Computer Networks 120 (2017) 186–197

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

A stable matching based elephant flow scheduling algorithm in data center networks

Yuxiang Zhang[a,b], Lin Cui[a,b,*], Yuan Zhang[a,b]

[a] Department of Computer Science, Jinan University, Guangzhou, P.R. China and Guangdong Key Laboratory of Big Data Analysis and Processing, Guangzhou, P.R. China

[b] Guangdong Key Laboratory of Big Data Analysis and Processing, Guangzhou, P.R. China

ARTICLE INFO

Article history:
Received 27 July 2016
Revised 29 December 2016
Accepted 7 April 2017
Available online 8 April 2017

Keywords:
Data center networking
Stable matching
Elephant flow

ABSTRACT

With the development of cloud computing in recent years, data center networks have become a hot topic in both industrial and academic communities. Previous studies have shown that elephant flows, which usually carry large amount of data, are critical to the efficiency of data centers. How to schedule elephant flows efficiently becomes an important issue for maintaining high performance and avoiding network congestion. In this paper, we study the efficient flow scheduling problem in data centers with a focus on elephant flows. By applying stable matching theory, the scheduling problem is modeled and proven to be NP-Hard. Then, we propose *Fincher*, an efficient scheme leveraging Software-Defined Networking (SDN) to reduce latency and avoid congestions in data centers. We have implemented *Fincher* with POX controller and Mininet. Extensive evaluation results demonstrate that *Fincher* can improve bisection bandwidth by 30% and reduce flow completion time by 28% on average compared to ECMP and Hedera.

1. Introduction

Driven by modern Internet applications and cloud computing technologies, data centers are being built dramatically around the world. To obtain high bandwidth and achieve fault tolerance, Data Center Networks (DCNs) are often designed with multiple paths between any two servers [1–3]. Multiple paths not only are useful in dealing with potential failures, but also can be exploited to enhance network performance. Specially, when traffic load of the network is getting larger, multiple paths can be used to reduce latency and improve throughput efficiently. However, scheduling on multiple paths is much more complex and challenging.

Grouping packets into flows provides a better view of data transmissions in data centers, and helps to model traffic scheduling problem on a more general aspect. Moreover, flow-based scheduling solves the reordering problem introduced by multi-path routing because data packets in the same flow are transferred sequentially [1,4]. Generally, flows in data centers can be categorized into two types: *elephant flow*, which carries lot of data or lives a long period; and *mice flow*, which is numerous but takes little resource [5]. Because elephant flow would take advantage of lots of network resources, elephant flows scheduling is a challenging problem in data centers which may cause congestion. Many existing research works and solutions have been proposed to schedule elephant flows [2,6,7].

Among all solutions, Equal Cost Multi-Path routing (ECMP) is the state-of-the-art for multi-path routing and load-balancing in data centers [5,8]. In ECMP, a switch locally decides the next hop from multiple equal cost paths by calculating a hash value, typically from the source and destination IP addresses, transport port numbers and protocol (IP five-tuple). ECMP is easy to implement and does not require per-flow statistics at switches. However, ECMP suffers from two major drawbacks when used in data center networks [9,10]. First, ECMP does not differentiate between mice flows and elephant flows. Thus, mice flows, which usually are latency-sensitive flows, often find themselves queued behind elephants in the egress ports, suffering from long queueing delays and poor flow completion times (FCT) [10,11]. Second, ECMP cannot effectively utilize available bandwidth due to hash collision [12].

In addition to ECMP, many other solutions are proposed, e.g., Hedera [4], Mahout [13], TinyFlow [7]. Those solutions can work effectively and improve performance of elephant flows scheduling when they are compared to ECMP. However, some of those solutions may cause unbalanced traffic or congestion, e.g., Hedera and Mahout, which would be limited as network utilization increased. And others may lead to packets reordering (which induce more overhead to fix it), e.g., TinyFlow and VL2. What's more, these schemes neglect statistics of switches memory to get a better scheduling [14,15].

* Corresponding author.
E-mail addresses: samuelzyx0924@gmail.com (Y. Zhang), tcuilin@jnu.edu.cn (L. Cui), michellinyuan@gmail.com (Y. Zhang).

http://dx.doi.org/10.1016/j.comnet.2017.04.018
1389-1286/

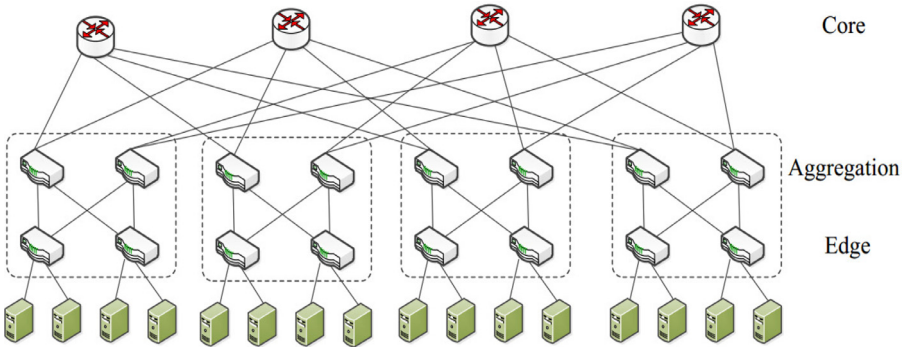


Fig. 1. 4-pod fat-tree topology.

Considering limitations of these solutions, in this paper, we try to find out a scheduling scheme that lead to both congestion avoidance and network performance optimization. To achieve these objectives, we use a strong tool – the *stable matching* theory to model the scheduling problem for elephant flows in data centers and solve this problem. The *stability* characteristic of elephant flows scheduling is carefully defined. And then we present *Fincher*, a novel stable matching based elephant flow scheduling algorithm, which can provide high utilization and low latency in data centers. Also, *Fincher* is an effective scheduling mechanism to avoid congestion because of the stability of the matching between flows and switches. *Fincher* leverages the global statistics of data centers to generate the preference lists of flows and switches and uses these preference lists to get a matching between flows and switches. By obtaining an optimal stable matching between flows and switches, *Fincher* can match all flows to their appropriate paths and achieve optimal performance of the network. The *stability* of output matching enables *Fincher* to avoid congestion which are mostly caused by elephant flows. Extensive evaluations results through Mininet show that *Fincher*'s performance is better than ECMP and Hedera in both bisection bandwidth and flow completion time (FCT). To the best of our knowledge, this is the first work using stable matching theory to perform elephant flow schedule in data centers.

The remainder of this paper is organized as follows. Section 2 describes the model of elephant flow scheduling problem in data centers. An efficient elephant flow scheduling scheme, *Fincher*, is proposed in Section 3. *Fincher*'s implementation details are elaborated in Section 4. Section 5 evaluates the performance of the proposed scheme. Related works and their differences compared to *Fincher* are given in Section 6. Finally, Section 7 concludes the paper.

2. System models

2.1. System overview

We consider a multi-tier data center network which is typically structured under a multi-root tree topology such as fat-tree network [1,3]. Fig. 1 shows an example of a 4-pod fat-tree. Notations that would be used in the following of this paper are summarized in Table 1.

Under a k-pod fat-tree topology, each switch has k ports. For inter-pod flows, there are $(k/2)^2$ equal shortest paths between any given pair of hosts in the network [1]. Each of these paths corresponds to a unique core switch. Thus, when a new inter-pod flow arrives, it needs to be assigned to an appropriate core switch, which determines a specific shortest path for the flow. Similar operation can be performed for intra-pod elephant flows, i.e., assigning a unique aggregation switch to represent the path for an intra-pod flow. Therefore, in the following of this paper, for ease of description, we use aggregation or core switches to represent the path assigned to inter-pod or intra-pod flows respectively.

Table 1
Denotations used in model and algorithm.

