Exploiting Efficient and Scalable Shuffle Transfers in Future Data Center Networks

Deke Guo, *Member, IEEE,* Junjie Xie, Xiaolei Zhou, *Student Member, IEEE,* Xiaomin Zhu, *Member, IEEE,* Wei Wei, *Member, IEEE,* Xueshan Luo

Abstract—Distributed computing systems like MapReduce in data centers transfer massive amount of data across successive processing stages. Such shuffle transfers contribute most of the network traffic and make the network bandwidth become a bottleneck. In many commonly used workloads, data flows in such a transfer are highly correlated and aggregated at the receiver side. To lower down the network traffic and efficiently use the available network bandwidth, we propose to push the aggregation computation into the network and parallelize the shuffle and reduce phases. In this paper, we first examine the gain and feasibility of the in-network aggregation with BCube, a novel server-centric networking structure for future data centers. To exploit such a gain, we model the in-network aggregation problem that is NP-hard in BCube. We propose two approximate methods for building the efficient IRS-based incast aggregation tree and SRS-based shuffle aggregation subgraph, solely based on the labels of their members and the data center topology. We further design scalable forwarding schemes based on Bloom filters to implement in-network aggregation over massive concurrent shuffle transfers. Based on a prototype and large-scale simulations, we demonstrate that our approaches can significantly decrease the amount of network traffic and save the data center resources. Our approaches for BCube can be adapted to other server-centric network structures for future data centers after minimal modifications.

Index Terms—Data center, shuffle transfer, data aggregation

1 Introduction

Data center is the key infrastructure for not only online cloud services but also systems of massively distributed computing, such as MapReduce [1], Dryad [2], CIEL [3], and Pregel [4]. To date, such large-scale systems manage large number of data processing jobs each of which may utilize hundreds even thousands of servers in a data center. These systems follow a data flow computation paradigm, where massive data are transferred across successive processing stages in each job.

Inside a data center, large number of servers and network devices are interconnected using a specific data center network structure. As the network bandwidth of a data center has become a bottleneck for these systems, recently many advanced network structures have been proposed to improve the network capacity for future data centers, such as DCell [5], BCube [6], Fat-Tree [7], VL2 [8], and BCN [9], [10]. We believe that it is more important to efficiently use the available network bandwidth of data centers, compared to just increasing the network capacity. Although prior work [11] focuses on scheduling network resources so as to improve the data center utilization, there has been relatively little work on directly lowering down the network traffic.

In this paper, we focus on managing the network activity at the level of transfers so as to significantly lower down the network traffic and efficiently use the available network bandwidth in future data centers. A transfer refers to the set of all data flows across successive stages of a data processing job. The many-to-many *shuffle* and many-to-one *incast* transfers are commonly needed to support those distributed computing systems. Such data transfers contribute most of the network traffic (about 80% as shown in [11]) and impose severe impacts on the application performance. In a shuffle transfer, the data flows from all senders to each receiver, however, are typically highly correlated. Many state-of-the-practice systems thus already apply aggregation functions at the receiver side of a shuffle transfer to reduce the output data size. For example, each reducer of a MapReduce job is assigned a unique partition of the key range and performs aggregation operations on the content of its partition retrieved from every mapper's output. Such aggregation operations can be the sum, maximum, minimum, count, top-k, KNN, et. al. As prior work [12] has shown, the reduction in the size between the input data and output data of the receiver after aggregation is 81.7% for Mapreduce jobs in Facebook.

Such insights motivate us to push the aggregation computation into the network, i.e., aggregating correlated flows of each shuffle transfer during the transmission process as early as possible rather than just at the receiver side. Such an in-network aggregation can significantly reduce the resulting network traffic of a shuffle transfer and speed up the job processing since the final input data to each reducer will be considerably decreased. It, however, requires involved flows to intersect at some rendezvous nodes that can cache and aggregate packets for supporting the in-network aggregation. In

- *D. Guo, J. Xie, X. Zhou, X. Zhu, and X. Luo are with the College of Information System and Management, National University of Defense Technology, Changsha 410073, P.R. China. E-mail: guodeke@gmail.com.*
- *W. Wei is with the College of Computer Science and Engineering, Xian University of Technology, Xian 710048, P.R. China. E-mail: weiwei@xaut.edu.cn.*

existing tree-based switch-centric structures [7], [8] of data centers, it is difficult for a traditional switch to do so due to the small buffer space shared by too many flows and limited packet processing capability.

Additionally, these server-centric network structures possess high link density, i.e., they provide multiple disjoint routing paths for any pair of servers. Consequently, they bring vast challenges and opportunities in cooperatively scheduling all flows in each shuffle transfer so as to form as many rendezvous servers as possible for performing the in-network aggregation. Currently, a shuffle transfer is just treated as a set of independently unicast transmissions that do not account for collective behaviors of flows. Such a method brings less opportunity for an efficient in-network aggregation.

In this paper, we examine the gain and feasibility of the in-network aggregation on a shuffle transfer in a server-centric data center. To exploit the gain of in-network aggregation, the efficient incast and shuffle transfers are formalized as the minimal incast aggregation tree and minimal shuffle aggregation subgraph problems that are NP-hard in BCube. We then propose two approximate methods, called IRS-based and SRS-based methods, which build the efficient incast aggregation tree and shuffle aggregation subgraph, respectively. We further design scalable forwarding schemes based on Bloom filters to implement in-network aggregation on massive concurrent shuffle transfers.

We evaluate our approaches with a prototype implementation and large-scale simulations. The results indicate that our approaches can significantly reduce the resultant network traffic and save data center resources. More precisely, our SRS-based approach saves the network traffic by 32.87% on average for a small-scale shuffle transfer with 120 members in data centers BCube$(6,k)$ for $2{\leq}k{\leq}8$. It saves the network traffic by 55.33% on average for shuffle transfers with 100 to 3000 members in a large-scale data center BCube$(8,5)$ with 262144 servers. For the in-packet Bloom filter based forwarding scheme in large data centers, we find that a packet will not incur false positive forwarding at the cost of less than 10 bytes of Bloom filter in each packet.

The rest of this paper is organized as follows. We briefly present the preliminaries in Section 2. We formulate the shuffle aggregation problem and propose the shuffle aggregation subgraph building method in Section 3. Section 4 presents scalable forwarding schemes to efficiently perform the in-network aggregation. We evaluate our methods and related work using a prototype and large-scale simulations in Section 5 and conclude this work in Section 6.

2 PRELIMINARIES

2.1 Data center networking

Many efforts have been made towards designing network structures for future data centers and can be roughly divided into two categories. One is switch-centric, which organizes switches into structures other than tree and puts the interconnection intelligence on switches. Fat-Tree [7], VL2 [8] and PortLand [13] fall into such a category, where servers only connect to the edge-level switches. Different from such structures, FBFLY [14] and HyperX [15] organize homogeneous switches, each connecting several servers, into the generalized hypercube [16]. For such switch-centric structures, there has been an interest in adding the in-network processing functionality to network devices. For example, new Cisco ASIC and arista application switches have provided a programmable data plane. Additionally, existing switches have been extended with dedicated devices like sidecar [17], [18]. Sidecar is connected to a switch via a direct link and receives all traffic. Such trends provide a software-defined approach to customize the networking functionality and to implement arbitrary in-network processing within such data centers.

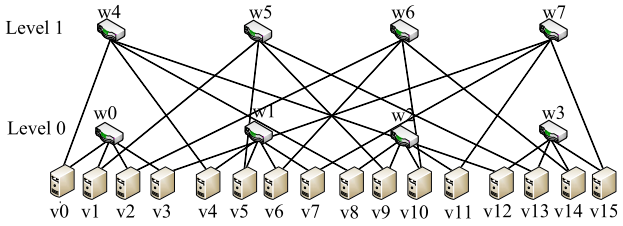


Fig. 1. A BCube(4,1) structure.

The second category is server-centric, which puts the interconnection intelligence on servers and uses switches only as cross-bars (or does not use any switch). BCube [6], DCell [5], CamCube [19], BCN [9], SWDC [20], and Scafida [21] fall into such a category. In such settings, a commodity server with ServerSwitch [22] card acts as not only an end host but also a mini-switch. Each server, thus, uses a gigabit programmable switching chip for customized packet forwarding and leverages its resource for enabling in-network caching and processing, as prior work [22], [23] have shown.

In this paper, we focus on BCube that is an outstanding server-centric structure for data centers. BCube$_0$ consists of n servers connecting to a n-port switch. BCube$_k$ ($k \geq 1$) is constructed from n BCube$_{k-1}$'s and n^k n-port switches. Each server in BCube$_k$ has $k{+}1$ ports. Servers with multiple NIC ports are connected to multiple layers of mini-switches, but those switches are not directly connected. Fig.1 plots a BCube$(4,1)$ structure that consists of $n^{k+1}{=}16$ servers and two levels of $n^k(k{+}1){=}8$ switches.

