of the desired paths between two servers/ToRs in a 4-radix Fattree topology.

Case #3: Virtual network embedding: In cloud computing, virtual data center (VDC) with bandwidth guarantees is an appealing model for cloud tenants due to its performance predictability in shared environments [8], [21], [47]. To accurately enforce such VDC abstraction over the physical topology with constrained bandwidth, one should be able to explicitly dictate which path to use in order to efficiently allocate and manage the bandwidth on each path (Section V-C).

Besides the above applications, the need for explicit path control has permeated almost every corner of data center designs and applications, from traffic engineering (*e.g.*, [9], [24]), energy-efficiency (*e.g.*, [23]), to network virtualization (*e.g.*, [8], [21], [47]), and so on. In Section V, we will study four of them.

B. XPath Overview

To enable explicit path control for general DCNs, XPath explicitly identifies an end-to-end path with a path ID and attempts to pre-install all desired paths using IP LPM tables of commodity switches, so that DCN applications can use these pre-installed explicit paths easily without dynamically setting up them. In what follows, we first introduce what the desired paths are, and then overview the XPath framework.

Desired paths: XPath does not try to pre-install all possible paths in a DCN because this is impossible and impractical. We observe that when designing DCNs, operators do not intend to use all possible paths in the routing. Instead, they use a set of desired paths which are sufficient to exploit the topology redundancy. This is the case for many recent DCN designs such as [3]–[5], [19], [20], [22], [33]. For example, in a k-radix Fattree [5], they exploit $k^2/4$ parallel paths between any two ToRs for routing (see Fig. 1 for desired/undesired paths on a 4-radix Fattree); whereas in an n-layer BCube [20], they use $(n+1)$ parallel paths between any two servers. These sets of desired paths have already contained sufficient parallel paths between any s-d pairs to ensure good load-balancing and handle failures. As the first step, XPath focuses on pre-installing all these desired paths.

XPath framework: Fig. 2 overviews XPath. As many prior DCN designs [13], [19], [20], [33], [42], in our implementation, XPath employs a logically centralized controller, called *XPath manager*, to control the network. The XPath manager has three main modules: routing table computation, path ID resolution, and failure handling. Servers have client modules for path ID resolution and failure handling.

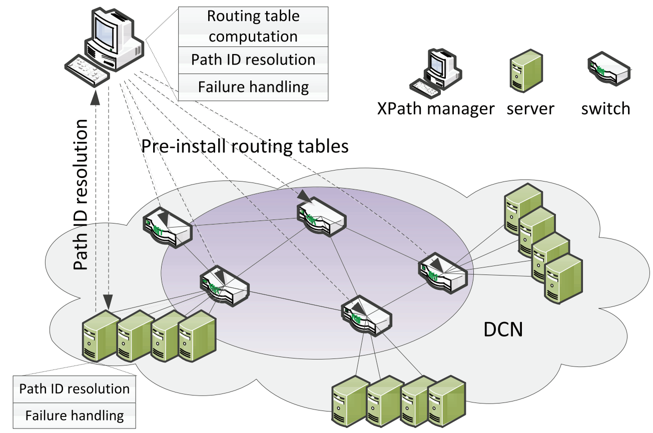


Fig. 2. The XPath system framework.

- *Routing table computation:* This module is the heart of XPath. The problem is how to compress a large number of desired paths (e.g., tens of billions) into IP LPM tables with 144 K entries. To this end, we design a two-step compression algorithm: paths to path sets aggregation (in order to reduce unique path IDs) and ID assignment for prefix aggregation (in order to reduce IP prefix based routing entries). We elaborate the algorithm and evaluate its scalability in Section III.
- *Path ID resolution:* In XPath, path IDs (in the format of 32-bit IP, or called routing IPs[3]) are used for routing to a destination, whereas the server has its own IP for applications. This entails path ID resolution which translates application IPs to path IDs. For this, the XPath manager maintains an IP-to-ID mapping table. Before a new communication, the source sends a request to the XPath manager resolving the path IDs to the destination based on its application IP. The manager may return multiple path IDs in response, providing multiple paths to the destination for the source to select. These path IDs will be cached locally for subsequent communications, but need to be forgotten periodically for failure handling. We elaborate this module and its implementation in Section IV-A.
- *Failure handling:* Upon a link failure, the detecting devices will inform the XPath manager. Then the XPath manager will in turn identify the affected paths and update the IP-to-ID table (i.e., disable the affected paths) to ensure that it will not return a failed path to a source that performs path ID resolution. The XPath source server handles failures by simply changing path IDs. This is because it has cached multiple path IDs for a destination, if one of them fails, it just uses a new live one instead. In the meanwhile, the source will request, from the manager, the updated path IDs to the destination. Similarly, upon a link recovery, the recovered paths will be added back to the IP-to-ID table accordingly. The source is able to use the recovered paths once the local cache expires and a new path ID resolution is performed.

  We note that XPath leverages failure detection and recovery outputs to handle failures. The detailed failure detection and recovery mechanisms are orthogonal to XPath, which focuses on explicit path control. In our implementation (Section IV-B), we adopt a simple TCP sequence based approach for proof-of-concept experiments, and we believe XPath can benefit from existing advanced failure detection and recovery literatures [17], [29].

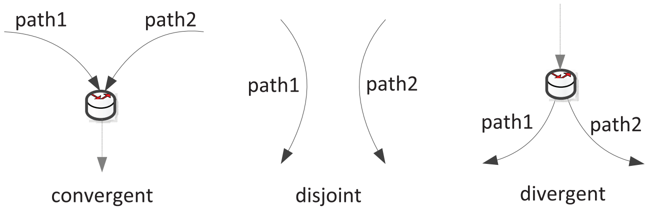


Fig. 3. Three basic relations between two paths.

Remarks: In this paper, XPath focuses on how to pre-install the desired paths, but it does not impose any constraint on how to use the pre-installed paths. On top of XPath, we can either let each server to select paths in a distributed manner, or employ an SDN controller to coordinate path selection between servers or ToRs in a centralized way (which we have taken in our implementation of this paper). In either case, the key benefit is that with XPath we do not need to dynamically modify the switches.

We also note that XPath is expressive and is able to pre-install all desired paths in large DCNs into commodity switches. Thus XPath's routing table recomputation is performed infrequently, and cases such as link failures or switch upgrade [28] are handled through changing path IDs rather than switch table reconfiguration. However, table recomputation is necessary for extreme cases like network wide expansion where the network topology has fundamentally changed.

III. XPath Algorithm and Scalability

We elaborate the XPath two-step compression algorithm in Section III-A and III-B. Then, we evaluate it on various large DCNs to show XPath's scalability in Section III-C.

A. Paths to Path Sets Aggregation (Step I)

The number of desired paths is large. For example, Fat-tree(64) has over 2^{32} paths between ToRs, more than what a 32-bit IP/ID can express. To reduce the number of unique IDs, we aggregate the paths that can share the same ID without causing routing ambiguity into a non-conflict path set, identified by a unique ID.

Then, what kinds of paths can be aggregated? Without loss of generality, two paths have three basic relations between each other, i.e., convergent, disjoint, and divergent as shown in Fig. 3. Convergent and disjoint paths can be aggregated using the same ID, while divergent paths cannot. The reason is straightforward: suppose two paths diverge from each other at a specific switch and they have the same ID $path1 = path2 = path\_id$, then there will be two entries in the routing table: $path\_id \rightarrow port_x$ and $path\_id \rightarrow port_y$, $(x \neq y)$. This clearly leads to ambiguity. Two paths can be aggregated without conflict if they do not cause any routing ambiguity on any switch when sharing the same ID.

[3] We use routing IPs and path IDs interchangeably in this paper.

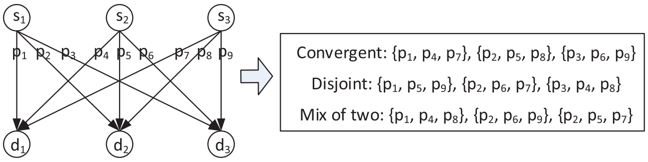

s1
s2
s3
p1 p2 p3 p4 p5 p6 p7 p8 p9
d1
d2
d3
Convergent: {p1, p4, p7}, {p2, p5, p8}, {p3, p6, p9}
Disjoint: {p1, p5, p9}, {p2, p6, p7}, {p3, p4, p8}
Mix of two: {p1, p4, p8}, {p2, p6, p9}, {p2, p5, p7}

Fig. 4. Different ways of path aggregation.

Problem 1: Given the desired paths $P = \{p_1, \cdots, p_n\}$ of a DCN, aggregate the paths into non-conflict path sets so that the number of sets is minimized.

We find that the general problem of paths to non-conflict path sets aggregation is NP-hard since it can be reduced from the Graph vertex-coloring problem [43]. Thus, we resort to practical heuristics.

Based on the relations in Fig. 3, we can aggregate the convergent paths, the disjoint paths, or the mix into a non-conflict path set as shown in Fig. 4. Following this, we introduce two basic approaches for paths to path sets aggregation: convergent paths first approach (CPF) and disjoint paths first approach (DPF). The idea is simple. In CPF, we prefer to aggregate the convergent paths into the path set first until no more convergent path can be added in; Then we can add the disjoint paths, if exist, into the path set until no more paths can be added in. In DPF, we prefer to aggregate the disjoint paths into the path set first and add the convergent ones, if exist, at the end.

The obtained CPF or DPF path sets have their own benefits. For example, a CPF path set facilitates many-to-one communication for data aggregation because such an ID naturally defines a many-to-one communication channel. A DPF path set, on the other hand, identifies parallel paths between two groups of nodes, and such an ID identifies a many-to-many communication channel for data shuffle. In practice, users may have their own preferences to define customized path sets for different purposes as long as the path sets are free of routing ambiguity.