Denotation	Description
F	The set of all flows
f_i	The ith flow in F, $i = 1, 2, \ldots, \lvert F\rvert$
r_i	Data rate of flow f_i
S	The set of all switches
s_j	The jth switch in $S (i = 1, 2, \ldots, \lvert S\rvert)$
c_j	Free memory space of switch s_j
M	Matching between switches and flows
$P(x)$	Preference list of x

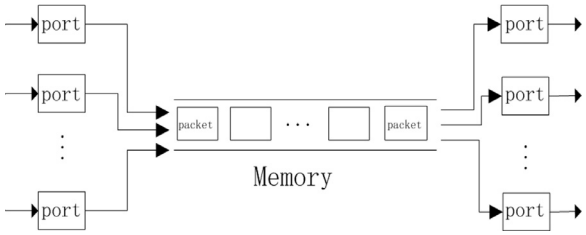


Fig. 2. Illustration of shared memory switch.

As mentioned above, there are mainly two objects, *switches and flows*, involved when performing flow scheduling in data center networks. Our goal is to find an appropriate matching between them:

1) *Switches*: a set of switches which can receive packets, store packets in memory and forward packets to other switches (or servers). Like most commodity switches, switches in data center usually are shared memory switches that aim to exploit statistical multiplexing gain through the use of logically common packet buffers available to all switch ports. [11] Unless otherwise stated, we mainly consider shared memory switches in the rest of the paper for ease of explanation. Moreover, the solution proposed in this paper can be easily extended to multi-queue switches, e.g., CIOQ (Combined Input/Output Queue) switch [10,15,16]. Fig. 2 shows an example of shared memory switch.

2) *Flows*: a set of flows which transfer data between two servers in data center. Flows would demand for and occupy memory resources of switches on the path.

The set of switches is denoted by $S = \{s_1, s_2, \ldots\}$, where $|S|$ is the total number of switches. For a switch s_j, its capacity, which is denoted by c_i, is defined by the amount of free space for incoming flows' packets. Specifically, within a time unit, a switch's capacity can be defined as the number of packets it can accept into its memory.

The set of flows is defined as $F = \{f_1, f_2, \ldots\}$, where $|F|$ is the total number of flows. Flows may have their own characteristics, e.g. data rate, living time, data size. In this paper, we focus on

flows' data rate and denote r_i to be data rate of flow f_i. Here, r_i can be the average pairwise traffic rate over a certain temporal interval, which can be set suitably long to match the dynamism of the environment while not responding to instantaneous traffic bursts. This is reasonable that many existing DC measurement studies suggest that DC traffic exhibits fixed-set hotspots that change slowly over time [17,18].

A switch can accept multiple flows, while each flow can be only forwarded to one critical switch, e.g., aggregation switch or core switch. To avoid congestion, switches' memory can not be overflowed. Thus, all switches must follow the following constraint: within a time unit, a switch s_j can accept multiple flows as long as the total flow data rate does not exceed its capacity, i.e., $c_j \geq r_1 + r_2 + \ldots + r_k$, where $f_1, f_2, \ldots f_k$ are all flows accepted by the s_j and $r_1, r_2, \ldots r_k$ are their data rates.

2.2. FSSM problem

According to the description above, there is a matching problem between flows and switches, and the resource requirement of flows and limited capacity of switches make the problem much more complex. In this section, we will apply the stable matching theory [19] for modeling of elephant flow scheduling problem in data center networks and prove this scheduling problem is NP-hard.

According to the stable matching theory, we build preference lists for both flows and switches as follows:

1) Each flow has a preference list of switches $P(f_i) = \{s_1, s_2, \ldots\}$. For each switch $s_k \in P(f_i)$, s_k can accept f_i and its capacity is sufficient, i.e., $c_k \geq r_i$. More specifically, according to the Section 2.1, an inter-pod flow's preference list consists of core switches; an intra-pod flow's preference list consists of aggregation switches. Flows always prefer switches which have more capacity due to the less possibility of congestion. Furthermore, by prioritizing switches with more available capacity can also reduce the flow's FCT (Flow Completion Time).

2) Each switch has a preference list of flows $P(s_i) = \{f_1, f_2, \ldots\}$. For each flow $f_k \in P(s_i)$, f_k can be accepted by s_i and the total data rate does not exceed capacity of s_i. The sequence order of flows in these lists can be determined by their data rates to fully utilize their forwarding capability.

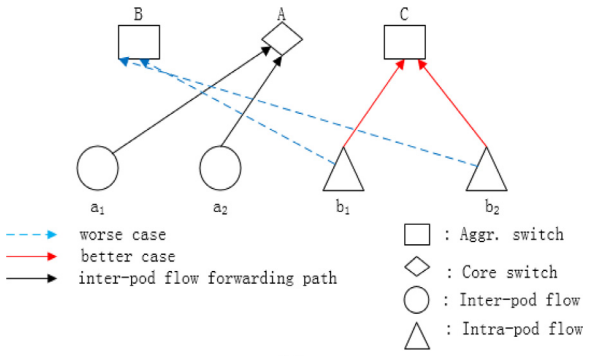
Under the preference lists for both flows and switches with the switch's memory occupancy constraint, our objective is to find an optimal matching between flows and switches, in which for each flow, no better switch can accept it, while for each switch, no larger flow is rejected when it still has enough available capacity. Let M be a *matching* between flows and switches. Denote $M(f_i)$ to be the switch which accepts f_i. Similarly, denote $M(s_i)$ to be the set of flows which are assigned to s_i. $x \succ_z y$ means that x is prioritized over y in the preference list $P(z)$. Let $E(f_i, s_j) = 1$ represent the event that f_i and s_j are not matched in M while s_j has enough capacity to accept f_i and both s_j and f_i prefer each other to their currently matched ones in M. When $E(f_i, s_j) = 1$, the pair (f_i, s_j) is a *blocking pair* which we define and elaborate it in Section 3.2

Then, we define the *FSSM (Flow-Switch Stable Matching)* problem as follows:

Definition 1. Given the set of S and F, the FSSM problem is to find a matching $M = \{(f_i, s_j) | f_i \in P(s_j), s_j \in P(f_i)\}$ with maximum cardinality

$$max\ |M|$$

$$s.t.\ c_j \geq \sum_{f_i \in M(s_j)} r_i$$



(a)

Switch	Capacity	Flow	Data rate
A	9	a1	2
B	10	a2	6
C	7	b1	2
		b2	4

(b)

Fig. 3. Example of confliction.

$$E(f_i, M(f_i)) = 0$$

$$|M(f_i)| \leq 1$$

Where $j = 1, 2, \ldots, |S|$ and $i = 1, 2, \ldots, |F|$.

The objective of *FSSM* problem is to obtain a matching between switches and flows with maximum cardinality and no blocking pairs. The first constraint ensures that capacity of all switches are not overflowed. And the last constraint ensures that each flow can only be matched to at most one switch.

Theorem 1. *The FSSM problem is NP-hard.*

Proof. The NP-hardness of this problem is proved by the following two steps.

Step 1: Obviously, this problem is in NP.

Step 2: To show the NP-hardness, we will show that the partition problem (PP) [20], which is already known NP-hard, is reducible to this problem in polynomial time.

The PP is defined as follows. Given a multiset S of integers (a_1, a_2, ... , a_n), is there a way to partition S into two subsets S_1 and S_2 such that the sum of the numbers in S_1 equals the sum of the numbers in S_2?

Consider a special case of FSSM problem with only two switches, i.e., s_1 and s_2, both with capacity 1. There are n flows (f_1, f_2, ... , f_n), and their demands are $r_i = 2 \cdot r_i/R$, where $R = \Sigma r_i$. Suppose that each switch has all flows in one tie, and all flows take the two switches in one tie, i.e., both switches and flows makes no distinction on each other. Therefore, all flows must be matched in M, and all switches fully accepted flows. The two set are $S_1 = \{f_i | M(f_i) = s_1\}$ and $S_2 = \{f_i | M(f_i) = s_2\}$. Therefore, the PP is reducible to the FSSM problem. □

2.3. Matching conflict

As the description above, each flow has a preference list over all the acceptable switches that have sufficient capacities. Similarly, each switch has a preference list over all the acceptable flows whose rate is smaller than its available capacity.

