Managing and Aggregating Data Transfers in Data Centers

Deke Guo*, Mo Li‡, Hai Jin†, Xuanhua Shi†, Lu Lu†
* Key Lab of Information System Engineering, National University of Defense Technology, China
† School of Computer Science and Technology, Huazhong University of Science and Technology, China
‡ School of Computer Engineering, Nanyang University, Singapore

***Abstract*—Distributed computing applications like MapReduce transfer massive amount of data between their successive processing stages. These data transfers, such as common shuffle and incast communication patterns, contribute most of the network traffic and thus have severe impacts on application performances. Despite such impacts, there has been relatively little work on decreasing the amount of traffic for computing such data transfers. We observe that the massive data flows in such a transfer already apply aggregate functions at the receiver side and the reduction in size between the input data and the output data is even pronounced.**

This motivates us to bring opportunities for performing inter-flow data aggregation during the transmission phase as early as possible rather than just at the receiver side. To this end, we first demonstrate the gain and feasibility of inter-flow data aggregation for data transfers in data centers with novel network structures. To achieve such a gain, such data transfers are normalized as the incast transfer. It is modeled as an incast minimal tree problem that is proved to be NP-hard in representative BCube and FBFLY data centers. We propose two approximate methods, the RS-based and ARS-based incast tree building methods, to generate an efficient incast tree based on only the labels of all incast members and the data center topology. We further present incremental methods to tackle the dynamic and fault-tolerant issues of the incast tree. Using a prototype implementation and large-scale simulations, we demonstrate that our method can significantly decrease the amount of network traffic, save the data center resources, and reduce the delay of the entire process of a job. Moreover, our proposals for BCube and FBFLY can also be applied to other novel data centers after minimal modifications.

I. Introduction

Large-scale data centers serve as infrastructures for not only online cloud applications, such as web search and email, but also systems for massively distributed computing frameworks, such as MapReduce [1], Dryad [2], CIEL [3], Pregel [4], and Spark [5]. To date, such systems manage large number of data processing jobs each of which may across hundreds even thousands of servers in a data center.

Such systems typically implement a data flow computation model, where massive data are transferred across successive processing stages. Such data transfers contribute most of the network traffic and impose severe impacts on the application performance and the data center utilization. Although prior work [6] improves the data center utilization by scheduling network resource, there has been relatively little work on decreasing the network traffic resulting from such data transfers.

We observe that such data flows across successive stages already apply aggregate functions at the receiver side in the state-of-the-practice. For example, during the shuffle phase of the MapReduce job, each reducer is assigned a unique partition of the key range produced by the map stage. Each reducer then pulls the content of such a partition from every mapper's output and perform some aggregation operations. As prior work [7] has shown, the reduction in size between the output data of all mappers and that of all reducers after aggregation is 81.7% for Facebook jobs. Such insights motivate us to think whether we can apply the same aggregate function to those flows during the transmission process as early as possible rather than just at the receiver side. If such an inter-flow data aggregation can be achieved, it would significantly decrease the traffic and occupy less data center resources. Moreover, it may speed up the entire computation of a job since the size of final input data to each reducer will be considerably reduced.

To this end, we manage and optimize the network activity at the level of transfers for aggregating correlated data flows across successive stages and achieving the above goals. A transfer denotes the set of all data flows across successive stages of a job. The many-to-many shuffle, many-to-one incast, and one-to-many multicast are common transfers and multicast does not involve the inter-flow data aggregation. We focus on the incast transfers since a shuffle transfer can be normalized as a set of independent incast transfers. Currently, an incast transfer is treated as a set of unicast transmissions that do not account for collective behaviors of flows. In such a way, an incast transmission has less opportunity to achieve the gain of inter-flow data aggregation.

For an incast transfer, the gain of inter-flow data aggregation can be achieved only if the involved flows have opportunities to intersect and cache at some devices in their routing paths. In a tree-based switch-centric data center, the buffer space available in a commodity switch is very limited, e.g., only 4MB. Although the high-end switch has larger buffer space than the commodity one, the buffer space is shared by more ports. Therefore, it is impractical for such tree-based structures to support the inter-flow data aggregation. Fortunately, many novel network structures [8], [9], [10], [11], [12] are proposed since the tree-based structures are increasingly difficult to meet the requirements of large-scale data centers. The server-centric [8], [9], [10] and the flat switch-centric [11], [12] network structures possess the in-network cache ability and a high link density. Moreover, they provide multiple routing paths for any pair of servers and bring opportunities to managing each incast transfer so as to form rendezvous for its flows.

In this paper, we first demonstrate the gain and feasibility of inter-flow data aggregation for an efficient incast transfer in the flat switch-centric and the server-centric data centers. To

exploit the maximum gain of the inter-flow data aggregation, the efficient incast transfer is formalized as the incast minimal tree problem. The challenge that arises here is how can we discover an incast minimal tree for any incast transfer. We prove that such a problem is NP-hard in BCube and FBFLY, two typical network structures with the outstanding topological properties compared to others in the same category.

We propose two approximate methods, AS-based and ARS-based methods, to build an efficient incast tree for any incast transfer so as to occupy less data center resources and incur less network traffic. We also present incremental methods to address the dynamic and fault-tolerant issues of an incast tree generated by our methods. We further evaluate our method and related work using a prototype implementation and large-scale simulations. The results demonstrate that our method can significantly reduce the network traffic, save data center resources, and speed up the computation of a job, compared to others. More precisely, our ARS-based method saves traffic by 39% on average for a small-scale incast transfer with 120 senders in data centers BCube$(6,k)$ where k ranges from 2 to 9. Additionally, it saves traffic by 59% on average for incast transfers with 100 to 4000 senders in a data center BCube$(8,5)$ with 262144 servers. Moreover, it can save more network traffic if the members of an incast transfer distribute in data centers in a managed manner rather than a random way.

The rest of this paper is organized as follows. We briefly describe the background and related work in Section II. We model the problem of efficient incast transfers that exploit the gain of inter-flow data aggregation in Section III. We propose the ARS-based incast tree building method and address the dynamic and fault-tolerant issues in Section IV. Section V presents some issues that can be further improved. We evaluate our methods and related work using a prototype and simulations in Section VI and conclude this work in Section VII.