B. *ID Assignment for Prefix Aggregation (Step II)*

While unique IDs can be much reduced through Step I, the absolute value is still large. For example, Fattree(64) has over 2 million IDs after Step I. We cannot allocate one entry per ID flatly with 144 K entries. To address this problem, we further reduce routing entries using ID prefix aggregation. Since a DCN is usually under centralized control and the IDs of paths can be coordinately assigned, our goal of Step II is to assign IDs to paths in such a way that they can be better aggregated using prefixes in the switches.

1) Problem Description: We assign IDs to paths that traverse the same egress port consecutively so that these IDs can be expressed using one entry via prefix aggregation. For example, in Table I, 8 path sets go through a switch with 3 ports. A naïve (bad) assignment will lead to an uncompressable routing table with 7 entries. However, if we assign the paths that traverse the same egress port with consecutive IDs (good), we can obtain a compressed table with 3 entries as shown in Table II.

TABLE I
PATH SET ID ASSIGNMENT

Path set	Egress port	ID assignment (bad)	ID assignment (good)
$pathset_0$	0	0	4
$pathset_1$	1	1	0
$pathset_2$	2	2	2
$pathset_3$	0	3	5
$pathset_4$	1	4	1
$pathset_5$	2	5	3
$pathset_6$	0	6	6
$pathset_7$	0	7	7

TABLE II
COMPRESSED TABLE VIA ID PREFIX AGGREGATION

Path set	ID	Prefix	Egress port
$pathset_{1,4}$	$0, 1$	$00*$	1
$pathset_{2,5}$	$2, 3$	$01*$	2
$pathset_{0,3,6,7}$	$4, 5, 6, 7$	$1{*}{*}$	0

To optimize for a single switch, we can easily achieve the optimal by grouping the path sets according to the egress ports and encoding them consecutively. In this way, the number of entries is equal to the number of ports. However, we optimize for all the switches simultaneously instead of one.

Problem 2: Let $T = \{t_1, t_2, \cdots, t_{|T|}\}$ be the path sets after solving Problem 1. Assigning (or ordering) the IDs for these path sets so that, after performing ID prefix aggregation, the largest number of routing entries among all switches is minimized.

In a switch, a block of consecutive IDs with the same egress port can be aggregated using one entry.[4] We call this an aggregateable ID block ($\mathbf{AIB}$). The number of such $\mathbf{AIB}$s indicates routing states in the switch.[5] Thus, we try to minimize the maximal number of $\mathbf{AIB}$s among all the switches through coordinated ID assignment.

To illustrate the problem, we use a matrix $\mathbf{M}$ to describe the relation between path sets and switches. Suppose switches have k ports (numbered as $1 \ldots k$), then we use $m_{ij} \in [0, k] (1 \leq i \leq |S|, 1 \leq j \leq |T|)$ to indicate whether t_j goes through switch s_i, and if yes, which the egress port is. If $1 \leq m_{ij} \leq k$, it means t_j goes through s_i and the egress port is m_{ij}, and 0 otherwise means t_j does not appear on switch s_i.

$$\mathbf{M} = \begin{array}{c} \\ s_1 \\ s_2 \\ s_3 \\ \vdots \\ s_{|S|} \end{array} \begin{array}{c} \begin{array}{ccccc} t_1 & t_2 & t_3 & \cdots & t_{|T|} \end{array} \\ \begin{pmatrix} m_{11} & m_{12} & m_{13} & \cdots & m_{1|T|} \\ m_{21} & m_{22} & m_{23} & \cdots & m_{2|T|} \\ m_{31} & m_{32} & m_{33} & \cdots & m_{3|T|} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ m_{|S|1} & m_{|S|2} & m_{|S|3} & \cdots & m_{|S||T|} \end{pmatrix} \end{array}$$

To assign IDs to path sets, we use $\mathtt{f}(t_j) = r$ $(1 \leq r \leq |T|)$ to denote that, with an ID assignment $\mathtt{f}$, the ID for t_j is r (or ranks the r-th among all the IDs). With $\mathtt{f}$, we actually permutate columns on $\mathbf{M}$ to obtain $\mathbf{N}$. Column r in $\mathbf{N}$

[4] The consecutiveness has local significance. Suppose path IDs 4, 6, 7 are on the switch (all exit through port p), but 5 are not present, then 4, 6, 7 are still consecutive and can be aggregated as $1 * * \rightarrow p$.

[5] Note that the routing entries can be further optimized using subnetting and supernetting [18], in this paper, we just use $\mathbf{AIB}$s to indicate entries for simplicity, in practice the table size can be even smaller.

corresponds to column t_j in $\mathbf{M}$, i.e., $[n_{1r}, n_{2r}, \ldots, n_{|S|r}]^{\mathbb{T}} = [m_{1j}, m_{2j}, \ldots, m_{|S|j}]^{\mathbb{T}}$.

$$\mathbf{N} = f(\mathbf{M}) = \begin{array}{c} \\ s_1 \\ s_2 \\ s_3 \\ \vdots \\ s_{|S|} \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & \ldots & |T| \end{array} \\ \begin{pmatrix} n_{11} & n_{12} & n_{13} & \cdots & n_{1|T|} \\ n_{21} & n_{22} & n_{23} & \cdots & n_{2|T|} \\ n_{31} & n_{32} & n_{33} & \cdots & n_{3|T|} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ n_{|S|1} & n_{|S|2} & n_{|S|3} & \cdots & n_{|S||T|} \end{pmatrix} \end{array}$$

With matrix $\mathbf{N}$, we can calculate the number of $\mathbf{AIB}$s on each switch. To compute it on switch s_i, we only need to sequentially check all the elements on the i-th row. If there exist sequential non-zero elements that are the same, it means all these consecutive IDs share the same egress port and belong to a same $\mathbf{AIB}$. Otherwise, one more $\mathbf{AIB}$ is needed. Thus, the total number of $\mathbf{AIB}$s on switch s_i is:

$$\mathbf{AIB}(s_i) = 1 + \sum_{r=1}^{|T|-1} (n_{ir} \oplus n_{i(r+1)}) \qquad (1)$$

where $u \oplus v = 1$ if $u \neq v$ (0 is skipped), and 0 otherwise. With (1), we can obtain the maximal number of $\mathbf{AIB}$s among all the switches: $\mathbf{MAIB} = \max_{1 \leq i \leq |S|} \{\mathbf{AIB}(s_i)\}$, and our goal is to find an $\mathtt{f}$ that minimizes $\mathbf{MAIB}$.

2) Solution: ID assignment algorithm: The above problem is NP-hard as it can be reduced from the 3-SAT problem [39]. Thus, we resort to heuristics. Our practical solution is guided by the following thought. Each switch s_i has its own local optimal assignment $\mathtt{f}_i$. But these individual local optimal assignments may conflict with each other by assigning different IDs to a same path set on different switches, causing an ID inconsistency on this path set. To generate a global optimized assignment $\mathtt{f}$ from the local optimal assignments $\mathtt{f}_i$s, we can first optimally assign IDs to path sets on each switch individually, and then resolve the ID inconsistency on each path set in an incremental manner. In other words, we require that each step of ID inconsistency correction introduces minimal increase on $\mathbf{MAIB}$.

Based on the above consideration, we introduce our ID_Assignment$(\cdot)$ in Algorithm 1. The main idea behind the algorithm is as follows.

- *First, we assign IDs to path sets on each switch individually.* We achieve the optimal result for each switch by simply assigning the path sets that have the same egress ports with consecutive IDs (lines 1–2).
- *Second, we correct inconsistent IDs of each path set incrementally.* After the first step, the IDs for a path set on different switches may be different. For any path set having inconsistent IDs, we resolve this as follows: we pick one ID out of all the inconsistent IDs of this path set and let other IDs be consistent with it provided that such correction leads to the minimal $\mathbf{MAIB}$ (lines 4–10). More specifically, in lines 6–9, we try each of the inconsistent IDs, calculate the associated $\mathbf{MAIB}$ if we correct the inconsistency with this ID, and finally pick the one that leads to the minimal $\mathbf{MAIB}$. The algorithm terminates after we resolve the ID inconsistencies for all the path sets.

Algorithm 1 Coordinated ID assignment algorithm

```
ID_Assignment($\mathbf{M}$)  /* $\mathbf{M}$ is initial matrix, $\mathbf{N}$ is output */;
1  foreach row $i$ of $\mathbf{M}$ (i.e., switch $s_i$) do
2      assign path sets $t_j (1 \leq j \leq |T|)$ having the same $m_{ij}$ values (i.e., egress ports) with consecutive IDs;
   /* path sets are optimally encoded on each switch locally, but one path set may have different IDs assigned with respect to different switches */;
3  $\mathbf{M}' \leftarrow \mathbf{M}$ with IDs specified for each $t_j$ in each $s_i$;
4  foreach column $j$ of $\mathbf{M}'$ (i.e., path set $t_j$) do
5      if $t_j$ has inconsistent IDs then
6          let $C = \{c_1, c_2, \cdots, c_k\}, (1 < k \leq |S|)$ be the set of inconsistent IDs;
7          foreach $c \in C$ do
8              tentatively use $c$ to correct the inconsistency by swapping $c_i$ with $c$ on each relevant switch;
9              compute $\mathbf{MAIB}$;
10         ID($t_j$) $\leftarrow c$ with the minimal $\mathbf{MAIB}$;
11 return $\mathbf{N} \leftarrow \mathtt{f}(\mathbf{M}')$; /* $\mathbf{M}'$ is inconsistency-free */
```

