DARD: Distributed Adaptive Routing for Datacenter Networks

TR-2011-01

Xin Wu Xiaowei Yang
Dept. of Computer Science, Duke University
{xinwu, xwy}@cs.duke.edu

ABSTRACT

Datacenter networks typically have many paths connecting each host pair to achieve high bisection bandwidth for arbitrary communication patterns. Fully utilizing the bisection bandwidth may require flows between the same source and destination pair to take different paths to avoid hot spots. However, the existing routing protocols have little support for load-sensitive adaptive routing. This work proposes DARD, a Distributed Adaptive Routing architecture for Datacenter networks. DARD allows each end host to adjust traffic from overloaded paths to underloaded ones without central coordination. We use openflow implementation and simulations to show that DARD can effectively use the network's bisection bandwidth. It outperforms previous solutions based on random flow-level scheduling by 10%, and performs similarly to previous work that assigns flows to paths using a centralized scheduler but without its scaling limitation. We use competitive game theory to show that DARD's flow scheduling algorithm is stable. It makes progress in every step and converges to a Nash equilibrium in finite steps. Our evaluation results suggest its gap to the optimal solution is likely to be small in practice.

1. INTRODUCTION

Datacenter network applications, *e.g.*, MapReduce and network storage, often require tremendous intra-cluster bandwidth [9] to transfer data among distributed components. Increasingly, the components of an application cannot always be placed on machines close to each other (*e.g.*, within a rack) for two main reasons. First, an application may use common services provided by a datacenter network, *e.g.*, DNS, web search, and storage. Those services may not reside in nearby machines. Second, the auto-scaling feature offered by a datacenter network [1, 5] allows an application to create dynamic instances when its workload increases. Where those instances will be placed depends on machine availability, and are not guaranteed to be close to the other instances of the application.

Therefore, it is important for a datacenter network to have high bi-section bandwidth to avoid hot spots between any pair of hosts. Today's datacenter networks often use commodity Ethernet switches to form multi-rooted tree topologies [23] (*e.g.*, fat-tree [8] or Clos topology [16]) to achieve this goal. A multi-rooted tree topology has multiple equal-cost paths connecting any host pair. A flow can use an alternative path if one path is overloaded.

However, it requires load-sensitive adaptive routing to fully take advantage of the multiple paths connecting a pair of hosts. Yet existing routing protocols have little support to load-sensitive routing. Existing work uses two approaches to evenly distribute flows among multiple paths: random flow-level scheduling and centralized scheduling. Equal-Cost-Multi-Path forwarding (ECMP) [21] is one representative example of the random flow-based scheduling. It forwards a packet according to a hash of the selected fields of the packet header. Since a flow's packets share the same hash value, they will take the same next hop to avoid the TCP packet reordering problem. Flow level Valiant Load Balancing (VLB) [16], which forwards a flow to a randomly chosen core switch, is another example. Random flow-level scheduling requires little computation and memory resources, and thus is scalable. However, multiple long flows (which we refer to as elephant flows) may collide on the same link to create a permanent bottleneck [9].

In contrast, Hedera [9] adopts a centralized scheduling approach. It detects elephant flows at the edge switches and collects the flow information at a centralized server, which will further calculate a flow assignment and distribute the decisions to the switches periodically. Centralized scheduling is able to obtain a nearly optimal flow assignment. However, it is not scalable for at least three reasons. First, the controller's crash degrades the entire system to ECMP. Second, the amount of control messages to or from the centralized controller is proportional to the product of network size and the traffic load, which may congest the bottleneck connecting the controller and the network. Third, if the network diameter is large, some switches may suffer from long flow setup delay. This delay will further cause asynchronous flow table update and degrade the adaptive flow scheduling to ECMP.

Given the limitations of existing mechanisms, we propose DARD, a distributed adaptive routing system for data center networks. Ideally, we intend to make DARD as scalable as random flow-level scheduling and achieve a flow assignment comparable with the centralized scheduling. Recent datacenter network traffic measurement studies [10, 16, 23] show that both short-term delay-sensitive flows and long-term throughput-sensitive flows exist in datacenter networks. For the rest of this paper, we use the term *elephant flow* to refer to a continuous flow set up by a TCP connection which lasts for at least 10 seconds. We focus our research on elephant flows for two reasons. First, a short-term flow usually lasts for seconds or even less than a second [10, 23]. It is not practical to shift a short-term flow from path to path in its life cycle. Meanwhile, existing work shows that random flow-level scheduling already performs well enough on short-term traffic [16]. Second, the quantity of elephant flows may not be as many as short-term flows, but wide area network measurement shows elephant flows may occupy a significant fraction of the total bandwidth [27]. An adaptive flow scheduling approach should be robust to this kind of traffic that stresses out the network resources.

In a high level description, DARD uses hierarchical addressing to facilitate flows between the same source destination pair to take different paths. It enables a source to monitor the state of all paths to the destination without flooding the network. The source further selfishly shifts flows off from overloaded paths to underloaded ones. We prove in appendix that this selfish-routing-like approach is stable and converges to a Nash equilibrium in finite steps.

We have implemented a DARD prototype on DeterLab [6] and a simulator on ns-2. Evaluation results show that when inter-pod traffic is dominant, DARD outperforms random flow-level scheduling and the performance gap to the centralized scheduling is small. When intra-pod traffic is dominant, DARD outperforms centralized scheduling. The results also show that 90% of the flows switch their paths less than 3 times in their life cycles, which means DARD introduces little path oscillation.

To the best of our knowledge, this is the first work that uses a distributed, scalable, and stable adaptive routing algorithm to load-balance datacenter network traffic. The rest of this paper is organized as follows. Section 2 describes DARD's design goals and system components. In Section 3, we introduce the system implementation details. We evaluate DARD in section 4. Related work is discussed in Section 5. Section 6 concludes our work.

2. DARD DESIGN

In this section, we describe DARD's design in detail. We first highlight the system design goals. Then we present an overview of the system and identify the design challenges. Finally, we introduce the detail mechanisms, including the hierarchical addressing scheme that encodes a path using the source and destination addresses, the on-demand path monitoring and the selfish flow scheduling.

2.1 Design Goals

Ideally, we strive to make DARD as scalable as random flow-level scheduling and achieve a flow assignment comparable to the centralized scheduling. More specifically, we want DARD to meet the following design goals.

Maximize the minimum flow rate. Maximizing the minimum flow rate is desirable because it does not harm the system efficiency and improves fairness to some extent. TCP's additive increase and multiplicative decrease (AIMD) adjustment algorithm combined with fair queuing tries to achieve max-min fairness. As shown in the appendix the minimum link bandwidth divided by the elephant flow numbers is a good approximation of the minimum flow rate. As a result, we use maximizing the minimum value of link bandwidth over the elephant flow numbers as the scheduling objective to simplify the system design.

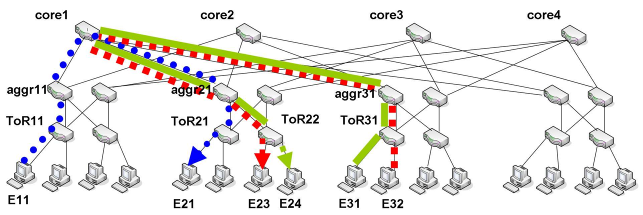
Monitor the network states and schedule the traffic without centralized coordination. As analyzed in Section 1, single point failure and asynchronous flow table update degrade centralized scheduling to ECMP. To make the system scalable, we strive to make DARD a complete distributed system, in which each component relies on itself to collect network states and to make decisions benefiting itself but being harmless to others.

Modify end hosts instead of switches. Modifying routers or switches is expensive, but the end host operating systems can be updated easily by applying software patches. As a result, we decide to make most changes to the end hosts. In addition, this modification should be transparent to existing applications.

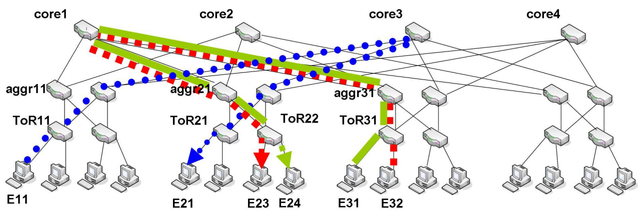
2.2 DARD Overview

In this section, we describe the system design intuition and a toy example to show how DARD works. We call the *B*andwidth of a switch-switch link *O*ver the *N*umber of elephant *F*lows via that link the $BoNF$. The $BoNF$ *of a path* refers to the minimum $BoNF$ along that path. Unless otherwise noted, we do not consider the host-switch link because a flow cannot bypass this first/last link by switching paths. Take Figure 1 as an example. There are three elephant flows: $Flow_0$ from E_{11} to E_{21}, $Flow_1$ from E_{31} to E_{24} and $Flow_2$ from E_{32} to E_{23}. All of them go through $core_1$. Assume the bandwidth of all the links is 1 unit, then the global minimum $BoNF$ is $\frac{1}{3}$, where 3 is the number of flows via the most congested link $core_1$-$aggr_{21}$. If a link has no flow, its $BoNF$ is ∞. In DARD, each source-destination pair monitors the equal-cost paths connecting them. We call the path goes through $core_i$ the $path_i$. A source-destination pair also maintains a vector, whose ith item is $path_i$'s $BoNF$. For example, (E_{11}, E_{21})'s vector is [$\frac{1}{3}$, 1, ∞, ∞] at step 0.

