DiFS: Distributed Flow Scheduling for Adaptive Routing in Hierarchical Data Center Networks

Wenzhi Cui
Department of Computer Science
The University of Texas at Austin
Austin, Texas, 78712
wc8348@cs.utexas.edu

Chen Qian
Department of Computer Science
University of Kentucky
Lexington, Kentucky, 40506
qian@cs.uky.edu

ABSTRACT

Data center networks leverage multiple parallel paths connecting end host pairs to offer high bisection bandwidth for cluster computing applications. However, state of the art routing protocols such as Equal Cost Multipath (ECMP) is load-oblivious due to static flow-to-link assignments. They may cause bandwidth loss due to flow collisions. Recently proposed centralized scheduling algorithm or host based adaptive routing that require network-wide condition information may suffer from scalability problems.

In this paper, we present Distributed Flow Scheduling (DiFS) based Adaptive Routing for hierarchical data center networks, which is a *localized and switch-only solution.* DiFS allows switches to cooperate to avoid over-utilized links and find available paths without centralized control. DiFS is scalable and can react quickly to dynamic traffic, because it is independently executed on switches and requires no synchronization. DiFS provides global bounds of flow balance based on local optimization. Extensive experiments show that the aggregate throughput of DiFS using various traffic patterns is much better than that of ECMP, and is similar to or higher than those of two representative protocols that use network-wide optimization.

Categories and Subject Descriptors

C.2.2 [**Computer Communication Networks**]: Network Protocols—*Routing Protocols*

Keywords

Adaptive Routing; Data Center Networks

1. INTRODUCTION

The growing importance of cloud-based applications and big data processing has led to the deployment of large-scale data center networks that carry tremendous amount of traffic. Recently proposed data center network architectures primarily focus on using commodity Ethernet switches to build hierarchical trees such as fat-tree [1] [20] [25] and Clos [11] [28]. These topologies provide multiple equal-cost paths for any pair of end hosts and hence significantly increase bisection bandwidth. To fully utilize the path diversity, an ideal routing protocol should allow flows to avoid over-utilized links and take alternative paths, called adaptive routing. In this work, we focus on the investigation of throughput improvement by *routing protocols.*

Most state of the art data center networks rely on layer-3 Equal Cost Multipath (ECMP) protocol [15] to assign flows to available links using static flow hashing. Being simple and efficient, however, ECMP is load-oblivious, because the flow-to-path assignment does not account current network utilization. As a result, ECMP may cause flow collisions on particular links and create hot spots.

Recently proposed methods to improve the bandwidth utilization in data center networks can be classified in three categories: *centralized, host-based, and switch-only.*

• *Centralized* solutions utilize the recent advances in Software Defined Networking (SDN), which allows a central controller to perform control plane tasks and install forwarding entries to switches via a special protocol such as OpenFlow [19]. A typical centralized solution Hedera [2] relies on a central controller to find a path for each flow or assign a single core switch to deal with all flows to each destination host. Centralized solutions may face scalability problems [21], because traffic in today's data center networks requires parallel and fast path selection according to recent measurement studies [5, 17].

• *Host-based* methods, such as DARD [25], can be run without a central control. These methods enable end systems to monitor the network bandwidth utilization and then select desired paths for flows based on network conditions. One major limitation of host-based approaches is that every host needs to monitor the states of all paths to other hosts. In a large network such as production data centers, great amounts of control messages would occur. For many applications such as Shuffle (described in Section 4.3), each DARD host may have to monitor the entire network, which also limits its scalability. In addition, all legacy systems and applications running these protocols need to be upgraded, which incurs a lot management cost. There are also a number of host-based solutions in the transport layer such as Data Center TCP (DCTCP) [3] and multipath TCP (MPTCP) [24]. These methods are out of the scope of this work because we only focus on routing protocols.

• The last type is *switch-only* protocols which is efficient and fully compatible to current systems and applications on

ANCS'14, October 20–21, 2014, Los Angeles, CA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2839-5/14/10 ...$15.00.
http://dx.doi.org/10.1145/2658260.2658266 .

end hosts. It has been argued that switch-only solutions hold the best promise for dealing with large-scale and dynamic data center traffic patterns [21]. ECMP is a typical switch-only routing protocol for load balance. Many existing switch-only protocols allow a flow take multiple paths at the same time (called flow splitting) to achieve high throughput [10,21,28]. Flow splitting may cause a high level of TCP packet reordering, resulting in throughput drop [18].

In this paper, we propose Distributed Flow Scheduling (DiFS), a switch-only routing protocol that is executed independently on the control unit of each switch. DiFS aims to balance flows among different links and improves bandwidth utilization for data center networks. DiFS does not need centralized control or changes on end hosts. In addition DiFS does not allow flow splitting and hence limits packet reordering.

Based on our observations, we categorize flow collisions in a hierarchical data center networks in two types, local and remote flow collisions. DiFS achieves load balancing by taking efforts in two directions. First, each switch uses the *Path Allocation* algorithm that assigns flows evenly to all outgoing links to avoid *local flow collisions*. Second, each switch also monitors its incoming links by running the Imbalance Detection algorithm. If a collision is detected, the switch will send an Explicit Adaption Request (EAR) message that suggests the sending switch of a flow to change its path. Upon receiving the EAR, the sending switch will run the *Explicit Adaption* algorithm to avoid *remote flow collisions*. Previous solutions such as Hedera [2] try to maximize the total achieved throughput across all elephant flows using global knowledge by balancing traffic load among core switches. However we show that load balance among core switches is not enough to achieve load balance among different links. DiFS effectively solves this problem using the control messages called Explicit Adaptation Requests.

We conduct extensive simulations to compare DiFS with three representative methods from different categories: ECMP (switch-only) [15], Hedera (centralized) [2], and Dard (host-based) [25]. Experimental results show that DiFS outperforms ECMP significantly in aggregate bisection bandwidth. Compared with the centralized solution Hedera and the host-based solution Dard, DiFS achieves comparable or even higher throughput and less out-of-order packets, for both small and large data center network topologies.

The rest of this paper is organized as follows. Section 2 introduces background knowledge of flow scheduling in data center networks. Section 3 presents the detailed architecture and algorithm design of DiFS. We evaluate the performance of DiFS and compare it with other solutions in Section 4. We conclude our work in Section 6.

2. BACKGROUND AND OVERVIEW OF DIFS

2.1 Data center topologies

Today's data center networks often use *multi-rooted hierarchical tree* topologies (e.g., fat-tree [1] and Clos [11] topologies) to provide multiple parallel paths between any pair of hosts to enhance the network bisection bandwidth, instead of using expensive high speed routers/switches. Our protocol *DiFS is designed for an arbitrary hierarchical tree topology* as long as the switch organization in every pod is the same. However for the ease of exposition and comparison

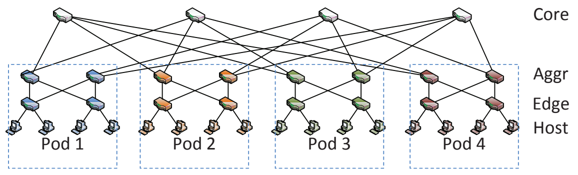


Figure 1: A fat tree topology for a datacenter network

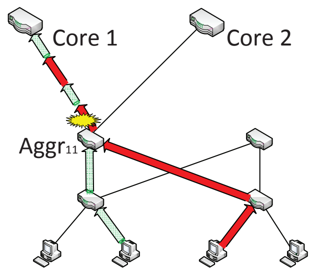
with existing protocols, we will use the fat-tree topology for our protocol description and experimental evaluation.