Essentially, BCube(n,k) is an emulation of a $k{+}1$ dimensional n-ary generalized hypercube [16]. In BCube(n,k), two servers, labeled as $x_kx_{k-1}...x_1x_0$ and $y_ky_{k-1}...y_1y_0$ are mutual one-hop neighbors in dimension j if their labels only differ in dimension j, where x_i and $y_i{\in}\{0,1,...,n{-}1\}$ for $0{\leq}i{\leq}k$. Such servers connect to a j level switch with a label $y_k...y_{j+1}y_{j-1}...y_1y_0$ in BCube(n,k). Therefore, a server and its $n{-}1$ 1-hop neighbors in each dimension are connected indirectly via a common switch. Additionally, two servers are j-hop

neighbors if their labels differ in number of j dimensions for $1 \leq j \leq k+1$. As shown in Fig.1, two servers v_0 and v_{15}, with labels 00 and 33, are 2-hop neighbors since their labels differ in 2 dimensions.

2.2 In-network aggregation

In a sensor network, sensor readings related to the same event or area can be jointly fused together while being forwarded towards the sink. In-network aggregation is defined as the global process of gathering and routing information through a multihop network, processing data at intermediate nodes with the objective of reducing energy consumption, thereby increasing network lifetime. In-network aggregation is a very important aspect of wireless sensor networks and have been widely studied, such as [24] and [25].

Costa et al have introduced the basic idea of such an in-network aggregation to the field of data centers. They built Camdoop [26], a MapReduce-like system running on CamCube data centers. Camdoop exploits the property that CamCube servers forward traffic to perform in-network aggregation of data during the shuffle phase and reduces the network traffic. CamCube differentiates itself from other server-centric structures by utilizing a 3-dimensional torus topology, i.e., each server is directly connected to six neighboring servers.

In the Camdoop, servers use the function getParent to compute the tree topology for an incast transfer. Given the locations of the receiver and a local server (a sender or an inner server), the function returns a one-hop neighbor, which is the nearest one towards the receiver in a 3D space compared to other five ones. In this way, each flow of an incast transfer identifies its routing path to the same receiver, independently. The unicast-based tree approach proposed in this paper shares the similar idea of the tree-building method in Camdoop.

In this paper, we cooperatively manage the network activities of all flows in a transfer for exploiting the benefits of in-network aggregation. That is, the routing paths of all flows in any transfer should be identified at the level of transfer not individual flows. We then build efficient tree and subgraph topologies for incast and shuffle transfers via the IRS-based and SRS-based methods, respectively. Such two methods result in more gains of in-network aggregation than the unicast-based method, as shown in Fig.2 and Section 5.

2.3 Stenier tree problem in general graphs

The Steiner Tree problems have been proved to be NP-hard, even in the special cases of hypercube, generalized Hypercube and BCube. Numerous heuristics for Steiner tree problem in general graphs have been proposed [27], [28]. A simple and natural idea is using a minimum spanning tree (MST) to approximate the Steiner minimum tree. Such kind of methods can achieve a 2-approximation. The Steiner tree problem in general graphs cannot be approximated within a factor of $1+\epsilon$ for sufficiently small ϵ [29]. Several methods, based on greedy strategies originally due to Zelikovsky [30], achieve better approximation ratio than MST-based methods, such as 1.746 [31], 1.693 [32], 1.55 [28]. Such methods have a similar idea. Indeed, they start from an MST and improve it step by step by using a greedy principle to choose a Steiner vertex to connect a triple of vertices.

3 Aggregating shuffle transfers

We first formulate an efficient shuffle transfer as building the minimal shuffle aggregation subgraph, an NP-hard problem, which can be applied to any family of data center topologies. We then propose efficient aggregation methods for incast and shuffle transfers, respectively.

3.1 Problem statement

We model a data center as a graph $G=(V, E)$ with a vertex set V and an edge set E. A vertex of the graph refers a switch or a datacenter server. An edge (u, v) denotes a link, through which u connects with v where $v, u \in V$.

Definition 1: A shuffle transfer has m senders and n receivers where a data flow is established for any pair of sender i and receiver j for $1 \leq i \leq m$ and $1 \leq j \leq n$. An incast transfer consists of m senders and one of n receivers. A shuffle transfer consists of n incast transfers that share the same set of senders but differ in the receiver.

In many commonly used workloads, data flows from all of m senders to all of n receivers in a shuffle transfer are highly correlated. More precisely, for each of its n incast transfers, the list of key-value pairs among m flows share the identical key partition for the same receiver. For this reason, the receiver typically applies an aggregation function, e.g., the sum, maximum, minimum, count, top-k and KNN, to the received data flows in an incast transfer. As prior work [12] has shown, the reduction in the size between the input data and output data of the receiver after aggregation is 81.7% for Facebook jobs.

In this paper, we aim to push aggregation into the network and parallelize the shuffle and reduce phases so as to minimize the resultant traffic of a shuffle transfer. We start with a simplified shuffle transfer with $n=1$, an incast transfer, and then discuss a general shuffle transfer. Given an incast transfer with a receiver R and a set of senders $\{s_1, s_2, ..., s_m\}$, message routes from senders to the same receiver essentially form an aggregation tree. In principle, there exist many such trees for an incast transfer in densely connected data center networks, e.g., BCube. Although any tree topology could be used, tree topologies differ in their aggregation gains. A challenge is how to produce an aggregation tree that minimizes the amount of network traffic of a shuffle transfer after applying the in-network aggregation.

Given an aggregation tree in BCube, we define the total edge weight as its cost metric, i.e., the sum of the

amount of the outgoing traffics of all vertices in the tree. As discussed in Section 2.1, BCube utilizes traditional switches and only data center servers enable the in-network caching and processing. Thus, a vertex in the aggregation tree is an aggregating vertex only if it represents a server and at least two flows converge at it. An aggregating vertex then aggregates its incoming flows and forwards a resultant single flow instead of multiple individual flows along the tree. At a non-aggregating vertex, the size of its outgoing flow is the cumulative size of its incoming flows. The vertices representing switches are always non-aggregating ones. We assume that the introduced traffic from each of m senders is unity (1 MB) so as to normalize the cost of an aggregation tree. In such a way, the weight of the outgoing link is one at an aggregating vertex and equals to the number of incoming flows at a non-aggregating vertex.

Definition 2: For an incast transfer, the minimal aggregation tree problem is to find a connected subgraph in $G{=}(V, E)$ that spans all incast members with minimal cost for completing the incast transfer.

The problem is translated to discover a minimal aggregation tree for an incast transfer in a data center. Consider a relaxation that each vertex is an aggregating one in a data center. This relaxation simplifies the minimal aggregation tree problem as the Steiner tree problem since the weight of each edge becomes one and thus the cost of the tree is the total number of edges. It is well-known that the Steiner tree problem in BCube is NP-hard. Therefore, the minimal aggregation tree of an incast transfer in BCube is NP-hard.

As aforementioned in Section 2.3, there are many approximate algorithms for Steiner tree problem. The time complexity of such algorithms for general graphs is of $O(m{\times}N^2)$, where m and N are the numbers of incast numbers and all servers in a data center. The time complexity is too high to meet the requirement of online tree building for incast transfers in production data centers which hosts large number of servers. For example, Google has more than 450,000 servers in 2006 in its 13th data center and Microsoft and Yahoo! have hundreds of thousands servers. On the other hand, such algorithms cannot efficiently exploit the topological feature of BCube, a densely connected data center network. For such reasons, we develop an efficient incast tree building method by exploiting the topological feature of BCube in Section 3.2. The resulting time complexity is of $O(m{\times}\log^3 N)$ in Theorem 1.

Definition 3: For a shuffle transfer, the minimal aggregation subgraph problem is to find a connected subgraph in $G{=}(V, E)$ that spans all shuffle members with minimal cost for completing the shuffle transfer.

Note that the minimal aggregation tree of an incast transfer in BCube is NP-hard and a shuffle transfer is normalized as the combination of a set of incast transfers. Therefore, the minimal aggregation subgraph of a shuffle transfer in BCube is NP-hard. For this reason, We develop an efficient shuffle subgraph construction method by exploiting the topological feature of BCube in Section 3.3.

After deriving the shuffle subgraph for a given shuffle, each sender greedily delivers its data flow toward a receiver along the shuffle subgraph if its output for the receiver is ready. All packets will be cached when a data flow meets an aggregating server. An aggregating server can perform the aggregation operation once a new data flow arrives and brings an additional delay. Such a scheme amortizes the delay due to wait and simultaneously aggregate all incoming flows. The cache behavior of the fast data flow indeed brings a little additional delay of such a flow. However, the in-network aggregation can lower down the job completion time due to significantly reduce network traffic and improve the utilization of rare bandwidth resource, as shown in Fig.7.