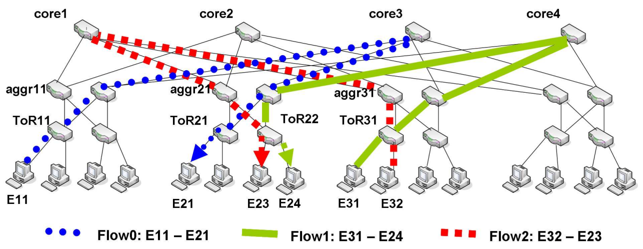
DARD's design goal is to maximize the minimum $BoNF$. The intuition is to let each source-destination pair to selfishly shift flows to increase the minimum $BoNF$. Figure 1 together with Table 1 shows a scheduling example. The *updated vector* in Table 1 means the updated $BoNF$ vector at the beginning of each round. The *estimated vector* is a source-destination pair's estimation of the $BoNF$ vector if it switches one flow to a different path. The source-destination pair in bold will shift one flow to a different path in the next round.



(a) Round 0



(b) Round 1



(c) Round 2

Figure 1: end hosts in DARD shift traffic selfishly from overloaded paths to underloaded ones.

Round 0 is the initial state. The source-destination pair (E_{11}, E_{21}) assumes there is no overlap among its four paths and further estimates if it shifts one flow off from $path_1$ to $path_3$, the vector will change from [$\frac{1}{3}$, 1, ∞, ∞] to [$\frac{1}{2}$, 1, 1, ∞]. The minimum $BoNF$ increases from $\frac{1}{3}$ to $\frac{1}{2}$. As a result, (E_{11}, E_{21}) shifts one flow to $path_3$, as shown in Figure 1(b).

In round 1, all sources-destination pairs first update their $BoNF$ vectors. (E_{31}, E_{24}) assumes the 4 paths connecting them are not overlapped and estimates if it shifts one flow off from $path_1$ to $path_4$, its vector will change from [$\frac{1}{2}$, $\frac{1}{2}$, 1, ∞] to [1, $\frac{1}{2}$, 1, 1]. Even though the minimum $BoNF$ is still $\frac{1}{2}$, the quantity of the minimum $BoNF$s decreases from 2 to 1. As a result, (E_{31}, E_{24}) shifts one flow to $path_4$, as shown in Figure 1(c).

In round 2, all sources-destination pairs first update their $BoNF$ vectors. Then each of the pairs estimates the new vector if it shifts one flow off from the path with smallest $BoNF$ to the path with the largest $BoNF$. (E_{31}, E_{24}) and (E_{32}, E_{23}) will find out this scheduling will eventually decrease the minimum $BoNF$ from 1 to $\frac{1}{2}$. (E_{11}, E_{21}) will notice shifting one flow to a different path does not increase the minimum $BoNF$ either. As a result, no flow shifting will be triggered at this round. The scheduling process converges after 2 rounds. To simplify this toy example, we make round 1 come before round 2. As a real system, each end host behaves independently without coordination.

In the above toy example, each source-destination pair assumes the paths connecting them do not share links, which is not true for fatree topologies. However we use this assumption for two reasons. First this assumption is only used to estimate the $BoNF$ vector in the next round. This rough estimation is good enough to prevent a scheduling that decreases the minimum $BoNF$. In other words, this rough estimation will not make the system perform worse. We prove in the appendix that DARD's flow scheduling algorithm is stable. It converges to a Nash equilibrium in finite rounds. Second, this assumption greatly simplifies the selfish scheduling process as described in Section 2.5.

round	src-dst pair	updated vector				estimated vector			
0	$\mathbf{E11 \rightarrow E21}$	$\frac{1}{3}$	1	∞	∞	$\mathbf{\frac{1}{2}}$	**1**	**1**	∞
	$E31 \rightarrow E24$	$\frac{1}{3}$	$\frac{1}{2}$	∞	∞	$\frac{1}{2}$	$\frac{1}{2}$	1	∞
	$E32 \rightarrow E23$	$\frac{1}{3}$	$\frac{1}{2}$	∞	∞	$\frac{1}{2}$	$\frac{1}{2}$	1	∞
1	$E11 \rightarrow E21$	$\frac{1}{2}$	∞	1	1	$\frac{1}{2}$	1	∞	1
	$\mathbf{E31 \rightarrow E24}$	$\frac{1}{2}$	$\frac{1}{2}$	1	∞	**1**	$\mathbf{\frac{1}{2}}$	**1**	**1**
	$E32 \rightarrow E23$	$\frac{1}{2}$	$\frac{1}{2}$	1	∞	1	$\frac{1}{2}$	1	1
2	$E11 \rightarrow E21$	1	∞	1	1	1	1	∞	1
	$E31 \rightarrow E24$	1	1	1	1	1	$\frac{1}{2}$	1	∞
	$E32 \rightarrow E23$	1	1	1	1	∞	$\frac{1}{2}$	1	1

Table 1: Bottleneck elephant flow numbers on each path

To achieve the system design goals and enable DARD to behave as we expected, we face three design challenges. First, how to build an multipath routing scheme with little modification to the switch. Second, how to monitor the network states in a distributed way and prevent flooding as much as possible. Third, how to design a distributed scheduling algorithm to increase the minimum $BoNF$ round by round.

2.3 Addressing and Routing in DARD

To prevent TCP suffering from reordering, a flow in DARD only uses a single path at any time. But DARD enables the flow to switch from path to path. We use a hierarchical addressing scheme that encodes a path using source and destination addresses as NIRA [7] did to achieve this design requirement.

Datacenter networks are usually constructed as a multi-rooted tree topology. Take Figure 2 as an example, all the switches and end hosts highlighted by solid circles form a tree with root $core_1$. Three other similar trees exist in the same topology. The strictly hierarchical structure facilitates adaptive routing through some special addressing rules [8]. We borrow the idea from NIRA [7] to separate an end-to-end path into *uphill* and *downhill* segments and encode a path in the source and destination addresses. In DARD, each of the core switch obtains a unique address prefix and then allocates nonoverlapping subdivisions of the address prefix to each of its subtrees. The subtrees will recursively allocate nonoverlapping subdivisions of the prefix to lower hierarchies. By this hierarchical prefix allocation scheme, each network device receives multiple IP addresses, each of which represents the device's position in one of the trees.

We use the private prefix 10.0.0.0/8 to illustrate how prefixes are allocated along the hierarchies. Every 6 bits of an address' last 24 bits represent one hierarchy. In Figure 2, $core_1$ is allocated with prefix 10.4.0.0/14. It then allocates prefixes 10.4.16.0/20 and 10.4.32.0/20 to its two subtrees. The subtree rooted from switch $aggr_{11}$ will further allocate prefixes 10.4.16.64/26 and 10.8.16.64/26 to lower hierarchies. Prefixes are allocated recursively until end hosts get multiple addresses.

For a general fat-tree topology consisting of p-port switches, each switch indexes its ports from 1 to p. We ignore the first 8 bits of the address since they are constant, treat every 6 bits of the last 24 bits as one group and represent each of the groups in decimal notation, *e.g.*, address 10.4.16.66 will be noted as (1, 1, 1, 2) and prefix 10.4.16.64/26 will be noted as (1, 1, 1, 0)/18. Each core is allocated a prefix $(core, 0, 0, 0)/6$, where $core$ stands for the index of the core switch in the range of $[1, \frac{p^2}{4}]$. A core then allocates a prefix $(core, port_{core}, 0, 0)/12$ to its subtrees, in which $port_{core}$ stands for the core's port index connecting that subtree. The aggregation switch further allocates prefixes in form of $(core,\ port_{core}, port_{aggr}, 0)/18$ to the lower hierarchy, where $port_{aggr}$ stands for the aggregation switch's port index connecting the subsequent branch. At the bottom of the hierarchy, every end host get $\frac{p^2}{4}$ addresses, each of which stands for its position in one of the trees. For example, in Figure 2 every end host gets four addresses, each of which is originated from the corresponding root. In case more IP addresses than network cards are assigned to each end host, we propose to use IP alias to configure multiple IP addresses to one network interface. The latest operating systems support a large number of IP alias to be added to one network interface, *e.g.*, Linux kernel 2.6 sets the limit to be $256K$ IP alias [3], Windows NT 4.0 has no limitation on the number of IP addresses that can be bound to a network interface [4], even the Linux kernals before 2.2 allows 255 IP alias on one network interface, which is sufficient to support the fattree topology containing 8192 end hosts.