A multi-rooted hierarchical tree topology has three vertical layers: edge layer, aggregate layer, and core layer. A *pod* is a management unit down from the core layer, which consists of a set of interconnected end hosts and a set of edge and aggregate switches that connect these hosts. As illustrated in Figure 1, a *fat-tree network* is built from a large number of k-port switches and end hosts. There are k pods, interconnected by $(k/2)^2$ core switches. Every pod consists of $(k/2)$ edge switches and $(k/2)$ aggregate switches. Each edge switch also connects $(k/2)$ end hosts. In the example of Figure 1, $k = 4$, and thus there are four pods, each of which consists of four switches.

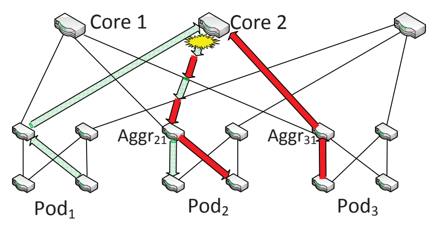
A path are a set of links that connect two end hosts. There are two kinds of paths in a fat-tree network: inter-pod path and intra-pod path. An intra-pod path interconnects two hosts within the same pod while an inter-pod path is a path that connects two end host in different pods. Between any pair of end hosts in different pods, there are $(k/2)^2$ equal-cost paths, each of which corresponds to a core switch. An end-to-end path can be split into two flow segments [26]: the *uphill segment* refers to the part of the path connecting source host to the switch in the highest layer (e.g., the core switch for an inter-pod path), and the *downhill segment* refers to the part connecting the switch in the highest layer to the destination host. Similar to existing work, we mainly focus our discussion on inter-pod flows, because intra-pod flows can be handled by a simpler version of the routing protocol.

2.2 Examples of flow collision and DiFS's solutions

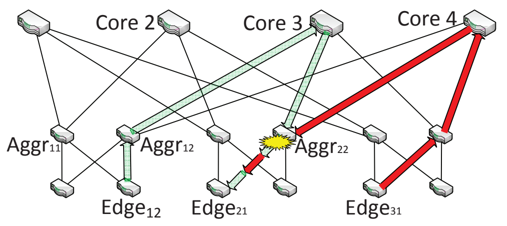
We define a flow collision as too many flows being transmitted to a same link, which may cause potential congestion. We show three types of flow collisions in Figure 2, where in each example some parts of the network are not shown for simplicity. If a switch experiences a flow collision on one of its links and can locally adjust the flow assignment to resolve the collision, such collision is called a *local collision*. Figure 2(a) shows an example of a local collision, where switch $Aggr_{11}$ forwards two flows to a same link. Local collisions may be caused by a bad flow assignment of static multi-pathing algorithms such as ECMP. Otherwise, the collision is called a *remote collision*. Figure 2(b) shows an example of Type 1 remote collision, where two flows take a same link from $Core_2$ to Pod_2. Type 1 remote collision may be caused by over-utilizing a core switch ($Core_2$ in this example). Hence some existing solutions propose to balance traffic among cores [2]. However balancing core utilization may not be enough. Another example of remote collision (Type 2) is shown in Figure 2(c), where core utilization is



(a) Local collision

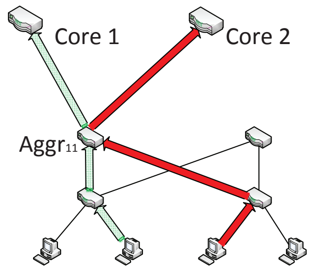


(b) Remote collision Type 1, caused by imbalance use of core switches

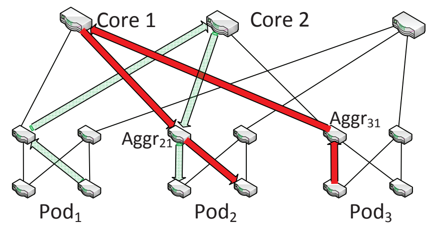


(c) Remote collision Type 2: load balance among core switches is not enough

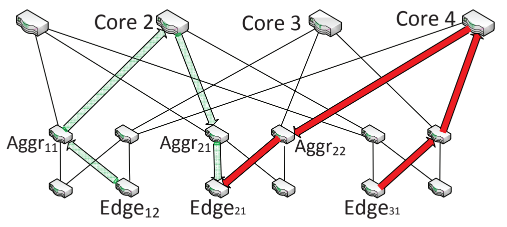
Figure 2: Three types of collisions. For simplicity we use two flows to indicate a collision. In practise, a collision may be caused by many flows.



(a) Path Allocation solves local collision.



(b) Solve remote collision Type 1: $Aggr_{21}$ sends back an EAR and suggest $Aggr_{31}$ to forward the flow to $Core1$.



(c) Solve remote collision Type 2: $Edge_{21}$ sends back an EAR and suggest $Edge_{12}$ to forward the flow to $Aggr11$.

Figure 3: Resolving collisions by Path Allocation and Explicit Adaption

balanced but flows still collide on the link from $Aggr_{22}$ to $Edge_{21}$. We also observe that location collisions happen in uphill segments and remote collisions happen in downhill segments.

Local collisions can be detected and resolved by local algorithms in a relatively easy way. DiFS uses the Path Allocation algorithm to detect flow-to-link imbalance and remove one of the flows to an under-utilized link, as shown in Figure 3(a). The key insight of DiFS to resolve remote collisions is to allow the switch in the downhill segment that detected flow imbalance to send an Explicit Adaption Request (EAR) message to the uphill segment. For the example of Figure 2(b), $Aggr_{21}$ can detect flow imbalance among the incoming links. It then sends an EAR to $Aggr_{31}$ in Pod_3 (randomly chosen between two sending pods), suggesting the flow to take the path through $Core_1$. $Aggr_{31}$ runs the Explicit Adaption algorithm and changes the flow path. That flow will eventually take another incoming link of $Aggr_{21}$ as shown in Figure 3(b). To resolve the collision in Figure 2(c), $Edge_{21}$ that detects flow imbalance sends back an EAR and suggest $Edge_{12}$ to forward the flow to $Aggr_{11}$. That flow will eventually go from $Aggr_{21}$ to $Edge_{21}$, as shown in Figure 3(c).

From the examples, the key observation is that *the incoming links of the aggregate (edge) switch in the downhill segment have one-to-one correspondence to the outgoing links of the aggregate (edge) switch in the uphill segment* in a multi-rooted tree. Therefore when an aggregate (edge) switch in the downhill segment detects imbalance and finds an under-utilized link, it can suggest the aggregate (edge) switch in the uphill segment to change the path to the "mirror" of the under-utilized link. In the example of Type 1 remote collision, $Aggr_{21}$ controls the flow to income from $Core_1$ by suggesting $Aggr_{31}$ to forward the flow to $Core_1$. In the example of Type 2 remote collision, $Edge_{21}$ controls the flow to income from $Aggr_{21}$ by suggesting $Edge_{12}$ to forward the flow to $Aggr_{11}$.

2.3 Classification of flows

In this paper, a flow is defined as a sequence of packets sent from a source host to a destination host using TCP. In our flow scheduling protocol, a flow can have only one path at any time. Allowing a flow to use multiple paths simultaneously may cause packet reordering and hence reduce the throughput. However, a flow is allowed to take multiple paths at different times in its life cycle.

Elephant and mice flows: Elephants are large, long-lived flows whose traffic amount is higher than a threshold. The other flows are called mice flows. Similar to many other work [2, 25], *our protocol focuses on elephant flows and intends to spread them as evenly as possible among all links.* All mice flows will be processed by ECMP, because recent work has shown that ECMP forwarding can perform load-balancing efficiently and effectively for mice flows [11]. Note that elephant flows do not necessarily require high demand of sending rates.