II. Background and related work

A. Network structures for data centers

Inside a data center, a large number of servers and switches are connected using a specific data center networking (DCN) structure. The current practice organizes switches into a tree-based structure. Recently, researchers have made growing efforts towards designing new data center network structures to replace the tree structure. Such new network structures can be roughly divided into three categories.

The first one is switch-centric, which organizes switches into structures other than tree and puts the interconnection intelligence on switches. Fat-Tree [13], VL2 [14], and PortLand [15] organize switches into variants of the tree structure in which servers only connect to the edge-level switches. Such proposals are called the tree-like switch centric structures. To conquer the shortages of tree-like structures, two flat switch-centric structures FBFLY [11] and HyperX [12] are proposed, which organizes high-radix switches, each connecting several servers, into generalized hypercube structures. They have no performance bottlenecks since all of the links and switches are used equally and thus outperform the tree-like structures.

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). DCell [8], BCube [9], FiConn [16], HCN [10], and CamCube [17] fall into the third category. CamCube differentiates itself from others by utilizing a 3-dimensional torus topology, i.e., each server is directly connected to six neighboring servers but without using any switch. The main problem of the torus topology is the long network diameter, compared to others. Such a problem causes a high communication latency and a low aggregate network throughput while mega data centers pursue the low latency, high network capacity and scalability. To achieve a low network diameter, the size of the torus topology has to be small-scale, compared to other topologies with the same network diameter, and thus cannot meet the requirement of large-scale data centers.

Among those proposals to improve the tree-based network structure for data centers, Bcube and FBFLY are two outstanding ones in the two categories.

FBFLY: The k-ary n-flat flattened butterfly (FBFLY) is a scalable yet low-diameter network that takes advantage of high port count switches [18]. The key insight behind FBFLY is to having the generalized multi-dimensional hypercube [19] interconnect all of the switches. More precisely, all of switches in each dimension connect with each other and each switch connects datacenter servers using its additional ports. Additionally, any pair of servers are not directly connected and each server only connects with one switch in the FBFLY. For example, for a 4-ary 3-flat FBFLY with c=2, there are $c\times k^{n-1} = 2\times 4^2$=32 servers and k^{n-1}=16 8-port switches each of which connects c=2 datacenter servers. Such a FBFLY only accommodates 16 servers if c=1.

BCube: Bcube is a server-centric network structure for a shipping-container data center that is typically of size 1k-4k servers. A BCube$_0$ is simply n servers connecting to a n-port switch. A BCube$_k$ ($k \leq 1$) is constructed from n BCube$_{k-1}$'s and n^k n-port switches. Each server in a 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, which consists of n^{k+1}=16 servers and two levels of $n^k(k+1) = 8$ switches.

Although such proposals for data centers, differ in the network structures, several consistent themes have emerged among them. First, a trend in data center design is to use low cost, commodity servers and switches. Second, such network structures possess a high link density and provide multiple routing paths for any pair of servers. Third, such networks structures are built in a regular way. Such trends bring challenges and opportunities in managing the data transfers.

B. Shuffle and incast transfers in data centers

We use the MapReduce as an example to provide a thorough analysis of two common transfer patterns, shuffle and incast, among distributed computing architectures in data centers. A MapReduce job consists of two successive stages, the map

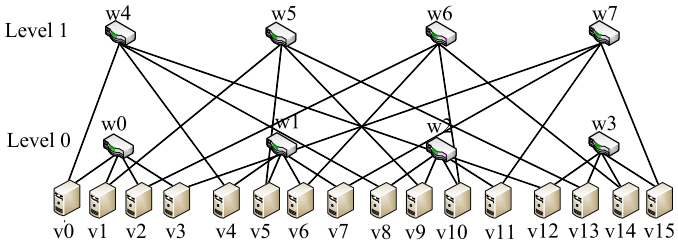


Fig. 1. A BCube(4,1) structure.

and the reduce, after the JobTracker has scheduled the sets of the map and reduce tasks. In the map stage, each of the map tasks will apply a user-defined map function to each input record and generates a list of key-value pairs. In the reduce stage, every reduce task applies a reduce function, usually an aggregate function, to each input record.

During the shuffle phase of the MapReduce job, each reducer is assigned a partition of the key range produced by the map stage and each reducer pulls the content of such a partition from every mapper's output. The data flows from two mappers to the identical reducer is highly correlated since the key range and its partitions are the same for all mappers. In such a setting, a reducer cannot fetch the output of a mapper until the mapper has finished and stored its final output to HDFS. The JobTracker is responsible to notify the location of the output of every mapper to every reducer. All map tasks must complete before the shuffle can complete, allowing the reduce phase to begin. Recently, there has been a growing interest in supporting data pipelining during the shuffle phase. To this end, each mapper is modified to push data to reducers [1], [20], [21].

No matter a MapReduce job utilizes the pull or push mechanism, a shuffle transfer is established between the map and reduce stages. Other distributed computing frameworks, like Dryad, CIEL, Pregel, and Spark, possess the similar constructs. Consequently, the shuffle transfer is one of the most common traffic patterns in data centers. In general, a shuffle consists of m senders, $S_1, S_2, ..., S_m$, and n receivers, $R_1, R_2, ..., R_n$, where a data flow d_{ij} is established for every pair of sender i and receiver j for $1 \leq i \leq m$ and $1 \leq j \leq n$. An incast consists of m senders and one of n receivers, where a data flow is established from every sender to the same receiver.

III. The problem of efficient incast transfer

We first demonstrate the gain and feasibility of inter-flow data aggregation for efficient incast transfer. The efficient incast transfer is then formalized as the incast minimal tree problem, an NP-hard problem. We finally discuss how to perform the inter-flow data aggregation given an incast tree.

A. Gain and feasibility of inter-flow data aggregation

The shuffle transfers are common communication pattern and are responsible for most of the network traffic in data centers. The flows from all senders to a given receiver, formed an incast transfer, are highly correlated during the shuffle phase of those distributed computing frameworks. The root cause is that the key range and its partitions are identical for all mappers. The data, a list of key-value pairs, among such flows thus share the identical key partition allocated to the same receiver. Therefore, for a key-value pair in one flow of an incast transfer there exists a key-value pair with the identical key in any other flow with high probability, as shown in Fig.2.