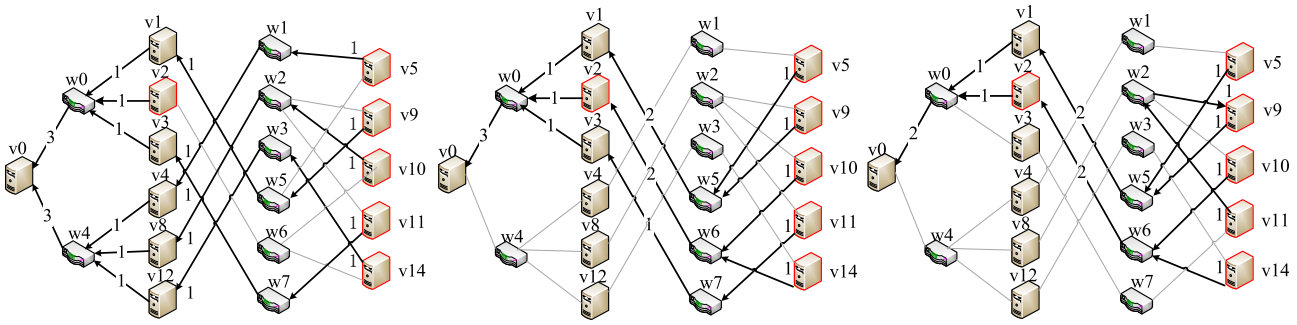
3.2 Incast aggregation tree building method

As aforementioned, building a minimal aggregation tree for any incast transfer in BCube is NP-hard. In this paper, we aim to design an approximate method by exploiting the topological feature of data center networks, e.g., BCube(n, k).

BCube(n, k) is a densely connected network structure since there are $k{+}1$ equal-cost disjoint unicast paths between any server pair if their labels differ in $k{+}1$ dimensions. For an incast transfer, each of its all senders independently delivers its data to the receiver along a unicast path that is randomly selected from $k{+}1$ ones. The combination of such unicast paths forms a ***unicast-based aggregation tree***, as shown in Fig.2(a). Such a method, however, has less chance of achieving the gain of the in-network aggregation.

For an incast transfer, we aim to build an efficient aggregation tree in a managed way instead of the random way, given the labels of all incast members and the data center topology. Consider an incast transfer consists of a receiver and m senders in BCube(n, k). Let d denote the maximum hamming distance between the receiver and each sender in the incast transfer, where $d{\leq}k{+}1$. Without loss of generality, we consider the general case that $d{=}k{+}1$ in this paper. An aggregation tree of the incast transfer can be expanded as a directed multistage graph with $d{+}1$ stages. The stage 0 only has the receiver. Each of all senders should appear at stage j if it is a j-hop neighbor of the receiver. Only such senders and the receiver, however, cannot definitely form a connected subgraph. The problem is then translated to identify a minimal set of servers for each stage and to identify switches between successive stages so as to constitute a minimal aggregation tree in BCube(n, k). The level and label of a switch between a pair of neighboring servers across two stages can be inferred from the labels of the two servers. We thus only focus on identify additional servers for each stage.

Identifying the minimal set of servers for each stage is an efficient approximate method for the minimal



(a) A unicast-based tree of cost 22 with 18 links. (b) An aggregation tree of cost 16 with 12 links. (c) An aggregation tree of cost 14 with 11 links.

Fig. 2. Different aggregation trees for an incast transfer with the sender set $\{v_2, v_5, v_9, v_{10}, v_{11}, v_{14}\}$ and the receiver v_0 in Bcube(4,1).

aggregation tree problem. For any stage, less number of servers incurs less number of flows towards the receiver since all incoming flows at each server will be aggregated as a single flow. Given the server set at stage j for $1 \leq j \leq d$, we can infer the required servers at stage $j-1$ by leveraging the topology features of BCube(n, k). Each server at stage j, however, has a one-hop neighbor at stage $j-1$ in each of the j dimensions in which the labels of the server and the receiver only differ. If each server at stage j randomly selects one of j one-hop neighbors at stage $j-1$, it just results in a unicast-based aggregation tree.

The insight of our method is to derive a common neighbor at stage $j-1$ for as many servers at stage j as possible. In such a way, the number of servers at stage $j-1$ can be significantly reduced and each of them can merge all its incoming flows as a single one for further forwarding. We identify the minimal set of servers for each stage from stage d to stage 1 along the processes as follows.

We start with stage $j=d$ where all servers are just those senders that are d-hops neighbors of the receiver in the incast transfer. After defining a routing symbol $e_j \in \{0, 1, ..., k\}$, we partition all such servers into groups such that all servers in each group are mutual one-hop neighbors in dimension e_j. That is, the labels of all servers in each group only differ in dimension e_j. In each group, all servers establish paths to a common neighboring server at stage $j-1$ that aggregates the flows from all servers in the group, called an *inter-stage aggregation*. All servers in the group and their common neighboring server at stage $j-1$ have the same label except the e_j dimension. Actually, the e_j dimension of the common neighboring server is just equal to the receiver's label in dimension e_j. Thus, the number of partitioned groups at stage j is just equal to the number of appended servers at stage $j-1$. The union of such appended servers and the possible senders at stage $j-1$ constitute the set of all servers at stage $j-1$. For example, all servers at stage 2 are partitioned into three groups, $\{v_5, v_9\}$, $\{v_{10}, v_{14}\}$, and $\{v_{11}\}$, using a routing symbol $e_2=1$, as shown in Fig.2(b). The neighboring servers at stage 1 of such groups are with labels v_1, v_2, and v_3, respectively.

The number of servers at stage $j-1$ still has an opportunity to be reduced due to the observations as follows. There may exist groups each of which only has one element at each stage. For example, all servers at the stage 2 are partitioned into three groups, $\{v_5, v_9\}$, $\{v_{10}, v_{14}\}$, and $\{v_{11}\}$ in Fig.2(b). The flow from the server $\{v_{11}\}$ to the receiver has no opportunity to perform the in-network aggregation.

To address such an issue, we propose an *intra-stage aggregation* scheme. After partitioning all servers at any stage j according to e_j, we focus on groups each of which has a single server and its neighboring server at stage $j-1$ is not a sender of the incast transfer. That is, the neighboring server at stage $j-1$ for each of such groups fails to perform the inter-stage aggregation. At stage j, the only server in such a group has no one-hop neighbors in dimension e_j, but may have one-hop neighbors in other dimensions, $\{0, 1, ..., k\} - \{e_j\}$. In such a case, the only server in such a group no longer delivers its data to its neighboring server at stage $j-1$ but to a one-hop neighbor in another dimension at stage j. The selected one-hop neighbor thus aggregates all incoming data flows and the data flow by itself (if any) at stage j, called an *intra-stage aggregation*.

Such an intra-stage aggregation can further reduce the cost of the aggregation tree since such a one-hop intra-stage forwarding increases the tree cost by two but saves the tree cost by at least four. The root cause is that for a sole server in a group at a stage j the nearest aggregating server along the original path to the receiver appears at stage $\leq j-2$. Note that the cost of an inter-stage or an intra-stage transmission is two due to the relay of an intermediate switch. For example, the server v_{11} has no neighbor in dimension $e_2=1$ at stage 2, however, has two neighbors, the servers v_9 and v_{10}, in dimension 0 at stage 2, as shown in Fig.2(b). If the server v_{11} uses the server v_9 or v_{10} as a relay for its data flow towards the receiver, prior aggregation tree in Fig.2(b) can be optimized as the new one in Fig.2(c). Thus, the cost of the aggregation tree is decreased by two while the number of active links is decremented by 1.

So far, the set of servers at stage $j-1$ can be achieved under a partition of all servers at stage j according to dimension e_j. The number of outgoing data flows from stage $j-1$ to its next stage $j-2$ is just equal to the cardinality of the server set at stage $j-1$. Inspired by such a fact, we apply the above method to other k settings of e_j and accordingly achieve other k server sets that can appear at the stage $j-1$. We then simply select the smallest server set from all $k+1$ ones. The setting of e_j is marked as the best routing symbol for stage j among all $k+1$ candidates. Thus, the data flows from stage j can achieve the largest gain of the in-network aggregation at the next stage $j-1$.

Similarly, we can further derive the minimal set of servers and the best choice of e_{j-1} at the next stage $j-2$, given the server set at $j-1$ stage, and so on. Here, the possible setting of e_{j-1} comes from $\{0, 1, ..., k\} - \{e_j\}$. Finally, server sets at all $k+2$ stages and the directed paths between successive stages constitute an aggregation tree. The behind routing symbol at each stage is simultaneously found. ***Such a method is called the IRS-based aggregation tree building method.***

Theorem 1: Given an incast transfer consisting of m senders in BCube(n, k), the complexity of the IRS-based aggregation tree building method is $O(m \times \log^3 N)$, where $N=n^{k+1}$ is the number of servers in a BCube(n, k).

Proof: Before achieving an aggregation tree, our method performs at most k stages, from stage $k+1$ to stage 2, given an incast transfer. The process to partition all servers at any stage j into groups according to e_j can be simplified as follows. For each server at stage j, we extract its label except the e_j dimension. The resultant label denotes a group that involves such a server. We then add the server into such a group. The computational overhead of such a partition is proportional to the number of servers at stage j. Thus, the resultant time complexity at each stage is $O(m)$ since the number of servers at each stage cannot exceed m. Additionally, the intra-stage aggregation after a partition of servers at each stage incurs additional $O(k \times m)$ computational overhead.