Let f_{ab} be a flow whose source is a and destination is b. A flow f_{ab} may be classified into four types for a particular switch s that runs DiFS: f_{ab} is a single-in-single-out (SISO) flow for switch s if and only if there are only one possible incoming link of s from a and one possible outgoing link of s to b. f_{ab} is a single-in-multi-out (SIMO) flow for switch s if and only if there are one incoming link of s from a and multiple outgoing links of s to b. f_{ab} is a multi-in-single-out (MISO) flow for switch s if and only if there are multiple incoming links of s from a and one outgoing link of s to b. A close look at fat-tree networks reveals that all inter-pod flows are SIMO for the edge and aggregate switches on the uphill segments, and are MISO for the edge and aggregate switches on the downhill segments. All inter-pod flows for core switches are SISO. Multi-in-multi-out (MIMO) flows may be defined similarly. However, there is no MIMO flow for any switch in a fat-tree network. They may appear in general topologies.

3. DIFS DESIGN

3.1 Components and Deployment

Typical switch architecture usually consists of two components: data plane and control plane. The data plane includes multiple network ports, as well as a flow/forwarding table and an output queue for each port. The control plane can perform general-purpose processing like collecting measurement results and install/modify the rules in the flow/forwarding tables of the data plane. As result, DiFS should be installed in the control plane of each switch. DiFS is also compatible to software defined networking such as OpenFlow [19]. The control logic of DiFS can be implemented in OpenFlow controllers which exchange control messages and update flow tables in OpenFlow switches accordingly. Compared with centralized algorithms that require a single controller responsible to the entire network, distributed and localized decision-making of DiFS offers tremendous scalability to SDN control. For example, OpenFlow switches in the same pod can be connected to one controller, which is physically close to these switches and able to handle the scheduling tasks. Controllers in different pods exchange control information that is much less than condition of the whole network.

3.2 Optimization goals

As a high-level description, DiFS intends to balance *the number of elephant flows* among all links in the network to utilize the bisection bandwidth and take the advantage of path diversity. We use the number of flows as the optimization metric instead of flow bandwidth consumption based on the following reasons:

1. A flow's maximum bandwidth consumption[1] can hardly be estimated. As shown in [2], a flow's current sending rate tells very little about its maximum bandwidth consumption. Hedera [2] uses global knowledge to perform flow bandwidth demand estimation. However, such method is not possible to be applied in distributed algorithms such as DiFS.

2. Using flow count only requires a switch to maintain a counter for each outgoing link. However, measurement of flow bandwidth consumption requires complicated traffic monitoring tools installed on each switch. Our method simplifies switch structure.

3. Using flow count as the metric, DiFS can achieve similar or even better performance compared with Hedera [2] and a variant of DiFS implementation that uses estimated bandwidth consumption as the metric. The results will be shown in Section 4.6.

Two optimization goals for load-balancing scenarios are desired:

Balanced Output (BO): For an edge switch s_e, let $o(s_a)$ be the number of SIMO flows on an outgoing link connecting the aggregate switch s_a. BO of edge switch s_e is achieved if and only if $o(s_{a1}) - o(s_{a2}) \leq \delta$, for any two aggregate switches s_{a1} and s_{a2}, where δ is a constant. Similarly we may define BO of an aggregate switch to cores. BO can be achieved by the Path allocation algorithm of DiFS with the smallest possible value of δ being 1.

Balanced Input (BI): For an aggregate switch s_a, let $i(c)$ be the number of MISO flows on an incoming link connecting the core c. BI of edge switch s is achieved if and only if $i(c_1) - i(c_2) \leq \delta$, for any two cores c_1 and c_2, where δ is a constant. Similarly we may define BI of an edge switch from aggregate switches. BI can be achieved by Explicit Adaptation of DiFS with the smallest possible value of δ being 1.

BO and BI do not interfere with each other, and hence a switch can achieve them at a same time. Although BO and BI of a switch are two kinds of optimization in a local view, we have proved that they provide global performance bounds of load balancing, as presented in Section 3.7. In Section 4 we further demonstrate that they can achieve high aggregate throughput via experiments.

3.3 Protocol Structure

DiFS uses a threshold to eliminate mice flows. Such threshold-based module can be installed on edge switches. It maintains the number of transmitted bytes of each flow. This monitoring task can be cost-efficient in switch resources using recent proposed techniques such as OpenSketch [27]. If the byte number of a flow is larger than a threshold value, the edge switch will label this flow as an elephant flow and mark the packet header to notify other switches on its path.

Each switch has a flow list which maintains three variables for every flow f: the incoming link identifier, denoted as L_i, the outgoing link identifier, denoted as L_o, and the last time this flow appeared, denoted as t. A switch also maintains two Port State Vectors (PSVs), V_i and V_o. The ith element in vector V_i is the number of flows coming from the ith incoming link. Likewise the ith element in vector V_o is the number of flows forwarded to the ith outgoing link.

There are three flow control modules in aggregate and edge switches: control loop unit, explicit adaptation unit, and path allocation unit. Control loops are run periodically by switches. The main objectives of the control loop unit are to detect imbalance of MISO flows among incoming links and send an Explicit Adaptation Request (EAR) if necessary. An EAR is a notification message sent along the *reverse flow path* to recommend switches in the flow's sending pod to choose a different path. An EAR also in-

[1]A flow's maximum bandwidth consumption, also called as flow demand, is the rate the flow would grow to in a fully non-blocking network.

Algorithm 1: Imbalance Detection in Control Loop

```
$S$ = the set of all MISO flows forwarded by this switch
for $f \in S$ do
    $L_i$ = incoming link of $f$
    $min$ = minimum value among elements in $V_i$ of $f$
    $\delta$ = imbalance threshold
    if $V_i[L_i] - min > T$ then
        compute a path recommendation $p$
        send a EAR($f$, $p$) to $L_i$
        Return
    end
end
```

cludes a path recommendation. When a switch receives an EAR, it runs the explicit adaptation unit and changes the output link of the designated flow in the EAR to that on the recommended path, if possible. Path Allocation Request (PAR) is another message to request flow scheduling. PAR includes a flow identifier and requires switches to allocate an available link for this flow. Switches treat a packet with a new flow identifier as a PAR. The sender needs to explicitly send a PAR only if path reservation is allowed to achieve a certain level of performance guarantee for upper-layer applications [4]. For a SIMO flow, the path allocation unit will assign an outgoing port for this flow based on link utilization. Detailed algorithms for these modules will be presented in the following subsections.

The time period between two control loops has limited impact to the convergence time of the whole protocol execution. We will show that DiFS converges quickly under a wide range of control loop period time in Section 4.5.

3.4 Control loop

Each DiFS switch continuously runs a control loop. At each iteration, the switch executes the following:

1. Remove disappeared flows. A flow may disappear from a switch due to several reasons. For example, the flow may have finished transmission or taken another path. In each iteration, the switch will delete a flow if the difference between current time and its last-appeared time t is larger than a threshold, which may be set to a multiple of the average round-trip time of flows.

2. Re-balance SIMO flows among all outgoing links. Removing disappeared flows may cause the change of flow numbers on links. Thus flow re-balancing is necessary.

3. Send an EAR if necessary. If the switch finds a MISO flow comes in a over-utilized link, the switch will recommend other switches to change the flow path by sending an EAR. In order to avoid TCP performance degrade caused by too many EARs, DiFS forces every switch to send at most one EAR at each iteration.

We detail the steps 2) and 3) as the follows.