One nice property of this hierarchical addressing scheme is that one end host address uniquely encodes the sequence of upper-level switches that allocate that address, *e.g.*, E_{11}'s address 10.4.16.66 uniquely encodes the address allocation sequence $core_1 \rightarrow aggr_{11} \rightarrow ToR_{11}$. A source-destination address pair can be further used to uniquely identify a path, *e.g.*, we can use the source-destination pair highlighted by dotted circles to uniquely encode the dotted path from E_{11} to E_{21} through $core_1$. We call the partial path encoded by source address the *uphill path* and the partial path encoded by destination address the *downhill path*. To shift flows from path to path, we can simply use different source and destination address combinations without dynamically reconfiguring the routing tables.

To forward a packet, each switch stores a *downhill table* and an *uphill table* as described in [7]. The uphill table keeps the entries for the prefixes allocated from upstream switches and the downhill table keeps the prefixes allocated to the downstream switches. Table 2 shows the switch $aggr_{11}$'s downhill and uphill table. When a packet arrives, a switch first looks up the destination address in the downhill table using the longest prefix matching algorithm. If a match is found, the packet will be forwarded to the corresponding downstream switch. Otherwise, the switch will look up the source address in the uphill table to forward the packet to the corresponding upstream switch. A core switch only has the downhill table since there is no higher hierarchies.

downhill talbe	
Prefix	Port
10.4.16.64/26	1
10.4.16.128/26	2
10.8.16.64/26	1
10.8.16.128/26	2
uphill talbe	
Prefix	Port
10.4.0.0/14	3
10.8.0.0/14	4

Table 2: **$aggr_{11}$'s downhill and uphill routing tables.**

In fact, the downhill-uphill-looking-up scheme is unnecessary for a fat-tree topology, because once a core switch is chosen as the intermediate node, the uphill and downhill paths are uniquely determined, *e.g.*, to forward a packet, $aggr_{11}$ only needs to look up the destination address in Table 3 to decide the exit port. However, not all multi-rooted trees share the same property, *e.g.*, picking a core switch as the intermediate node cannot determine either the uphill path or the downhill path for a Clos network. As a result, we need both uphill and downhill tables for any generic multi-rooted trees. The downhill-uphill-looking-up scheme modifies current switch's forwarding algorithm. However an increasing number of switches support highly customized forwarding policies. In our implementation described in Section 3, we use OpenFlow enabled switch to support this forwarding algorithm.

routing talbe	
Prefix	Port
10.4.16.64/26	1
10.4.16.128/26	2
10.8.16.64/26	1
10.8.16.128/26	2
10.4.0.0/14	3
10.4.0.0/14	4

Table 3: **Ordinary routing table works for $aggr_{11}$.**

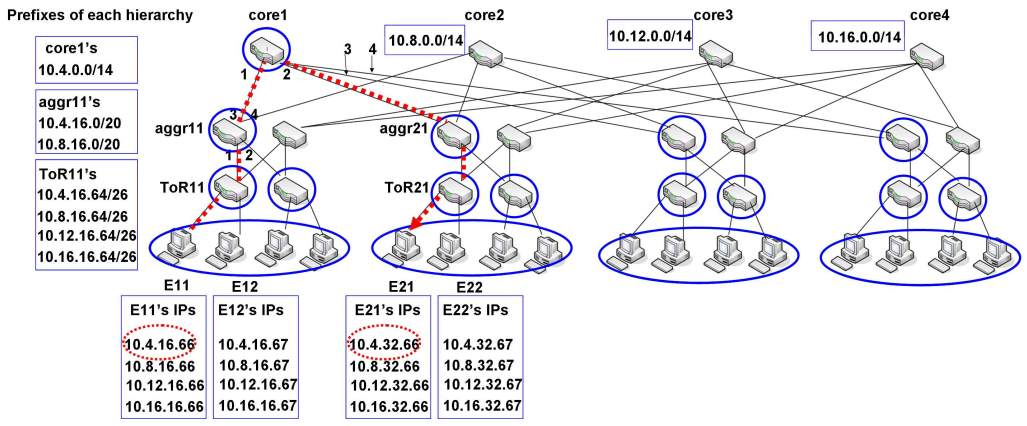


Figure 2: **DARD's addressing and routing. Prefix 1.0.0.0/8 is allocated hierarchically along the tree rooted at $core_1$. E_{11}'s address 1.1.1.2 encodes the uphill path ToR_{11}-$aggr_{11}$-$core_1$. E_{21}'s address 1.2.1.2 encodes the downhill path $core_1$-$aggr_{21}$-ToR_{21}.**

Each network component is also assigned a location independent IP address, *ID*, which uniquely identifies the component and is used for making TCP connections. The mapping from IDs to underlying IP addresses is maintained by a DNS-like system and cached locally. To deliver a packet from a source to a destination, the source encapsulates the packet with a proper source-destination address combination. Switches in the middle will forward the packet according to the encapsulated packet header. When the packet arrives at the destination, it will be decapsulated and passed to upper layer protocols.

2.4 Monitoring Path State

DARD utilizes ECMP as the default routing mechanism. Once a source detects an elephant flow, it starts to monitor all the available paths to the destination and shift flows from overloaded paths to underloaded ones by choosing different source-destination address pairs. This section will describe the monitoring scheme in detail.

Designing a scalable monitoring scheme is challenging because we need to monitor all paths' states without flooding the network. We introduce two techniques to resolve this challenge, *On-demand Monitoring* and *Path State Assembling*.

2.4.1 On-demand Monitoring

A $monitor$ running on a source end host is responsible to track the $BoNF$s of all the paths connecting the corresponding source-destination ToR switch pair, *e.g.*, in Figure 1(a), the monitor running on E_{32} tracks the number of elephant flows of all the four paths between ToR_{31} and ToR_{22}.

We propose the *On-demand Monitoring* scheme to limit the number of monitors, in which a monitor is generated only when it is necessary. In DARD, once an end host detects an outgoing elephant flow, it will check if there is already a monitor tracking the corresponding source-destination ToR switch pair. If not, the source end host will then generate a new monitor. For example, in Figure 1(a), When E_{32} detects $Flow_2$, it does not have the monitor to track the paths' states between ToR_{31} and ToR_{22}. Therefor it will generate such a monitor. Suppose before $Flow_2$ being transfered, another elephant flow from E_{32} to E_{24} is detected. Since both (E_{32}, E_{23}) and (E_{32}, E_{24}) share the same source-destiantion ToR switch pair (ToR_{31}, ToR_{22}), no new monitor will be generated at E_{32}. E_{32} will release this monitor when all its elephant flows to ToR_{22} are transfered.

2.4.2 Path State Assembling

One intuitive method to monitor $BoNF$s of all the paths connecting a source-destination ToR switch pair is probing. A source sends probe packets periodically along all the paths. The $BoNF$ field of the probe packet is updated at each hop and echoed back at the destination. This method requires to flood probes and to modify switches to update the probe packets.

We propose the *Path State Assembling* scheme to resolve probing's unscalable problem. We assume that switches support state querying. This assumption is valid because existing "networking operating system", *e.g.*, OpenFlow and NOX, which provides convenient state querying APIs, has been deployed in commercial switches [19]. In DARD, a switch's state refers to the exit ports' bandwidths and the number of elephant flows. The high level idea of Path State Assembling is to let each monitor periodically query every switch along all the paths for the switch states and assemble the collected states to $BoNF$ of each path. Figure 3 illustrates an example, in which elephant flows are transferring from E_{21} to E_{31}, and a monitor is running on E_{21} to track the $BoNF$ along all the paths connecting ToR_{21} and ToR_{31}. Instead of sending probe messages along all the four paths, the monitor sends queries to the switches highlighted by dotted circles, asking for the states at their exit ports. After receiving all the replies, the monitor assembles the collected flow numbers and bandwidths to the $BoNF$s of all the paths.

The set of switches to query is determined by the topologies and configurations. For both fat-tree and Clos network topologies, this set of switches include (1) the source ToR

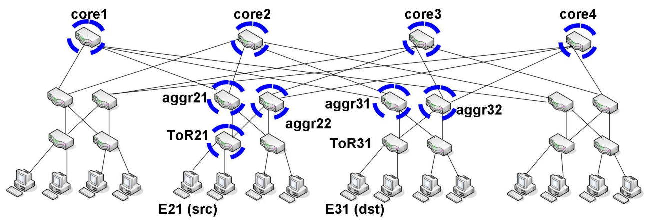


Figure 3: **Path State Assembling.**

Algorithm selfish flow scheduling

```
1:  for each monitor $M$ on an end host do
2:      $max\_index = 0$; $max\_BoNF = 0.0$;
3:      $min\_index = 0$; $min\_BoNF = \infty$;
4:      for each $i \in [1, M.PV.length]$ do
5:          if $M.FV[i] > 0$ and
            $max\_BoNF < M.PV[i].BoNF$ then
6:              $max\_BoNF = M.PV[i].BoNF$;
7:              $max\_index = i$;
8:          else if $min\_BoNF > M.PV[i].BoNF$ then
9:              $min\_BoNF = M.PV[i]$.BoNF;
10:             $min\_index = i$;
11:         end if
12:     end for
13: end for
14: if $max\_index \neq min\_index$ then
15:     $estimation = \frac{M.PV[max\_index].bandwidth}{M.PV[max\_index].flow\_numbers+1}$
16:     if $estimation - min\_BoNF > \delta$ then
17:         shift one elephant flow from path $min\_index$ to
            path $max\_index$.
18:     end if
19: end if
```

switch, (2) the aggregation switches directly connected to the source ToR switch, (3) all the core switches and (4) the aggregation switches directly connected to the destination ToR switch. These four groups of switches covers all the equal-cost paths between the source and destination ToRs.