In the IRS-based building method, we conduct the same operations under all of $k+1$ settings of e_{k+1}. This produces $k+1$ server sets for stage k at the cost of $O((k+1)^2 \times m)$. In summary, the total computation cost of generating the set of servers at stage k is $O((k+1)^2 \times m)$. At stage k, our approach can identify the server set for stage $k-1$ from k candidates resulting from k settings of e_k since e_{k+1} has exclusively selected one from the set $\{0, 1, 2, ..., k\}$. The resultant computation cost is thus $O(k^2 \times m)$. In summary, the total computation cost of the IRS-based building method is $O(k^3 \times m)$ since it involves at most k stages. Thus, Theorem 1 proved. □

Our method for minimal aggregation tree in the BCube and the method for minimal Steiner tree in the hypercube proposed in [33] share similar process. Such two methods differ from those MST-based and improved approximate methods for general graphs in Section 2.3 and are in effect a $(1+\epsilon)$ approximation, as shown in [33]. The evaluation result indicates that our method outperforms the approximate algorithm for general graphs, whose approximate ratio is less than 2 [27].

Algorithm 1 Clustering all nodes in a graph $G'=(V', E')$

Require: A graph $G' = (V', E')$ with $|V'| = n$ vertices.

1: Let $groups$ denote an empty set;
2: **while** G' is not empty **do**
3: Calculate the degree of each vertex in V';
4: Find the vertex with the largest degree in G'. Such a vertex and its neighbor vertices form a group that is added into the set $groups$;
5: Remove all elements in the resultant group and their related edges from G'.

3.3 Shuffle aggregation subgraph building method

For a shuffle transfer in BCube(n, k), let $S=\{s_1, s_2, ..., s_m\}$ and $R=\{r_1, r_2, ..., r_n\}$ denote the sender set and receiver set, respectively. An intrinsic way for scheduling all flows in the shuffler transfer is to derive an aggregation tree from all senders to each receiver via the IRS-based building method. The combination of such aggregation trees produces an ***incast-based shuffle subgraph*** in $G=(V, E)$ that spans all senders and receivers. The shuffle subgraph consists of $m \times n$ paths along which all flows in the shuffle transfer can be successfully delivered. If we produce a unicast-based aggregation tree from all senders to each receiver, the combination of such trees results in a ***unicast-based shuffle subgraph***.

We find that the shuffle subgraph has an opportunity to be optimized due to the observations as follows. There exist $(k+1) \times (n-1)$ one-hop neighbors of any server in BCube(n, k). For any receiver $r \in R$, there may exist some receivers in R that are one-hop neighbors of the receiver r. Such a fact motivates us to think whether the aggregation tree rooted at the receiver r can be reused to carry flows for its one-hop neighbors in R. Fortunately, this issue can be addressed by a practical strategy. The flows from all of senders to a one-hop neighbor $r_1 \in R$ of r can be delivered to the receiver r along its aggregation tree and then be forwarded to the receiver r_1 in one hop. In such a way, a shuffle transfer significantly reuses some aggregation trees and thus utilizes less data center resources, including physical links, servers, and switches, compared to the incast-based method. We formalize such a basic idea as a NP-hard problem in Definition 4.

Definition 4: For a shuffle transfer in BCube (n, k), the minimal clustering problem of all receivers is how to partition all receivers into a minimal number of groups under two constraints. The intersection of any two groups is empty. Other receivers are one-hop neighbors of a receiver in each group.

Such a problem can be relaxed to find a minimal dominating set (MDS) in a graph $G'=(V', E')$, where V' denotes the set of all receivers of the shuffle transfer, and for any $u, v \in V'$, there is $(u, v) \in E'$ if u and v are one-hop

neighbor in BCube(n, k). In such a way, each element of MDS and its neighbors form a group. Any pair of such groups, however, may have common elements. This is the only difference between the minimal dominating set problem and the minimal clustering problem. Finding a minimum dominating set is NP-hard in general. Therefore, the minimal clustering problem of all receivers for a shuffle transfer is NP-hard. We thus propose an efficient algorithm to approximate the optimal solution, as shown in Algorithm 1.

The basic idea behind our algorithm is to calculate the degree of each vertex in the graph G' and find the vertex with the largest degree. Such a vertex (a head vertex) and its neighbors form the largest group. We then remove all vertices in the group and their related edges from G'. This brings impact on the degree of each remainder vertex in G'. Therefore, we repeat the above processes if the graph G' is not empty. In such a way, Algorithm 1 can partition all receivers of any shuffle transfer as a set of disjoint groups. Let α denote the number of such groups.

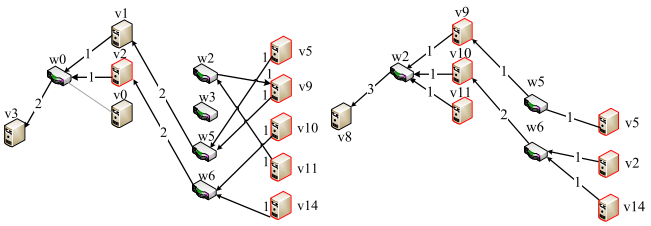
We define the cost of a shuffle transfer from m senders to a group of receivers R_i=$\{r_1, r_2, ..., r_g\}$ as C_i, where $\sum_{i\geq 1}^{\alpha} |R_i|$=$n$. Without loss of generality, we assume that the group head is r_1 in R_i. Let c_j denote the cost of an aggregation tree, produced by our IRS-based building method, from all m senders to a receiver r_j in the group R_i where $1 \leq j \leq |R_i|$. The cost of such a shuffle transfer depends on the entry point selection for each receiver group R_i.

1) $C_i^1 = |R_i| \times c_1 + 2(|R_i| - 1)$ if the group head r_1 is selected as the entry point. In such a case, the flows from all of senders to all of the group members are firstly delivered to r_1 along its aggregation tree and then forwarded to each of other group members in one hop. Such a one-hop forwarding operation increases the cost by two due to the relay of a switch between any two servers.
2) $C_i^j = |R_i| \times c_j + 4(|R_i| - \beta - 1) + 2 \times \beta$ if a receiver r_j is selected as the entry point. In such a case, the flows from all of senders to all of the group members are firstly delivered to r_j along its aggregation tree. The flow towards each of β one-hop neighbors of r_j in R_i, including the head r_1, reaches its destination in one-hop from r_j at the cost of 2. The receiver r_j finally forwards flows towards other $|R_i| - \beta - 1$ receivers, each of which is two-hops from r_j due to the relay of the group head r_1 at the cost of 4.

Given C_i^j for $1 \leq j \leq |R_i|$, the cost of a shuffle transfer from m senders to a receiver group R_i=$\{r_1, r_2, ..., r_g\}$ is given by

$$C_i = \min\{C_i^1, C_i^2, ..., C_i^{|R_i|}\}. \quad (1)$$

As a result, the best entry point for the receiver group R_i can be found simultaneously. It is not necessary that the best entry point for the receiver group R_i is the group head r_1. Accordingly, a shuffle subgraph from m senders to all receivers in R_i is established. ***Such a method is called the SRS-based shuffle subgraph building method.***



(a) A tree of cost 14 with 11 active links. (b) A tree of cost 12 with 9 active links.

Fig. 3. Examples of aggregation trees for two incast transfers, with the same sender set $\{v_2, v_5, v_9, v_{10}, v_{11}, v_{14}\}$ and the receiver v_3 or v_8.

We use an example to reveal the benefit of our SRS-based building method. Consider a shuffle transfer from all of senders $\{v_2, v_5, v_9, v_{10}, v_{11}, v_{14}\}$ to a receiver group $\{v_0, v_3, v_8\}$ with the group head v_0. Fig.2(c), Fig.3(a), and Fig.3(b) illustrate the IRS-based aggregation trees rooted at v_0, v_3, and v_8, respectively. If one selects the group head v_0 as the entry point for the receiver group, the resultant shuffle subgraph is of cost 46. When the entry point is v_3 or v_8, the resultant shuffle subgraph is of cost 48 or 42. Thus, the best entry point for the receiver group $\{v_0, v_3, v_8\}$ should be v_8 not the group head v_0.

In such a way, the best entry point for each of the α receiver groups and the associated shuffle subgraph can be produced. The combination of such α shuffle subgraphs generates the final shuffle subgraph from m senders to n receivers. Therefore, the cost of the resultant shuffle subgraph, denoted as C, is given by $C = \sum_{i\geq 1}^{\alpha} C_i$.

4 Scalable Forwarding Schemes for Performing In-network Aggregation

We start with a general forwarding scheme to perform the in-network aggregation based on a shuffle subgraph. Accordingly, we propose two more scalable and practical forwarding schemes based on in-switch Bloom filters and in-packet Bloom filters, respectively.

4.1 General forwarding scheme

As MapReduce-like systems grow in popularity, there is a strong need to share data centers among users that submit MapReduce jobs. Let β be the number of such jobs running in a data center concurrently. For a job, there exists a JobTracker that is aware of the locations of m map tasks and n reduce tasks of a related shuffle transfer. The JobTracker, as a shuffle manager, then calculates an efficient shuffle subgraph using our SRS-based method. The shuffle subgraph contains n aggregation trees since such a shuffle transfer can be normalized as n incast transfers. The concatenation of the job *id* and the receiver *id* is defined as the identifier of an incast transfer and its aggregation tree. In such a way, we can differentiate all incast transfers in a job or across jobs in a data center.