Re-balance SIMO flows. The purpose of re-balancing SIMO flows is to achieve BO, i.e., let the flow count difference of any two outgoing links be smaller than the predefined threshold δ. The solution seems to be trivial: a switch can simply move flows on overloaded links to underloaded ones. However this simple method could cause oscillations of network status. Consider a switch s moves a random flow f from link l_1 to l_2 for load balance. Later by receiving an EAR from another switch, s will be suggested to move f from l_2 to l_1 to avoid remote collisions. During the next control loop, s will again move f to l_1 to l_2 and so on. Such oscillation will never stop. One obvious downside of oscillations is that they will incur packet reordering and hurt TCP performance. To resolve this problem, we maintain a priority value for each flow in the flow list. When the link assignment of a flow is changed based upon the suggestion from an EAR, the priority of the flow is increased by one. When a switch re-balances SIMO flows, it should first move flows that have less priority values. This strategy intends to let flows whose assignments are changed by EARs be more stable and avoid oscillations.

Imbalance detection and path recommendation for EAR. For fairness concern, at each iteration the switch will scan each MISO flows in a random order. The imbalance detection is also in a threshold basis, which is presented in Algorithm 1.

Due to lack of global view of flow distribution, the EAR receiver should be told how to change the flow's path. Therefore the EAR sender should include a path recommendation, *which does not necessarily need to be a complete path.* In a fat-tree, both aggregate and edge switches are able to detect load imbalance and recommend an alternative path *only based on local link status.*

For the flow collision example of Figure 2(b), $Aggr_{21}$ will notice the load imbalance among incoming links and send an EAR to $Aggr_{31}$ (randomly selected between senders of the two collided flows). The path recommendation in this EAR is just $Core_1$. $Aggr_{31}$ will receive the EAR and change the flow to the output link connected with $Core_1$, and this flow will eventually come from another incoming link of $Aggr_{21}$ that was under-utilized, as shown in Figure 3(b).

For the flow collision example of Figure 2(c), $Edge_{21}$ can detect it by comparing two incoming links and then send an EAR to $Edge_{12}$ in the uphill segment. The path recommendation here is just $Aggr_{11}$. When $Edge_{12}$ let the flow take $Aggr_{11}$, the flow will eventually take another incoming link to $Edge_{21}$ and hence resolves the collision as shown in Figure 3(c).

As a matter of fact, in a fat-tree network a path recommendation can be specified by either a recommended core or a recommended aggregate switch in the uphill segment. For other topologies, more detailed path specification might be needed.

For an intra-pod flow, the path consists of two edge switches and one aggregate switch. If the aggregate switch detects load imbalance, it can also send an EAR to the edge switch in the previous hop and suggest the edge switch to send the flow to another aggregate switch. In fact, this is the one difference in our protocol when it treats intra-pod and inter-pod flows.

3.5 Operations upon receiving a PAR

In order to keep all links output balanced, we use a distributed greedy algorithm to select an outgoing link for each flow requested by a PAR. When a switch received a PAR, it first check how many outgoing links can lead to the destination. If there is only one link, then the switch will simply use this link. If there are multiple links to which this flow can be forwarded, the switch will select an local optimal link for this flow. The algorithm first find the set of links with

Algorithm 2: Path Allocation

```
Input: Path Allocation Request $PAR$
Output: None
$f$ = flow identifier in $PAR$
$S$ = set of links that can reach $f$'s destination
if $|S| > 1$ then
    $min$ = minimal value among all $V_o[l]$, $l \in S$
    for $l \in S$ do
        if $V_o[l] > min$ then
            $S = S - \{l\}$
        end
    end
    $L_o$ = a random element in $S$
    increase $V_o[L_o]$ by 1
else
    $L_o$ = the first element of $S$
end
record the incoming link $L_i$ of $f$
record the outgoing link $L_o$ of $f$
update the access time $t$ of $f$
```

Algorithm 3: Explicit Adaptation of switch s

```
Input: Explicit Adaptation Request $EAR$
Output: None
$f$ = flow identifier in $EAR$
$r$ = recommended core or aggregate switch in $EAR$
$L_i$ = current incoming link of $f$
$L_o$ = current outgoing link of $f$
if $r$ and $s$ are connected and sending $f$ to $r$ can lead to
the destination of $f$ then
    $L$ = the outgoing link connecting $r$
    if $V_o[L] >= V_o[L_o]$ then
        move a flow currently on $L$ to $L_o$
    move $f$ to the outgoing link $L$
    update the link variables of changed links
else
    forward EAR to $L_i$
end
```

the minimum number of outgoing flows. If there are more than one links in this set, the algorithm will randomly select a link from the set.

3.6 Operations upon receiving an EAR

An EAR includes a flow identifier and a path recommendation. As mentioned, for a fat-tree network a path recommendation can be specified by either a recommended core or a recommended uphill aggregate switch. When a switch received an EAR, it first checks if it can move the requested flow f to the recommended core or aggregate switch. If not, it will forward this EAR further towards the reverse path of f. If moving f will cause imbalance among outgoing links, the switch swaps f with another flow on the recommended link. The complete algorithm is described in Algorithm 3.

EARs may also cause network status oscillations. Consider the following scenario in Fig 4, where only part of the network is shown. $flow_1$ and $flow_2$ collide on the same link from SW_4 to SW_1 but the link from SW_5 to SW_1 is free. SW_1 may send an EAR to SW_2 and suggest SW_2 to send $flow_1$ to SW_5, in the purpose of resolving the remote collision at SW_1. After receiving the EAR, SW_2 swaps the outgoing links of $flow_1$ and $flow_3$. However at the same time SW_3 may send an EAR to SW_2 and suggest SW_2 to send $flow_4$ to SW_4. SW_2 should then swap the outgoing links of $flow_2$ and $flow_4$. As a result the collisions still exist. By keeping executing the protocol, oscillations happen and the network status cannot converge. To deal with the problem, we allow random spans in control loops. There is some non-negligible time difference between the control loops of SW_1 and SW_3. In this way, SW_3 may notice that its collision has already been solved after SW_2 swaps the outgoing links of $flow_1$ and $flow_3$ and will not send another EAR.

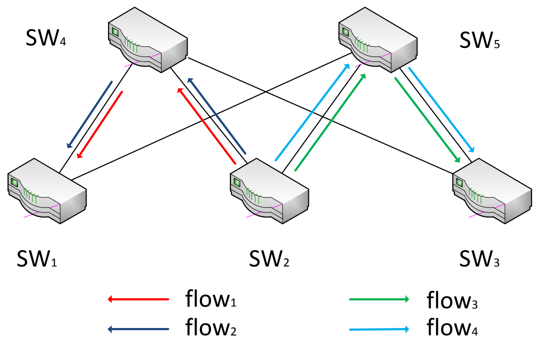


Figure 4: Oscillation problem caused by EARs

3.7 Bounds on global flow balance

The local optimization on switches can lead to global performance bounds as introduced in this section.

We provide a bound on flow balance among aggregate switches in a same pod by the following theorem:

THEOREM 3.1. *In a k-pod fat-tree, suppose every edge switch achieves BO with δ. Let $n(s_a)$ be the number of flows that are sending to aggregate switch s_a. Then we have $MAX_a - MIN_a \leq \delta \cdot k/2$, where MAX_a is the maximum $n(s_a)$ value among all aggregate switches in the pod, MIN_c is the minimum $n(s_a)$ value among all aggregate switches in the pod.*

We further prove a bound on flow balance among core switches by the following theorem:

THEOREM 3.2. *In a k-pod fat-tree, suppose every edge and aggregate switch achieves BO with $\delta = 1$. Let $n(c)$ be the number of flows that are sending to core c. Then we have $MAX_{all} - MIN_{all} \leq 3k$, where MAX_{all} is the maximum $n(c)$ value among all cores and MIN_{all} is the minimum $n(c)$ value among all cores.*

Similarly we have a bound of flow balance in the receiving side.