2.5 Selfish Flow Scheduling

Each monitor tracks the path states by periodically sending queries to related switches and assembling replies to $BoNF$ of each path. Link l's state, noted as S_l, consists of a triple ($bandwidth$, $flow\_numbers$, $BoNF$), where the $BoNF$ is the $bandwidth$ divided by $flow\_numbers$. Path p's state, noted as S_p, is the state of the most congested link along that path, *i.e.*, the S_l with the smallest $BoNF$. In DARD each monitor maintains two vectors, the *path state vector* (PV), whose ith item is the state of the ith path, and the *flow vector* (FV), whose ith item is the number of elephant flows the source is sending along the ith path.

As described in Section 2.2, a source end host will selfishly shift elephant flows from overloaded paths to underloaded ones round by round. However, if the end host contributes no flow to the overloaded path, it cannot shift a flow off from it. We call the path not transferring any elephant flow from source to destination the *inactive path*. For example, in Figure 1(b), even though E_{11}'s monitor detects $path_1$ is the most congested path, E_{11} cannot shift traffic off from the path since it does not contribute to the congestion. Algorithm *selfish flow scheduling* illustrates one round of the selfish path switching process.

Line 15 estimates the $BoNF$ of $path_{max\_index}$ if one more flow is scheduled on it. The δ in line 16 is a threshold to prevent unnecessary path switching. Shifting one flow to an underloaded path may improve some flows' throughputs while decrease others'. If we set δ to be 0, line 16 is to make sure a path switching will not decrease the global minimum $BoNF$. If we set δ to be larger than 0, it is a trade off between performance and stability.

3. IMPLEMENTATION

To test DARD's performance in real datecenter networks, we implemented a prototype and deployed it in a fattree topology consisting of 4-port switches on DeterLab [6]. We also implemented a simulator based on ns-2 to evaluate DARD's performance in larger topologies.

3.1 Emulator

We set up a fat-tree topology using 4-port PCs acting as the switches and configure IP addresses according to the hierarchical addressing scheme described in Section 2.3. All switches run OpenFlow. NOX [20] is used as the centralized controller. We use "out-of-band control", in which control and data planes use different networks. Data plane bandwidth is configured as $100Mbps$. We implement a NOX component which configures all switches' flow tables during the initialization. This component allocates the downhill table to flow table 0 and the uphill table to flow table 1 to enforce a higher priority for the downhill table. All entries are set to be permanent. One thing to note is that even though we are using OpenFlow as the underlying infrastructure, we do not rely on the centralized controller. We use NOX only once to initialize the static flow tables.

A daemon program is running on every end host. It has three components. An *Elephant Flow Detector* detects an elephant flow if a TCP connection (determined by source and destination IPs and ports) has lasted for more than 10s. Then the daemon checks if an existing monitor is tracking all the paths connecting the source and destination ToR switches. If not, a new monitor is inserted into the monitor list. A $Monitor$ tracks the $BoNF$ of all the equal-cost paths connecting the source and destination ToR switches as described in Section 2.4. It queries switches for their states using the interfaces of the *aggregate flow statistics* provided by OpenFlow infrastructure. We set the querying interval to be 1 seconds. A *Flow Scheduler* maintains the monitor linked list and shifts elephant flows from overloaded paths to underloaded ones according to the Algorithm *selfish flow scheduling*, where we set the δ to be $10Mbps$. The scheduling interval is 5 seconds plus a random time from $[0s, 5s]$. This random time is to prevent synchronized path switching. We use the Linux IP-in-IP tunneling as the encapsulation/decapsulation module. All the mappings from IDs to underlying IP addresses are maintained in one configuration file, which is kept at every end host.

3.2 Simulator

To evaluate DARD's performance in larger topologies, we build a DARD simulator on ns-2, which captures the system's packet level behavior. The simulator can support $p = 32$ fattree topology and $D_I = D_A = 16$ Clos network [16]. The topology and traffic patterns are passed in as tcl configuration files. A link's bandwidth is $1Gbps$ and its delay is $0.01ms$. The queue size is set to be the delay-bandwidth product. We use source routing to assign a path to a flow. TCP New Reno is used to transfer large files from host to host. A monitor queries related switches for their states every 1 seconds. An end host starts a new round of scheduling every 5 seconds plus a random time from [0s, 5s]. To speed up the simulation, we remove unnecessary packet headers and disable most tracing functions.

4. EVALUATION

This section describes the evaluation of DARD using DeterLab testbed and ns-2 simulation. We focus this evaluation on two metrics, (1) large file transfer time, which measures the efficiency of DARD's flow scheduling algorithm; and (2) the times an elephant flow switches its paths in its life cycle, which shows how stable DARD is. We compare DARD with both random flow level scheduling and centralized scheduling on heterogeneous topologies, including fattree, Clos network and a traditional 8-core-3-tier topology where the oversubscription is larger than 1 [2]. We first introduce the traffic patterns used in our evaluation, and then analyze our evaluation results.

4.1 Traffic Patterns

Since we cannot obtain commercial datacenter traffic traces, we use the three traffic patterns introduced in [8] for both our testbed and simulation evaluations. (1) *Random*, where an end host sends elephant flows to any other end host in the topology with a uniform probability; (2) *Staggered(ToRP,PodP)*, where an end host sends elephant flows to another end host connecting to the same ToR switch with probability $ToRP$, to any other end host in the same pod with probability $PodP$ and to the end hosts in different pods with probability $1 - ToRP - PodP$. In our evaluation we set $ToRP$ to be 0.5 and $PodP$ to be 0.3; (3) *Stride(step)*, where an end host with index x sends elephant flows to the end host with index $((x + step)\ mod\ (number\_of\_hosts)) + 1$. For a specific topology we choose a proper $step$ to make sure the source and destination end hosts are in different pods.

4.2 Testbed Results

We generate the source-destination pairs following the three traffic patterns described in section 4.1. For each source-destination pair, flow inter-arrival times follow exponential distribution. We vary each source-destination pair's expected flow generating rate from 1 to 10 per second. Each elephant flow is a FTP connection transferring a 128MB file. The experiment lasts for 5 minutes. We track the start and end time of every elephant flow and calculate the average file transfer time during the 5 minutes. We also run ECMP, in which the hashing function is defined as the source and destination IP addresses and ports modulo the number of paths, and calculate the *improvement of DARD over ECMP* using formula (1), where avg_T_{ECMP} is the average file transfer time using ECMP, and avg_T_{DARD} is the average file transfer time using DARD.

$$improvement = \frac{avg\_T_{ECMP} - avg\_T_{DARD}}{avg\_T_{ECMP}} \qquad (1)$$

Figure 4 shows the improvement $vs.$ the flow generating rate under different traffic patterns. For the stride traffic pattern, DARD outperforms ECMP within expectation, because in each step, DARD shifts flows from overloaded paths to underloaded ones and increases the minimum flow throughput. We find both random and staggered traffic share an interesting pattern. When the flow generating rate is low, ECMP and DARD have almost the same performance because ECMP's randomness is capable of exploring the limited path diversity since a large portion of elephant flows are within the same pod. As the flow generating rate increases, cross-pod flows congest the switch-to-switch links, in which case DARD reschedules the flows sharing the same bottleneck and improves the average file transfer time. When flow generating rate becomes even higher, the end host-switch links are occupied by flows within the same pod and thus become the bottleneck, in which case DARD helps little.

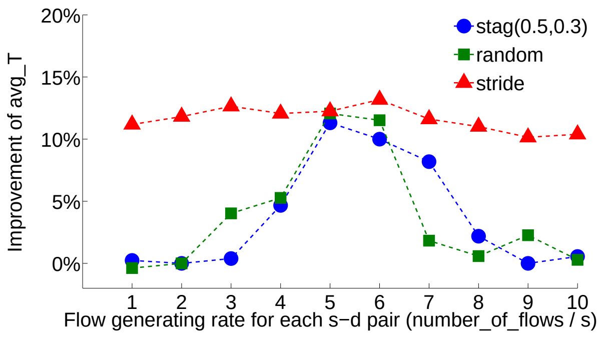


Figure 4: **File transfer improvement. Measured on testbed.**

We also implement a modified version of VLB in the testbed. Since VLB randomly chooses core switches to forward a flow, it may also introduce permanent collisions as ECMP does. As a result, in our VLB implementation, a flow randomly picks a core switch every 10 seconds to prevent the permanent collision. We note it as periodical VLB ($pVLB$).

Figure 5 shows the CDF of file transfer time for the three scheduling approaches under stride traffic pattern. DARD improves the average file transfer time by enhancing the fairness, in which both the fastest and slowest flow perform toward an average case.