Conceptually, given an aggregation tree all involved servers and switches can conduct the in-network aggregation for the incast transfer via the following operations. Each sender (a leaf vertex) greedily delivers a data flow along the tree if its output for the receiver has generated and it is not an aggregating server. If a data flow meets an aggregating server, all packets will be cached. Upon the data from all its child servers and itself (if any) have been received, an aggregating server performs the in-network aggregation on multiply flows as follows. It groups all key-value pairs in such flows according to their keys and applies the aggregate function to each group. Such a process finally replaces all participant flows with a new one that is continuously forwarded along the aggregation tree.

To put such a design into practice, we need to make some efforts as follows. For each of all incast transfers in a shuffle transfer, the shuffle manager makes all servers and switches in an aggregation tree are aware of that they join the incast transfer. Each device in the aggregation tree thus knows its parent device and adds the incast transfer identifier into the routing entry for the interface connecting to its parent device. Note that the routing entry for each interface is a list of incast transfer identifiers. When a switch or server receives a flow with an incast transfer identifier, all routing entries on its interfaces are checked to determine where to forward or not.

Although such a method ensures that all flows in a shuffle transfer can be successfully routed to destinations, it is insufficient to achieve the inherent gain of in-network aggregation. Actually, each flow of an incast transfer is just forwarded even it reaches an aggregating server in the related aggregation tree. The root cause is that a server cannot infer from its routing entries that it is an aggregating server and no matter knows its child servers in the aggregation tree. To tackle such an issue, we let each server in an aggregation tree maintain a pair of $id{-}value$ that records the incast transfer identifier and the number of its child servers. Note that a switch is not an aggregation server and thus just forwards all received flows.

In such a way, when a server receives a flow with an incast transfer identifier, all $id{-}value$ pairs are checked to determine whether to cache the flow for the future aggregation. That is, a flow needs to be cached if the related *value* exceeds one. If yes and the flows from all its child servers and itself (if any) have been received, the server aggregates all cached flows for the same incast transfer as a new flow. All routing entries on its interfaces are then checked to determine which interface the new flow should be sent out. If the response is null, the server is the destination of the new flow. If a flow reaches a non-aggregating server, it just checks all routing entries to identify an interface the flow should be forwarded.

4.2 In-switch bloom filter based forwarding scheme

One trend of modern data center designs is to utilize a large number of low-end switches for interconnection for economic and scalability considerations. The space of fast memory in such kind of switches is relatively narrow and thus is quite challenging to support massive incast transfers by keeping the incast routing entries. To address such a challenge, we propose two types of Bloom filter based incast forwarding schemes, namely, in-switch Bloom filter and in-packet Bloom filter.

A Bloom filter consists of a vector of m bits, initially all set to 0. It encodes each item in a set X by mapping it to h random bits in the bit vector uniformly via h hash functions. To judge whether an element x belongs to X with n_0 items, one just needs to check whether all hashed bits of x in the Bloom filter are set to 1. If not, x is definitely not a member of X. Otherwise, we infer that x is a member of X. A Bloom filter may yield a false positive due to hash collisions, for which it suggests that an element x is in X even though it is not. The false positive probability is $f_p \approx (1{-}e^{-h\times n_0/m})^h$. From [34], [35], f_p is minimized as $0.6185^{m/n_0}$ when $h{=}(m/n_0)\ln 2$.

For the in-switch Bloom filter, each interface on a server or a switch maintains a Bloom filter that encodes the identifiers of all incast transfers on the interface. When the switch or server receives a flow with an incast transfer identifier, all the Bloom filters on its interfaces are checked to determine which interface the flow should be sent out. If the response is null at a server, it is the destination of the packet. Consider that a server is a potential aggregating server in each incast transfer. When a server receives a flow with an incast identifier, it first checks all $id{-}value$ pairs to determine whether to cache the flow for the future aggregation. Only if the related $value$ is 1 or a new flow is generated by aggregating cached flows, the server checks all Bloom filters on its interfaces for identifying the correct forwarding direction.

The introduction of Bloom filter can considerably compress the forwarding table at each server and switch. Moreover, checking a Bloom filter on each interface only incurs a constant delay, only h hash queries, irrespective of the number of items represented by the Bloom filter. In contrast, checking the routing entry for each interface usually incurs $O(\log\gamma)$ delay in the general forwarding scheme, where γ denotes the number of incast transfers on the interface. Thus, Bloom filters can significantly reduce the delay of making a forwarding decision on each server and switch. Such two benefits help realize the scalable incast and shuffle forwarding in data center networks.

4.3 In-packet bloom filter based forwarding scheme

Both the general and the in-switch Bloom filter based forwarding schemes suffer non-trivial management overhead due to the dynamic behaviors of shuffle transfers. More precisely, a new scheduled Mapreduce job will

bring a shuffle transfer and an associated shuffle subgraph. All servers and switches in the shuffle subgraph thus should update its routing entry or Bloom filter on related interface. Similarly, all servers and switches in a shuffle subgraph have to update their routing table once the related Mapreduce job is completed. To avoid such constraints, we propose the in-packet Bloom filter based forwarding scheme.

Given a shuffle transfer, we first calculate a shuffle subgraph by invoking our SRS-based building method. For each flow in the shuffle transfer, the basic idea of our new method is to encode the flow path information into a Bloom filter field in the header of each packet of the flow. For example, a flow path from a sender v_{14} to the receiver v_0 in Fig.2(c) is a sequence of links, denoted as $v_{14}\rightarrow w_6$, $w_6\rightarrow v_2$, $v_2\rightarrow w_0$, and $w_0\rightarrow v_0$. Such a set of links is then encoded as a Bloom filter via the standard input operation [34], [36]. Such a method eliminates the necessity of routing entries or Bloom filters on the interfaces of each server and switch. We use the link-based Bloom filters to encode all directed physical links in the flow path. To differentiate packets of different incast transfers, all packets of a flow is associated with the identifier of an incast transfer the flow joins.

When a switch receives a packet with a Bloom filter field, its entire links are checked to determine along which link the packet should be forwarded. For a server, it should first check all $id{-}value$ pairs to determine whether to cache the flow for the future aggregation. That is, the packet should be forwarded directly if the related $value$ is 1. Once a server has received all $value$ flows of an incast transfer, e.g., server v_{10} collects $value$=3 flows in Fig.3(b), such flows are aggregated as a new flow. To forward such packets, the server checks the Bloom filter with its entire links as inputs to find a forwarding direction.

Generally, such a forwarding scheme may incur false positive forwarding due to the false positive feature of Bloom filters. When a server in a flow path meets a false positive, it will forward flow packets towards another one-hop neighbors besides its right upstream server. The flow packets propagated to another server due to a false positive will be terminated with high probability since the in-packet Bloom filter just encodes the right flow path. Additionally, the number of such mis-forwarding can be less than one during the entire propagation process of a packet if the parameters of a Bloom filter satisfy certain requirements.

For a shuffle transfer in BCube(n,k), a flow path is at most $2(k+1)$ of length, i.e., the diameter of BCube(n,k). Thus, a Bloom filter used by each flow packet encodes $n_0 = 2(k+1)$ directed links. Let us consider a forwarding of a flow packet from stage i to stage $i{-}1$ for $1{\le}i{\le}k$ along the shuffle subgraph. The server at stage i is an i-hop neighbor of the flow destination since their labels differ in i dimensions. When the server at stage i needs to forward a packet of the flow, it just needs to check i links towards its i one-hop neighbors that are also $(i{-}1)$-hops neighbors of the flow destination. The flow packet, however, will be forwarded to far away from the destination through other $k{+}1{-}i$ links that should be ignored directly when the server makes a forwarding decision. Among the i links on the server at stage i, $i{-}1$ links may cause false positive forwarding at a given probability when checking the Bloom filter field of a flow packet. The packet is finally sent to an intermediate switch along the right link. Although the switch has n links, only one of them connects with the server at stage $i{-}1$ in the path encoded by a Bloom filter field in the packet. The servers along with other $n{-}2$ links of the switch are still i-hops neighbors of the destination. Such $n{-}2$ links are thus ignored when the switch makes a forwarding decision. Consequently, no false positive forwarding appears at a switch.

In summary, given a shuffle subgraph a packet will meet at most k servers before reaching the destination and may occur false positive forwarding on each of at most $\sum_{i=1}^{k-1} i$ server links at probability f_p. Thus, the number of resultant false positive forwarding due to deliver a packet to its destination is given by

$$f_p{\times}\sum_{i=1}^{k-1} i = 0.6185^{\frac{m}{2(k+1)}} \times k \times (k{-}1)/2, \qquad (2)$$

here $n_0{=}2(k{+}1)$. If we impose a constriant that the value of Formula (2) is less than 1, then

$$m \ge 2(k{+}1) \times \log_{0.6185} \frac{2}{k \times (k{-}1)}. \qquad (3)$$

We can derive the requried size of Bloom filter field of each packet from Formula (3). We find that the value of k is usually not large for a large-scale data center, as shown in Section 5.5. Thus, the Bloom filter field of each packet incurs additional traffic overhead.

5 Performance Evaluation

We start with our prototype implementation. We then evaluate our methods and compare with other works under different data center sizes, shuffle transfer sizes, and aggregation ratios. We also measure the traffic overhead of our in-packet Bloom filter based forwarding scheme.