In Fig. 5 we use a simple example to walk readers through the algorithm. Given $\mathbf{M}$ with 6 path sets across 3 switches, we first encode each switch optimally. This is achieved by assigning path sets having the same egress port with consecutive IDs. For example, on switch s_1, path sets t_1, t_2, t_3, t_5 exit from $port_1$ and t_4, t_6 from $port_2$, then we encode t_1, t_2, t_3, t_5 with IDs 1, 2, 3, 4 and t_4, t_6 with 5,6 respectively. We repeat this on s_2 and s_3, and achieve $\mathbf{M}'_0$ with $\mathbf{MAIB} = 4$. However, we have inconsistent IDs (marked in red) for all path sets. For example, t_1 has different IDs 1, 3, 1 on s_1, s_2, s_3 respectively. Then, we start to correct the inconsistency for each path set. For t_1 with inconsistent IDs 1, 3, 1, we try to correct with IDs 1 and 3 respectively. To correct with ID 1, we exchange IDs 3 and 1 for t_1 and t_2 on switch s_2, and get $\mathbf{MAIB} = 5$. To correct with ID 3, we exchange IDs 1 and 3 for t_1 and t_3 on switch s_1 and s_3, and get $\mathbf{MAIB} = 4$. We thus choose to correct with ID 3 and achieve $\mathbf{M}'_1$ as it has minimal $\mathbf{MAIB} = 4$. We perform the same operation for the remaining path sets one by one and finally achieve $\mathbf{M}'_6$ with $\mathbf{MAIB} = 4$. Therefore, the final ID assignment is $\mathtt{f} : (t_1, t_2, t_3, t_4, t_5, t_6) \rightarrow (3, 2, 1, 4, 6, 5)$.

We note that the proposed algorithm is not optimal and has room to improve. However, it is effective in compressing the routing tables as we will show in our evaluation. One problem is the time cost as it works on a large matrix. We intentionally designed our Algorithm 1 to be of low time complexity, i.e., $O(|S|^2|T|)$ for the $|S| \times |T|$ matrix $\mathbf{M}$. Even though, we find that when the network scales to several thousands, it cannot return a result within 24 hours (see Table IV). Worse, it is possible that $|S| \sim 10^{4-5}$ and $|T| \sim 10^6$ or more for large DCNs. In such cases, even a linear time algorithm can be slow, not to mention any advanced algorithms.

Speedup with equivalence reduction: To speed up, we exploit DCN topology characteristics to reduce the runtime of our algorithm. The observation is that most DCN topologies are regular

$$\mathbf{M} = \begin{array}{c} \\ s_1 \\ s_2 \\ s_3 \end{array}\!\!\begin{array}{c} \begin{array}{cccccc} t_1 & t_2 & t_3 & t_4 & t_5 & t_6 \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 2 & 1 & 2 \\ 2 & 1 & 1 & 2 & 3 & 4 \\ 1 & 2 & 2 & 2 & 3 & 2 \end{pmatrix} \end{array} \to \mathbf{M}'_0 = \begin{array}{c} \\ s_1 \\ s_2 \\ s_3 \end{array}\!\!\begin{array}{c} \begin{array}{cccccc} t_1 & t_2 & t_3 & t_4 & t_5 & t_6 \end{array} \\ \begin{pmatrix} 1(1) & 1(2) & 1(3) & 2(5) & 1(4) & 2(6) \\ 2(3) & 1(1) & 1(2) & 2(4) & 3(5) & 4(6) \\ 1(1) & 2(2) & 2(3) & 2(4) & 3(6) & 2(5) \end{pmatrix} \end{array} \to \mathbf{M}'_1 = \begin{array}{c} \\ s_1 \\ s_2 \\ s_3 \end{array}\!\!\begin{array}{c} \begin{array}{cccccc} t_1 & t_2 & t_3 & t_4 & t_5 & t_6 \end{array} \\ \begin{pmatrix} 1(3) & 1(2) & 1(1) & 2(5) & 1(4) & 2(6) \\ 2(3) & 1(1) & 1(2) & 2(4) & 3(5) & 4(6) \\ 1(3) & 2(2) & 2(1) & 2(4) & 3(6) & 2(5) \end{pmatrix} \end{array} \to$$

$$\mathbf{M}'_2 = \begin{array}{c} \\ s_1 \\ s_2 \\ s_3 \end{array}\!\!\begin{array}{c} \begin{array}{cccccc} t_1 & t_2 & t_3 & t_4 & t_5 & t_6 \end{array} \\ \begin{pmatrix} 1(3) & 1(2) & 1(1) & 2(5) & 1(4) & 2(6) \\ 2(3) & 1(2) & 1(1) & 2(4) & 3(5) & 4(6) \\ 1(3) & 2(2) & 2(1) & 2(4) & 3(6) & 2(5) \end{pmatrix} \end{array} \to \ldots \to \mathbf{M}'_6 = \begin{array}{c} \\ s_1 \\ s_2 \\ s_3 \end{array}\!\!\begin{array}{c} \begin{array}{cccccc} t_1 & t_2 & t_3 & t_4 & t_5 & t_6 \end{array} \\ \begin{pmatrix} 1(3) & 1(2) & 1(1) & 2(4) & 1(6) & 2(5) \\ 2(3) & 1(2) & 1(1) & 2(4) & 3(6) & 4(5) \\ 1(3) & 2(2) & 2(1) & 2(4) & 3(6) & 2(5) \end{pmatrix} \end{array} \to \mathbf{N} = \begin{array}{c} \\ s_1 \\ s_2 \\ s_3 \end{array}\!\!\begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 2 & 2 & 1 \\ 1 & 1 & 2 & 2 & 4 & 3 \\ 2 & 2 & 1 & 2 & 2 & 3 \end{pmatrix} \end{array}$$

Fig. 5. Walk-through example on Algorithm 1: for any element $x(y)$ in $\mathbf{M_{k'}}$ $(1 \leq k \leq 6)$, x is the egress port and y is the ID assigned to a path set t_j on switch s_i, red/green ys mean inconsistent/consistent IDs for path sets.

TABLE III
RESULTS OF XPATH ON THE 4 WELL-KNOWN DCNS

DCNs	Nodes #	Links #	**Original paths#**	**Max. entries # without compression**	**Path sets # after Step I compression**	**Max. entries # after Step I compression**	**Max. entries # after Step II compression**
Fattree(8)	208	384	**15,872**	**944**	**512**	**496**	**116**
Fattree(16)	1,344	3,072	**1,040,384**	**15,808**	**8,192**	**8,128**	**968**
Fattree(32)	9,472	24,576	**66,977,792**	**257,792**	**131,072**	**130,816**	**7,952**
Fattree(64)	70,656	196,608	**4,292,870,144**	**4,160,512**	**2,097,152**	**2,096,128**	**64,544**
BCube(4, 2)	112	192	**12,096**	**576**	**192**	**189**	**108**
BCube(8, 2)	704	1,536	**784,896**	**10,752**	**1,536**	**1,533**	**522**
BCube(8, 3)	6,144	16,384	**67,092,480**	**114,688**	**16,384**	**16,380**	**4,989**
BCube(8, 4)	53,248	163,840	**5,368,545,280**	**1,146,880**	**163,840**	**163,835**	**47,731**
VL2(20, 8, 40)	1,658	1,760	**31,200**	**6,900**	**800**	**780**	**310**
VL2(40, 16, 60)	9,796	10,240	**1,017,600**	**119,600**	**6,400**	**6,360**	**2,820**
VL2(80, 64, 80)	103,784	107,520	**130,969,600**	**4,030,400**	**102,400**	**102,320**	**49,640**
VL2(100, 96, 100)	242,546	249,600	**575,760,000**	**7,872,500**	**240,000**	**239,900**	**117,550**
HyperX(3, 4, 40)	2,624	2,848	**12,096**	**432**	**192**	**189**	**103**
HyperX(3, 8, 60)	31,232	36,096	**784,896**	**4,032**	**1,536**	**1,533**	**447**
HyperX(4, 10, 80)	810,000	980,000	**399,960,000**	**144,000**	**40,000**	**39,996**	**8,732**
HyperX(4, 16, 100)	6,619,136	8,519,680	**17,179,607,040**	**983,040**	**262,144**	**262,140**	**36,164**

and many nodes are equivalent (or symmetric). These equivalent nodes are likely to have similar numbers of routing states for any given ID assignment, especially when the path sets are symmetrically distributed. The reason is that for two equivalent switches, if some path sets share a common egress port on one switch, most of these path sets, if not all, are likely to pass through a common egress port on another switch. As a result, no matter how the path sets are encoded, the ultimate routing entries on two equivalent switches tend to be similar. Thus, our hypothesis is that, by picking a representative node from each equivalence node class, we can optimize the routing tables for all the nodes in the topology while spending much less time.

Based on the hypothesis, we improve the runtime of Algorithm 1 with equivalence reduction. This speedup makes no change to the basic procedure of Algorithm 1. Instead of directly working on $\mathbf{M}$ with $|S|$ rows, the key idea is to derive a smaller $\mathbf{M}^*$ with fewer rows from $\mathbf{M}$ using equivalence reduction, i.e., for all the equivalent nodes s_is in $\mathbf{M}$ we only pick one of them into $\mathbf{M}^*$, and then apply ID_Assignment($\cdot$) on $\mathbf{M}^*$. To this end, we first need to compute the equivalence classes among all the nodes, and there are many fast algorithms available for this purpose [12], [16], [30]. This improvement enables our algorithm to finish with much less time for various well-known DCNs while still maintaining good results as we will show subsequently.

C. Scalability Evaluation

Evaluation setting: We evaluate XPath's scalability on 4 well-known DCNs: Fattree [5], VL2 [19], BCube [20], and HyperX [4]. Among these DCNs, BCube is a server-centric structure where servers act not only as end hosts but also relay nodes for each other. For the other 3 DCNs, switches are the only relay nodes and servers are connected to ToRs at last hop. For this reason, we consider the paths between servers in BCube and between ToRs in Fattree, VL2 and HyperX.