Figure 6 shows DARD's CDF of path switch times under different traffic patterns. For the staggered traffic, around 90% of the flows stick to their original path assignment. This indicates when most of the flows are within the same pod or even the same ToR switch, the bottleneck is most likely located at the end host-switch links, in which case DARD helps little. On the other hand, for the stride traffic, where all flows are cross pods, around 40% of the flows do not switch their paths. Another 50% switch their paths for only 1 or 2 times. The largest path switch time is 3, which is smaller than the number of available paths ($\frac{p^2}{4} = 4$) connecting a source-destination pair. The average path switch time is 0.9. This small number indicates that DARD is stable and no flow switches its paths back and forth. We contribute this property to the random time added to the scheduling interval which prevents the synchronized flow scheduling. The path switch time distribution of the random traffic is between that of the staggered and stride traffic, because the quantity of random traffic's intra-pod flows is in the middle of staggered and stride traffic'.

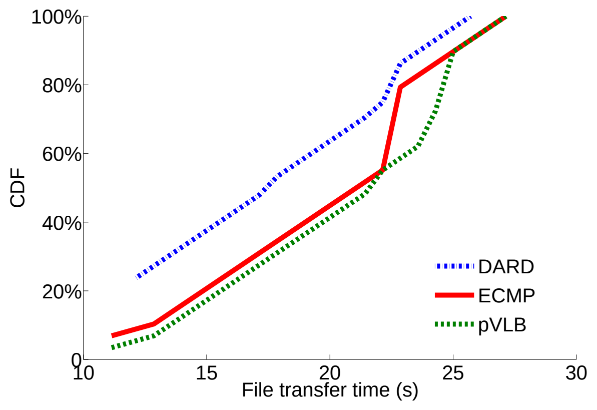


Figure 5: **File transfer time on $p = 4$ fattree topology under stride traffic pattern. Measured on testbed**

4.3 Simulation Results

4.3.1 Results on Fattree Topologies

To compare DARD with centralized scheduling, in our ns-2 simulation we implement both the demand-estimation and simulated annealing algorithm described in Hedera [9], and set the scheduling interval to be 5 seconds. We compare the above four approaches (Simulated annealing, DARD, ECMP and pVLB) on p-pod fattree topologies with $1Gbps$ bandwidth. We use the three traffic patterns described in section 4.1. Flow inter-arrival times follow the exponential distribution with $0.2s$ as the expectation for each source-destination pair. An elephant flow transfers a $128MB$ file. Every experiment lasts for 120s in ns-2.

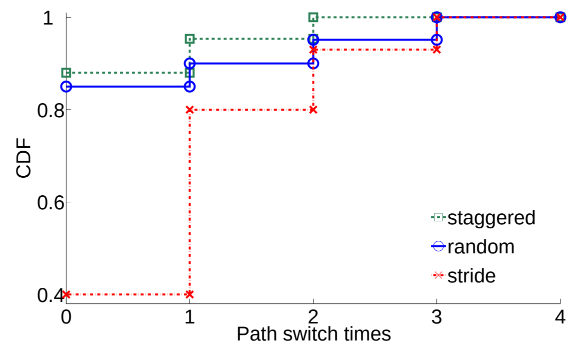


Figure 6: **Path switch times on $p = 4$ fattree topology. Measured on testbed**

	$p = 8$		$p = 16$		$p = 32$	
	90%	max	90%	max	90%	max
random	1	1	2	5	2	8
staggered	1	1	1	4	1	11
stride	1	2	2	3	3	9

Table 5: **DARD's 90-percentile and maximum path switch times on fattree topologies. Measured on ns-2.**

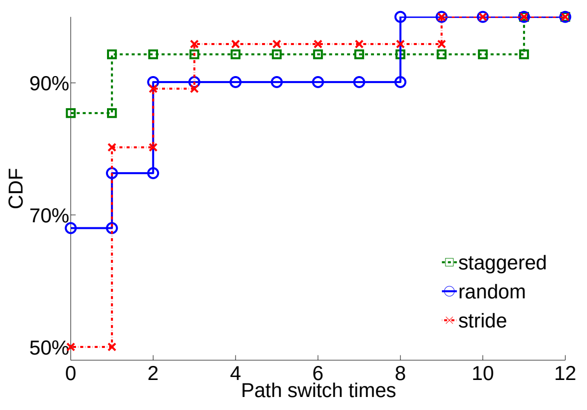


Figure 8: **Path switch times on $p = 32$ fattree topology under different traffic patterns. Measured on ns-2.**

Figure 7 shows the CDF of file transfer time on $p = 32$ fattree topology under different traffic patterns. We have three main observations. (1) Under stride traffic pattern, both simulated annealing and DARD outperform ECMP and pVLB. The simulated annealing outperforms DARD by less than 10%. (2) For the staggered traffic pattern, the simulated annealing does not improve the file transfer time much, because it does not schedule the traffic in granularity of a single flow, but assigns core switches to destination hosts to limit the searching space. This optimization fails to approach the optimal solution when intra-pod traffic are dominant. On the other hand, when intra-pod traffic are dominant, even though

	$p = 8$			$p = 16$			$p = 32$		
	random	staggered	stride	random	staggered	stride	random	staggered	stride
Simulated Annealing	26.5	34.2	18.7	30.4	34.6	21.8	21.0	23.1	18.5
DARD	26.1	31.3	20.6	31.2	32.5	21.8	21.7	21.1	20.3
ECMP	29.9	35.0	24.8	34.2	34.9	25.4	23.8	24.5	22.3
pVLB	30.2	34.8	25.1	33.9	35.1	25.3	24.4	23.7	22.8

Table 4: Average file transfer time (s) on fattree topologies. Measured on ns-2.

	$D_I = D_A = 4$			$D_I = D_A = 8$			$D_I = D_A = 16$		
	random	staggered	stride	random	staggered	stride	random	staggered	stride
Simulated Annealing	27.8	30.0	31.4	30.9	21.0	25.4	18.2	19.7	21.7
DARD	27.4	29.2	32.6	30.5	19.5	26.8	18.8	18.3	22.9
ECMP	32.4	30.7	35.8	32.6	20.8	29.5	20.5	19.7	25.3
pVLB	32.4	30.8	35.6	32.5	21.4	30.0	20.5	19.4	26.0

Table 6: Average file transfer time (s) on Clos topologies. Measured on ns-2.

	$D_I = D_A = 4$		$D_I = D_A = 8$		$D_I = D_A = 16$	
	90%	max	90%	max	90%	max
random	1	2	1	3	2	6
staggered	0	0	1	2	1	7
stride	1	2	2	4	3	8

Table 7: DARD's 90-percentile and maximum path switch times on Clos topologies. Measured on ns-2.

path diversity is limited and the bottlenecks are very likely on the end host-switch links, DARD is still capable of shifting flows to underloaded paths and improving performance. (3) The random traffic pattern lies somewhere in the middle. Both simulated annealing and DARD improve the file transfer time. DARD and the simulated annealing's curves are close to each other. We find the same pattern in smaller fattree topologies. Table 4 shows the comparison of average file transfer time under different traffic patterns. On the 8-pod fattree topology, DARD even outperforms the simulated annealing.

Figure 6 shows the CDF of DARD's path switch times on $p = 32$ fattree topology under different traffic patterns. It shares the same characteristic as Figure 6. DARD adjusts paths most frequently under the stride traffic pattern. But for staggered traffic, only a small percentage of flows switch their paths. The number of paths for a cross-pod flow is $\frac{p^2}{4} = 256$. We find that even the maximum path switch times are much smaller than this number, which means a flow finishes transferring before it exploring all the available paths. This further indicates DARD is stable and introduces little path oscillation. We find the same pattern of path switch times on smaller fattree topologies. Table 5 summarizes the 90-percentile and the maximum path switch times on different fattree topologies and under different traffic patterns.

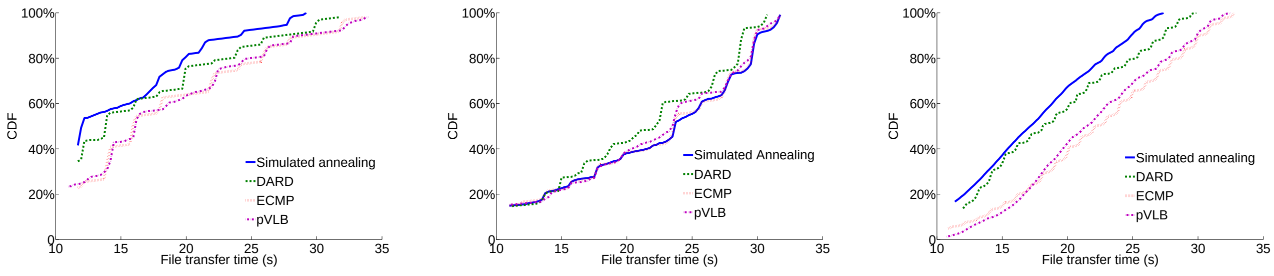
4.3.2 Results on Other Topologies

DARD and the other flow scheduling mechanisms are designed for "multi-rooted trees" [9] in datacenter networks. Besides fattree topology, Clos network is another horizontal expandable topology whose oversubscription is 1 [16]. We construct Clos topologies of different sizes in the ns-2 simulator. The link bandwidth is $1Gbps$. An elephant flow transfers a $128MB$ file. We use the three traffic patterns described in Section 4.1. Flow inter-arrival times follow the exponential distribution with 0.2s as the expectation for each source-destination pair. The simulation lasts for $120s$ in ns-2. The key difference between fattree and Clos network is that in fattree topology we can uniquely identify a path for a source-destination pair by assigning a core switch, but in Clos network, we need both uphill and downhill aggregation switches to uniquely identify a path connecting a source-destination pair. As a result, in the simulated annealing implementation, we assign uphill and downhill aggregation switch pairs, instead of core switches, to destination hosts. In the VLB implementation, a flow randomly chooses a uphill and downhill aggregation switch pair instead of a core switch.