5.1 The prototype implementation

Our testbed consists of 61 virtual machines (VM) hosted by 6 servers connected with an Ethernet. Each server equips with two 8-core processors, 24GB memory and a 1TB disk. Five of the servers run 10 virtual machines as the Hadoop virtual slave nodes, and the other one runs 10 virtual slave nodes and 1 master node that acts as the shuffle manager. Each virtual slave node supports two map tasks and two reduce tasks. We extend the Hadoop to embrace the in-network aggregation on any shuffle transfer. We launch the built-in wordcount job with the combiner, where the job has 60 map tasks and 60 reduce tasks, respectively. That is, such a job employs one map task and reduce task from each of 60 VMs. We

associate each map task ten input files each with 64M. A shuffle transfer from all 60 senders to 60 receivers is thus achieved. The average amount of data from a sender (map task) to the receiver (reduce task) is about 1M after performing the combiner at each sender. Each receiver performs the count function. Our approaches exhibit more benefits for other aggregation functions, e.g., the sum, maximum, minimum, top-k and KNN functions.

To deploy the wordcount job in a BCube$(6, k)$ data center for $2 \leq k \leq 8$, all of senders and receivers (the map and reduce tasks) of the shuffle transfer are randomly allocated with BCube labels. We then generate the SRS-based, incast-based, and unicast-based shuffle subgraphs for the shuffle transfer via corresponding methods. Our testbed is used to emulate a partial BCube(6,k) on which the resultant shuffle subgraphs can be conceptedly deployed. To this end, given a shuffle subgraph we omit all of switches and map all of intermediate server vertices to the 60 slave VMs in our testbed. That is, we use a software agent to emulate an intermediate server so as to receive, cache, aggregate, and forward packets. Thus, we achieve an overlay implementation of shuffle transfer. Each path between two neighboring servers in the shuffle subgraph is mapped to a virtual link between VMs or a physical link across servers in our testbed.

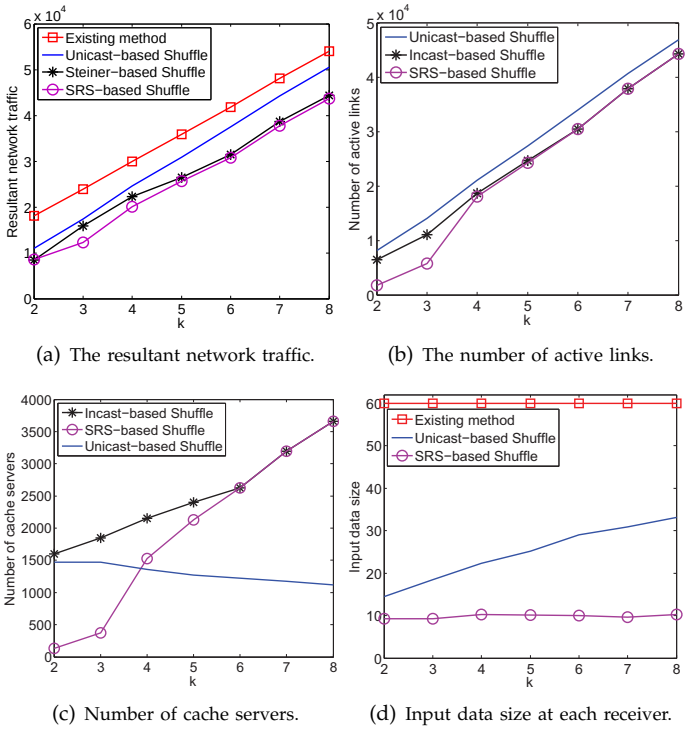
We compare our SRS-based shuffle subgraph against the incast-based shuffle subgraph, the Steiner-based one, the unicast-based one, and the existing method in terms of four metrics. They are the resultant network traffic, the number of active links, the number of cache servers, and the input data size at each receiver. The network traffic denotes the sum of network traffic over all edges in the shuffle subgraph. Actually, the existing method is just the unicast-based method but does not perform the in-network aggregation.

The Steiner-based shuffle subgraph is similar to the incast-based one but each aggregation tree results from the Steiner-tree algorithm. The Steiner-tree algorithm we choose is the one described in [27], whose benefit is the computation speed. The algorithm works as follows. At first, a virtual complete graph is generated upon the incast members. Then, a minimum spanning tree is calculated on the virtual complete graph. Finally, the virtual link in the virtual complete graph is replaced by the shortest path between any two incast members in the original topology, with unnecessary links deleted.

5.2 Impact of the data center size

Consider a shuffle transfer with 60 senders and 60 receivers that are randomly selected in a data center with BCube(6,k) as its network structure. We conduct experiments and collect the performance metrics after completing such a shuffle transfer along different shuffle subgraphs. Fig.4 shows the changing trends of the performance metrics on average, among 100 rounds of experiments, under different methods and settings of k.

Fig.4(a) indicates that our SRS-based, the Steiner-based, and the unicast-based methods considerably save



(a) The resultant network traffic. (b) The number of active links.

(c) Number of cache servers. (d) Input data size at each receiver.

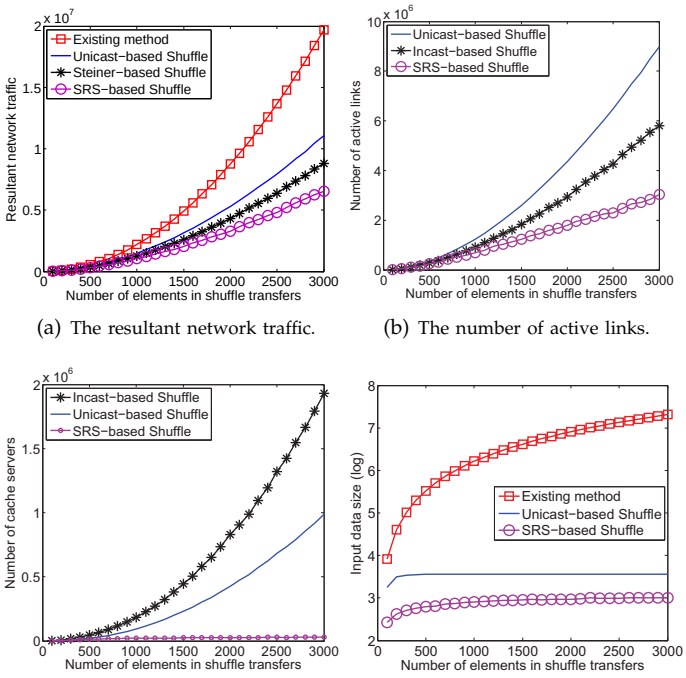
Fig. 4. The changing trends of four performance metrics for shuffle transfers with 60 senders and 60 receivers in BCube(6,k), where k ranges from 2 to 8.

the resultant network traffic compared to the existing method, irrespective of the data center size, i.e., 6^{k+1}. Table 1 shows that the SRS-based method causes less network traffic than the incast-based method. More precisely, the SRS-based, incast-based, Steiner-based, and unicast-based methods save the network traffic by 32.87%, 32.69%, 28.76%, and 17.64% on average compared to the existing method. Such results demonstrate the large gain of in-network aggregation in BCube(6,k) even for a small shuffle transfer. Additionally, our SRS-based and incast-based methods considerably outperform the unicast-based one due to the following reason.

Our two methods schedule all of involved flows at the level of a single or a group of incast transfers while the unicast-based method does it at the level of each individual flow. Actually, the number of aggregating servers in the incast-based method increases while that in the unicast-based method decreases along with the increase of k, as shown in Fig.4(c). Thus, the incast-based method always has more opportunity to performing the in-network aggregation than the unicast-driven method. The SRS-based method improves the incast-based one by reusing as many incast aggregation trees as possible and thus utilizes less number of aggregating servers and active links, as shown in Fig.4(c) and Fig.4(b). Note that

TABLE 1
The network traffic under the same settings of Figure 4.

Shuffle	k=2	k=3	k=4	k=5	k=6	k=7
SRS-based	8544	12250	20102	25608	30806	37754
Incast-based	8730	12298	20098	25606	30806	37754



(a) The resultant network traffic. (b) The number of active links.

(c) Number of cache servers. (d) Input data size at each receiver.

Fig. 5. The changing trends of two performance metrics along with the increase of elements in a shuffle transfer in BCube(8,5) with 262144 servers, where the number of senders equals to that of the receivers.

the SRS-based method becomes the incast-based method when $k \geq 6$. The reason is that the data center is too large such that a small number of receivers, e.g., randomly selected 60 receivers, do not contain any pair of one-hop neighbors. Consequently, each partition of all receivers just covers one receiver. Besides such benefits, our SRS-based method significantly reduces the size of input data at each receiver, as shown in Fig.4(d), and thus reduces the delay during the reduce phase of a job.

In summary, our SRS-based and incast-based methods support a small size shuffle transfer well with less data center resources and network traffic compared to the unicast-based and existing methods, irrespective of the size of the data center. Moreover, our SRS-based and incast-based methods can improve the performance of the Steiner-based method at some extent. The root cause is that our methods can efficiently exploit the topological feature of BCube and utilize more aggregating servers than the Steiner-based method although they have similar number of active links.