But in a practical scenario, considering the capacity constraints of switches, the mutual preference between switches and flows

may cause conflict during flow scheduling. Fig. 3 shows an example of such confliction. There are two elephant flows: flow *a* and flow *b*. Suppose flow *a* is an inter-pod flow and its data rate is 8; flow *b* is an intra-pod flow and its data rate is 6. Also there is a core switch *A* with capacity of 9, an aggregation switch *B* with capacity of 10 and an aggregation switch *C* with capacity of 7. *A* is connected with *B* and *C* respectively. In the case 1, an inter-pod flow *a* chooses *A* as its appropriate core switch while *B* is on this forwarding path. Meanwhile, another intra-pod flow *b* has chosen *B* as its appropriate Aggr. switch. This may cause congestion in *B*. The reason is that the inter-pod flows scheduling neglects whether the aggregation switch on this forwarding path has enough capacity to transmit this flow, which may cause retransmission of flows or congestion on aggregation switches. We call this the conflict between intra-pod flows and inter-pod flows on aggregation switches. We believe case 2 is a better matching for these flows and switches, because it can avoid congestion. So we need to find a way to get rid of case 1.

Our solution for this conflict is to perform scheduling for intra-pod flows and inter-pod flows separately, while they are all scheduled by the same controller. Furthermore, as the main reason of the conflict is the limited resources of aggregation switches, we monitor the status information of aggregation switches' resource during scheduling. When flows to be scheduled contain both intra-pod flows and inter-pod flows, these information would be monitored by the controller to avoid exhausting the aggregation switches' capacity and avoid congestion. Specifically, When we do the matching for an inter-pod flow, we choose an appropriate core switch for this flow as well as modifying status of corresponding Aggr. switch according to its resource consumption to ensure that Aggr. switch's resource is not overused.

3. Fincher scheme

In this section, we will present *Fincher*, a stable matching based solution for the Flow-Switch Stable Matching Problem.

3.1. Threshold of remained capacity

The problem in Definition 1 is actually a stable matching problem. The most classic and simple stable matching problem is Stable Marriage (SM) problem [21]. In SM problem, each woman is only matched to one man, meanwhile, each man is matched to one woman. Stable Marriage problem is a one-to-one matching and its solution is called Deferred Acceptance (DA) algorithm. DA algorithm is an effective solution that it uses the deferred acceptance mechanism to handle the matching between men and women. However, in FSSM problem, a switch can be matched to multiple flows, i.e., a many-to-one matching problem. Furthermore, the resource requirement and capacity in FSSM are all non-unitary compared with traditional stable marriage problem. Thus, we can not directly apply DA algorithm to solve FSSM.

For example, suppose there are two switches with capacity of 10 and 9 respectively. Also there are two flows respectively and their demands are 6 and 4. If we directly apply the DA for this case, these two flows may match to the switch which has capacity 10 and the other is free. So in this case, DA algorithm becomes a greedy matching solution which may cause congestion. Furthermore, this matching mechanism may keep some switches busy but let other switches free which is not we want. Thus directly apply DA algorithm for FSSM problem is not an effective solution. So, we need a better way to handle the challenge of load unbalance.

Next, we introduce a threshold-based method to achieve balance among all switches when applying stable matching. The threshold T is defined as the minimum remained capacity of switches during each scheduling round. Switches can not accept more flows when its remained capacity is less than or equal to T. The value of T should be chosen carefully. A smaller T may cause unbalanced load and a larger T may cause some flows can not be matched to any switches. We now give one definition of threshold T as follows:

$$T = \frac{\sum c_j - \sum r_i}{|S|} \tag{1}$$

Thus, the threshold T can ensure that each switch handle the same size of flows during each scheduling for load balancing. However, as described above, if T is too large, some flows would not be matched to any switches. So if threshold T is defined in (1), we may not get maximum cardinality. We need to find a new way to obtain an appropriate value for T to meet our requirement.

Theorem 2. *With the threshold defined in (1), in the worst case, there are at most* $(|S| - 1)$ *flows, which are not matched.*

Proof. We assume that there are $|S|$ flows not matched, and their rates are $r_1, r_2, \ldots, r_{|S|}$ and $r_1 \geq r_2 \geq \ldots \geq r_{|S|}$. The worst case is that each switch has the same remained capacity, which means $(c_1 - T) = (c_2 - T) = \ldots = (c_{|S|} - T)$. According to the definition of T in (1), we know that

$$\sum r_i = \sum (c_j - T) \tag{2}$$

Now, there are $|S|$ free flows and $|S|$ switches. We discuss as follow: i) $r_1 = r_2 = \ldots = r_{|S|}$, we can know that $r_1 = r_2 = \ldots = r_{|S|} = (c_1 - T) = (c_2 - T) = \ldots = (c_{|S|} - T)$. It means all flows can match a switch because switches have enough remained capacity to accept flows. So it contradicts our consumption. ii) $r_a \neq r_b$, we can know that $r_i \leq (c_j - T)$ so that s_j which has enough remained capacity to accept f_i , which contradicts our assumption. □

According to Theorem 2, there are at most $|S| - 1$ flows, which are not matched to any switches, in the worst case. Inspired by this, we can adjust the T to let switches accepting more flows to handle the challenge we discussed above. We let the threshold T minus the data rate of flow which is ranked $|F| + 1 - |S|$ to assure all flows would match to a switch, because the flow ranked $|F| + 1 - |S|$ is the largest flow in the last $|S|$ - 1 flows that are not matched in the worst case. For the ease of explanation, we denote Δ to be the data rate of flow which ranked $(|F| + 1 - |S|)$. So we can obtain the new definition of threshold T as follows.

Definition 2. The new threshold of switches' capacity is

$$T = \frac{\sum c_j - \sum r_i}{|S|} - \Delta \tag{3}$$

More specifically, for aggregation switches, S is the set of aggregation switches in the same pod, Σc_j is the sum of the capacity of the aggregation switches at this pod and Σr_i is the sum of data rates of all flows which are generated in this pod. Similarly, for core switch, S is all core switches, Σc_j is the sum of the capacity of all core switches, and Σr_i is the sum of data rates of all inter-pod flows.

3.2. Stability of FSSM

As we have described above, each flow has a preference list of switches and each switch has a preference list of flows. So we define the blocking pair as follows [19,22].

Definition 3. A flow - switch pair $(f_i , s_j) \notin M$ is a blocking pair if following conditions hold:

$$(c_j - T) \leq r_i \text{ and } (c_j + \sum r'_i - T) \geq r_i \tag{4}$$

where $f_i \succ_{s_j} M(f_i)$, $f'_i \in F$, and $s_j \succ_{f_i} M(f_i)$.

Definition 4. A flow-switch matching M is stable if it does not contain any blocking pairs.[1]

According to the definition of the stability of FSSM problem, we need to find a solution to get the stable matching with no blocking pairs.

3.3. Fincher algorithm

To solve the conflict between flows and switches, and find a stable matching between them, we present our modified Gale–Shapley algorithm [7,21], which is called *Fincher* algorithm: elephant Flow scheduling based on stable matching in data center networks.

The detailed steps of *Fincher* are shown in Algorithm 1, which is guaranteed to find a stable matching for a given problem. The key idea is to ensure that, whenever a flow is rejected by a switch, any less preferable flows will never be accepted by the switch, even if it has enough capacity to do so.

The algorithm starts with a set of flows and a set of switches. Each flow and switch is set to be free initially. Then, the algorithm enters a propose-reject procedure. Whenever there are free flows and their preference lists of switches are not empty, we randomly pick one, say f_i, which its preference list contains all the switches that have not yet rejected it. If s_j has sufficient capacity, it holds the offer. Otherwise, it sequentially rejects offers from preferable flows f_i' until it can take the offer, in the order of its preference. If it still cannot do so even after rejecting all the f_i's, f_i is rejected. Whenever a switch rejects a flow, it updates the best_rejected variable, and at the end all flows ranked lower than best_rejected are removed from its preference list, i.e., line 17∼ 19. The switch is also removed from preference list of all these flows, as it will never accept their offers. Especially, in line 6 and line 14, $c_j = c_j$ - r_i means 'occupying' the switches remain capacity, but the calculation of s_j capacity is a little different for intra-pod flows and inter-pod flows. For intra-pod flows, it means that we only calculating the aggregation switches' remaining free memory space which is on flows' forwarding path. For inter-pod flows, it means calculating both aggregation switch and core switches remain capacity which are on flows' forwarding path.