THEOREM 3.3. *In a k-pod fat-tree, suppose all aggregate switches in a same pod achieve BI with $\delta = 1$. Let $n(s_e)$ be the number of flows that are sending to edge switch s_e. Then we have $MAX_e - MIN_e \leq k/2$, where MAX_e is the maximum $n(s_e)$ value among all edge switches in the pod and MIN_e is the minimum $n(s_e)$ value among all edge switches in the pod.*

The proof is similar to that of Theorem 3.1. Proofs in this section can be found in the appendix.

Note that the values we provide in the theorems are only bounds of the difference between the maximum and minimum flow numbers. In practice, however, *the actual differences are much lower than these bounds.*

3.8 Failures Recovery

Switches must take network failures into consideration in performing flow assignments. A network failure may be a switch failure, a link failure, or a host failure. Failures may also be classified into reachability failures and partial failures. Reachability failures refer to those failures that can cause one or more end hosts unreachable. For example, crash of an edge switch can make $(k/2)$ hosts unreachable. DiFS can tolerate such kind of failures because our algorithm relies on local, soft state collected at run time. Only flows towards the unreachable hosts are affected.

Partial failures, i.e., individual link or port failures on edge and aggregate switches, can cause performance degradation due to loss of equal-cost paths. However, DiFS can cope with such kind of failures with a simple modification. When a link or switch experiences such failure, other switches connected to the switch/link can learn the loss of capacity from underlying link state protocols. These switches then move the flows on the failed link to other available links, or send EARs to notify the other switches. We do not present the details of failure recovery due to space limit, but they are implemented in our experiments.

Loss or delay of EARs on congested link may make DiFS degrade into local link balanced algorithm like ECMP. To avoid additional cost, control messages are delivered using UDP. However, a switch will keep sending control messages at each control loop if previous flow collisions have not been resolved. In the experiments, we also take the loss and delay of control messages into consideration. Experiments show that DiFS still converges in short time under congestion. Therefore the loss or delay of control messages has limited impact to network convergence. Besides, in order to avoid the oscillation problem caused by EARs arrived in the same time, we add a random time slot to the interval between two adjacent control loop.

4. EXPERIMENTAL RESULTS

In this section, we evaluate the performance of DiFS by comparing it with three representative adaptive routing solutions from different categories: ECMP (switch-only) [15], Hedera (centralized) [2], and Dard (host-based) [25]. *Note that both Hedera and Dard use global network information which is not available to switch-only methods.*

4.1 Methodology

Most existing studies use custom-built simulators to evaluate data center networks at large scale [2] [24] [22] [21]. Simulation is able to show the scalability of the protocols for large networks with dynamic traffic patterns, while testbed experiments can only scale to up to tens of hosts. We find many of them use a certain level of abstraction for TCP, which may result in inaccurate throughput results. [2] To perform experiments with accurate results, we developed a packet-level stand-alone simulator in which DiFS as well as other protocols are implemented in detail.[3] *Our simulator models individual packets, hence we believe it can better demonstrate real network performance.* TCP New Reno is implemented in detail as the transportation layer protocol. Our simulator models each link as a queue whose size is the delay-bandwidth product. A link's bandwidth is 1 Gbps and its average delay is 0.01 ms. Our switch abstraction maintains finite shared buffers and forwarding tables. In our experiments, we simulate multi-rooted tree topologies in different sizes. We use 16-host networks as small topologies and 1024-host networks for bulk analysis.

DiFS is compared with ECMP, Hedera, and Dard. For ECMP we implemented a simple hash function which uses the flow identifier of each tcp packet as the key. We implemented the Simulated Annealing scheduler of Hedera, which achieves the best performance among all schedulers proposed in Hedera [2]. We set the control loop period of Hedera to 0.01 seconds and Simulated Annealing iteration to 1000, both of which are exactly the same as their implementation. We also set the period of distributed control loop to 0.01 second for DiFS. As mentioned in Section 3.2, we focus on balancing the number of elephant flows among links. We use 100KB as the elephant threshold, same to the value used by other work [25].

Performance criteria. We evaluate the following performance criteria.

Aggregate throughput is the measured throughput of various traffic patterns using proposed routing protocols on the corresponding data center topology. It reflects how a routing protocol can utilize the topology bandwidth.

Flow completion time characterizes the time to deliver a flow, which is important because many data center applications are latency-aware. Besides the comparison of flow completion time among different protocols, we also care about the fairness of flow completion time of different flows routed by a same protocol.

Packet out-of-order ratio. Although all protocols in our experiments do not split flows, dynamic routing will still cause some out-of-order packets. The out-of-order ratio is measured to see whether a protocol will hurt TCP performance.

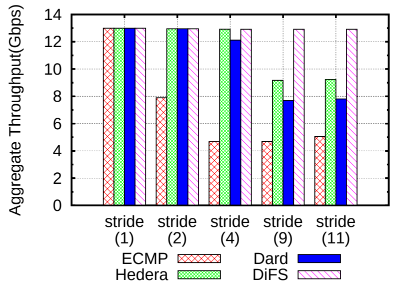
Convergence time is important to measure the stability of a dynamic routing protocol.

Control overhead. We measure the control message overhead in bytes.

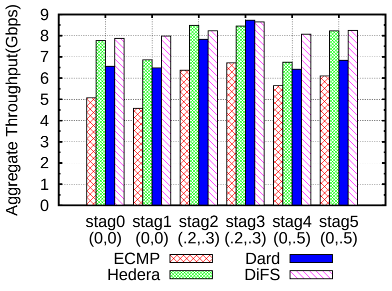
Traffic patterns. Similar to [2] and [25], we created a group of traffic patterns as our benchmark communication suite. These patterns are considered typical for cluster computing applications and can be either static or dynamic. For static traffic patterns, all flows are permanent. Dynamic traffic patterns refer to those in which flows start at different times. In this paper, we evaluate the performance of DiFS against dynamic patterns similar to data shuffle in cluster computing applications such as MapReduce [8]. The static patterns used by our experiments are described as follows:

[2]For example, the simulator developed in [2] only simulates each flow without performing per-packet computation, and uses predicted sending rate instead of implementing TCP. The simulator that implements MPTCP [24] has been used for performance evaluation by many other projects [22] [21]. However, it does not implement TCP ACKs and assume ACKs can all be successfully delivered.

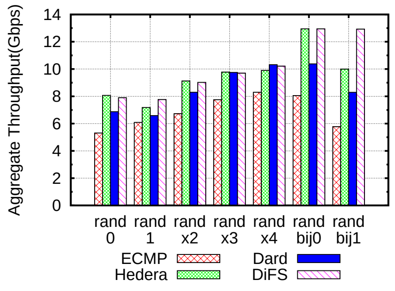
[3]We have also implemented DiFS on NS2, but experienced very slow speed when using NS2 for data center networks. We guess the existing studies do not use NS2 due to the same reason.