5.3 Impact of the shuffle transfer size

Consider that a MapReduce-like job may sometimes involve several hundreds even thousands map and reduce tasks. We will evaluate our methods over shuffle transfers with varying number of members. It is difficult for our testbed to execute a large-scale wordcount job due to its limited resources. We thus conduct extensively simulations to demonstrate the scaling property of our methods. To achieve a shuffle transfer with m senders and n receivers, $m=n=50\times i$ for $1 \leq i \leq 30$, we created a synthetic wordcount job. The wordcount job provides the input data of 64M for each sender, generated by the built-in RandomTextWriter of Hadoop. The data transmission from each sender to the receiver is controlled to be 1M on average. Fig.5 shows the changing trends of the performance metrics along with the increase of the number of elements, i.e., $m+n$, in shuffle transfers in BCube(8,5). The number of servers in BCube(8,5) is 262144 that is large enough for a data center.

Our results indicate that the SRS-based, incast-based, Steiner-based, and unicast-based shuffle methods significantly save the network traffic by 55.33%, 55.29%, 44.89%, and 34.46% on average compared to the existing method as $m=n$ ranges from 50 to 1500. Such benefits of the three methods are more notable as the increase of $m=n$ and thus demonstrate the gain of the in-network aggregation in large shuffle transfers. Moreover, our SRS-based method always utilizes less links (hence less servers and network devices) compared to the unicast-based and incast-basded methods, as shown in Fig.5(b).

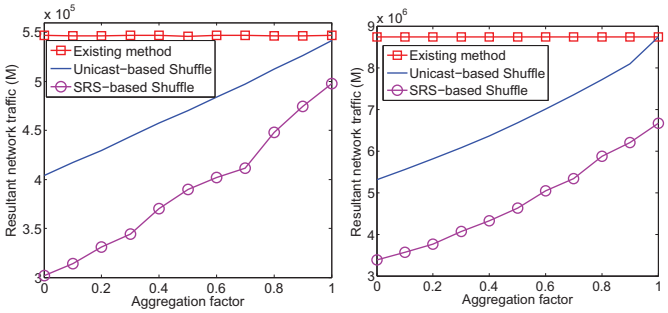
On the other hand, the incast-based method always exploits more aggregating servers than the unicast-based method as the shuffle size increases, as shown in Fig.5(c). Thus, the incast-based method always has more opportunity to performing the in-network aggregation than the unicast-driven method. This is the fundamental reason why our incast-based method causes less network traffic than the unicast-based method. Our SRS-based method improves the incast-based one by reusing as many incast aggregation trees as possible. Consequently, the SRS-based method achieves the largest gain of the in-network aggregation with the least number of aggregating servers and active links compared to the incast-based and unicast-based methods, as shown in Fig.5(b) and Fig.5(c). Besides the above benefits, our SRS-based method significantly reduces the size of input data at each receiver, as shown in Fig.5(d), and thus reduces the delay during the reduce phase of a job.

In summary, our SRS-based and incast-based approaches can support a shuffle transfer of any size well at the cost of less data center resources and network traffic compared to the unicast-based and existing methods.

5.4 Impact of the aggregation ratio

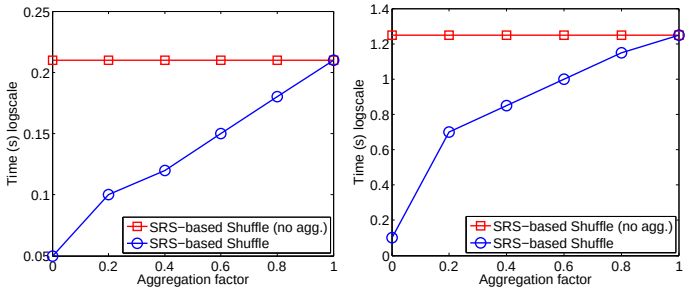
Recall that we make an assumption for easing our problem analysis in Section 3.1 and large-scale simulations. Data flows each of which consists of key-value pairs can be aggregated as a new one whose size is the largest size among such flows. That is, the set of keys in each involved data flow is the subset of that in the largest data flow. We further evaluate our SRS-based approach under a more general shuffle transfer.

Given s data flows towards to the same receiver in a general shuffle transfer, let f_i denote the size of the i^{th} data flow for $1 \leq i \leq s$. Let δ denote the aggregation ratio among any number of data flows, where $0 \leq \delta \leq 1$. After



(a) A shuffle transfer, m=n=250. (b) A shuffle transfer, m=n=1000.

Fig. 6. The changing trends of the network traffic along with the increase of aggregation factor under different settings of shuffle transfers in BCube(8,5).



(a) A shuffle transfer, m=n=250. (b) A shuffle transfer, m=n=1000.

Fig. 7. The changing trends of the delay along with the increase of aggregation factor under different settings of shuffle transfers in BCube(8,5).

aggregating such s flows, the size of the new data flow is given by

$$\max\{f_1, f_2, ..., f_s\} + \delta \times \big(\sum_{i=1}^{s} f_i - \max\{f_1, f_2, ..., f_s\}\big).$$

Our analysis in Section 3.1 and the above large-scale simulations fall into a special scenario, i.e., δ=0, where the in-network aggregation on flows achieves the largest gain. On the contrary, δ=1 is another special scenario where any two of the s data flows do not share any key. In such a case, the in-network aggregation on the s data flows does not bring any gain; hence, the unicast-driven method is equivalent to the existing method. Here, we evaluate our SRS-based approach in a more general scenario where $0 \leq \delta \leq 1$.

We measure the resultant network traffic of the SRS-based, unicast-based, and existing methods under two representative shuffle transfers in BCube(8,5) as δ ranges from 0 to 1. Fig.6 indicates that our SRS-based method always incurs much less network traffic than other two methods. Such results demonstrate the gain of the in-network aggregation for more general shuffle transfers. Assume that the value of the random variable δ follows a uniform distribution (if any). In such a case, our SRS-based method saves the resultant network traffic by 28.78% in Fig.6(a) and 45.05% in Fig.6(b) compared to the existing method, respectively. Thus, our SRS-based method outperforms other two methods in more general scenario, irrespective of the size of shuffle transfer.

We further compare of our SRS-based shuffle and SRS-based shuffle (no agg.) methods in terms of the completion time of map-reduce jobs on average. As expected, when the aggregation ration approaches to 1 the performance of such two methods is the same, because there is no opportunity to aggregate packets on-path. As aggregation ration decreases, our SRS-based shuffle method always achieves the lower shuffle and reduces time due to reduce the number of packets forward and increase the available bandwidth. This is clearly seen in Fig.7(a) and Fig.7(b).

TABLE 2
The minimum size of Bloom filter field in each packet.

	k=2	k=3	k=4	k=5	k=6	k=7
Bits (m)	<19	19	38	58	79	102
BCube(6,k)	216	1296	7776	46656	279936	1679616

5.5 The size of bloom filter in each packet

Among the three forwarding schemes for performing the in-network aggregation, we prefer the in-packet Bloom filter based forwarding scheme. The only overhead of such a scheme is the traffic overhead resulting from not only the false positive forwarding but also the Bloom filter field in each packet. We reveal that, for a packet of any shuffle transfer, no false positive forwarding occurs during the entire forwarding process along with a SRS-based shuffle subgraph only if Formula (3) holds.

For data centers with network structures BCube$(6, k)$s, Table 2 shows the minimum size of Bloom filter field in each packet. The traffic overhead due to the Bloom filter field in each packet increases along with the increase of k. The overhead, however, is less than 10 bytes in data centers, with no more than 279936 servers, that is large enough for a production data center.

6 Conclusion

In many commonly used workloads, shuffle transfers contribute most of the network traffic and make the network bandwidth become a bottleneck in data centers. To efficiently use the available network bandwidth, we propose to aggregate correlated data flows during their transmission process. To this end, we model the minimal incast aggregation tree and shuffle aggregation subgraph problems that are NP-hard. We propose two approximate methods, IRS-based and SRS-based methods, for building an efficient incast aggregation tree and shuffle aggregation subgraph. We then present scalable forwarding schemes using Bloom filters to implement in-network aggregation. We evaluate our method and related work using a prototype and large-scale simulations. The results show that our method can significantly reduce the network traffic and save the data center resources compared to others.

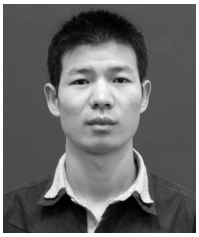
ACKNOWLEDGMENTS

The authors would like to thank anonymous reviewers for their constructive comments. The work is partially supported by the NSFC under Grant No. 61170284, the National Basic Research Program (973 program) under Grant No. 2014CB347800, and the China Postdoctoral Science Foundation under Grant No. 2013M542370.