The key insights behind this paper are two observations as follows. First, the receiver typically applies an aggregate function, e.g., the sum, maximum, minimum, and average, to the received data from all flows in such an incast transfer. Second, although the combiner at each sender has aggregated its output before delivering to the receiver, there exist considerable opportunities for the data aggregation between different flows. Such insights motivate us to think whether we can apply the same aggregate function to those flows in the incast transfer during the shuffle phase rather than just at the receiver side. If such an inter-flow data aggregation can be applied, it would not hurt the correctness of the computing result at the receiver. On the contrary, it can significantly reduce the number of flows and the volume of transferring data during the shuffle phase.

For an incast transfer, the gain of inter-flow data aggregation can be achieved only if its flows have opportunities to intersect and cache at some network devices in their physical paths. The tree-like switch-centric structures utilize a large number of commodity, shallow buffered switches and less number of high-end switches. In current practice, the buffer space available in a commodity switch is very limited, e.g., only 4MB of buffer shared among 48 1Gbps ports and two 10Gbps ports for a 48-ports switch. Although the high-end switch has larger buffer space than the commodity one, the buffer space is allocated to more switch ports. Therefore, it is difficult for a shared switch in the paths of multiple flows to cache packets from such flows for an inter-flow aggregation.

For the server-centric structures for data centers, the commodity servers each with multi-ports act as not only end hosts, but also mini-switches. In practice, a server can use a PCI-E interface to connect a ServerSwitch card [22] that uses a gigabit, programmable switching chip for customized packet forwarding. A ServerSwitch card can leverage the server CPU for advanced in-network packet processing due to the high throughput and low latency between the switching chip and server CPU. As prior work [22], [23] has shown, a server with a ServerSwitch card can enable new data center networking services, e.g., in-network packet cache. The server-centric structures thus have brought opportunities for achieving the inter-flow data aggregation in data centers. If some flows in an incast transfer intersect at a server, the early arrived packets can be cached on the server for the inter-flow data aggregation. After such an advanced processing to those flows, a new flow with a list of new packets are generated and forwarded to the receiver of the incast transfer by the caching server.

For the flat switch-centric networks for data centers, all flows in the incast transfer are delivered along paths consisting of high-radix switches. The packets of such flows are traditionally not cached at the rendezvous switch for advanced processing due to the limited buffer. If all servers utilize ServerSwitch-like cards for the connections with high-radix

switches, the incast transfer can achieve the gain of inter-flow data aggregation as follows. For a rendezvous switch of some flows in an incast transfer, it can leverage the resources of one server connecting with it, for caching and advanced processing the packets from all those flows.

In summary, the inter-flow data aggregation can significantly reduce the number of flows and the volume of transferring data during the shuffle phase without hurting the accuracy of the output of the receiver. Such gains, however, can be achieved in a data center only if it utilizes a flat switch-centric network structure or a server-centric network structure. It is difficult for a data center with a tree-like network structure to achieve the inter-flow data aggregation in practice. If existing high-end and commodity switches are replaced by software-based or FPGA-based switches [24], [25] in data centers, it would be possible to perform inter-flow data aggregation in tree-like switch-centric structures.

B. Incast minimal tree problem

For a given data center, we model it as a graph $G = (V, E)$ with a vertex set V and an edge set E. A vertex of the graph corresponds to a switch or a datacenter server. An edge (u, v) denotes a link, through which u connects with v. The network structure of the data center contributes to the edge set E.

In this paper, our motivation is to minimize the amount of the resultant traffic in a shuffle or an incast transfer by applying the inter-flow data aggregation method for the correlated flows in the transfer. We focus on the incast transfer since a shuffle transfer is equivalent to multiple independent incast transfers. Given an incast transfer with a receiver R and a set of senders $\{S_1, S_2, ..., S_m\}$ in a data center, we need to get an incast tree from the graph G=(V, E). The data flow from each sender thus can be delivered to the receiver r along the resultant tree. There exist so many such kinds of trees for an incast transfer in densely connected data center networks, e.g., BCube and FBFLY. A challenging issue that arises here is how can we identify an incast tree that results in less number of network traffic after using the method of inter-flow data aggregation.

For any incast tree, the vertices are divided into two parts: terminals and nonterminal vertices. The terminals are the given vertices that have to be included in the tree, i.e., the receiver and all senders in the incast transfer. The nonterminal vertices can be datacenter servers or switches and are used to form a connected subgraph with the terminal vertexes together. A vertex can achieve the inter-flow data aggregation if it enables the in-network cache and receives at least two incoming flows in the incast tree. Such kind of vertices are called aggregating vertices and others are the non-aggregating ones. Note that the generated flow at a terminal vertex acts as an incoming flow to it. As we have shown in Section III-A, the resultant flow of inter flow data aggregation is assumed to be the same size as each incoming flow at an aggregating vertex. Thus, the size of the outgoing flow of the aggregating vertex is just equal to that of one incoming flow. For a non-aggregating vertex, the size of its outgoing flow is the cumulative of its incoming flows.

Given an incast tree, we define its cost metric as the volume of introduced network traffic into the data center for completing the incast transfer along the tree. More precisely, the cost of an incast tree is the total edge weight, i.e., the sum of the amount of the outgoing traffics of all vertices in the incast tree except the receiver. Without loss of generality, the size of traffic resulting from each of the m senders is assumed to be unity (1 MB) so as to normalize the cost of an incast tree. In such a way, the weight of the outgoing link is one at an aggregating vertex and is equal to the number of incoming flows at a non-aggregating vertex. Note that the generated flow at a terminal acts as an incoming flow to it.

Definition 1: For an incast transfer, the incast minimal tree problem means to find a connected subgraph in G=(V, E) spanning all terminals with minimal total cost for completing the incast transfer.

The problem is then translated as how can we discover an incast minimal tree for an incast transfer in a data center. We will prove that the incast minimal tree problem in typical data center network structures such as BCube and FBFLY is NP-hard in Theorem 1.