For each DCN, we vary the network size (Table III). We consider $k^2/4$ paths between any two ToRs in Fattree(k), $(k+1)$ paths between any two servers in BCube(n, k), D_A paths between any two ToRs in VL2(D_A, D_I, T), and L paths between any two ToRs in HyperX(L, S, T).[6] These paths do not enumerate all possible paths in the topology, however, they cover all desired paths sufficient to exploit topology redundancy in each DCN.

Our scalability experiments run on a Windows server with an Intel Xeon E7-4850 2.00 GHz CPU and 256 GB memory.

Main results: Table III shows the results of XPath algorithm on the 4 well-known DCNs, which demonstrates XPath's high scalability. Here, for paths to path sets aggregation we used CPF.

We find that XPath can effectively pre-install up to tens of billions of paths using tens of thousands of routing entries for very large DCNs. Specifically, for Fattree(64) we express 4 billion paths with 64 K entries; for BCube(8,4) we express 5 billion paths with 47 K entries; for VL2(100,96,100) we express 575 million paths with 117 K entries; for HyperX(4,16,100) we express 17 billion paths with 36 K entries. These results suggest that XPath can easily pre-install all desired paths into IP LPM

[6] DCNs use different parameters to describe their topologies. In Fattree(k), k is the number of switch ports; in BCube(n, k), n is the number of switch ports and k is the BCube layers; in VL2(D_A, D_I, T), D_A/D_I are the numbers of aggregation/core switch ports and T is the number of servers per rack; in HyperX(L, S, T), L is the number of dimensions, S is the number of switches per dimension, and T is the number of servers per rack.

TABLE IV
TIME COST OF ID ASSIGNMENT ALGORITHM WITH AND WITHOUT EQUIVALENCE REDUCTION FOR THE 4 DCNs

DCNs	Time cost (Second)	
	No equivalence reduction	Equivalence reduction
Fattree(16)	8191.121000	0.078000
Fattree(32)	>24 hours	4.696000
Fattree(64)	>24 hours	311.909000
BCube(8, 2)	365.769000	0.046000
BCube(8, 3)	>24 hours	6.568000
BCube(8, 4)	>24 hours	684.895000
VL2(40, 16, 60)	227.438000	0.047000
VL2(80, 64, 80)	>24 hours	3.645000
VL2(100, 96, 100)	>24 hours	28.258000
HyperX(3, 4, 40)	0.281000	0.000000
HyperX(4, 10, 80)	>24 hours	10.117000
HyperX(4, 16, 100)	>24 hours	442.379000

tables with 144 K entries, and in the meanwhile XPath is still able to accommodate more paths before reaching 144 K.

Understanding the ID assignment: The most difficult part of the XPath compression algorithm is Step II (i.e., ID assignment), which eventually determines if XPath can pre-install all desired paths using 144 K entries. The last two columns of Table III contrast the maximal entries before and after our coordinated ID assignment for each DCN.

We find that XPath's ID assignment algorithm can efficiently compress the routing entries by 2 × to 32 × for different DCNs. For example, before our coordinated ID assignment, there are over 2 million routing entries in the bottleneck switch (i.e., the switch with the largest routing table size) for Fattree(64), and after it, we achieve 64K entries via prefix aggregation. In the worst case, we still compress the routing states from 240 K to 117 K in VL2(100,96,100). Furthermore, we note that the routing entries can be further compressed using traditional Internet IP prefix compression techniques, e.g., [18], as a post-processing step. Our ID assignment algorithm makes this prefix compression more efficient.

We note that our algorithm has different compression effects on different DCNs. As to the 4 largest topologies, we achieve a compression ratio of $\frac{2{,}096{,}128}{64{,}544} = 32.48$ for Fattree(64), $\frac{262{,}140}{36{,}164} = 7.25$ for HyperX(4,16,100), $\frac{163{,}835}{47{,}731} = 3.43$ for BCube(8,4), and $\frac{239{,}900}{117{,}550} = 2.04$ for VL2(100,96,100) respectively. We believe one important decisive factor for the compression ratio is the density of the matrix $\mathbf{M}$. According to (1), the number of routing entries is related to the non-zero elements in $\mathbf{M}$. The sparser the matrix, the more likely we achieve better results. For example, in Fattree(64), a typical path set traverses $\frac{1}{32}$ aggregation switches and $\frac{1}{1024}$ core switches, while in VL2(100,96,100), a typical path set traverses $\frac{1}{2}$ aggregation switches and $\frac{1}{50}$ core switches. This indicates that $\mathbf{M}_{\text{Fattree}}$ is much sparser than $\mathbf{M}_{\text{VL2}}$, which leads to the effect that the compression on Fattree is better than that on VL2.

Time cost: In Table IV, we show that equivalence reduction speeds up the runtime of the ID assignment algorithm. For example, without equivalence reduction, it cannot return an output within 24 hours when the network scales to a few thousands. With it, we can get results for all the 4 DCNs within a few minutes even when the network becomes very large. This is acceptable because it is one time pre-computation and we do not require routing table re-computation as long as the network topology does not change.

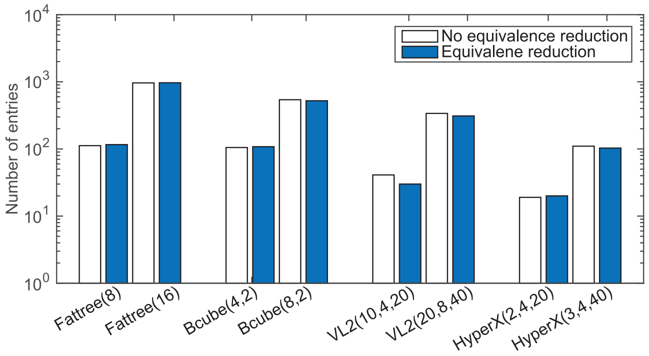


Fig. 6. Effect of ID assignment algorithm with and without equivalence reduction for the 4 DCNs.

Effect of equivalence reduction: In Fig. 6, we compare the performance of our ID assignment with and without equivalence reduction. With equivalence reduction, we use $\mathbf{M}^*$ (i.e., part of $\mathbf{M}$) to perform ID assignment, and it turns out that the results are similar to that without equivalence reduction. This partially validates our hypothesis in Section III-B2. Furthermore, we note that the algorithm with equivalence reduction can even slightly outperform that without it in some cases. This is not a surprising result since both algorithms are heuristic solutions to the original problem.

Results on randomized DCNs: We note that most other DCNs such as CamCube [3] and CiscoDCN [15] are regular and XPath can perform as efficiently as above. However, in some recent work such as Jellyfish [41] and SWDC [40], the authors also discussed random graphs for DCN topologies. XPath's performance is unpredictable for random graphs. But for all the Jellyfish topologies we tested, in the worst case, XPath still manages to compress over 1.8 billion paths with less than 120K entries. The runtime varies from tens of minutes to hours or more depending on the degree of symmetry of the random graph.

CPF vs DPF: To make a comparison between CPF and DPF (Section III-A), we study two compression ratios in Fig. 7, i.e., maximum routing entries without compression to maximum routing entries after Step I compression (MRE#-0/MRE#-1), and maximum routing entries after Step I compression to maximum routing entries after Step II compression (MRE#-1/MRE#-2). We make the following observations.

First, as to the paths to path sets aggregation (Step I compression), for all the 4 DCNs, CPF has a higher compression ratio than DPF. One reason is that a CPF path set can possibly hold more paths than a DPF path set. For example, in Fattree, we observe that a CPF path set contains the convergent paths to one destination node from all the other nodes as sources, while a DPF path set contains the paths from half of the nodes as sources to the other half as destinations.

Second, as to the ID assignment for prefix aggregation (Step II compression), we find that CPF has a higher ratio for Fattree and HyperX while DPF has a higher ratio for VL2. One reason for this is that, as mentioned above, this compression ratio has a correlation with the density of matrix $\mathbf{M}$. The path

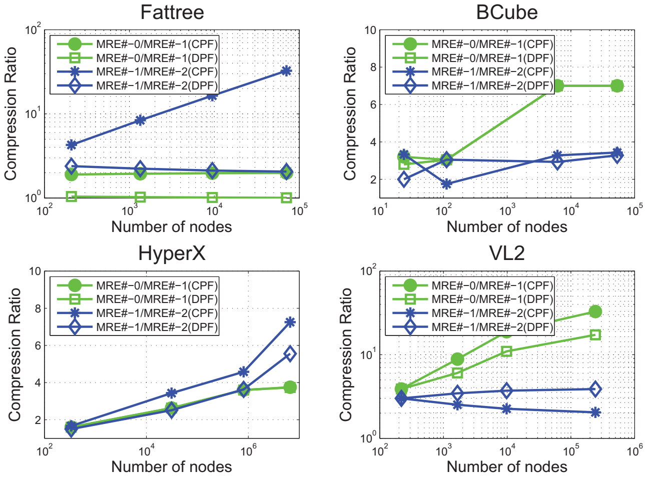


Fig. 7. CPF vs DPF: the compression ratio of maximum routing entries without compression to maximum routing entries after Step I compression (MRE#-0/MRE#-1) and the compression ratio of maximum routing entries after Step I compression to maximum routing entries after Step II compression (MRE#-1/MRE#-2).

sets generated by CPF form a sparser $\mathbf{M}$ in Fattree and HyperX, while the path sets computed by DPF lead to a sparser $\mathbf{M}$ in VL2.

Third, the overall compression effect of CPF is better than that of DPF for the 4 DCN topologies we have evaluated. However, we believe there also exist topologies where DPF has better performance.