Figure 9 shows the CDF of file transfer time under above settings. When the network is dominated by inter-pod traffic (i.e., the stride traffic), DARD can improve the file transfer time considerably. The improvement of simulated annealing over DARD is less than 10%. When the network is dominated by intra-pod traffic (i.e., the staggered traffic), DARD can still explore the path diversities and improve the file transfer time. Table 6 summarizes DARD's average file transfer time on different Clos networks. It flows the same pattern as we discovered in Table 4. Figure 10 shows that even the flow's maximum path switch times is much smaller than the available paths ($2D_A$) connecting a source-destination pair. This indicates that DARD introduces little path oscillation in Clos networks.

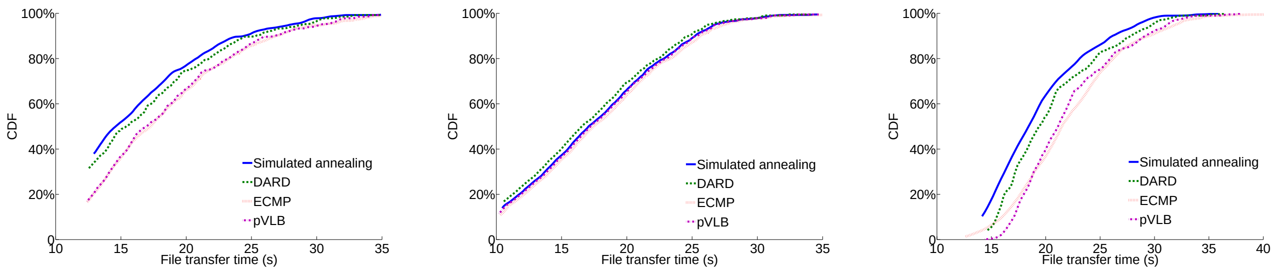
Both Table 4 and 6 show that pVLB introduces some randomness to the performance. Even though pVLB can prevent permanent elephant flow collisions, a TCP connection may suffer from packet loss or reordering because of frequent path switching. As a result, in most cases, pVLB performs similarly to ECMP.

Both fattree topology and Clos network have the 1 oversubscription, which is not always true in existing datacenter networks. As a result, we compare the four scheduling approaches on a 8-core-3-tier topology [2], whose access layer's oversubscription is $2.5 : 1$ and aggregation layer's oversubscription is $1.5 : 1$. We use the same traffic generating mechanisms, the same bandwidth setting and the same



(a) random (b) staggered (c) stride

Figure 7: **CDF of file transfer time on $p = 32$ fattree topology under different traffic patterns. Measured on ns-2.**



(a) random (b) staggered (c) stride

Figure 9: **CDF of file transfer time on $D_I = D_A = 16$ Clos topology under different traffic patterns. Measured on ns-2.**

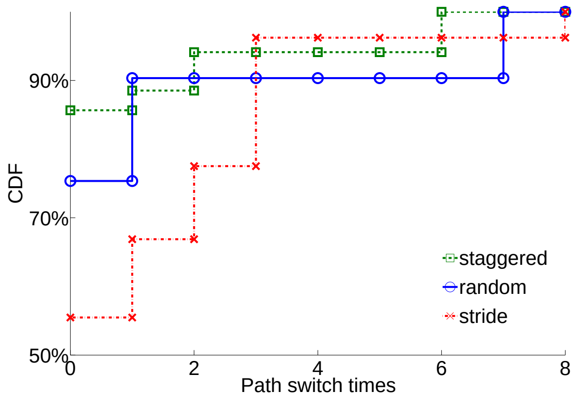


Figure 10: **Path switch times on $D_I = D_A = 16$ Clos topology under different traffic patterns. Measured on ns-2.**

file size as previous simulations.

Figure 11 shows the CDF of file transfer time on this 8-core-3-tier topology. It shares the same characteristic as the results on fattree and Clos topologies. Under staggered traffic pattern, where intra-pod traffic is dominant, DARD outperforms both the centralized scheduling and random flow level scheduling. Under the stride traffic pattern, where the inter-pod traffic is dominant, DARD outperforms random flow level scheduling and the gap to the centralized scheduling's performance is small. Figure 12 shows the CDF of

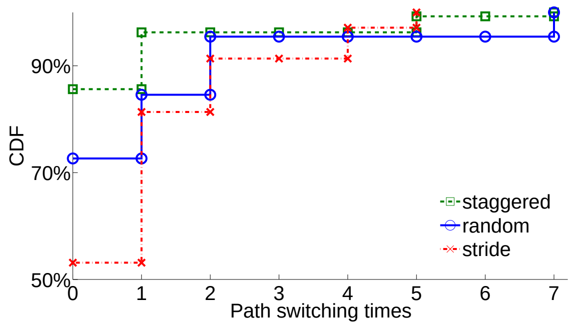
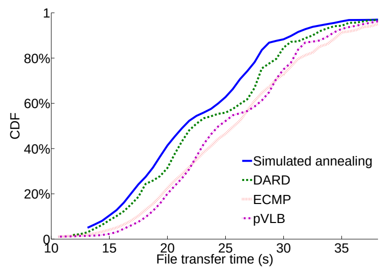


Figure 12: **Path switch times on 8-core-3-tier topology under different traffic patterns.**

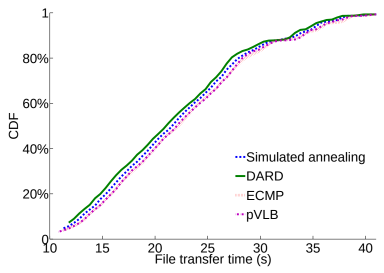
DARD's path switch times on the 8-core-3-tier topology under different traffic patterns. We find that 90% of the flows shift their paths no more than twice. This indicates DARD introduces little path oscillation and is stable on topologies whose oversubscription is larger than 1.

4.3.3 Comparison with Load-sensitive Scheduling

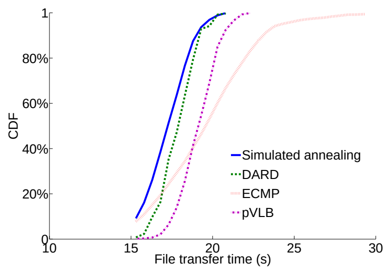
In this section, we compare DARD with TeXCP [22], a distributed online intra-domain traffic engineering approach. TeXCP adaptively moves traffic from over-utilized to under-utilized paths without oscillation. We implement TeXCP in ns-2 according to [22], in which each ToR switch pair main-



(a) random



(b) staggered



(c) stride

Figure 11: **CDF of file transfer time on 8-core-3-tier topology under different traffic patterns. Measured on ns-2.**

tains states for all the available paths connecting the two of them and probes the network states every $10ms$ (The default probe interval is $200ms$. However since the RTT in datacenter is in granularity of $1ms$ or even smaller, we decrease this probe interval). We do not implement the flowlet [29] mechanisms in the simulator, thus each ToR switch schedules at the packet level. As required in [22], we set the control interval to be five times of the probe interval. We use the same traffic patterns and topology configurations as previous experiments.

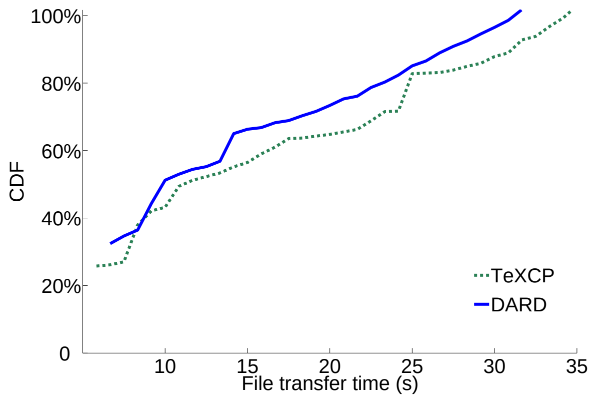


Figure 13: **File transfer time on $p = 4$ fattree topology under stride traffic pattern. Measured on ns-2.**

Figure 13 shows the CDF of DARD and TeXCP's file transfer time on a $p = 4$ fattree topology under stride traffic pattern. We find DARD outperforms TeXCP slightly. By tracking the link parameters, we find the bisection bandwidth for both DARD and TeXCP are close to each other. We further measure every elephant flow's retransmission rate, which is defined as the number of retransmitted packets over the number of unique packets. Figure 14 shows TeXCP has a higher retransmission rate than DARD. In other words, even though TeXCP can fully utilize the bisection bandwidth, some of the packets are retransmitted and thus its goodput is not as high as DARD. Packet level scheduling assigns packets belonging to the same flow to different paths with different RTTs, and causes packets reordering. This reordering problem eventually triggers retransmission.