Theorem 1: The incast minimal tree problem for an incast transfer in a BCube network is NP-hard.

Proof: Recall that the cost of an incast tree of an incast transfer is just the total edge weight. Among all vertices except the receiver in the tree, the weight of the outgoing link is one at an aggregating vertex and is equal to the number of incoming flows at a non-aggregating vertex. Therefore, the weight of the outgoing link of a non-aggregating vertex would exceed one if the vertex has more than one incoming flows. Such kind of vertices correspond to the switches in data centers. If we treat such vertices as aggregating ones, the weights of their outgoing links would become one. Such a relaxation can simply the incast minimal tree problem as a Steiner tree problem since the weight of each edge in the tree is one and thus the cost of the tree is just the total number of edges. The Steiner tree problem in BCube is NP-hard, as prior work [9], [26] has shown. Therefore, the incast minimal tree problem in a generic BCube network is NP-hard. ■

Lemma 1: The incast minimal tree problem for an incast transfer in a FBFLY network is NP-hard.

Proof: In a FBFLY network, all switches are organized as a generalized hypercube structure. In principle, a BCube network actually emulates a generalized cube by having all servers in each dimension connecting with each other through a switch. Therefore, the incast minimal tree problem in a FBFLY network is also NP-hard since the problem in a generic BCube network is NP-hard, as proved in Theorem 1. ■

To meet the requirement of online tree building for incast transfers with large number of senders, we aim to design an approximate algorithm by exploiting the topological feature of future data center networks.

In densely connected data center networks, there are multiple available unicast paths between any server pair. For example, BCube(n, k) exists k+1 equal-cost paths between any two servers if their labels differ in k+1 dimensions. An

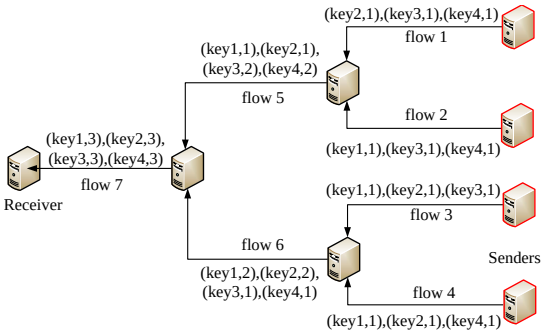


Fig. 2. An example of inter-flow data aggregation for an incast transfer.

intrinsic solution to the incast tree problem is that each sender independently delivers its data to the receiver along a unicast path and the incast tree is thus formed by the combination of such unicast paths. Such a unicast-driven routing protocols are undesirable since independent path selections by receivers can result in many unnecessary intermediate links. Moreover, the resultant incast tree could not achieve the desirable benefit of inter-flow data aggregation since the number of aggregating vertices in the tree is often very small. As a result, such kind of solutions cannot considerably save the network traffic of an incast transfer. We thus need a deterministic approach for building an efficient incast tree for arbitrary incast transfer.

Without loss of generality, we take an example in the BCube topology shown in Fig.1. We assume that the receiver is v_0 and the sender set is $\{v_2, v_5, v_9, v_{10}, v_{11}, v_{14}\}$. If using unicast-driven routing, the resultant incast tree can have 18 links, as shown in Fig.3(a). The cost of such an incast tree is 22 and the tree has no aggregating vertex. However, an efficient incast tree for the same incast transfer consists of only 12 links if we construct in the way shown in Fig.3(c). The total cost of the resultant incast tree is 16 and there exist two aggregating vertices, v_1 and v_2.

C. Inter-flow data aggregation based on an incast tree

For an incast transfer, the incast manager can associate it with an efficient incast tree that can be achieved by our incast-tree building methods, as we will show in Section IV. The problem is then translated as how to perform the inter-flow data aggregation in the incast transfer given an incast tree.

The incast manager first makes all servers in the incast tree be aware of the tree structure by broadcasting the resultant incast tree to them. In such a way, each involved server at the incast transfer will know its parent server and child servers (if any) and can be an aggregating server if it has more than one child servers or has one child but it is a sender. For example, the servers v_0, v_1, v_2, and v_3 in Fig.3(c) are aggregating servers.

Each sender of the incast transfer greedily delivers a data flow toward the receiver along the incast 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. Note that the server-centric and flat switch-centric structures for data centers can naturally support the in-network cache. Upon the data from all its children and itself (if any) have been received, an aggregating server performs the inter-flow data aggregation among $i \leq m$ flows as follows. It groups all key-value pairs in such flows according to their keys and applies the aggregate function at the receiver to each group. Such a process finally replaces the i flows with a new one that is continually forwarded to the receiver along the incast tree.

Note that an aggregating server may perform the aggregation operation once a data flow arrives. Such a scheme amortizes the delay due to simultaneously aggregate all incoming flows from itself and its child servers. At the root of the tree, i.e., the receiver of the incast transfer, all of received data are aggregated using the reduce function. Fig.2 depicts an example of the inter flow data aggregation based on an incast tree. We can see that flows 1 and 2 are merged as a new flow 5, flows 3 and 4 are aggregated as a new flow 6. The flows 5 and 6 are then forwarded to the receiver and aggregated as a flow 7.

Those distributed computing frameworks like MapReduce, however, suffer the computation skew, as shown in prior work [27], [28], [29]. The map tasks exhibit significant skews, where their runtime depends on more than just input size and therefore vary dramatically and unpredictably. When such a skew arises, some map tasks take significantly longer to process their input data than others[1], slowing down the entire computation of a job. Therefore, other senders have opportunities to transfer their data flows along an incast tree before the slowest sender in an incast transfer completes its task. If each sender independently sends its data to the receiver once it completes, the time for delivering the data flow from the slowest sender to the receiver is just the time consumed by the incast transfer. If all senders deliver their data flows along an incast tree, the data flow from the slowest sender introduces a few additional time at each aggregating server (if any). Note that only one subtree is influenced by the lowest sender. The number of aggregating servers for a data flow is at most the tree depth minus one and is at most k in a BCube(n,k) network. Such additional time is a little bit compared to the slowest map task takes and thus has a trivial impact on the entire time for completing a job.