IV. Implementation and Experiments

We have implemented XPath on both Windows and Linux platforms, and deployed it on a 54-server Fattree testbed with commodity switches for experiments. This paper describes the implementation on Windows. In what follows, we first introduce path ID resolution (Section IV-A) and failure handling (Section IV-B). Then, we present testbed setup and basic XPath experiments (Section IV-C).

A. Path ID Resolution

As introduced in Section II-B, path ID resolution addresses how to resolve the path IDs (i.e., routing IPs) for a destination. To achieve fault-tolerant path ID resolution, there are two issues to consider. First, how to distribute the path IDs of a destination to the source. The live paths to the destination may change, for example, due to link failures. Second, how to choose the path for a destination, and enforce such path selection in existing networks.

These two issues look similar to the name resolution in existing DNS. In practice, it is possible to return multiple IPs for a server, and balance the load by returning different IPs to the queries. However, integrating the path ID resolution of XPath into existing DNS may challenge the usage of IPs, as legacy applications (on socket communication) may use IPs to differentiate the servers instead of routing to them. Thus, in this paper, we develop a clean-slate XPath implementation on the XPath manager and end servers. Each server has its original name and IP address, and the routing IPs for path IDs are not related to DNS.

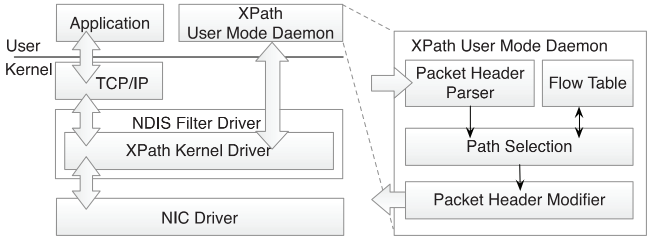


Fig. 8. The software stacks of XPath on servers.

To enable path ID resolution, we implemented a XPath software module on the end server, and a module on the XPath manager. The end server XPath software queries the XPath manager to obtain the updated path IDs for a destination. The XPath manager returns the path IDs by indexing the IP-to-ID mapping table. From the path IDs in the query response, the source selects one for the current flow, and caches all (with a timeout) for subsequent communications.

To maintain the connectivity to legacy TCP/IP stacks, we design an IP-in-IP tunnel based implementation. The XPath software encapsulates the original IP packets within an IP tunnel: the path ID is used for the tunnel IP header and the original IP header is the inner one. After the tunnel packets are decapsulated, the inner IP packets are delivered to destinations so that multi-path routing by XPath is transparent to applications. Since path IDs in Fattree end at the last hop ToR, the decapsulation is performed there. The XPath software may switch tunnel IP header to change the paths in case of failures, while for applications the connection is not affected. Such IP-in-IP encapsulation also eases VM migration as VM can keep the original IP during migration.

We note that if VXLAN [44] or NVGRE [34] is introduced for tenant network virtualization, XPath IP header needs to be the outer IP header and we will need 3 IP headers which looks awkward. In the future, we may consider more efficient and consolidated packet format. For example, we may put path ID in the outer NVGRE IP header and the physical IP in NVGRE GRE Key field. Once the packet reaches the destination, the host OS then switches the physical IP and path ID.

In our implementation, the XPath software on end servers consists of two parts: a Windows Network Driver Interface Specification (NDIS) filter driver in kernel space and a XPath daemon in user space. The software stacks of XPath are shown in Fig. 8. The XPath filter driver is between the TCP/IP and the Network Interface Card (NIC) driver. We use the Windows filter driver to parse the incoming/outgoing packets, and to intercept the packets that XPath is interested in. The XPath user mode daemon is responsible for path selection and packet header modification. The function of the XPath filter driver is relatively fixed, while the algorithm module in the user space daemon simplifies debugging and future extensions.

In Fig. 8, we observe that the packets are transferred between the kernel and user space, which may degrade the performance. Therefore, we allocate a shared memory pool by the XPath driver. With this pool, the packets are not copied and both

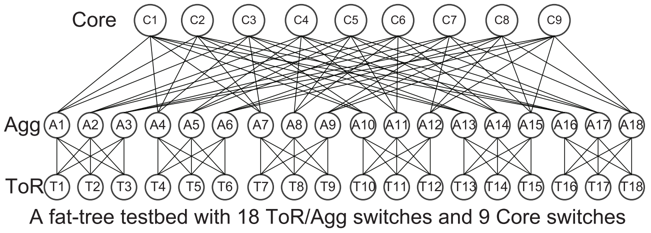


Fig. 9. Fattree(6) testbed with 54 servers. Each ToR switch connects 3 servers (not drawn).

the driver and the daemon operate on the same shared buffer. We tested our XPath implementation (with tunnel) and did not observe any visible impact on TCP throughput at Gigabit line rate.

B. Failure Handling

As introduced in Section II-B, when a link fails, the devices on the failed link will notify the XPath manager. In our implementation, the communication channel for such notification is out-of-band. Such out-of-band control network and the controller are available in existing production DCNs [46].

The path IDs for a destination server are distributed using a query-response based model. After the XPath manager obtains the updated link status, it may remove the affected paths or add the recovered paths, and respond to any later query with the updated paths.

For proof-of-concept experiments, we implemented a failure detection method with TCP connections on the servers. In our XPath daemon, we check the TCP sequence numbers and switch the path ID once we detect that the TCP has retransmitted a data packet after a TCP timeout. The motivation is that the TCP connection is experiencing bad performance on the current path (either failed or seriously congested) and the XPath driver has other alternative paths ready for use. We note that this TCP based approach is sub-optimal and there are faster failure detection mechanisms such as BFD [17] or F10 [29] that can detect failures in 30 μs, which XPath can leverage to perform fast rerouting (combining XPath with these advanced failure detection schemes is our future work). A key benefit of XPath is that it does not require route re-convergence and is loop-free during failure handling. This is because XPath pre-installs the backup paths and there is no need to do table re-computation unless all backup paths are down.

C. Testbed Setup and Basic Experiments

Testbed setup: We built a testbed with 54 servers connected by a Fattree(6) network (as shown in Fig. 9) using commodity Pronto Broadcom 48-port Gigabit Ethernet switches. On the testbed, there are 18 ToR, 18 Agg, and 9 Core switches. Each switch has 6 GigE ports. We achieve these 45 virtual 6-port GigE switches by partitioning the physical switches. Each ToR connects 3 servers; and the OS of each server is Windows Server 2008 R2 Enterprise 64-bit version. We deployed XPath on this testbed for experimentation.

IP table configuration: On our testbed, we consider 2754 explicit paths between ToRs (25758 paths between end hosts). After running the two-step compression algorithm, the number of routing entries for the switch IP tables are as follows, ToR: $31 \sim 33$, Agg: 48, and Core: 6. Note that the Fattree topology is symmetric, the numbers of routing entries after our heuristic are almost the same for the switches at the same layer, which confirms our hypothesis in Section III-B2 that equivalent nodes are likely to have similar numbers of entries.

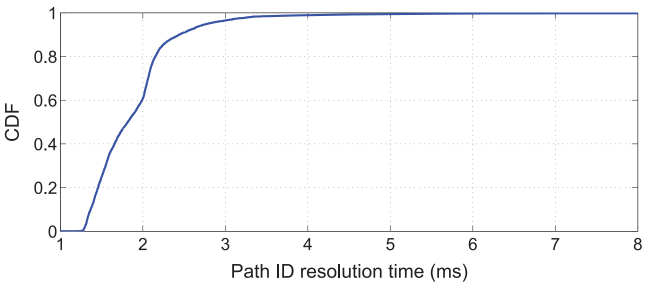


Fig. 10. The CDF of path ID resolution time.

Path ID resolution time: We measure the path ID resolution time at the XPath daemon on end servers: from the time when the query message is generated to the time the response from the XPath manager is received. We repeat the experiment 4000 times and depict the CDF in Fig. 10. We observe that the 99-th percentile latency is 4ms. The path ID resolution is performed for the first packet to a destination server that is not found in the cache, or cache timeout. A further optimization is to perform path ID resolution in parallel with DNS queries.

XPath routing with and without failure: In this experiment, we show basic routing of XPath, with and without link failures. We establish 90 TCP connections from the 3 servers under ToR T1 to the 45 servers under ToRs T4 to T18. Each source server initiates 30 TCP connections in parallel, and each destination server hosts two TCP connections. The total link capacity from T1 is$3 \times 1\text{G} = 3\text{G}$, shared by 90 TCP connections.

Given the 90 TCP connections randomly share 3 up links from T1, the load should be balanced overall. At around 40 seconds, we disconnect one link (T1 to A1). We use TCP sequence based method developed in Section IV-B for automatic failure detection and recovery in this experiment. We then resume the link at time around 80 seconds to check whether the load is still balanced. We log the goodput (observed by the application) and show the results for three connections versus time in Fig. 11. Since we find that the throughput of all 90 TCP connections are very similar, we just show the throughput of one TCP connection for each source server.

We observe that all the TCP connections can share the links fairly with and without failure. When the link fails, the TCP connections traversing the failed link (T1 to A1) quickly migrate to the healthy links (T1 to A2 and A3). When the failed link recovers, it can be reused on a new path ID resolution after the timeout of the local cache. In our experiment, we set the cache timeout value as 1 second. However, one can change this parameter to achieve satisfactory recovery time for resumed links. We also run experiments for other traffic patterns, e.g., ToR-to-ToR and All-to-ToR, and link failures at different locations, and find that XPath works as expected in all cases.