Moreover, *Fincher* needs solutions for mice flow scheduling. For mice flows, some existing schemes are efficiently handling their scheduling, e.g. ECMP, [9,23]. These schemes help us for scheduling mice flows (we choose ECMP in our experiments).

Theorem 3. *Fincher algorithm can always output a stable matching M within $O(|F|^2)$.*

Proof. We prove the stability of the outcome by introducing a contradiction. Suppose that *Fincher* produces a matching M with a blocking pair (f_i, s_j), i.e., there is at least one flow f_i' is worse than f_i to s_j in $M(s_j)$. Since $s_j \succ_{f_i} M(f_i)$, f_i must have proposed to s_j and been rejected. When f_i was rejected, f_i' was either rejected before f_i, or was made unable to propose to s_j because s_j is removed from the preferences of all the flows ranked lower than f_i. Thus f_i' doesn't match a switch, which contradicts with the assumption.

It also proves the existence of stable matching, as *Fincher* algorithm terminates within $O(|F|^2)$ in the worst case. □

[1] Considering the complexity and existence of a stable matching, we only consider one type of blocking pair for stable matching. For more detailed definitions on blocking pairs and stable matching, please refer to [19].

4. Implementation

In this section, we discuss implementation of the prototype of *Fincher* on POX [24], which is an open source SDN controller. The overview of all components of *Fincher* are described in Fig. 4.

Fincher is implemented with about 400 lines of Python code based on the implementation of Hedera.[2] Generally, *Fincher* mainly contains three modules: *Scheduler, Statistics Collector* and *Monitor*. We elaborate their details as follow:

Scheduler: The Scheduler is the core part of our *Fincher* which implements the Algorithm 1 described in Section 3 as a module

Algorithm 1 Fincher algorithm.

Input: Flows F, Switches S, the preference list of F & S
Output: a stable matching M

```
1: set all flows and switches free & calculate the threshold T
2: M = ∅
3: while ∃ f_i who is free do
4:   s_j = f_i's highest ranked switch in preference list
5:   if (c_j - T) ≥ r_i then
6:     M = M ∪ (f_i , s_j) , c_j = c_j - r_i
7:   else
8:     find all f_i' matched to s_j so far such that f_i' ≺_{s_j} f_i
9:     repeat
10:      M = M \ (f_i' , s_j) in order of the preference list of s_j
11:      c_j = c_j + r_i' , best_rejected = f_i
12:    until (c_j - T) ≥ r_i or all f_i' are rejected
13:    if c_j ≥ r_i then
14:      M = M ∪ (f_i , s_j) , c_j = c_j - r_i
15:    else
16:      f_i becomes free, best_rejected = f_i
17:    end if
18:    for each f∈s_j's preference list && f ≺_{s_j} best_rejected do
19:      delete f from s_j's preference list, s_j from f's preference
         list
20:    end for
21:  end if
22: end while
```

of POX. After Statistics Collector got the statistics of both flows and switches in discrete time ticks, it would trigger the Scheduler pulled the data through the interface between these two modules. Meanwhile, Scheduler is implemented as POX module so it 'knows' network topology when it connect to the SDN-enabled network (which it can list all switches). Then Scheduler generates the preference lists of flows and switches according to the statistics from Statistics Collector and starts to match flows and switches which follow the algorithm described in 3. The matching result which stands for forwarding paths which would be installed upon network via packet-out message later and flows would follow these rules to transmitting data.

Statistics Collector: The Statistics Collector is used for network statistics collection. We use POX controller to handle the interface with Monitor (via Openflow). In every α time period which is defined as scheduling time unit, Statistics Collector triggers the statistics collection procedure to get data from Monitor using OFPT_STATS_REQUEST and OFPT_STATS_REPLY messages. α is a parameter which can be set and changed by operators according to current network status (in our experiment, α is set to be 10 s, which is a heuristic setting that most flow would finish transmitting under this time slot). After getting all statistical data, Statis-

[2] The source code of Hedera in our implementation is obtained from: https://bitbucket.org/msharif/hedera/src.

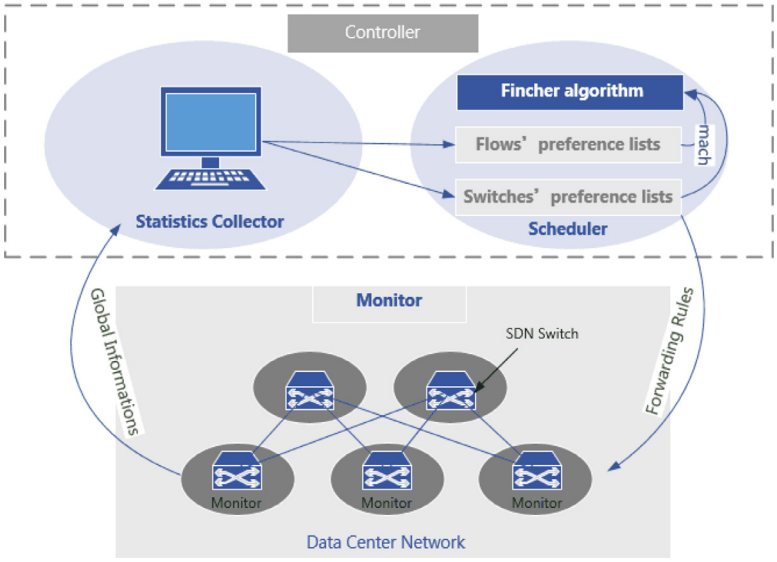


Fig. 4. Overview of *Fincher*.

tics Collector sends a signal to Scheduler through the interface between them.

Monitor: Monitor collects statistics of flows and switches which would be used matching. Some statistics like flows' data rate and switches' memory occupancy which are crucial data for matching procedure and these data must be monitored and obtained in switches. Monitor collect these statistics and waiting for OFPT_STATS_REQUEST message and then push the statistics to Statistics Collector when it got a request. Monitor is implemented in a Linux queueing discipline (qdisc) kernel module for monitoring switch and flow states. Because Openflow does not allow to report neither the memory availability of switches or the data rate of flows. So we customized the Openflow to support that switches can report the memory availability to controller under our customized protocol. Customized Openflow adds a new field to report switch memory occupancy. Then Monitor would send these statistics to Statistics Collector under customized OFPT_STATS_REPLY messages.

5. Evaluation

We have evaluated the performance of *Fincher* in a relatively large scale with higher path diversity using POX and Mininet platform. Mininet [25] is a high fidelity network emulator. Routing is completely under the control of a centralized POX [24] controller. In this section, we evaluate the effectiveness and scalability of *Fincher* over a series of experiments.

5.1. Evaluation setup

We have implemented *Fincher* on a POX controller, which is connected to the Mininet through LAN. The POX controller is running on a server with a 4-core Intel i5-3210 2.5 GHz CPU and 4GB memory, installed upon Ubuntu 14.04 LTS 64-bit. A fat-tree topology is constructed in Mininet as the network topology, which is commonly used in data centers.

Our experiments are conducted with the fat-tree topology scaled from 4-pod to 8-pod. And the network contains 36–200 nodes and 4–8 core switches which means 4–16 equal-cost paths between any pair of hosts at different pods. Note that all links in the topology are set to 1 Gbps because of the limited switching ability on a single machine. The buffer size at each switch output port is configured to 200, 180, 170, 160 packets [7], in order to simulate the diversity of background traffic. The scheduling scheme for mice flows is ECMP. The scheduling interval α is set to be 10 s.

Benchmark Workloads: We evaluated *Fincher* with a set of synthetic and realistic workloads. Similar to previous works [1,4,26], our synthetic workload is a group of communication patterns according to the following styles:

Stride (i). We index all hosts in the topology from left to right. A host with index x sends data to the host with index $(x+i)$ mod (num of hosts).

Staggered Prob (EdgeP, PodP). A host sends to another host under the same edge switch with probability of *EdgeP*, and to another host within the same pod with probability of *PodP*, and to the rest of the network with probability of $1 - EdgeP - PodP$.