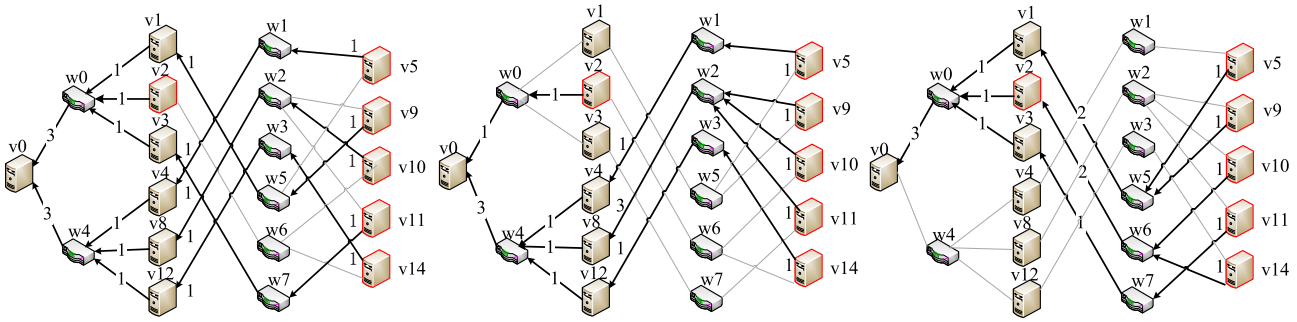
IV. Efficient Incast Tree Building

We start with the RS-based and ARS-based incast tree building methods. We then present incremental methods to tackle the dynamic behaviors of incast members and the fault-tolerant issues of the incast tree. We also study the usability of our methods in other mainstream structures for data centers.

A. Incast tree building method

Theorem 1 proves that building an incast minimal tree for any incast transfer in BCube, a typical data center architecture, is NP-hard. The unicast-driven incast tree has less chance of achieving the gain of inter-flow data aggregation and cannot save the network traffic effectively. In this paper, we propose to build an efficient incast tree in a managed and deterministic way. For an incast transfer, there exists an incast manager,

[1]Prior work [29] have shown that the slowest map task takes more than twice as long to complete as the second slowest map task, which is still five times slower than the average.



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

Fig. 3. Different incast 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 a Bcube(4,1).

e.g., the JobTracker of a MapReduce job, that are aware of the senders and receiver. The incast manager then calculates an efficient incast tree using an approximate method, given the incast membership distribution and the data center topology.

The topology of a BCube(n,k) can be abstracted as a $k+1$ dimensional n-ary generalized Hypercube [19] with the same set of servers. For a BCube(n,k) and the generalized Hypercube, two servers $x_k x_{k-1}...x_1 x_0$ and $y_k y_{k-1}...y_1 y_0$ are called the mutual j dimension 1-hop neighbors if their labels differ in only the j dimension, where x_i and $y_i \in \{0,1,...,n-1\}$ for $0 \le i \le k$. Therefore, a server has $n-1$ 1-hop neighbors in each dimension. The difference is that such two servers directly connect with each other in the generalized Hypercube while connect to a j level switch with a label $y_k...y_{j+1}y_{j-1}...y_1 y_0$, i.e., $x_k...x_{j+1}x_{j-1}...x_1 x_0$, in a BCube$(n,k)$. In this case, a server and its all neighbors in each dimension are connected indirectly via a common switch. Additionally, any two servers are called the mutual j-hop neighbors if their labels differ in number of j dimensions for $0 \le j \le k$.

Consider an incast transfer consists of a receiver R and a set of senders $\{S_1, S_2, ..., S_m\}$ in a BCube(n,k). Assume that each receiver is labeled as $r_k r_{k-1}...r_1 r_0$ and a sender is labeled as $s_k s_{k-1}...s_1 s_0$, where $r_i \in \{0,1,...,n-1\}$ and $s_i \in \{0,1,...,n-1\}$ for $0 \le i \le k$. It is clear that the hamming distance between the receiver and every sender is at most $k+1$. Without loss of generality, we assume that the maximum hamming distance between the receiver and every sender is $k+1$ and discuss other cases in Section V. Thus, the shortest paths from all senders to the receiver can be expanded as a directed multistage graph with $k+2$ stages. The stage 0 only has the receiver. The servers at the stage j must be some of the j-hop neighbors of the receiver, i.e., their labels differ in j dimensions compared to that of the receiver, for $1 \le j \le k+1$. Additionally, there must exist a set of switches as relays between two successive stages since servers are not directly connected in a BCube(n,k).

First of all, each of all senders should appear at the stage j if it is a j-hop neighbor of the receiver. For example, senders v_5, v_9, v_{11}, and v_{14} are at the stage 2 while another sender v_2 is at the stage 1, as shown in Fig.3. Only such senders and the receiver in a BCube(n,k), however, cannot definitely form a connected subgraph in the level of a generalized hypercube. The problem is then translated as how to select some additional servers for each stage and switches between successive stages so as to constitute an incast tree in a BCube(n,k), with low cost of supporting the incast transfer. Before presenting our approach, we first make two definitions on the incast tree.

Definition 2: For a server set A at the j stage and a server set B at the $j+1$ stage in an incast tree for any j ranging from 0 to k, we call A covers B if and only if for each server $u \in B$, there exists a directed path from itself to a server $v \in A$. We call A strictly covers B if any subset of A does not cover B.

Recall that a pair of neighboring servers across two successive stages connect to a common switch whose label and level can be derived from the labels of the two servers. For such reason, the switch in each directed path between two successive stages can be identified and thus we only focus on the selection of additional servers for each stage between successive stages. Additionally, we will not explicitly present the switch between a pair of neighboring servers that are reachable with each other in a BCube(n,k). The insight behind Definition 2 is to ensure that a data flow from a server, in the $j+1$ stage, toward the receiver can be sent to a server in the j stage.