(a) Stride traffic pattern



(b) Staggered traffic pattern



(c) Random traffic pattern

Figure 5: Aggregate throughput comparison on small topologies

1. *Stride*(i): A host with index x sends data to a host with index $(x+i) mod(num\_hosts)$, where num_hosts is the number of all hosts in the network. This traffic pattern stresses out the links between the core and the aggregation layers with a large i.
2. *Staggered*(P_e, P_p): A host sends data to another host in the same edge layer with probability P_e, and to host in the same pod (but in the different edge layer) with probability P_p, and to hosts in different pods with probability $1 - P_e - P_p$.
3. *Random*: A host sends one elephant flow to some other end host in the same network with a uniform probability. This is a special case of *Randx*(x) where $x = 1$.
4. *Randx*(x): A host sends x elephant flows to any other end host in the same topology with a uniform probability.
5. *Randbij*: A host sends one elephant flow to some other host according to a bijective mapping of all hosts. This is a special case of *Random* pattern which may be created by certain cluster computing applications.

4.2 Small Topology Simulation Results

In this set of experiments, 16 hosts (acting as clients) first establish TCP connections with some designated peers (acting as servers) according to the specified traffic pattern. After that, these clients begin to send elephant flows to their peers constantly. Each experiment lasts 60 seconds and each host measures the incoming throughput during the whole process. We use the results for all hosts in the middle 40 seconds as the aggregate throughput.

Figure 5(a) shows the average aggregate throughput for a variety of Stride traffic patterns with different parameters. For stride parameter $i = 1$, all three methods have good performance. DiFS achieves highest throughput for all i values and outperforms ECMP significantly when i is greater than 2. DiFS has significant lead over Hedera and Dard when $i = 9$ and 11. Note a larger value of i indicates less traffic locality. Hence DiFS is more robust than the other methods for traffic locality.

Figure 5(b) shows the average aggregate throughput for Staggered patterns. Similar to the Stride results, DiFS has the highest throughput for most cases. In two cases (stag2(.2,.3) and stag3(.2,.3)), DiFS's throughput is marginally less than that of Hedera and Dard respectively. We might find that the absolute bandwidth values of all three methods in this set of experiments are less than those in the Stride experiments. According to our results on non-blocking switches and links (not shown in the figure), the average throughput for Staggered is also limited to 10-12 Gbps due to the hotspots created by the traffic pattern. DiFS results are actually very close to the limit.

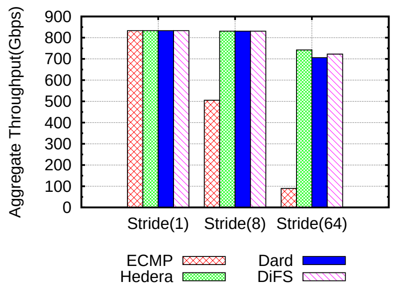
Figure 5(c) depicts the throughput for Random patterns. For all cases except one, DiFS outperforms the other three protocols. In Random experiments, DiFS outperforms ECMP in the average throughput by at least 33% for most traffic patterns. For particular patterns, this value can be higher than 100%. Compared to Hedera and Dard that uses global information, DiFS achieves higher throughput for the Randbij1 pattern and similar throughput for the others. We suspect there are two major reasons why Hedera achieves less bandwidth compared to DiFS: First, Hedera ignores intra pod flows and degrades to ECMP when intra pod flows are dominant. Second, Hedera with Simulated Annealing does not assign an explicit path for each flow. Instead Hedera assigns a core switch for every single host, which may result in bottlenecks on the links connecting aggregate switches and edge switches.

	ECMP	Hedera	Dard	DiFS
Shuffle time (s)	249.82	204.87	210.83	179.48
Aver. completion time (s)	224.78	178.53	191.25	157.20
Aver. throughput (Gbps)	4.31	5.30	4.61	6.10
Aver. out-of-order to in-order ratio	0.006	0.006	0.006	0.006
Max. out-of-order to in-order ratio	0.643	0.750	0.750	0.4
Aver. out-of-order window size	0.00	14.75	13.72	28.66
Max. out-of-order window size	0.00	69.00	68.00	123.00

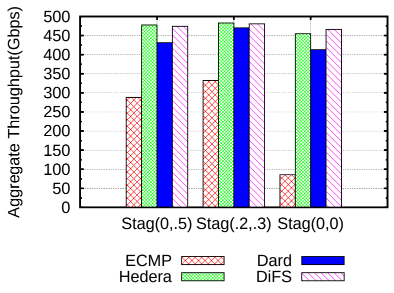
Table 1: results of shuffle experiments

4.3 MapReduce Traffic: Data Shuffle

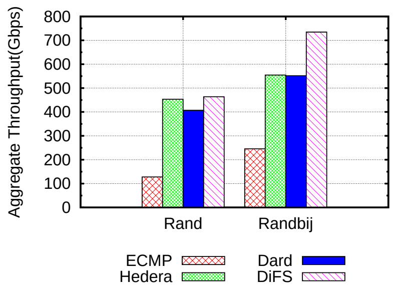
We conduct experiments of all-to-all Data Shuffle in the 16-host multi-rooted tree topology to evaluate the performance of DiFS under dynamic traffic patterns. Data Shuffle



(a) Stride traffic pattern

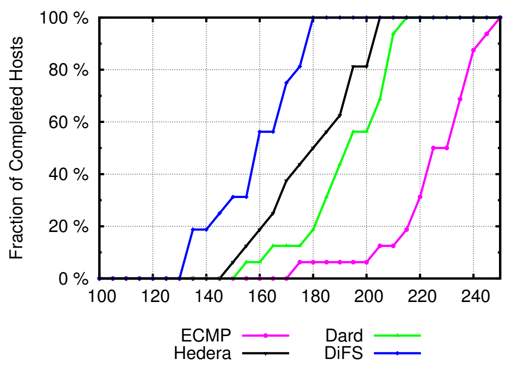


(b) Staggered traffic pattern



(c) Random traffic pattern

Figure 6: Aggregate throughput comparison for bulk analysis



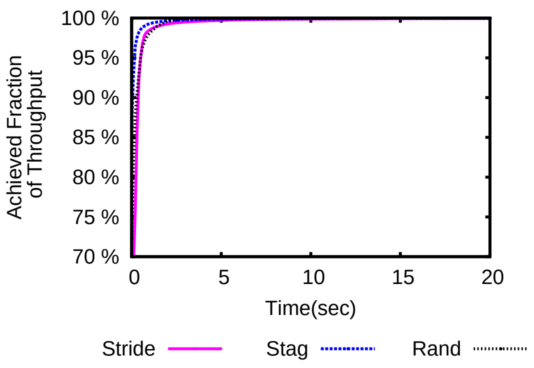


Figure 7: CDF of host completion time for data shuffle

Figure 8: Convergence time of DiFS in the 1024-host network

is an important operation for MapReduce-like applications. Each host (acting as reducer) in the network will sequentially receive a large amount of data (500MB in our simulation) from all other hosts (acting as mapper) using TCP. Therefore in total it is a 120 GB Data Shuffle. In order to avoid unnecessary hotspots, each host will access other hosts in a random order. We also assume there is no disk operation during the whole process. We measure the shuffle time, average completion time, and average throughput of the three methods. The shuffle time is the total time for the 120 GB Shuffle operation. The average completion time is the average value of the completion time of every host in the network. The average aggregate throughput refers to the sum of average throughput of every host.

We also measure two variables described in [28] during the Shuffle period in order to reflect the packet reordering problem. The first variable is the ratio of the number of packets delivered out-of-order to the number of packets provided in-order in TCP by the senders. The second variable is the out-of-order packet window size, defined as the average gap in the packet sequence numbers observed by the receivers.

Table 1 shows that our algorithm outperforms ECMP by 28%, Hedera by around 13%, and Dard by 15%, in aggregate throughput. DiFS achieves the least shuffle time and average completion time per flow. In addition DiFS causes less packet reordering compared to Hedera. ECMP has the least out-of-order packets because it is a static scheduling algorithm.