Random. A host sends to a random destination not in the same pod as itself with uniform probability. We include bijective mappings and ones where hotspots are present.

Performance Evaluation: We compare *Fincher* to both ECMP and Hedera.

ECMP hashes the five-tuples, i.e., *src* and *dst IP*, *src* and *dst port* number, and protocol number, and selects a path for each flow according to flow's hash value. It is a proactive scheme which needs flows' five-tuples to choose a forwarding path and it is widely used for multi-path routing. Hedera is a centralized flow scheduling system for fat-tree, which detects large flows exceeding 100 Mbps at edge switches, and selects a suitable path with suitable path according to the result of the estimated demand of large flows.

We have evaluated the performance of *Fincher*, Hedera and ECMP on various performance metrics, such as normalized bisection bandwidth and flow completion time (FCT). Moreover, we also calculated the signaling overhead of all three solutions above. The signaling overhead was obtained by carefully calculated the number of packets used in communication between Controller and switches and the size of packet-in and packet-out message in customized protocol. Wireshark [27] is used for detecting the signaling packets and analyze the overhead of these packets.

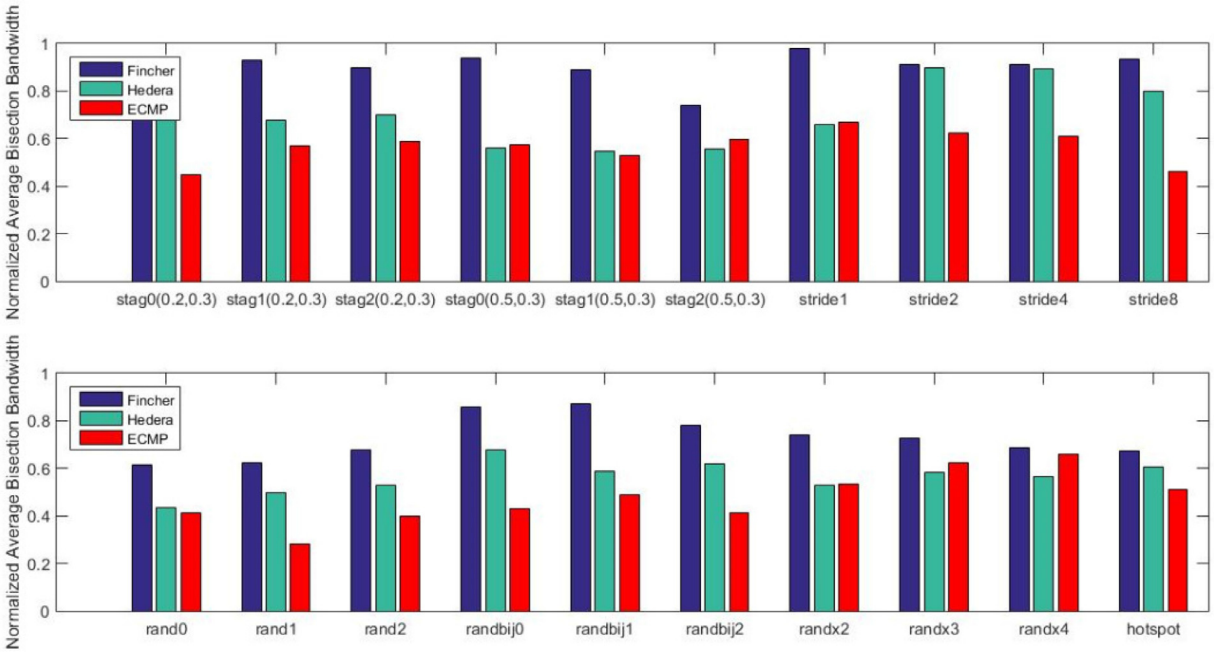


Fig. 5. 4-pod fat-tree normalized bisection bandwidth.

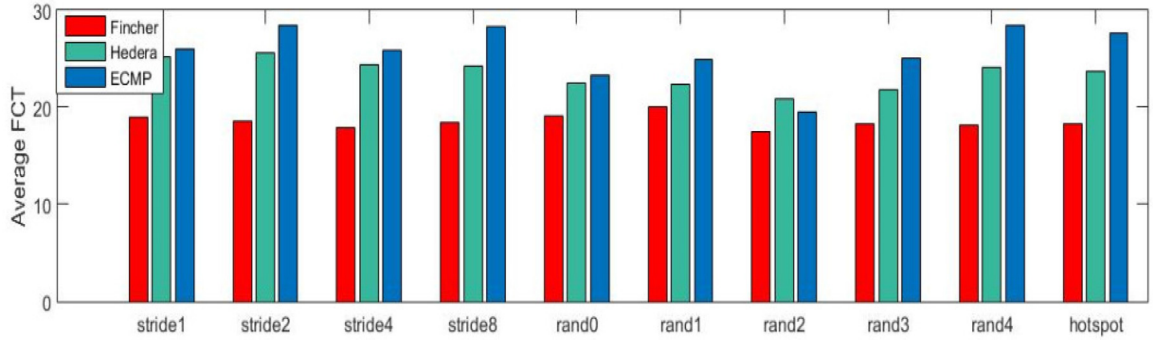


Fig. 6. Average FCT.

5.2. Evaluation results and analysis

We first evaluated the effectiveness of *Fincher* over a couple of analysis metrics: (i) *Fincher*'s improvement in bisection bandwidth performance; (ii) *Fincher*'s ability to avoid congestion in network. Both experiments compare *Fincher* to ECMP and Hedera to demonstrate the actual improvement in performance. Then we evaluated *Fincher*'s scalability over some experiments. We tested *Fincher*'s performance over larger scale network and larger network load to demonstrate *Fincher*'s ability of handling congestion and larger scale network.

We evaluated *Fincher*'s ability to promote network bisection bandwidth. Fig. 5 shows the overall normalized bisection bandwidth performance of ECMP, Hedera and *Fincher* in a fat-tree network. In most of simulations, *Fincher* algorithm has the best performance. Then, we also let each host send a 125MB elephant flow to its companion host. Fig. 6 shows each elephant flows average FCT of ECMP, Hedera and *Fincher* and it also shows all flows' performance in FCT under different scheme. From the Fig. 6, we can know that the performance of Fincher is the best, and the performance of ECMP is the worst. Fig. 7 shows that *Fincher* achieves 14% to 35% reduction and average 29% reduction in the FCT of elephant flow compared to ECMP and average 29%. It also shows that *Fincher* achieves 10% to 28% reduction and average 21% reduction in the FCT of elephant flow compared to Hedera. For these two parts of the experiment, we know that *Fincher* is a more effective scheme.

Fig. 8 shows the performance improvement of *Fincher* compared to both ECMP and Hedera. From the Fig. 8, we know that 90% elephant flows finished transmission in less than 20 s and all flows finished transmission in less than 25 s when flows are scheduled by *Fincher* scheme. And also, the Fig. 8 shows that 90% elephant flows finished transmission in less than 26 s and all finished transmission in less than 35 s when flows are scheduled by Hedera scheme and 90% elephant flows finished transmission in less than 38 s and all finished transmission in less than 50 s when flows are forwarded under ECMP algorithm. This means most of flows under *Fincher* scheduled get better performance.