We propose to identify other necessary servers besides the senders at each stage in an expansion way from the stage $k+1$ to the stage 1, recursively. A constraint must be satisfied to constitute an efficient incast tree. The server set at any stage $j-1$ strictly covers the server set at the successive stage j for $1 \le j \le k+1$. Given the server set at the stage j, we leverage the topology features of a BCube(n,k) to infer the required servers at the stage $j-1$ under such a constraint for $1 \le j \le k-1$. The number of such server sets at the stage $j-1$, however, may not be unique in a BCube(n,k), a densely connected data center network. The root cause is that each server in the stage j has a mutual neighbor at the stage $j-1$ in each of j dimensions in which the labels of the current server and the receiver differ. Such a feature motivates us to define a routing sequence for each incast transfer.

Definition 3: Let $e_1 e_2...e_j...e_k e_{k+1}$ be a routing sequence, with $k+1$ routing symbols. It is one of $(k+1)!$ permutations of the set $\{0,1,2,...,k\}$. The stage j of an incast tree is associated with a routing symbol e_j for $1 \le j \le k+1$. For each server $X = x_k...x_j...x_1 x_0$ in the stage j and the receiver,

1) If their labels differ in dimension e_j, the server selects

one of its n mutual neighbors in dimension e_j whose label is identical in dimension e_j but different in $j-1$ dimensions compared to the receiver's label.

2) Otherwise, their labels differ in at least one dimension from dimensions e_{k+1} to e_{j+1}. Let the rightmost such dimension be $\overline{e_j}$. Thus, the server X selects one neighbor in dimension $\overline{e_j}$ whose label is identical in dimension $\overline{e_j}$ compared to the receiver's label.

In such a way, the selected neighbor of the server X appears at the stage $j-1$ in the incast tree.

Given an incast transfer and its associated routing sequence $e_1e_2...e_j...e_ke_{k+1}$, we can derive an incast tree under the above constraint along the processes as follows.

We start with any stage j in the $k+1$ stages and assume that j=$k+1$, without loss of generality. Upon the servers at the stage j are given, we partition all such servers into groups such that all servers in each group are mutual one-hop neighbors in the dimension e_i, i.e., their labels differ only in the e_i dimension. The size of each group is at most n due to their $k+1$ dimensional n-ary labels. The number of groups is at most the total number of senders in the incast transfer. In each group, the labels of all servers differ in j dimensions compared to the receiver's label, i.e., they are j-hop neighbors of the receiver. The next server along the path from each server in the group to the receiver should differ in $j-1$ dimensions and appears in the $j-1$ stage. Note that each server in the group has j selections about the next server towards the receiver. Here, we require that the paths from all servers in the group to the receiver share a common next server at the stage $j-1$.

To this end, we have each server in a group selecting the next sever using the scheme shown in Definition 3. Thus, all servers in a group and their common next server have the same label except the e_j dimension. Actually, the e_j dimension of the common next server is just equal to the receiver's label at the e_j dimension. As shown in Fig.3(b), all servers at the stage 2 are partitioned into three groups, $\{5\}$, $\{9,10,11\}$, and $\{14\}$, using a routing symbol e_2=0. The next servers at the stage 1 of such groups are with labels 4, 8, and 12, respectively. In such a way, the data flows from the group toward the receiver can achieve the desired inter-flow data aggregation at the stage $j-1$ if exists, as shown in Fig.3(b). Otherwise, there is less opportunity to get the inter-flow data aggregation at the stage $j-1$, as shown in Fig.3(a). Note that our method still works well if the group has only one server. The neighbor of such a single server at the stage $j-1$ is determined by $\overline{e_j}$, as discussed in Definition 3. The single data flow from the server toward the receiver has no gains of inter-flow data aggregation at the next stage.

After processing other groups in the stage j via the same method, the server set at the stage $j-1$ that strictly covers the server set at the stage j are achieved. Note that some of senders may also appear on the $j-1$ stage. So far, the set of all servers on the $j-1$ stage are the union of the two parts. We can thus apply our method to infer the required servers at the stage $j-2$, and so on. Finally, all servers in all $k+2$ stages and the directed paths between successive stages constitute an incast tree for the given incast transfer in a BCube(n,k). The resultant tree is called the routing sequence based incast tree, abbreviated as RS-based incast tree.

Theorem 2: Given an incast transfer consisting of m senders in a BCube(n,k) and a routing sequence, the complexity of the RS-based incast tree building method is $O(k\times m)$

Proof: Consider that our method has to apply to at most k stages, from stage $k+1$ to stage 2, in an incast tree. The process to partition all servers at a stage j as groups according to e_j can be simplified as follows. For each existing server at the 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 and infer the common next server of all possible servers in the group. The upcoming servers in the group need not calculate the common next server at the stage $j-1$ again. The resultant computational overhead at the stage j is proportional to the number of servers at such stage. Thus, the time complexity at each stage is $O(m)$ since the number of servers at each stage cannot exceed m. The time complexity of our method is $O(k\times m)$ due to involve at most k stages. Thus, Theorem 2 proved. ■

B. Incast minimal tree building method

A routing sequence $e_1e_2...e_j...e_ke_{k+1}$ can yield an RS-based incast tree for any incast transfer in a BCube(n,k). Inspired that there exist at most $(k+1)!$ routing sequences for an incast transfer, we can derive $(k+1)!$ incast trees each of which meets the constraint that the server set at any stage strictly covers the server set at the successive stage. Such incast trees for an incast transfer achieve different gains of inter-flow data aggregation and thus may have diverse tree costs. For example, 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, we can derive an incast tree under a routing sequence e_1e_2=10, as shown in Fig.3(b), and another incast tree under a routing sequence e_1e_2=01, as shown in Fig.3(c). The incast tree under e_1e_2=10 is of cost 18 with 14 links while that under e_1e_2=01 is of cost 16 with 12 links, where the tree cost is defined in Section III-B.

A challenging issue that arises here is how can we efficiently find an RS-based incast tree with the minimal cost among at most $(k+1)!$ candidates for an incast transfer in a BCube(n,k). An intrinsic way is to generate all possible RS-based incast trees by applying our method under each of the $(k+1)!$ routing sequences. We then calculate the cost of each RS-based incast tree and select the one with the minimal cost after a round of comparison. Such an intrinsic way consists of three processes and suffers high computation overhead as follows.