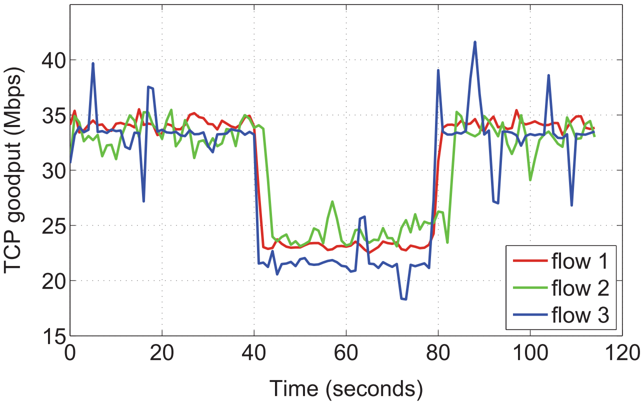


Fig. 11. TCP goodput of three connections versus time on three phases: no failure, in failure, and recovered.

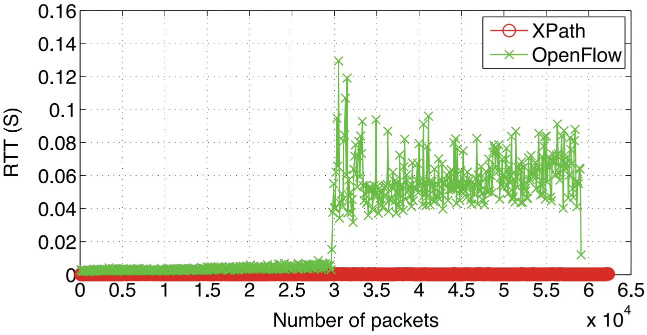
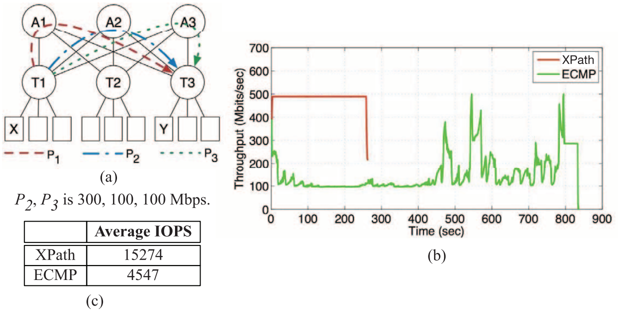


Fig. 12. Pre-installation under XPath vs dynamic installation under OpenFlow as packet/flow increases.



(a)

P_2, P_3 is 300, 100, 100 Mbps.

	Average IOPS
XPath	15274
ECMP	4547

(c)

(b)

Fig. 13. XPath utility case #1: we leverage XPath to make necessary bandwidth easier to implement for provisioned IOPS.

D. Pre-Installation Vs Dynamic Installation

In this experiment, we compare pre-installation using XPath with dynamic installation using OpenFlow [10], [24] in terms of supporting a large number of flows. For the experiment setting, we continuously send out 40 KB-size TCP flows from a source to a destination with 3 switches on the path. We vary destination TCP ports to emulate different flows. For XPath, we pre-install all routing entries of the desired paths into IP LPM tables of the switches. For OpenFlow, it dynamically installs flow entries for new flows into the generic flow tables of OpenFlow switches via the controller during runtime.

We measured the average RTTs with the number of packets/flows in Fig. 12. We make two observations: 1) Dynamic installation under OpenFlow has slightly higher average RTTs than pre-installation under XPath. The reason is that, to set up a flow on an N-switch path, OpenFlow requires $O(N)$ control packets for flow entry installation, whereas XPath only requires $O(1)$ control packet for path ID resolution on end-host. 2) There is an abrupt increase under OpenFlow when the flow count hits 1K ($\sim$ 30K packets), while XPath maintains persistent low latency. We note that our OpenFlow switch has 2K hardware forwarding entries, when the hardware flow table is full, the new flows will be automatically installed in the software forwarding table, which is much slower [1]. Thus, when the flow count increases to over 1K (about 2K rules installed[7]), the hardware flow table becomes full. After that, any subsequent new flows will enter the software flow table, which significantly inflates the forwarding delay of the message.

We also implemented rule replacement algorithm in which we let new flows replace the old ones in the hardware table when it is full, and we observed bad performance as well. Specifically, we measure the dynamic rule installation time of OpenFlow with 3 switches on the path using POX as the controller. We find it takes over 9.6 ms to replace an old rule with a new rule, i.e., the time from a new packet arrives at the switch until a new rule is effectively working. In contrast, pre-installation under XPath is not restricted by table size and its path ID resolution time is relatively small—4ms at 99th percentile as measured in Section IV-C.

The takeaway of this experiment is that XPath complements existing OpenFlow-based dynamic solutions, e.g., [24], [28], in terms of explicit path control, and pre-installation under XPath can maintain persistent low latency for a large number of flows.

V. XPath Applications

To showcase XPath's utility, we use it for explicit path support in four applications. The key is that, built on XPath, applications can freely choose which path to use without worrying about how to set up the path and the time cost or overhead of setting up the path. In this regard, XPath emerges as an interface for applications to use explicit paths conveniently, but does not make any choice on behalf of them.

A. XPath for Provisioned IOPS

In cloud services, there is an increasing need for provisioned IOPS. For example, Amazon EBS enforces provisioned IOPS for instances to ensure that disk resources can be accessed with high and consistent I/O performance whenever you need them [27]. To enforce such provisioned IOPS, it should first provide necessary bandwidth for the instances [11]. In this experiment, we show XPath can be easily leveraged to use the explicit path with necessary bandwidth.

As shown in Fig. 13(a), we use background UDP flows to stature the ToR-Agg links and leave the remaining bandwidth on 3 paths (P_1, P_2 and P_3) between X-Y as 300 Mpbs, 100 Mbps, and 100 Mbps respectively. Suppose there is a request for provisioned IOPS that requires 500 Mbps necessary bandwidth (The provisioned IOPS is about 15000 and the chunk size is 4 KB.). We now leverage XPath and ECMP to write 15 GB data ($\approx$ 4 million chunks) through 30 flows

[7] Note that each flow requires two OpenFlow rules for both directions.

from X to Y, and measure the achieved IOPS respectively. The storage we used for the experiment is Kingston V + 200 120G SSD, and the I/O operations on the storage are sequential read and sequential write.

From Fig. 13(c), it can be seen that using ECMP we cannot provide the necessary bandwidth between X-Y for the provisioned IOPS although the physical capacity is there. Thus, the actual achieved IOPS is only 4547, and the write under ECMP takes much longer time than that under XPath as shown in Fig. 13(c). This is because ECMP performs random hashing and cannot specify the explicit path to use, hence it cannot accurately make use of the remaining bandwidth on each of the multiple paths for end-to-end bandwidth provisioning. In contrast, XPath can be easily leveraged to provide the required bandwidth due to its explicit path control. With XPath, we explicitly control how to use the three paths and accurately provide 500 Mbps necessary bandwidth, achieving 15274 IOPS.

B. *XPath for Network Updating*

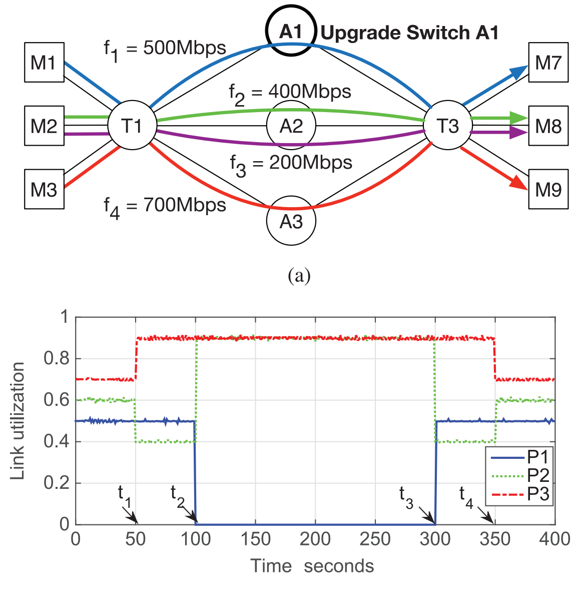
In production data centers, DCN update occurs frequently [28]. It can be triggered by the operators, applications and various networking failures. zUpdate [28] is an application that aims to perform congestion-free network-wide traffic migration during DCN updates with zero loss and zero human effort. In order to achieve its goal, zUpdate requires explicit routing path control over the underlying DCNs. In this experiment, we show how XPath assists zUpdate to accomplish DCN update and use a switch firmware upgrade example to show how traffic migration is conducted with XPath.

In Fig. 14(a), initially we assume 4 flows (f_1, f_2, f_3 and f_4) on three paths (P_1, P_2 and P_3). Then we move f_1 away from switch A_1 to do a firmware upgrade for switch A_1. However, neither P_2 nor P_3 has enough spare bandwidth to accommodate f_1 at this point of time. Therefore we need to move f_3 from P_2 to P_3 in advance. Finally, after the completion of firmware upgrade, we move all the flows back to original paths. We leverage XPath to implement the whole movement.

In Fig. 14(b), we depict the link utilization dynamics. At time t_1, when f_3 is moved from P_2 to P_3, the link utilization of P_2 drops from 0.6 to 0.4 and the link utilization of P_3 increases from 0.7 to 0.9. At time t_2, when f_1 is moved from P_1 to P_2, the link utilization of P_1 drops from 0.5 to 0 and the link utilization of P_2 increases from 0.4 to 0.9. The figure also shows the changes of the link utilization at time t_3 and t_4 when moving f_3 back to P_2 and f_1 back to P_1. It is easy to see that with the help of XPath, P_1, P_2 and P_3 see no congestion and DCN update proceeds smoothly without loss.