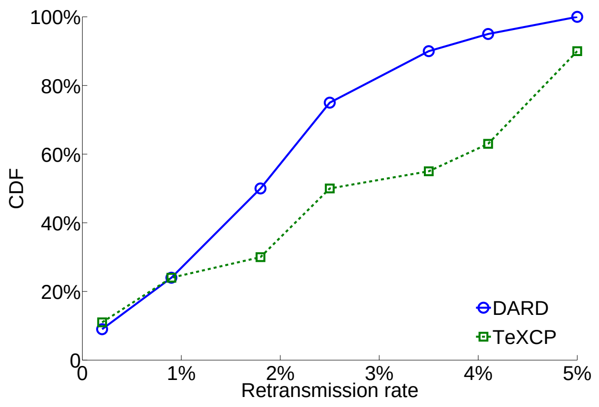


Figure 14: **DARD and TeXCP's TCP retransmission rate on $p = 4$ fattree topology. Measured on ns-2.**

It is possible that scheduling traffic in granularity of a flowlet (TCP packet burst) would reduce TeXCP's retransmission rate. However, the small RTT in datacenter networks may require a more accurate timing mechanisms for flowlet scheduling. We leave this problem as our future work.

4.3.4 Communication Overhead

To evaluate DARD's communication overhead, we trace the control messages for both DARD and centralized scheduling on a $p = 8$ fattree topology in ns-2. DARD's communication overhead is mainly introduced by the periodical probes, which include both queries from hosts and replies from switches. This communication overhead is bounded by the size of the topology, because in the worst case, the system only needs to handle all pair probes. However, for centralized scheduling, ToR switches need to report elephant flows to the centralized controller and the controller will further update some switches' flow tables. As a result, the communication overhead is bounded by the number of flows, which could be potentially very large.

Figure 15 shows how much of the bandwidth is taken by control messages given different workload. With the increase of the workload, there are three stages. The first stage is where elephant flow numbers is less than $1.5k$, in which case DARD's control messages take less bandwidth than centralized scheduling's. The reason is mainly because centralized scheduling's control message size is larger than DARD's (a message from a ToR switch to the controller takes 80 bytes and a message from the controller to a switch takes 72 Bytes. On the other hand, a message from a host to a switch takes 48 bytes and a message from a switch to a host takes 32 bytes). The second stage is where the elephant flow number is between $1.5K$ and $3K$, in which case DARD's control messages take more bandwidth. That is because more source-destination pairs in DARD need to probe all paths states. The third stage is where more than $3K$ flows compete for the limited network resources, in which case DARD's probe traffic is bounded by the topology size. One thing to note is that the centralized scheduling's communication overhead does not increase proportionally to the workload. That is mainly because when the traffic pattern is dense enough, even the centralized scheduling cannot easily find out a better flow allocation and thus few messages are sent from the controller to switches.

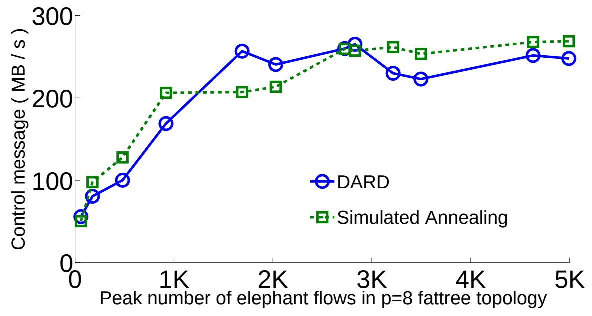


Figure 15: DARD and Centralized Scheduling's communication overhead on $p = 8$ fattree topology. Measured on ns-2.

In sum, both testbed and simulation evaluations show that for a wide range of datacenter topologies, DARD can considerably improve the elephant flow transfer time compared with random flow-level scheduling. When inter-pod traffic are dominant, the improvement of centralized scheduling over DARD is less than 10%. When the intra-pod traffic becomes dominant, DARD can even outperform centralized scheduling in some cases. DARD's randomness in control interval calculation prevents flows from synchronized shifting and introduces little path oscillation.

5. RELATED WORK

Commodity datacenter network design. Traditional datacenter network typically consists of trees of switches and routers with more expensive equipments moving up the topology hierarchy [2]. However, non-uniform bandwidth among datacenter components limits the utilization of the aggregated bandwidth at the edge of the network. On the other hand, commodity switches are becoming available with high port speed and low price. OpenFlow [25] further provides switches with a programmable infrastructure. As a result, people start to redesign the datacenter networks using commodity switches as the building blocks. Fat-tree [8] and PortLand [28] propose to build up datacenter networks using fattree topology. VL2 [16] and Monsoon [18] construct commodity datacenter networks with the Clos topology. All these systems provide multiple equal-cost paths connecting any pair of hosts. DARD provides a generic flow scheduling mechanism for all the above datacenter networks.

Centralized flow scheduling. In addition to Hedera [9], Ethane [13] and 4D [17] also schedule flows in a centralized manner. Being unscalable is the major concern of these centralized scheduling approaches. DARD, on the other hand, adopts distributed scheduling approach and thus is scalable. Based on our evaluation, The performance gap between DARD and the centralized scheduling is small. When intra-pod traffic is dominant, DARD even outperforms the centralized scheduling.

Distributed flow scheduling. ECMP [21] and flow level VLB [24] are two dominant distributed flow scheduling approaches in datacenter networks. We observed that these distributed scheduling algorithms introduce some randomness to the performance. That is mainly because both of them do not rely on traffic patterns to make scheduling decisions. DARD, on the other hand, schedules flows according to the traffic patterns and makes improvement in every step.

There are extensive literature on distributed multipath flow scheduling for general topologies. MATE [14] and TeXCP [22] perform distributed traffic engineering within a network through explicit congestion feedback, which is not supported by current commodity switches. Our evaluation shows that TeXCP's packet level scheduling causes reordering and packet retransmission. It is still unclear whether scheduling traffic in granularity of flowlet will improve TeXCP's performance in datacenter. Geoffray *et al.* proposed to conduct packet level scheduling in high performance networks, which may bring significant reordering problem to TCP flows [15]. VLANNet [26] schedules flows along multiple paths by dynamically modifying routing tables, which may cause temporary loops in routes.

6. CONCLUSION

This paper proposes DARD, a distributed adaptive routing system for datacenter networks. DARD allows each end-host to selfishly shift elephant flows from overloaded paths to underloaded ones. Our analysis shows that this algorithm converges to a Nash equilibrium in finite steps. Testbed emulation and ns-2 simulation show that DARD outperforms random flow-level scheduling when the bottlenecks are not at the edge, and outperforms centralized scheduling when intra-pod traffic is dominant. Evaluation results also show that DARD introduces little path oscillation. 90% of the flows shift less than 3 times in their life cycles.

7. REFERENCES

[1] Amazon elastic compute cloud. http://aws.amazon.com/ec2.

[2] Cisco Data Center Infrastructure 2.5 Design Guide. http://www.cisco.com/en/US/docs/solutions/Enterprise/Data_Center/DC_Infra2_5/DCI_SRND_2_5_book.html.

[3] IP alias limitation in linux kernel 2.6. http://lxr.free-electrons.com/source/net/core/dev.c#L935.

[4] IP alias limitation in windows nt 4.0. http://support.microsoft.com/kb/149426.

[5] Microsoft Windows Azure. http://www.microsoft.com/windowsazure.

[6] Microsoft Windows Azure. http://www.isi.deterlab.net/.

[7] Nira: a new internet routing architecture. In *FDNA '03: Proceedings of the ACM SIGCOMM workshop on Future directions in network architecture*, pages 301–312, New York, NY, USA, 2003. ACM.

[8] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. *SIGCOMM Comput. Commun. Rev.*, 38(4):63–74, 2008.

[9] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proceedings of the 7th ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, Apr. 2010.

[10] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th annual conference on Internet measurement*, IMC '10, pages 267–280, New York, NY, USA, 2010. ACM.

[11] J.-Y. Boudec. Rate adaptation, congestion control and fairness: A tutorial, 2000.

[12] C. Busch and M. Magdon-Ismail. Atomic routing games on maximum congestion. *Theor. Comput. Sci.*, 410(36):3337–3347, 2009.

[13] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: taking control of the enterprise. *SIGCOMM Comput. Commun. Rev.*, 37(4):1–12, 2007.

[14] A. Elwalid, C. Jin, S. Low, and I. Widjaja. Mate: Mpls adaptive traffic engineering, April 2001.

[15] P. Geoffray and T. Hoefler. Adaptive routing strategies for modern high performance networks. In *Proceedings of the 2008 16th IEEE Symposium on High Performance Interconnects*, pages 165–172, Washington, DC, USA, 2008. IEEE Computer Society.