ECMP suffers from lower bisection bandwidth when flows are hashed to the same path (maybe it is due to hash collision). Hashing on the same path leads to congestion and thus increased latency and retransmission. And *Fincher* uses the stable matching theory to assign all flows to appropriate paths and its matching mechanism is effectively avoid congestion. And also *Fincher* leverages the threshold to ensure load balance and avoid underlying congestion so that network capacity is utilized more efficiently. Furthermore, we leverage SDN architecture, which aggregates the network-wide control logic and information into a logically cen-

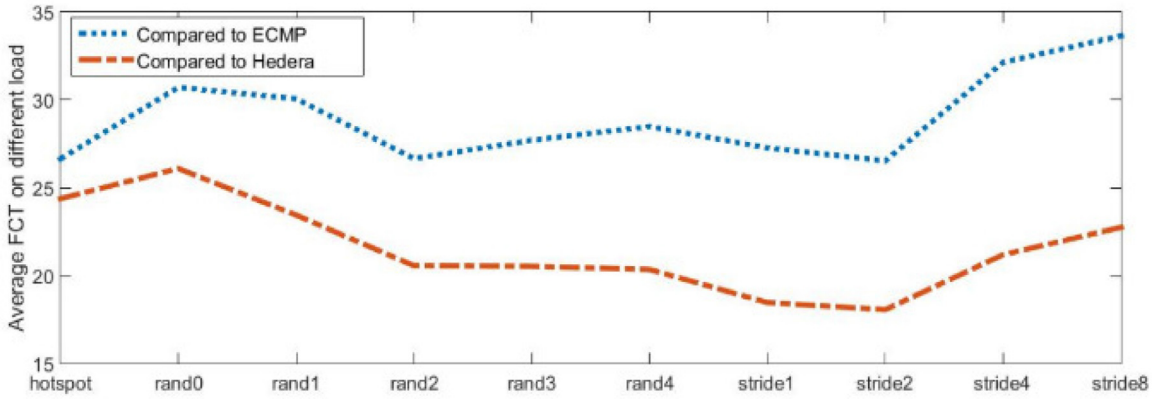


Fig. 7. Reduction of average FCT.

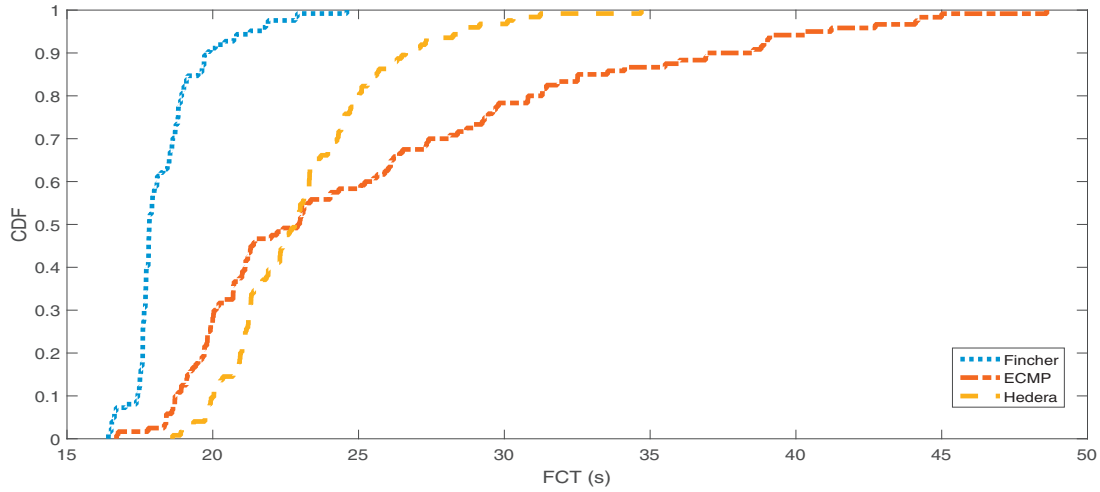


Fig. 8. CDF of FCT.

tralized software component, *Fincher* can simply use global information of flows and switches to fully utilize the capability of the network. Hedera focus on assigning one flow to current best path, but it ignore that this path may be assigned to another flow so that it causes conflict and both flow cannot get the best performance. But our *Fincher* scheme matches each flow to an appropriate path so that it will get the global best optimal paths for all flows. It means that the path which is assigned to a flow may be not the best path but it's an appropriate path so that all flows would get an optimal matching and promote whole network's performance. Thus, *Fincher* has better performance compared to ECMP and Hedera.

Since stable matching is applied for elephant flow scheduling which would get an optimal result for both data transmission and network capacity, both of the performance of bisection bandwidth and FCT show *Fincher*'s effectiveness.

By the way, we evaluated *Fincher*'s overhead when doing scheduling. As we mentioned before, *Fincher* do scheduling every 10 s so that it can handle the dynamic traffic. We got the information of signaling overhead and Table 2 shows the detail. Our scheme triggers scheduling every 10 s and the Monitor module would send flows statistics to Controller under our customized protocol. The packet-out message in customized protocol is 30 bytes. The total overhead is 2904 bytes in 4-pod experiment and 34560 bytes in 8-pod experiment which we believe consumed very little of network capacity.

Then we evaluate *Fincher*' s scalability over a larger network load and a larger network scale. We evaluate *Fincher* scheme on a 8-pod fat-tree which is a larger network compared to 4-pod fat-tree. Fig. 11 show that *Fincher* got the best performance. It shows that *Fincher* perform well even in a larger scale network. We believe that is an effective mechanism to handle network congestion avoidance and keep network on a high utilization. Fig. 9 shows the average FCT of flows when the number of elephant flows is increasing. In Fig. 9, when the load is increasing, flows FCT is increasing undoubtedly, but the performance of *Fincher* is the best in all cases, while performance of ECMP is the worst. FCT of Fincher is always the lowest while the FCT of ECMP is the highest. In Fig. 10, we compared *Fincher* and Hedera to ECMP. Fig. 10 shows that *Fincher* achieves 27%–31% reduction and average 29% reduction in the FCT of elephant flow compared to ECMP. On the other hand, Hedera achieves 8%–22% reduction and average 16% reduction in the FCT of elephant flow compared to ECMP. For this, we know that *Fincher* has a better performance and *Fincher* can still perform well under a larger network load. We believe that even in a larger network, *Fincher* can function well. In this scenario, *Fincher*'s performance is better than Hedera's. *Fincher* schedules flows with

Table 2
Cost with increasing problem size in different scale.

	Packets number	Total size
4-pod	12	2904Byte
8-pod	768	34560Byte

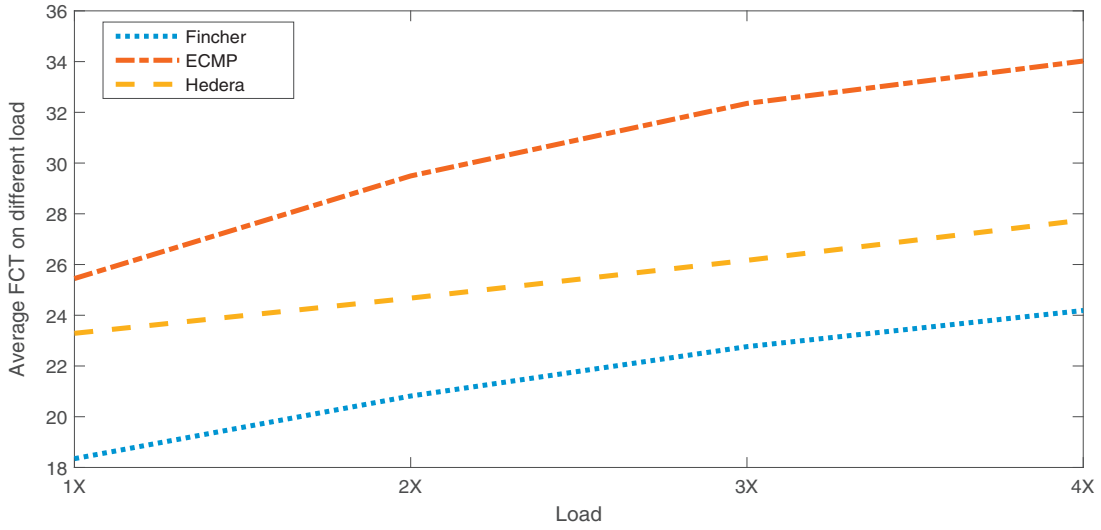


Fig. 9. Average FCT in different load.