C. *Virtual Network Enforcement With XPath*

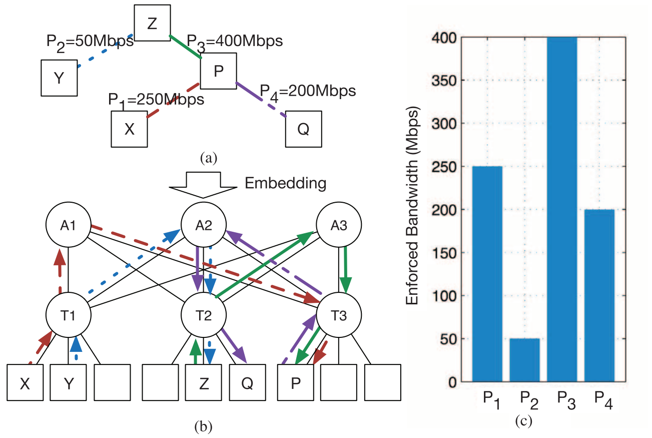
In cloud computing, virtual data center (VDC) abstraction with bandwidth guarantees is an appealing model due to its performance predictability in shared environments [8], [21], [47]. In this experiment, we show XPath can be applied to enforce virtual networks with bandwidth guarantees. We assume a simple SecondNet-based VDC model with 4 virtual links, and the bandwidth requirements on them are 50 Mbps, 200 Mbps, 250 Mbps and 400 Mbps respectively as shown in Fig. 15(a). We then leverage XPath's explicit path control to embed this VDC into the physical topology.



(a)

(b)

Fig. 14. XPath utility case #2: we leverage XPath to assist zUpdate [28] to accomplish DCN update with zero loss. (a) Path P_1: T1 →A1 →T3; P_2: T1 →A2 →T3; P_3: T1 →A3 →T3, (b) Time t_1: move f_3 from P_2 to P_3; t_2: move f_1 from P_1 to P_2; t_3: move f_1 from P_2 to P_1; t_4: move f_3 from P_3 to P_2.



(a)

(b)

(c)

Fig. 15. XPath utility case #3: we leverage XPath to accurately enforce VDC with bandwidth guarantees.

In Fig. 15(b), we show that XPath can easily be employed to use the explicit paths in the physical topology with enough bandwidth to embed the virtual links. In Fig. 15(c), we measure the actual bandwidth for each virtual link and show that the desired bandwidth is accurately enforced. However, we found that ECMP cannot be used to accurately enable this because ECMP cannot control paths explicitly.

D. *Map-Reduce Data Shuffle With XPath*

In Map-reduce applications, many-to-many data shuffle between the map and reduce stages can be time-consuming. For example, Hadoop traces from Facebook show that, on average, transferring data between successive stages accounts for 33% of

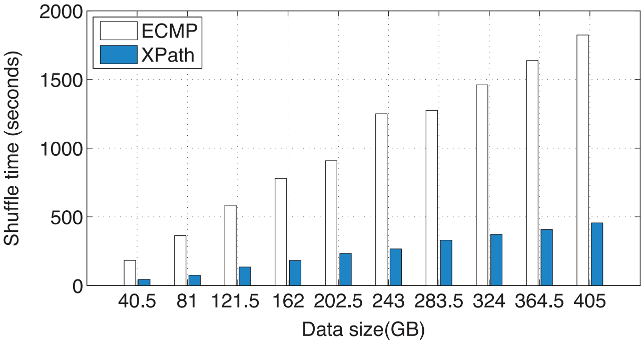


Fig. 16. XPath utility case #4: we leverage XPath to select non-conflict paths to speed up many-to-many data shuffle.

the running times of jobs [14]. Using XPath, we can explicitly express non-conflict parallel paths to speed up such many-to-many data shuffle. Usually, for a m-to-n data shuffle, we can use $(m + n)$ path IDs to express the communication patterns. The shuffle patterns can be predicted using existing techniques [35].

In this experiment, we selected 18 servers in two pods of the Fattree to emulate a 9-to-9 data shuffle by letting 9 servers in one pod send data to 9 servers in the other pod. We varied the data volume from 40 G to over 400 G. We compared XPath with ECMP.

In Fig. 16, it can be seen that by using XPath for data shuffle, we can perform considerably better than randomized ECMP hash-based routing. More specifically, it reduces the shuffle time by over 3× for most of the experiments. The reason is that XPath's explicit path IDs can be easily leveraged to arrange non-interfering paths for shuffling, thus the network bisection bandwidth is fully utilized for speedup.

VI. Related Work

The key to XPath is explicit path control. We note that many other approaches such as source routing [38], MPLS [37], OpenFlow [31] and the like, can also enable explicit path control. However, each of them has its own limitation.

OpenFlow [31] has been used in many recent proposals (e.g., [6], [9], [23], [24], [28]) to enable explicit path control. OpenFlow can establish fine-grained explicit routing path by installing flow entries in the switches via the OpenFlow controller. But in current practice, there are still challenges such as small flow table size and dynamic flow entries setup that need to be solved. For example, the on-chip OpenFlow forwarding rules in commodity switches are limited to a small number, typically 1–4 K. To handle this limitation, recent solutions, e.g., [24], dynamically change, based on traffic demand, the set of live paths available in the network at different times through dynamic flow table configurations, which could potentially introduce non-trivial implementation overhead and performance degradation. XPath addresses such challenge by pre-installing all desired paths into IP LPM tables. In this sense, XPath complements existing OpenFlow-based solutions in terms of explicit path control, and in the meanwhile, the OpenFlow framework may still be able to be used as a way for XPath to pre-configure the switches and handle failures.

Source routing is usually implemented in software and slow paths, and not supported in the hardware of the data center switches, which typically only support destination IP based routing. Compared to source routing, XPath is readily deployable without waiting for new hardware capability; and XPath's header length is fixed while it is variable for source routing with different path lengths.

With MPLS, paths can also be explicitly set up before data transmission using MPLS labels. However, XPath is different from MPLS in following aspects. First, because MPLS labels only have local significance, it requires a dynamic Label Distribution Protocol (LDP) for label assignments. In contrast, XPath path IDs are unique, and we do not need such a signaling protocol. Second, MPLS is based on exact matching (EM) and thus MPLS labels cannot be aggregated, whereas XPath is based on longest prefix matching (LPM) and enables more efficient routing table compression. Furthermore, MPLS is typically used only for traffic engineering in core networks instead of application-level or flow-level path control. In addition, it is reported [7], [24] that the number of tunnels that existing MPLS routers can support is limited.

SPAIN [32] builds a loop-free tree per VLAN and utilizes multiple paths across VLANs between two nodes, which increases the bisection bandwidth over the traditional Ethernet STP. However, SPAIN does not scale well because each host requires an Ethernet table entry per VLAN. Further, its network scale and path diversity are also restricted by the number of VLANs supported by Ethernet switches, e.g., 4096.

PAST [42] implements a per-address spanning tree routing for data center networks using the MAC table. PAST supports more spanning trees than SPAIN, but PAST does not support multi-paths between two servers, because a destination has only one tree. This is decided by the MAC table size and its exact matching on flat MAC addresses.

Both SPAIN and PAST are L2 technologies. Relative to them, XPath builds on L3 and harnesses the fast-growing IP LPM table of commodity switches. One reason we choose IP instead of MAC is that it allows prefix aggregation. It is worth noting that our XPath framework contains both SPAIN and PAST. XPath can express SPAIN's VLAN or PAST's spanning tree using CPF, and it can also arrange paths using DPF and perform path ID encoding and prefix aggregation for scalability.

Finally, there are various DCN routing schemes that come with specific topologies, such as those introduced in Fattree [5], PortLand [33], BCube [20], VL2 [19], ALIAS [45], and so on. For example, PortLand [33] leverages Fattree topology to assign hierarchical Pseudo-MACs to end hosts, while VL2 [19] exploits folded Clos network to allocate location-specific IPs to ToRs. These topology-aware addressing schemes generally benefit prefix aggregation and can lead to very small routing tables, however they do not enable explicit path control and still rely on ECMP [33] or Valiant Load Balancing (VLB) [19] for traffic spreading over multiple paths. Relative to them, XPath enables explicit path control for general DCN topologies.

VII. Conclusion

XPath is motivated by the need for explicit path control in DCN applications. At its very core, XPath uses a path ID to identify an end-to-end path, and pre-installs all the desired path IDs between any s-d pairs into IP LPM tables of commodity switches using a two-step compression algorithm. Through extensive evaluation and implementation, we show that XPath is scalable and easy to implement with existing commodity switches. Finally, we used testbed experiments to show that XPath can directly benefit many popular DCN applications.

ACKNOWLEDGMENT

C. Lan and H. Zhao were interns with Microsoft Research Asia when they worked on this project. The authors would like to thank their shepherd G. Porter and the anonymous NSDI and IEEE/ACM TRANSACTIONS ON NETWORKING reviewers for their feedback and suggestions.

REFERENCES

[1] Pica8, “Pica8-datasheet-48x1gbe-p3290-p3295,” [Online]. Available: http://www.pica8.com/

[2] Arista, “Arista 7050QX,” 2013 [Online]. Available: http://www.aristanetworks.com/media/system/pdf/Datasheets/7050QX-32_Datasheet.pdf

[3] H. Abu-Libdeh, P. Costa, A. Rowstron, G. O'Shea, and A. Donnelly, “Symbiotic routing in future data centers,” in *Proc. SIGCOMM*, 2010, pp. 51–62.

[4] J. H. Ahn, N. Binkert, A. Davis, M. McLaren, and R. S. Schreiber, “HyperX: Topology, routing, and packaging of efficient large-scale networks,” in *Proc. SC*, 2009, Art. no. 41.