Figure 7 depicts the cumulative distribution function (CDF) of host completion time of the three methods. As observed from this figure, by the time DiFS finishes Shuffle operations, around 50% hosts of Hedera have completed their jobs and only 20% hosts of Dard and 5% hosts of ECMP have finished their jobs. In general DiFS fishes flows much faster than all other protocols. All four methods have obvious variation in completion time of different flows.

4.4 Large Topology Simulation Results

Figure 6 shows the aggregate throughput comparison using a 1024-host fat-tree network ($k = 16$). We can find that ECMP performs worse in a large topology, compared with its performance in the 16-host network using the same traffic patterns. We suspect this is because the chances of collisions in path assignment for static hash functions increase when topology gets larger. We also noticed that the performance gap between Hedera and DiFS shrinks in the 1024-host network compared to that in the 16-host network due to the decreased portion of intra pod flows. However, DiFS still has the highest aggregate throughput in general except for two traffic patterns among the three figures.

4.5 Convergence speed and control overhead

Convergence speed.

Convergence speed is a critical performance metric for DiFS, because DiFS is a distributed solution rather than a centralized algorithm. We measure the convergence speed

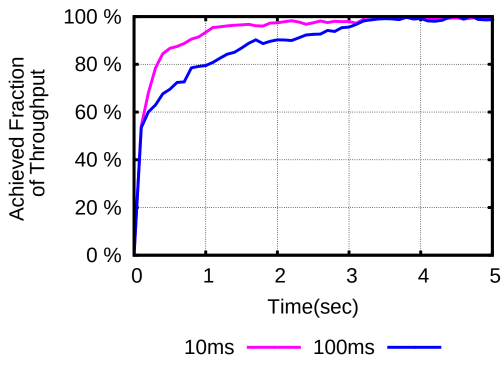


Figure 9: Convergence time of DiFS with different control loops

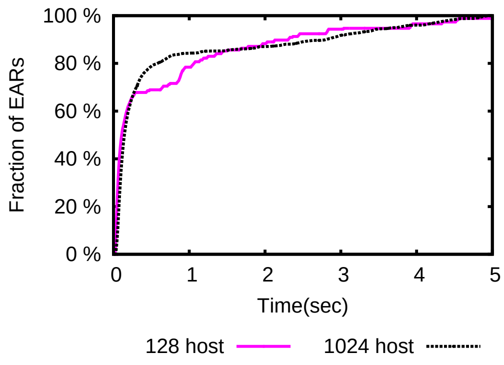


Figure 10: Cumulative distribution of EAR-receiving times

of DiFS for different traffic patterns using fat-tree topologies. In Figure 8 we show the achieved fraction of throughput of DiFS versus time for different traffic patterns in the 1024-host network. Even with Random traffic our algorithm may still converge to a steady state within 5 seconds. We also compare the convergence speed against the frequency of control loops in 1024 host networks using Randbij patterns. Figure 9 compares the convergence speed of DiFS with 10 ms control loops to 100 ms control loops. Although smaller frequency yields longer converge time, the throughput still converge to relatively stable state in three seconds and achieves more than 80% throughput in the first second. We may conclude that our protocol is robust under different frequencies of control loops.

Control Overhead.

As a distributed solution, the computation cost of DiFS is very low because switch only needs to consider its local flows. Hence we mainly focus on the communication overhead of DiFS, which is measured by the number of EAR messages. Aside from communication overhead, too many EAR messages may cause performance degradation because flows may be requested to change their paths back and forth.

Table 2 shows the number of EARs sent by switches under random traffic patterns in fat-tree networks with different sizes. In the measurement, we assume the size of each message is 26 Bytes, which includes the size of flow identifier and the address of recommended core or aggregate switch in an EAR. As shown in the table, for an 8192-host fat-tree network, DiFS only generates control messages in a total size of around 1 MB. Figure 10 shows the CDF of EAR-receiving times. Within 5 seconds, all EARs have sent and received, and around 80% EARs are received in the first second.

k	Host	EAR	Control Overhead (KB)
4	16	4	0
8	128	304	7.72
16	1024	4113	104.43
32	8192	45183	1147.22

Table 2: control overhead of DiFS for random traffic patterns

4.6 Flow count versus flow bandwidth consumption

DiFS use the number of elephant flows as the metric for load balancing. Obviously not all elephant flows have equal bandwidth consumptions, i.e., sending rates. As discussed in Section 3.2, DiFS cannot estimate the flow bandwidth consumption due to lack of global information. A substitution for bandwidth consumption estimation is to measure the sending rate of each flow on the current path. Unfortunately, a flow's current sending rate doest not reflect its maximum bandwidth consumption [2]. We also implemented a variant of DiFS which uses measured flow sending rate as the metric for load balancing, denoted as DiFS-FM. We compare both algorithms in Figure 11(a) and Figure 11(b). The results tell that DiFS-FM has similar performance compared to DiFS that uses flow count. Therefore there is no need to deploy a particular module to keep measuring sending rates in switches.

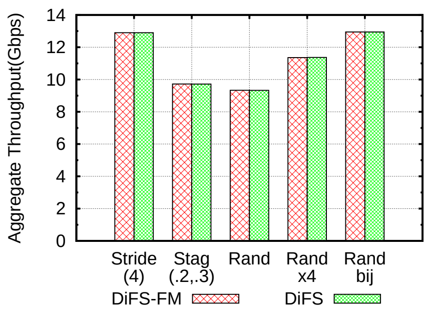
4.7 Summary of results

To summarize the performance evaluation, we compare the important properties of adaptive routing protocols in Table 3. Our results has shown that DiFS can achieve similar or even higher throughput than Hedera and Dard that require network-wide information for routing decisions. As a local, switch-only solution, DiFS does not have the limitations of central and host-based methods such as bottleneck of a single controller and massive monitoring messages. Compared to the state-of-art networking techniques, DiFS only requires either the SDN support or simple special switch logic.

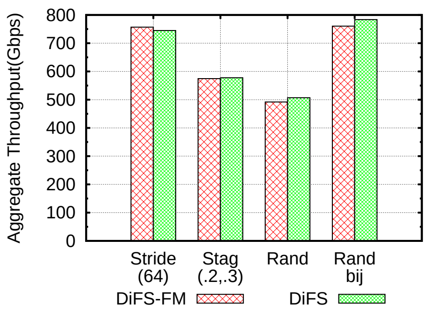
5. RELATED WORKS

Recently there have been a great number of proposals for data center network topologies that provide high bisection bandwidth [11–14, 20]. However, current routing protocols like ECMP [15] usually suffer from elephant flow collisions and bandwidth loss. Application layer scheduling like Orchestra [7] usually focuses on higher level scheduling policies such as transfer prioritizing and ignores multipathing issues in data center networks. Transport layer solutions like DCTCP [3] and MPTCP [24] optimize the resource share on fixed paths among flows. This work focuses on adaptive routing solutions.

Centralized flow routing [2, 6] usually relies on a central controller and schedules flow path at every control interval. Aside from the additional hardware and software support for



(a) 16-host network



(b) 1024-host network

Figure 11: Flow bandwidth measurement vs flow counting

Table 3: Important properties of adaptive routing protocols.

	ECMP [15]	Hedera [2]	Dard [25]	DiFS (this work)
Network throughput	Benchmark	Higher than ECMP	High than ECMP	≈ Hedera and Dard
Flow completion	Benchmark	faster than ECMP and Dard	faster than ECMP	faster than other three
Decision making	Local info.	Network-wide info.	Network-wide info.	Local info.
Scalability problem?	Scalable	Bottleneck of a single controller	Massive monitoring msgs	Scalable
Compatibility	Standard	SDN support & monitoring tools	Changes on hosts	SDN support or switch logic

communication and computation, centralized solutions may be hard to scale out due to the single point of the controller. Recent research [5,17] shows that centralized solutions must employ parallelism and fast route computation heuristics to support observed traffic patterns.