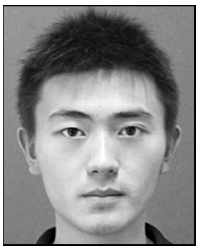
REFERENCES

[1] T. Condie, N. Conway, P. Alvaro, and J. M. Hellerstein, "Mapreduce online," in *Proc. 7th USENIX NSDI*, CA, USA, 2010, pp. 313–328.
[2] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey, "Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language," in *Proc. OSDI*, San Diego, California, USA, 2008, pp. 1–14.
[3] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand, "CIEL: A universal execution engine for distributed data-flow computing," in *Proc. 8th USENIX NSDI*, Boston, MA, USA, 2011, pp. 227–236.
[4] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proc. ACM SIGMOD*, Indiana, USA, 2010, pp. 135–146.
[5] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu, "Dcell: A scalable and fault-tolerant network structure for data centers," in *Proc. ACM SIGCOMM*, Seattle, Washington, USA, 2008.
[6] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, "Bcube: A high performance, server-centric network architecture for modular data centers," in *Proc. ACM SIGCOMM*, Barcelona, Spain, 2009.
[7] M. A. Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. ACM SIGCOMM*, Seattle, Washington, USA, 2008.
[8] A. Greenberg, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, and P. P. and, "Vl2: A scalable and flexible data center network," in *Proc. ACM SIGCOMM*, Barcelona, Spain, 2009.
[9] D. Guo, T. Chen, D. Li, Y. Liu, X. Liu, and G. Chen, "Bcn: Expansible network structures for data centers using hierarchical compound graphs," in *Proc. 30th IEEE INFOCOM*, Shanghai, China, 2011, pp. 61–65.
[10] D. Guo, T. Chen, D. Li, M. Li, Y. Liu, and G. Chen, "Expansible and cost-effective network structures for data centers using dual-port servers," *IEEE Transactions on Computers*, vol. 62, no. 7, pp. 1303–1317, 2013.
[11] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra," in *Proc. ACM SIGCOMM*, Toronto, ON, Canada, 2011, pp. 98–109.
[12] Y. Chen, A. Ganapathi, R. Griffith, and R. H. Katz, "The case for evaluating mapreduce performance using workload suites," in *Proc. 19th IEEE/ACM MASCOTS*, Singapore, 2011, pp. 390–399.
[13] R. Mysore, A. Pamboris, and N. Farrington, "Portland: A scalable fault-tolerant layer 2 data center network fabric," in *Proc. ACM SIGCOMM*, 2009.
[14] D. Abts, M. R. Marty, P. M. Wells, P. Klausler, and H. Liu, "Energy proportional datacenter networks," in *Proc. 37th ACM ISCA*, Saint-Malo, France, 2010, pp. 338–347.
[15] J. H. Ahn, N. L. Binkert, A. Davis, M. McLaren, and R. S. Schreiber, "Hyperx: topology, routing, and packaging of efficient large-scale networks," in *Proc. ACM/IEEE Conference on High Performance Computing(SC)*, Portland, Oregon, USA, 2009.
[16] L. N. Bhuyan and D. P. Agrawal, "Generalized hypercube and hyperbus structures for a computer network," *IEEE Transactions on Computers*, vol. 33, no. 4, pp. 323–333, 1984.
[17] S. Alan, K. Srikanth, and S. E. Gn, "Sidecar: building programmable datacenter networks without programmable switches," in *HotNets*, Monterey, CA, USA, 2010.
[18] P. Costa, "Bridging the gap between applications and networks in data centers," *Operating Systems Review*, vol. 47, no. 1, pp. 3–8, 2013.
[19] H. Abu-Libdeh, P. Costa, A. Rowstron, G. OShea, and A. Donnelly, "Symbiotic routing in future data centers," in *Proc. ACM SIGCOMM*, New Delhi, India, 2010.
[20] J.-Y. Shin, B. Wong, and E. G. Sirer, "Small-world datacenters," in *Proc. ACM SoCC*, Cascais, Portugal, 2011, p. 2.
[21] L. Gyarmati and T. Trinh, "Scafida: A scale-free network inspired data center architecture," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 5, pp. 5–12, Oct. 2010.
[22] G. Lu, C. Guo, Y. Li, and Z. Zhou, "Serverswitch: A programmable and high performance platform for data center networks," in *Proc. 8th NSDI*, Boston, MA, USA, 2011, pp. 15–28.
[23] J. Cao, C. Guo, G. Lu, Y. Xiong, Y. Zheng, Y. Zhang, Y. Zhu, and C. Chen, "Datacast: a scalable and efficient reliable group data delivery service for data centers," in *Proc. ACM CoNEXT*, Nice, France, Dec. 2012, pp. 37–48.
[24] L. Yu and J. Li, "Grouping-based resilient statistical en-route filtering for sensor networks," in *INFOCOM*, Rio de Janeiro, Brazil, 2009, pp. 1782–1790.
[25] L. A. Villas, A. Boukerche, H. S. Ramos, H. A. B. F. de Oliveira, R. B. de Araujo, and A. A. F. Loureiro, "Drina: A lightweight and reliable routing approach for in-network aggregation in wireless sensor networks," *IEEE Transactions on Computers*, vol. 62, no. 4, pp. 676–689, 2013.
[26] P. Costa, A. Donnelly, and A. R. G. OShea, "Camdoop: Exploiting in-network aggregation for big data applications," in *Proc. 9th NSDI*, San Jose, CA, Apr. 2012.
[27] L. Kou, G. Markowsky, and L. Berman, "A fast algorithm for steiner trees," *Acta Informatica (Historical Archive)*, vol. 15, no. 2, pp. 141–145, Jun. 1981.
[28] G. Robins and A. Zelikovsky, "Tighter bounds for graph steiner tree approximation," *SIAM J. Discrete Math*, vol. 19, no. 1, pp. 122–134, 2005.
[29] B. Bloom, "The steiner tree problem with edge lengths 1 and 2," *Information Processing Letters*, vol. 32, pp. 171–176, 1989.
[30] A. Z. Zelikovsky, "The 11/6-approximation algorithm for the steiner problem on networks," *Algorithmica*, vol. 9, no. 5, pp. 463–470, 1993.
[31] P. Berman and V. Ramaiyer, "Improved approximation algorithms for the steiner tree problem," *Journal of Algorithms*, vol. 17, no. 3, pp. 381–408, 1994.
[32] A. Z. Zelikovsky, "Better approximation bounds for the network and euclidean steiner tree problems," *SIAM Journal on Computing*, vol. 26, no. 3, pp. 857–869, 1997.
[33] T. Jiang and Z. M. D. Pritikin, "Near optimal bounds for steiner tree in the hypercube," *SIAM Journal on Discrete Mathematics*, vol. 40, no. 5, pp. 1340–1360, 2011.
[34] D. Guo, J. Wu, H. Chen, Y. Yuan, and X. Luo, "The dynamic bloom filters," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 1, pp. 120–133, 2010.
[35] D. Guo, Y. Liu, X. Li, and P. Yang, "False negative problem of counting bloom filter," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 5, pp. 651–664, 2010.
[36] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485–509, 2005.

Deke Guo received the B.S. degree in industry engineering from Beijing University of Aeronautic and Astronautic, Beijing, China, in 2001, and the Ph.D. degree in management science and engineering from National University of Defense Technology, Changsha, China, in 2008. He is an Associate Professor with the College of Information System and Management, National University of Defense Technology, Changsha, China. His research interests include distributed systems, software-defined networking, data center networking, wireless and mobile systems, and interconnection networks. He is a member of the ACM and the IEEE.

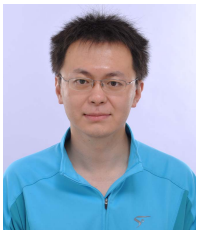


Junjie Xie received the B.S. degree in computer science and technology from Beijing Institute of Technology, Beijing, China, in 2013. He is currently working toward the M.S. degree in College of Information System and Management, National University of Defense Technology, Changsha, China. His research interests include distributed systems, data centers, software defined networks and interconnection networks.



Xiaolei Zhou received the B.S. degree in information management from Nanjing University, Nanjing, China, in 2009, and the M.S. degree in military science from National University of Defense Technology, China, in 2011. He is currently working toward the Ph.D. degree in College of Information System and Management, National University of Defense Technology, China. His current research interests include wireless sensor networks, Internet of things, and data center networking. He is a student member of the IEEE.

Xiaomin Zhu received the B.S. and M.S. degrees in computer science from Liaoning Technical University, Liaoning, China, in 2001 and 2004, respectively, and the Ph.D. degree in computer science from Fudan University, Shanghai, China, in 2009. In the same year, he won the Shanghai Excellent Graduate. He is currently an associate professor with the College of Information System and Management at National University of Defense Technology, Changsha, China. His research interests include scheduling and resource management in green computing, cluster computing, cloud computing, and multiple satellites. He has published more than 50 research articles in refereed journals and conference proceedings such as IEEE TC, IEEE TPDS, JPDC, JSS and so on. He is a member of the IEEE, the IEEE Communication Society, and the ACM.



Wei Wei received the M.S. and Ph.D. degrees from Xian Jiaotong University, Xian, China, in 2011 and 2005, respectively. He is an assistant Professor at Xian University of Technology. His research interests include wireless sensor network, pervasive computing, and distributed computing. He is a member of the IEEE.

Xueshan Luo received the B.S. degree in information engineering from Huazhong Institute of Technology, Wuhan, China, in 1985, and the M.S. and Ph.D. degrees in system engineering from the National University of Defense Technology, Changsha, China, in 1988 and 1992, respectively. Currently, he is a professor of Information System and Management, National University of Defense Technology. His research interests are in the general areas of information system and operation research.