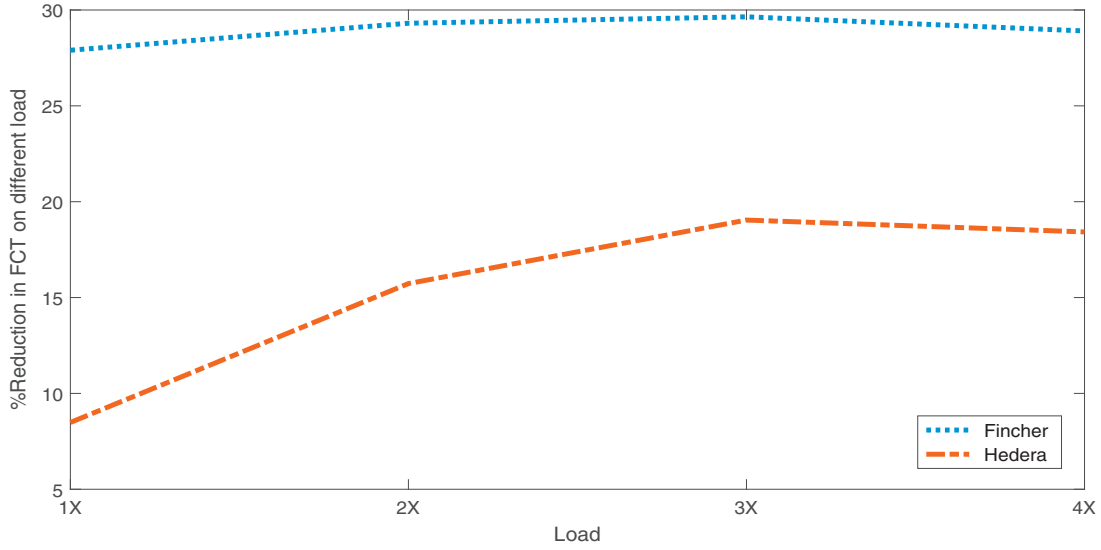


Fig. 10. Reduction of average FCT in different load.

global information of the network to get a stable matching between switches and flows and the stability of this matching ensures the network avoids congestion. Thus, when network load increased, *Fincher* can still handle the conflict of all flows' demand and avoid the congestion.

6. Related work

Elephant flow scheduling in data center network has been an active research area in recent years. There are many existing works in this area.

Al-Fares et al. proposed a centralized flow scheduler for fat-tree in [1]. Once a flow grows above a threshold, the edge switch will send a notification to the central scheduler with the information of large flows. The central scheduler is responsible for reassigning paths to large flows. Hedera [4] is another typical centralized flow scheduling system for fat-tree. Hedera detects large flows at edge switches, and selects a path according to the result of the estimated demand of large flows. Mahout [13] improves the scalability of elephant detection with a host-based mechanism that monitors the host socket buffers to detect elephants. When it detects an elephant flow, centralized controller will assign the least congested path for this elephant flow. Baatdaat [14] uses spare network capacity to mitigate the performance degradation of heavily utilized links. PIAS [15] is a DCN flow scheduling mechanism that aims to minimize FCT with Shortest Job First (SJF) principle and using priority queues. Those schemes neglect memory occupancy and data rate which are an important signal of congestion and a characteristic of flows. Meanwhile some of these work focus on current scheduling flow's effectiveness which may lead to unbalanced traffic. *Fincher* not only focuses on the flows' performance improvement, but also takes some methods to avoid underlying congestion. Moreover, *Fincher* is a global optimal solution.

Another direction of flows scheduling is focusing on transferring of a small size granularity of data. VL2 [5] makes use of two techniques, VLB (Valiant Load Balancing) and ECMP (Equal Cost MultiPath), to offer hot-spot-free performance. VL2 implements VLB by sending flows through a randomly-chosen intermediate switch, and ECMP by distributing flows across paths with equal cost. TinyFlow, a novel scheduling approach for elephant flows,

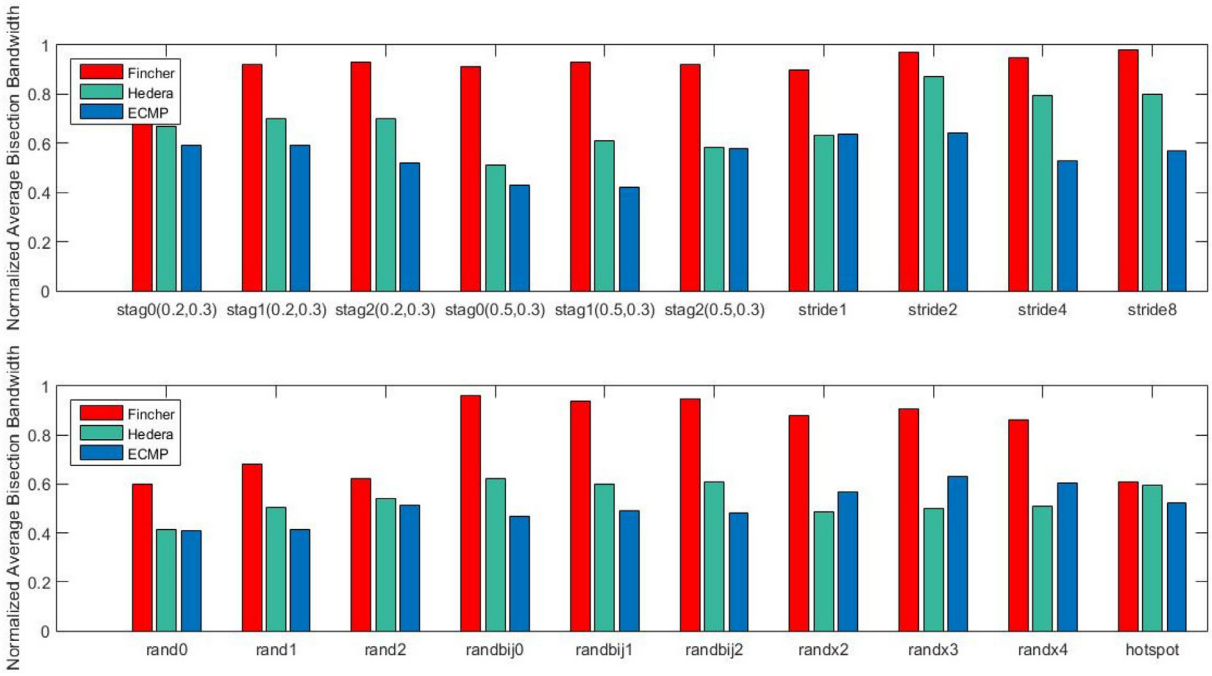


Fig. 11. 8-pod fat-tree normalized bisection bandwidth.

was presented in [7]. TinyFlow changes the traffic characteristics of data center networks to be amenable to ECMP by breaking elephants into mice, and leverages ECMP to achieve load balance. J. Cao et al. proposed [9] Digit-Reversal Bouncing (DRB) algorithm to balance all flows based on per-packet. These works achieve good performance but actually causes reordering and large overhead. The above solutions may cause reordering problem (though some proposals figure out reordering by some methods, they cause lots of overhead), and *Fincher* focus on 'flows' so that it can be more effective and avoid the reordering problem.

There are some existing solutions for handling congestion in DCNs. Detail [10], which proposed a new network stack, attempts to reduce flow completion time in DC networks for web site load times by using flow priorities and switch port buffer occupancies to determine next hop behavior. DCTCP [11] enabled ECN notification which would tags packets when the switch's memory occupancy excess a threshold. Then hosts would limit its data rate according to the fractions of ECN-tagged packets. Presto [8] select an end-to-end forwarding path for each flows at servers under round robin mechanism. MicroTE [28] uses short-term traffic patterns and partial predictability to achieve its goals. s DCell [29] and SPAIN [30], use servers to determine packet routing or relaying, which requires modification to end hosts. Above research works gave us another aspect of flows scheduling, congestion control and load balancing and enlightened us to improve *Fincher*.

The key idea of *Fincher* is applying stable matching algorithm in elephant flow scheduling in data center networks. We note that some research work has applied stable matching in network problem. Xu et al. proposed Anchor [22], which is a general resource management architecture that uses the stable matching framework and matches VMs with heterogeneous resource needs to servers. Cui et al. propose a MSA algorithm, aiming to find a stable matching, where both the preference of femtocells and that of users are satisfied under a backhaul capacity constraint [31]. In [32], they modeled the dynamic controller assignment (DCA) problem as a stable matching problem, and proposed a hierarchically two-phase algorithm that integrates key concepts from both stable matching theory and coalitional games to reduce controller response time. These works inspired us to apply stable matching theory on flows scheduling.