The process of generating all RS-based incast trees for an incast transfer will incur $O((k+1)!\times k\times m)$ computation overhead since it needs to calculate at most $(k+1)!$ incast trees each of which incurs $O(k\times m)$ computation overhead. Additionally, calculating the cost of an RS-based incast tree needs to check the weight of all paths across all successive stages. The resultant computational overhead is $O(k\times m)$ since the number of paths between any successive stages is at most m. It thus generates $O((k+1)!\times k\times m)$ computation overhead

to identify the cost for at most $(k+1)!$ incast trees. Finally, it incurs $(k+1)!$ comparison overhead to identify the incast tree with the minimal cost from at most $(k+1)!$ incast trees. In summary, the time complexity of such an intrinsic way of inferring an incast with the minimal cost is $O((k+1)!\times k\times m)$.

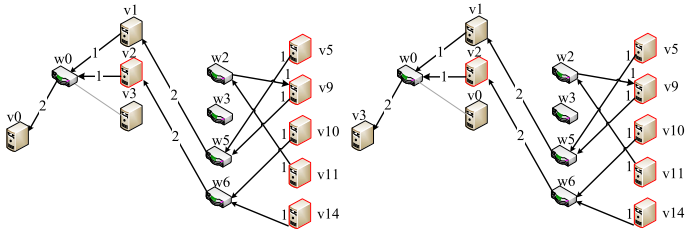
We further improve such an intrinsic method by proposing an efficient method as follows. We start with any stage j in the $k+1$ stages and assume that $j=k+1$, without loss of generality. After defining a routing symbol $e_j \in \{0,1,...,k\}$, all servers at the stage j can be partitioned into groups from each of which we can identify an additional server for the stage $j-1$ using our RS-based method. Thus, the number of partitioned groups is just equal to the number of appended servers at the next stage. The union of such appended servers and the potential senders at the stage $j-1$ constitute the server set at such a stage. Recall that the outgoing flows from the stage j may be merged at the aggregating servers if exist, the number of outgoing data flows from the stage $j-1$ is just equal to the size of the server set at such stage. Inspired by such a fact, we apply our RS-based method to other k settings of e_j and accordingly achieve other k server sets that can appear at the stage $j-1$. So far, we can simply select the server set with the smallest cardinality among all $k+1$ ones. The related setting of e_j is marked as the best choice for the stage j among all $k+1$ candidates. In such a way, the data flows from the stage j can achieve larger gains of inter-flow data aggregation at the stage $j-1$.

Upon the smallest server set at the $j-1$ stage is given, we can achieve the smallest server set at the successive stage $j-2$ and know the best choice of e_{j-1} at the stage $j-1$, and so on. Finally, server sets at all $k+2$ stages and the directed paths between successive stages constitute an incast tree. The behind routing sequence of the resultant incast tree is simultaneously determined. Such a method is called the ARS-based building method that generates the resultant ARS-based incast tree.

We use an example to reveal the benefit of the ARS-based building method. Consider the incast transfer shown in Fig.3. All senders at the stage 2 are partitioned into three groups, $\{5\}$, $\{9,10,11\}$, and $\{14\}$, using a routing symbol e_2=0. The resultant server set at the stage 1 is $\{2,4,8,12\}$, as shown in Fig.3(b). In the case of e_2=1, all senders at the stage 2 are partitioned into three groups, $\{5,9\}$, $\{10,14\}$, and $\{11\}$, and the resultant server set at the stage 1 is $\{1,3,4\}$, as shown in Fig.3(c). Therefore, we definitely select the server set $\{1,3,4\}$ for the stage 1 and e_2=1 for the stage 2. The selected incast tree in Fig.3(c) has lower cost than the one in Fig.3(b).

Theorem 3: Given an incast transfer consisting of m senders in a BCube(n,k), the complexity of the ARS-based incast tree building method is $O(k^2\times m)$.

Proof: Consider that our method has to apply to at most k stages, from stage $k+1$ to stage 2, in an incast tree. As shown in the proof of Theorem 2, the resultant computation cost at the stage $k+1$ for determining the server set at the stage k is $O(m)$ given a routing symbol e_{k+1}. In the ARS-based building method, we conduct the same operations under all of $k+1$ settings of e_{k+1}. This generates $k+1$ server sets for the stage k at the cost of $O((k+1)\times m)$ computation cost. Additionally, it incurs $O((k+1)\times m)$ computation cost to identify the server set with the smallest cardinality from the $k+1$ server set. In summary, the total computation cost of the ARS-based building method is $O((k+1)\times m)$ at the stage $k+1$.



(a) With the receiver v_0 (b) With the receiver v_3

Fig. 4. An improved incast tree of cost 14 with 11 links 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 or v_3.

At the stage k, the ARS-based method can identify the smallest server for the stage $k-1$ from k candidates resulting from k settings of e_k since e_{k+1} has exclusively selected one from the set $\{1,2,...,k+1\}$. The resultant computation cost is thus $O(k\times m)$ at the stage k. In summary, the total computation cost of the ARS-based building method is $O(\frac{k(k+3)}{2}\times m)$ since it involves at most k stages. Thus, Theorem 3 proved. ■

Recall that the basic idea behind the ARS-based building method is to identify the best routing symbol for each stage and then partition all servers at each stage as groups according to the associated routing symbol. At each stage j, all servers in a group establish direct paths to the same server, at the stage $j-1$, which aggregates the flows from all servers in the group and is determined by the routing symbol and the group servers. Consequently, the resultant incast tree for any incast transfer can fully exploit the benefits of inter-flow data aggregation across two successive stages.

We find that the incast tree may still have an opportunity to be optimized at each stage, especially at the largest stage, due to the observations as follows. There may exist groups each of which only has one element at each stage in the resultant incast tree. For example, the best incast tree 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, is determined by a routing sequence e_1e_2=01, as shown in Fig.3(c). In such a tree, all servers at the stage 2 are partitioned into three groups, $\{5,9\}$, $\{10,14\}$, and $\{11\}$. We can see that the flow from the server 11 to the receiver has no opportunity to perform the inter-flow data aggregation.