[16] 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. *SIGCOMM Comput. Commun. Rev.*, 39(4):51–62, 2009.

[17] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A clean slate 4d approach to network control and management. *SIGCOMM Comput. Commun. Rev.*, 35(5):41–54, 2005.

[18] A. Greenberg, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Towards a next generation data center architecture: scalability and commoditization. In *PRESTO '08: Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, pages 57–62, New York, NY, USA, 2008. ACM.

[19] T. Greene. Researchers show off advanced network control technology. http://www.networkworld.com/news/2008/102908-openflow.html.

[20] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. Nox: towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, 2008.

[21] C. Hopps. Analysis of an Equal-Cost Multi-Path Algorithm. RFC 2992, 2000.

[22] S. Kandula, D. Katabi, B. Davie, and A. Charny. Walking the tightrope: Responsive yet stable traffic engineering. In *In Proc. ACM SIGCOMM*, 2005.

[23] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of data center traffic: measurements & analysis. In *IMC '09: Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, pages 202–208, New York, NY, USA, 2009. ACM.

[24] M. Kodialam, T. V. Lakshman, and S. Sengupta. Efficient and robust routing of highly variable traffic. In *In Proceedings of Third Workshop on Hot Topics in Networks (HotNets-III)*, 2004.

[25] 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.*, 38(2):69–74, 2008.

[26] S. Miura, T. Boku, T. Okamoto, and T. Hanawa. A dynamic routing control system for high-performance pc cluster with multi-path ethernet connection. In *IPDPS*, pages 1–8, 2008.

[27] T. Mori, R. Kawahara, S. Naito, and S. Goto. On the characteristics of internet traffic variability: Spikes and elephants. *Applications and the Internet, IEEE/IPSJ International Symposium on*, 0:99, 2004.

[28] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. Portland: a scalable fault-tolerant layer 2 data center network fabric. *SIGCOMM Comput. Commun. Rev.*, 39(4):39–50, 2009.

[29] S. Sinha, S. Kandula, and D. Katabi. Harnessing TCPs Burstiness using Flowlet Switching. In *3rd ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)*, San Diego, CA, November 2004.

Appendix

A. EXPLANATION OF THE OBJECTIVE

We assume TCP is the dominant transport protocol in datacenter, which tries to achieve max-min fairness when combined with fair queuing. Each endhost shifts flows from overloaded paths to underloaded ones to increase its observed minimum $\frac{bandwidth}{flow\_numbers}$, *i.e.*, the $BoNF$. This section explains given max-min fair bandwidth allocation, the global minimum $BoNF$ is the lower bound of the global minimum flow rate, thus increasing the minimum $BoNF$ actually increases the global minimum flow rate.

Theorem 1. Given max-min fair bandwidth allocation for any network topology and any traffic pattern, the global minimum $BoNF$ is the lower bound of global minimum flow rate.

First we define a bottleneck link according to [11]. A link l is a bottleneck for a flow f if and only if (a) link l is fully utilized, and (b) flow f has the maximum rate among all the flows using link l. This definition is on the assumption that max-min fair bandwidth allocation is achieved.

Given max-min fair bandwidth allocation, link l_i has the fair share rate $BoNF_i = \frac{bandwidth_i}{flow\_numbers_i}$. Suppose link l_0 has the minimum fair share rate $BoNF_0$. Flow f has the global minimum flow rate, min_rate. Link l_f is flow f's bottleneck. Theorem 1 claims $min\_rate \geq BoNF_0$. We prove this theorem using contradiction.

According to the bottleneck definition, min_rate is the maximum flow rate on link l_f, and thus $\frac{bandwidth_{l_f}}{flow\_numbers_{l_f}} \leq min\_rate$. Suppose $min\_rate < BoNF_0$, we get

$$\frac{bandwidth_{l_f}}{flow\_numbers_{l_f}} < BoNF_0 \qquad \text{(A1)}$$

(A1) is conflict with $BoNF_0$ being the minimum fair share rate. As a result, the minimum $BoNF$ is the lower bound of the global minimum flow rate.

In DARD, every endhost tries to increase its observed minimum $BoNF$ in each round, thus the global minimum $BoNF$ increases, so does the global minimum flow rate.

B. CONVERGENCE PROOF

We now formalize DARD's flow scheduling algorithm and prove that this algorithm can converge to a Nash equilibrium in finite steps.

The proof is a special case of a *congestion game* [12], which is defined as $(F, G, \{r^f\}_{f \in F})$. F is the set of all the flows. $G = (V, E)$ is an directed graph. r^f is a set of routes that can be used by flow f. A *strategy* $s = [r_{i_1}^{f_1}, r_{i_2}^{f_2}, \ldots, r_{i_{|F|}}^{f_{|F|}}]$ is a collection of routes, in which the i_kth route in r^{f_k}, $r_{i_k}^{f_k}$, is used by flow f_k to deliver the traffic.

For a strategy s and a link j, the *link state* $S_j(s)$ is a triple ($bandwidth_j$, $flow\_numbers_j$, $BoNF_j$), where the $BoNF_j$ is defined as $\frac{bandwidth_j}{flow\_numbers_j}$. For a route r, the *route state* $S_r(s)$ is the link state with the smallest $BoNF$ over all links in r. The *system state* $S(s)$ is the link state with the smallest $BoNF$ over all links in E. A *flow state* $S_f(s)$ is the corresponding route state, *i.e.*, $S_f(s) = S_r(s)$, flow f is using route r.

Notation s_{-k} refers to the strategy s without $r_{i_k}^{f_k}$, *i.e.* $[r_{i_1}^{f_1}, \ldots, r_{i_{k-1}}^{f_{k-1}}, r_{i_{k+1}}^{f_{k+1}}, \ldots, r_{i_{|F|}}^{f_{|F|}}]$. $(s_{-k}, r_{i_{k'}}^{f_k})$ refers to the strategy $[r_{i_1}^{f_1}, \ldots, r_{i_{k-1}}^{f_{k-1}}, r_{i_{k'}}^{f_k}, r_{i_{k+1}}^{f_{k+1}}, \ldots, r_{i_{|F|}}^{f_{|F|}}]$. Flow f_k is *locally optimal* in strategy s if

$$S_{f_k}(s).BoNF \geq S_{f_k}(s_{-k}, r_{i_{k'}}^{f_k}).BoNF \qquad \text{(B1)}$$

for all $r_{i_{k'}}^{f_k} \in r^{f_k}$. *Nash equilibrium* is a state where all flows are locally optimal. A strategy s^* is *global optimal* if for any strategy s, $S(s^*).BoNF \geq S(s).BoNF$.

Theorem 2. If there is no synchronized flow scheduling, Algorithm *selfish flow scheduling* will increase the minimum $BoNF$ round by round and converge to a Nash equilibrium in finite steps. The global optimal strategy is also a Nash equilibrium strategy.

For a strategy s, the *state vector* $SV(s) = [v_0(s), v_1(s), v_2(s), \ldots]$, where $v_k(s)$ stands for the number of links whose $BoNF$ is located at $[k\delta, (k+1)\delta)$, where δ is a positive parameter, *e.g.*, $10Mbps$, to cluster links into groups. As a result $\sum_k v_k(s) = |E|$. A small δ will group the links in a fine granularity and increase the minimum $BoNF$. A large δ will improve the convergence speed. Suppose s and s' are two strategies, $SV(s) = [v_0(s), v_1(s), v_2(s), \ldots]$ and $SV(s') = [v_0(s'), v_1(s'), v_2(s'), \ldots]$. We define $s = s'$ when $v_k(s) = v_k(s')$ for all $k \geq 0$. $s < s'$ when there exists some K such that $v_K(s) < v_K(s')$ and $\forall k < K, v_k(s) \leq v_k(s')$. It is easy to show that given three strategies s, s' and s'', if $s \leq s'$ and $s' \leq s''$, then $s \leq s''$.

Given a congestion game $(F, G, \{r^f\}_{f \in F})$ and δ, there are only finite number of state vectors. According to the definition of "$=$" and "$<$", we can find out at least one strategy $\widetilde{s}$ that is the *smallest*, *i.e.*, for any strategy s, $\widetilde{s} \leq s$. It is easy to see that this $\widetilde{s}$ has the largest minimum $BoNF$ or has the least number of links that have the minimum $BoNF$ and thus is the global optimal.

If only one flow f selfishly changes its route to improve its $BoNF$, making the strategy change from s to s', this action decreases the number of links with small $BoNF$s and increases the number of links with larger $BoNF$s. In other words, $s' < s$. This indicates that asynchronous and selfish flow movements actually increase global minimum $BoNF$ round by round until all flows reach their locally optimal state. Since the number of state vectors is limited, the steps converge to a Nash equilibrium is finite. What is more, because $\widetilde{s}$ is the smallest strategy, no flow can have a further movement to decrease $\widetilde{s}$, *i.e* every flow is in its locally optimal state. Hence this global optimal strategy $\widetilde{s}$ is also a Nash equilibrium strategy.