[5] M. Al-Fares, A. Loukissas, and A. Vahdat, “A scalable, commodity data center network architecture,” in *Proc. ACM SIGCOMM*, 2008, pp. 63–74.

[6] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, “Hedera: Dynamic flow scheduling for data center networks,” in *Proc. NSDI*, 2010, p. 19.

[7] D. Applegate and M. Thorup, “Load optimal MPLS routing with N+M labels,” in *Proc. IEEE INFOCOM*, pp. 555–565.

[8] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, “Towards predictable datacenter networks,” in *Proc. SIGCOMM*, 2011, pp. 242–253.

[9] T. Benson, A. Anand, A. Akella, and M. Zhang, “MicroTE: Fine grained traffic engineering for data centers,” in *Proc. CoNEXT*, 2010, Art. no. 8.

[10] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, “A NICE way to test openflow applications,” in *Proc. NSDI*, 2012, p. 10.

[11] Amazon Web Services, “I/O characteristics,” [Online]. Available: http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-io-characteristics.html

[12] K. Chen *et al.*, “Generic and automatic address configuration for data centers,” in *Proc. SIGCOMM*, 2010, pp. 39–50.

[13] K. Chen *et al.*, “OSA: An optical switching architecture for data center networks with unprecedented flexibility,” in *Proc. NSDI*, 2012, p. 18.

[14] M. Chowdhury, M. Zaharia, J. Ma, M. Jordan, and I. Stoica, “Managing data transfers in computer clusters with orchestra,” in *Proc. SIGCOMM*, 2011, pp. 98–109.

[15] Cisco, “Data center: Load balancing data center services,” 2004.

[16] P. T. Darga, K. A. Sakallah, and I. L. Markov, “Faster symmetry discovery using sparsity of symmetries,” in *Proc. 45th DAC*, 2008, pp. 149–154.

[17] Cisco, “Bidirectional forwarding detection,” 2006 [Online]. Available: http://www.cisco.com/c/en/us/td/docs/ios/12_0s/feature/guide/fs_bfd.html

[18] R. Draves, C. King, S. Venkatachary, and B. Zill, “Constructing optimal IP routing tables,” in *Proc. IEEE INFOCOM*, 1999, pp. 88–97.

[19] A. Greenberg *et al.*, “VL2: A scalable and flexible data center network,” in *Proc. ACM SIGCOMM*, 2009, pp. 51–62.

[20] C. Guo *et al.*, “BCube: A high performance, server-centric network architecture for modular data centers,” in *Proc. SIGCOMM*, 2009, pp. 63–74.

[21] C. Guo *et al.*, “SecondNet: A data center network virtualization architecture with bandwidth guarantees,” in *Proc. CoNEXT*, 2010, Art. no. 15.

[22] C. Guo *et al.*, “DCell: A scalable and fault-tolerant network structure for data centers,” in *Proc. SIGCOMM*, 2008, pp. 75–86.

[23] B. Heller *et al.*, “ElasticTree: Saving energy in data center networks,” in *Proc. NSDI*, 2010, p. 17.

[24] C.-Y. Hong *et al.*, “Achieving high utilization with software-driven WAN,” in *Proc. ACM SIGCOMM*, 2013, pp. 15–26.

[25] C. Hopps, “Analysis of an equal-cost multi-path algorithm,” RFC 2992, 2000.

[26] Broadcom, “Broadcom Strata XGS Trident II,” [Online]. Available: http://www.broadcom.com

[27] Amazon Web Services, “Provisioned I/O-EBS,” [Online]. Available: https://aws.amazon.com/ebs/details

[28] H. Liu *et al.*, “zUpdate: Updating data center networks with zero loss,” in *Proc. ACM SIGCOMM*, 2013, pp. 411–422.

[29] V. Liu, D. Halperin, A. Krishnamurthy, and T. Anderson, “F10: A fault-tolerant engineered network,” in *Proc. NSDI*, 2013, pp. 399–412.

[30] B. D. McKay, “Practical graph isomorphism,” *Congressus Numer.*, 1981.

[31] N. McKeown *et al.*, “OpenFlow: Enabling innovation in campus networks,” *Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, 2008.

[32] J. Mudigonda, P. Yalagandula, and J. Mogul, “SPAIN: COTS datacenter ethernet for multipathing over arbitrary topologies,” in *Proc. NSDI*, 2010, p. 18.

[33] R. N. Mysore *et al.*, “PortLand: A scalable fault-tolerant layer 2 data center network fabric,” in *Proc. SIGCOMM*, 2009, pp. 39–50.

[34] “NVGRE,” [Online]. Available: http://en.wikipedia.org/wiki/NVGRE

[35] Y. Peng *et al.*, “HadoopWatch: A first step towards comprehensive traffic forecasting in cloud computing,” in *Proc. IEEE INFOCOM*, 2014, pp. 19–27.

[36] Amazon Web Services, “Announcing provisioned IOPS for Amazon EBS,” [Online]. Available: http://aws.amazon.com/about-aws/whats-new/2012/07/31/announcing-provisioned-iops-for-amazon-ebs/

[37] E. Rosen, A. Viswanathan, and R. Callon, “Multiprotocol label switching architecture,” RFC 3031, 2001.

[38] “Source routing,” [Online]. Available: http://en.wikipedia.org/wiki/Source_routing

[39] “Boolean satisfiability problem,” [Online]. Available: http://en.wikipedia.org/wiki/Boolean_satisfiability_problem

[40] J.-Y. Shin, B. Wong, and E. G. Sirer, “Small-world datacenters,” in *Proc. ACM SoCC*, 2011, Art. no. 2.

[41] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey, “Jellyfish: Networking data centers randomly,” in *Proc. NSDI*, 2012, p. 17.

[42] B. Stephens, A. Cox, W. Felter, C. Dixon, and J. Carter, “PAST: Scalable ethernet for data centers,” in *Proc. CoNEXT*, 2012, pp. 49–60.

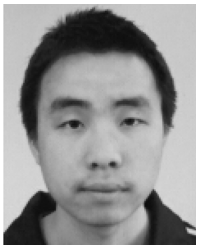
[43] “Graph vertex coloring,” [Online]. Available: http://en.wikipedia.org/wiki/Graph_coloring

[44] “VXLAN,” [Online]. Available: http://en.wikipedia.org/wiki/Virtual_Extensible_LAN

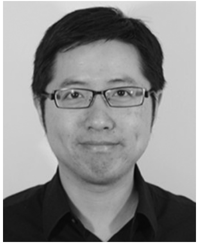
[45] M. Walraed-Sullivan *et al.*, “ALIAS: Scalable, decentralized label assignment for data centers,” in *Proc. SoCC*, 2011, Art. no. 6.

[46] X. Wu *et al.*, “NetPilot: Automating datacenter network failure mitigation,” in *Proc. SIGCOMM*, 2012, pp. 419–430.

[47] D. Xie, N. Ding, Y. C. Hu, and R. Kompella, “The only constant is change: Incorporating time-varying network reservations in data centers,” in *Proc. SIGCOMM*, 2012, pp. 199–210.



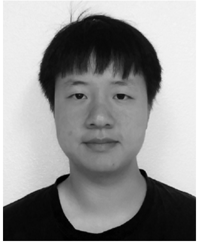
Shuihai Hu received the B.S. degree in computer science from University of Science and Technology of China, Hefei, China, in 2013, and is currently pursuing the Ph.D. degree in computer science at Hong Kong University of Science and Technology, Hong Kong. His current research interests are in the area of data center networks.



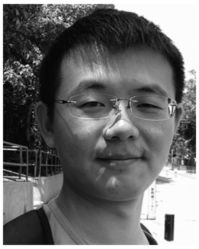
Kai Chen is an Assistant Professor with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong. He received a Ph.D. degree in computer science from Northwestern University, Evanston, IL in 2012. His research interest includes networked systems design and implementation, data center networks, and cloud computing.

Haitao Wu received his Bachelor degree in telecomm. engineering and his Ph.D. in telecommunication and information systems in 1998 and 2003 respectively, both from Beijing University of Post and Telecommunications (BUPT). He was a member of IEEE. He joined the Wireless and Networking Group, Microsoft Research Asia (MSRA), in 2003. He was transferred to Microsoft Azure product group on data center networking in 2014. His research interests include datacenter networks, QoS, TCP/IP, P2P, and wireless networks.

Wei Bai received the B.E. degree in information security from Shanghai Jiao Tong University, China, in 2013. He is currently pursuing the Ph.D. degree in computer science in Hong Kong University of Science and Technology. His current research interests are in the area of data center networks.



Chang Lan received the B.Eng. degree in computer science from Tsinghua University, Beijing, China, in 2013. He is currently working towards the Ph.D. degree in computer science at the University of California, Berkeley, CA, USA. His research focus on software defined networking and network function virtualization, and he also works on security and privacy.



Hao Wang is a Ph.D. student in Department of Electrical and Computer Engineering, University of Toronto. He received his B.E. degree in information security and M.E. degree in software engineering both from Shanghai Jiao Tong University in 2012 and 2015 respectively. His research interests include load balancing schemes in DCN and distributed computing optimization.

Hongze Zhao is a Ph.D. student at Duke University, working with Prof. Xiaowei Yang. His research interest includes computer networks, security and network diagnostics. He also enjoys writing code and learning new programming techniques.

Chuanxiong Guo is a Principal Software Engineering Manager at Microsoft Azure Networking. Before that, he was a Senior Researcher in the Wireless and Networking Group of Microsoft Research Asia (MSRA). He received his Ph.D. degree from the Institute of Communications Engineering in Nanjing, China. His areas of interest include: networked systems design and implementation at scale, data center networking, network troubleshooting, network security, networking support for operating systems and applications, and Cloud Computing. He is currently working on data center networking and Cloud Computing.