Host-based solutions [25] enable end hosts select flow path simultaneously to enhance parallelism. Dard [25] allows each host to select flow path based on network conditions. However, Dard has potential scalability issues due to massive monitoring messages to every host. Besides, deployment of host-based solutions requires updates on legacy systems and applications.

Switch-only protocols [9,16,21,28] are also proposed. However most of them require flow splitting which may cause significant packet reordering. TeXCP [16], as an online distributed Traffic Engineering protocols, performs packet-level load balancing by using splitting schemes like FLARE [23]. Localflow [21] refines a naive link balancing solution and minimizes the number of flows that are split. Zahavi *et al.* [28] also describes a general distributed adaptive routing architecture for Clos networks [11]. Dixit *et al.* [9] uses random packet spraying to split flows to multiple paths to minimize the hurts to TCP. DiFS does not split a flow in order to avoid packet reordering.

6. CONCLUSION

This paper proposes DiFS, a local, lightweight, and switch-only protocol for adaptive flow routing in data center networks. Switches running DiFS cooperate to achieve flow-to-link balance by avoiding both local and remote collisions. Experimental results show that our algorithm can outperform the well-known distributed solution ECMP, a centralized scheduling algorithm Hedera, and a host-based protocol Dard. We will investigate flow scheduling for general network topologies in future work.

7. ACKNOWLEDGMENTS

This research is supported by University of Kentucky Faculty Startup funding. We thank the anonymous reviewers of ANCS'14 for their constructive comments.

8. REFERENCES

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proc. of ACM SIGCOMM*, 2008.

[2] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: dynamic flow scheduling for data center networks. In *Proceedings of USENIX NSDI*, 2010.

[3] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Dctcp: Efficient packet transport for the commoditized data center. In *Proceedings of ACM SIGCOMM*, 2010.

[4] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *Proc. of SIGCOMM*, 2011.

[5] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of ACM IMC*, 2010.

[6] T. Benson, A. Anand, A. Akella, and M. Zhang. Microte: fine grained traffic engineering for data centers. In *Proc. of ACM CoNEXT*, 2011.

[7] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with orchestra. In *Proc. of ACM SIGCOMM*, 2011.

[8] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 2008.

[9] A. Dixit, P. Prakash, Y. C. Hu, and R. R. Kompella. On the impact of packet spraying in data center networks. In *Proc. of IEEE INFOCOM*, 2013.

[10] A. Dixit, P. Prakash, and R. R. Kompella. On the efficacy of fine-grained traffic splitting protocols in data center networks. In *Proceedings of ACM SIGCOMM*, 2011.

[11] 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 *Proceedings of ACM SIGCOMM*, 2009.

[12] A. Greenberg, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Towards a next generation data center architecture: Scalability and commoditization. In *Proc. of ACM PRESTO*, 2008.

[13] C. Guo et al. Dcell: a scalable and fault-tolerant network structure for data centers. In *Proc. of ACM SIGCOMM*, 2008.

[14] C. Guo et al. Bcube: a high performance, server-centric network architecture for modular data centers. In *Proc. of ACM SIGCOMM*, 2009.

[15] C. Hopps. Analysis of an equal-cost multi-path algorithm. *RFC 2992*, 2000.

[16] S. Kandula, D. Katabi, B. Davie, and A. Charny. Walking the tightrope: responsive yet stable traffic engineering. In *Proceedings of ACM SIGCOMM*, 2005.

[17] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of data center traffic: measurements & analysis. In *Proceedings of ACM IMC*, 2009.

[18] K. C. Leung, V. Li, and D. Yang. An overview of packet reordering in transmission control protocol (tcp): Problems, solutions, and challenges. *IEEE Transactions on Parallel and Distributed Systems*, 2007.

[19] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 2008.

[20] R. N. Mysore et al. Portland: a scalable fault-tolerant layer 2 data center network fabric. In *Proceedings of ACM SIGCOMM*, 2009.

[21] S. Sen, D. Shue, S. Ihm, and M. J. Freedman. Scalable, opitmal flow routing in datacenters via local link balancing. In *Proceedings of ACM CoNEXT*, 2013.

[22] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey. Jellyfish: Networking data centers randomly. In *Proc. of USENIX NSDI*, 2012.

[23] S. Sinha, S. Kandula, and D. Katabi. Harnessing tcps burstiness using flowlet switching. In *Proc. of ACM HotNets*, 2004.

[24] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, implementation and evaluation of congestion control for multipath tcp. In *Proceedings of USENIX NSDI*, 2011.

[25] X. Wu and X. Yang. Dard: Distributed adaptive routing for datacenter networks. In *Proceedings of IEEE ICDCS*, 2012.

[26] X. Yang, D. Clark, and A. Berger. Nira: A new inter-domain routing architecture. *IEEE/ACM Transactions on Networking*, 2007.

[27] M. Yu, L. Jose, and R. Miao. Software defined traffic measurement with opensketch. In *Proc. of USENIX NSDI*, 2013.

[28] E. Zahavi, I. Keslassy, and A. Kolodny. Distributed adaptive routing for big-data applications running on data center networks. In *Proceedings of ACM/IEEE ANCS*, 2012.

Appendix

Proof of Theorem 3.1:

Proof. Let x and y be arbitrary two aggregate switches. Let n_{ae} be the number of flows from edge switch e to aggregate switch a.

$$n(x) = \sum n_{xe}$$

$$n(y) = \sum n_{ye}$$

Since $|n_{xe} - n_{ye}| \leq \delta$ for every edge switches e and there are $k/2$ edge switches in a pod,

$$|n(x) - n(y)| \leq \sum |n_{xe} - n_{ye}| \leq \delta \cdot k/2$$

Hence $MAX_a - MIN_a \leq \delta \cdot k/2$. $\square$

Proof of Theorem 3.2:

Proof. The $(k/2)^2$ cores can be divided into $k/2$ groups $g_1, g_2, ..., g_{k/2}$, each of which contains $k/2$ cores that receive flows from a same group of aggregate switches.

Suppose x and y are two cores. If they belong to a same group, we can prove $n_x - n_y \leq k/2$ using a way similar to the proof of Theorem 3.1. Consider that they belong to different groups. For a pod p, x and y connect to two different switches in p, because they are in different core groups. Let s_{a1} and s_{a2} denote the switches connecting to x and y respectively. We have $n(s_{a1}) - n(s_{a2}) \leq k/2$ according to Theorem 3.1. Hence the average numbers of flows from s_{a1} and s_{a2} to each core are $\frac{n(s_{a1})}{k/2}$ and $n(s_{a2})/2$ respectively.

$$\frac{n(s_{a1})}{k/2} - \frac{n(s_{a2})}{k/2} \leq 1$$

Let n_{pc} denote the number of flows from pod p to core c. We have $n_{px} - \frac{n(s_{a1})}{k/2} \leq 1$ (BO of s_{a1}), and $\frac{n(s_{a2})}{k/2} - n_{py} \leq 1$ (BO of s_{a2}). Hence

$$n_{px} - n_{py} \leq 1 + \frac{n(s_{a1})}{k/2} - \frac{n(s_{a2})}{k/2} + 1 \leq 3$$

$$n_x - ny = \sum_p n_{px} - \sum_p n_{py} = \sum_p (n_{px} - n_{py}) \leq 3k$$

$\square$