Above solutions lack effective methods or need a better performance for avoidance of underlying congestion. And congestion avoidance is as important as high throughput and low latency. *Fincher* uses stable matching to model the conflicts between flows and switches, achieving load balance of the network, avoiding congestion and reducing FCT of flows.

7. Conclusion

In this paper, we studied the challenge of elephant flow scheduling and proposed a solution, *Fincher*, by applying stable matching theory. Through finding a *stable* matching between flows and core/aggregation switches, *Fincher* can effectively improve the performance of large flow and avoid congestion. Experimental evaluation based on POX and Mininet demonstrated *Fincher*'s effectiveness on reducing the completion time of the flows and fully utilizing network capacity both in large scale and small scale network. Future work is needed to improve the performance of the algorithm by considering more other factors, e.g., dynamics of network conditions.

Acknowledgement

This work is partially supported by Chinese National Research Fund (NSFC) Project No. 61402200 and 61602210, the Science and Technology Planning Project of Guangdong Province of China with No. 2014A040401027 and 2015A030401043.

References

[1] M. Al-Fares, A. Loukissas, A. Vahdat, A scalable, commodity data center network architecture, Comput. Commun. Rev. 38 (2008) 63–74. ACM SIGCOMM.
[2] C. Hong, M. Caesar, P. Godfrey, Finishing flows quickly with preemptive scheduling, Comput. Commun. Rev. 42 (4) (2012) 127–138. ACM SIGCOMM.
[3] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, et al., Jupiter rising: a decade of clos topologies and centralized control in google's datacenter network, Comput. Commun. Rev. 45 (4) (2015) 183–197. ACM SIGCOMM.

[4] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, A. Vahdat, Hedera: Dynamic flow scheduling for data center networks, in: Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation, 2010. 19–19.
[5] A. Greenberg, J.R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D.A. Maltz, P. Patel, S. Sengupta, Vl2: a scalable and flexible data center network, Comput. Commun. Rev. 39 (2009) 51–62. ACM SIGCOMM.
[6] S. Hu, K. Chen, H. Wu, W. Bai, C. Lan, H. Wang, H. Zhao, C. Guo, Explicit path control in commodity data centers: Design and applications, in: 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), 2015, pp. 15–28.
[7] H. Xu, B. Li, Tinyflow: Breaking elephants down into mice in data center networks, in: IEEE 20th International Workshop on Local & Metropolitan Area Networks (LANMAN), 2014 , IEEE, 2014, pp. 1–6.
[8] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, A. Akella, Presto: Edge-based load balancing for fast datacenter networks, in: Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, ACM, 2015, pp. 465–478.
[9] J. Cao, R. Xia, P. Yang, C. Guo, G. Lu, L. Yuan, Y. Zheng, H. Wu, Y. Xiong, D. Maltz, Per-packet load-balanced, low-latency routing for clos-based data center networks, in: Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies, ACM, 2013, pp. 49–60.
[10] D. Zats, T. Das, P. Mohan, D. Borthakur, R. Katz, Detail: reducing the flow completion time tail in datacenter networks, Comput. Commun. Rev. 42 (4) (2012) 139–150. ACM SIGCOMM.
[11] M. Alizadeh, A. Greenberg, D.A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, M. Sridharan, Data center tcp (dctcp), Comput. Commun. Rev. 41 (4) (2011) 63–74. ACM SIGCOMM.
[12] A. Dixit, P. Prakash, Y.C. Hu, R.R. Kompella, On the impact of packet spraying in data center networks, in: INFOCOM, 2013 Proceedings IEEE, IEEE, 2013, pp. 2130–2138.
[13] A.R. Curtis, W. Kim, P. Yalagandula, Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection, in: INFOCOM, 2011 Proceedings IEEE, IEEE, 2011, pp. 1629–1637.
[14] F.P. Tso, G. Hamilton, R. Weber, C.S. Perkins, D.P. Pezaros, Longer is better: exploiting path diversity in data center networks, in: IEEE 33rd International Conference on Distributed Computing Systems (ICDCS), 2013, IEEE, 2013, pp. 430–439.
[15] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, H. Wang, Information-agnostic flow scheduling for commodity data centers, in: 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), 2015, pp. 455–468.
[16] S. Hu, W. Bai, K. Chen, C. Tian, Y. Zhang, H. Wu, Providing bandwidth guarantees, work conservation and low latency simultaneously in the cloud, in: Proceedings of , 2016.
[17] T. Benson, A. Akella, D.A. Maltz, Network traffic characteristics of data centers in the wild, in: Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement, ACM, 2010, pp. 267–280.
[18] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, R. Chaiken, The nature of data center traffic: measurements & analysis, in: Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference, ACM, 2009, pp. 202–208.
[19] K. Iwama, S. Miyazaki, A survey of the stable marriage problem and its variants, in: International Conference on Informatics Education and Research for Knowledge-Circulating Society, 2008. ICKS 2008, IEEE, 2008, pp. 131–136.
[20] Partition Problem. [Online]. http://en.wikipedia.org/wiki/Partition_problem.
[21] D. Gale, L.S. Shapley, College admissions and the stability of marriage, Am. Math. Monthly (1962) 9–15.
[22] H. Xu, B. Li, Anchor: a versatile and efficient framework for resource management in the cloud, IEEE Trans. Parallel Distrib. Syst., 24 (6) (2013) 1066–1076.
[23] H. Xu, B. Li, Repflow: Minimizing flow completion times with replicated flows in data centers, in: IEEE INFOCOM 2014-IEEE Conference on Computer Communications, IEEE, 2014, pp. 1581–1589.
[24] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner, Openflow: enabling innovation in campus networks, Comput. Commun. Rev. 38 (2) (2008) 69–74. ACM SIGCOMM.
[25] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, N. McKeown, Reproducible network experiments using container-based emulation, in: Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies, ACM, 2012, pp. 253–264.
[26] K. He, E. Rozner, K. Agarwal, Y.J. Gu, W. Felter, J. Carter, A. Akella, Ac/dc tcp: Virtual congestion control enforcement for datacenter networks, in: Proceedings of the 2016 Conference on ACM SIGCOMM, ACM, 2016, pp. 244–257.
[27] "Wireshark network protocol analyzer" http://www.wireshark.org.
[28] T. Benson, A. Anand, A. Akella, M. Zhang, Microte: Fine grained traffic engineering for data centers, in: Proceedings of the Seventh Conference on Emerging Networking Experiments and Technologies, ACM, 2011, p. 8.
[29] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, S. Lu, Dcell: a scalable and fault-tolerant network structure for data centers, Comput. Commun. Rev. 38 (2008) 75–86. ACM SIGCOMM.
[30] J. Mudigonda, P. Yalagandula, M. Al-Fares, J.C. Mogul, Spain: Cots data-center ethernet for multipathing over arbitrary topologies., in: NSDI, 10, 2010, pp. 18–33.
[31] L. Cui, W. Jia, A novel mutual-preference-based admission control mechanism for hybrid femtocells system, IEEE Trans.Veh. Technol., 62 (9) (2013) 4555–4564.
[32] T. Wang, F. Liu, J. Guo, H. Xu, Dynamic sdn controller assignment in data center networks: Stable matching with transfers, Proceedings of INFOCOM, 2016.

Yuxiang Zhang received the B.E. degree in network engineering from Jinan University, China, in 2013. He is currently pursuing the Master Degree in computer science in Jinan University. His current research interests are in the area of data center networks.

Lin Cui is currently with the Department of Computer Science at Jinan University, Guangzhou, China. He received the Ph.D. degree from City University of Hong Kong in 2013. He has broad interests in networking systems, with focuses on the following topics: cloud data center resource management, data center networking, software defined networking (SDN), virtualization, distributed systems as well as wireless networking.

Yuan Zhang is studying for the Master Degree at Jinan University, Guangzhou, China, she major in computer application technology. In 2016, she received the Bachelor Degree from Southwest University for Nationalities, Chengdu, China. Now her main research are software defined networking (SDN) and data center networking.