To address such an issue, we propose an intra-stage scheme to optimize the ARS-based incast tree for an incast transfer. After partitioning all servers at any stage j according to its routing symbol e_j in the ARS-based incast tree, the only server in a group has no mutual neighbors in the dimension e_j at the stage j. It, however, may have mutual neighbors in other dimensions except e_j at the stage j. In such a case, the sole server in a group no longer delivers its data along the path to the receiver in the ARS-based incast tree but forwards its data flow to one of such neighbors. The selected neighbor can thus aggregate all incoming data flows and the data flow by itself (if any) at the same stage.

Such an intra-stage aggregation can reduce the cost of the incast tree since an one-hop intra-stage forwarding operation increases the tree cost by two but decreases 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 path to the receiver in the ARS-based incast tree is at the stage $j-2$. Note that the cost of once inter-stage or intra-stage transmission is two due to the relay of an intermediate switch. For example, the server 11 has no neighbor in the dimension e_2=1, however, has two neighbors, the servers 9 and 10, in the dimension 0 at the stage 2, as shown in Fig.3(c). If the server 11 uses the server 9 or 10 as a relay for its data flow toward the receiver, prior incast tree can be optimized as the one, as shown in Fig.4(a). We can see that the cost of the incast tree is decreased by two and the number of active links is decremented by 1.

C. Dynamical behaviors of senders

When a new sender S_{m+1} joins an existing incast transfer with a set of senders $\{S_1, S_2, ..., S_m\}$ and a receiver R in a BCube(n,k), the incast tree should be updated to embrace such a new sender. An intrinsic way is to generate a new incast tree by invoking the ARS-based building method given the set of senders $\{S_1, S_2, ..., S_{m+1}\}$. Such a scheme, however, not only incurs $O(k^2 \times m)$ computation overhead at the incast manager but also consumes additional network bandwidth for disseminating the resultant new incast tree to its all servers.

To address such an issue, we prefer to update the incast tree in an incremental way. First of all, the incast manager keeps the routing sequence associated with the incast minimal tree before the joining of a new sender. Upon the incast manager schedules a new sender S_{m+1} into the existing incast transfer, it derives a unicast path from S_{m+1} to R by invoking the RS-based building method at the cost of $O(k)$ computation overhead. In such a case, the unicast transfer is a special case of the incast transfer since it only involves one sender.

The combination of the unicast path and prior incast tree constitutes a final incast tree for the current incast transfer. To enable all servers in the final incast tree be aware of such changes, the incast manager only needs to propagate the path structure to all servers along the path. In such a way, each server in the final incast tree is aware of the subtree rooted at itself that is sufficient for achieving the inter-flow data aggregation. Additionally, the final incast tree obeys our RS-based building method, and the path from each prior sender to the receiver R does not suffer any change.

When a sender S_j leaves the incast transfer, the incast tree should be updated to embrace such a change in an incremental way. First of all, the incast manager extracts the unicast path from the sender S_j to the receiver R in the existing incast tree and achieves the resultant incast tree for the new incast transfer. Note that extracting the unicast path for the leaving sender needs to decrease the weight of each related edge in the existing incast tree by one and the edge with weight 0 will be removed from the tree. To enable all servers in the final incast tree be aware of such changes, the incast manager only needs to propagate the extracted path to all servers along the path. In such a way, each server in the final incast tree is aware of the new subtree rooted at itself. Such a process also does not change the unicast path from other senders to the receiver.

In summary, our incast tree building method can gracefully address the event that a sender dynamically join or leave an incast transfer, since it does not change the sender-to-receiver paths of other senders in the incast transfer. In a MapReduce job, a master server schedules an idle server to replace a failed map server. Such a process consists of the leaving of a failed sender and the joining of a new sender.

D. Dynamical behaviors of the receiver

Similarly, the receiver R of a given incast transfer may also be replaced with a new one, denoted as R^n, after the incast manager generates an incast tree using our methods. In such a case, the incast tree should be updated to embrace such a change. The incast manager may generate a new incast tree by using the ARS-based building method, given the set of senders $\{S_1, S_2, ..., s_m\}$ and the new receiver R^n. Such an intrinsic scheme not only incurs $O(k^2 \times m)$ computation overhead but also consumes network bandwidth for disseminating the new incast tree to its all servers and canceling prior incast tree.

To address such an issue, we prefer to update the incast tree in an incremental way. First of all, the incast manager keeps the routing sequence, $e_1 e_2 ... e_j ... e_k e_{k+1}$, associated with the incast minimal tree for prior incast transfer. Given prior receiver R and the new receiver R^n, we compare their labels along $k+1$ dimensions, $e_1, e_2, ..., e_k, e_{k+1}$, one by one. If their labels differ only in the dimension e_j, i.e., R^n and R are mutual neighbors in dimension e_j, the two incast trees each for one receiver are equivalent from stage $k+1$ to stage j but differ from stage j to stage 0. In other words, not only the server sets but also the directed paths across two successive stages are identical from stage $k+1$ to stage j at the two incast trees.

Inspired by such a fact, the incast manager can partially reuse prior incast tree rooted at R and just recalculates the tree structure from stage j to stage 0 for efficiently generating the incast tree rooted at R_n as follows.

1) Given the routing symbols, $e_1, e_2, ..., e_j$, and the server set at the stage e_j, the tree structure from stage j to stage 0 can be calculated by using the RS-based building method.
2) If R^n appears in prior incast tree rooted at R, its subtree in such a tree still exists in the incast tree rooted at R^n but should appear from stage 0 in the new incast tree. The behind reason is the change of location of R^n between the two incast trees.

We can conclude that the smaller the j such a method is more efficient. If R^n is just one neighbor of R in dimension e_1 (j=1), the incast manager only needs to accordingly adjust the directed paths across the stage 1 and the stage 0. Fig.4 illustrates two incast trees with the same sender set but different receivers v_0 and v_3. Given the incast tree rooted at v_0 and the related routing sequence $e_1 e_2$=01, we can generate the incast tree rooted at v_3 by just adjusting the structure from

stage 1 to stage 0 in prior incast tree since v_0 and v_3 are mutual neighbors in dimension e_1=0. We can see that the two incast trees are identical from stage 2 to stage 1 but are different from stage 1 to stage 0.

Given an incast tree rooted at R with a routing sequence $e_1e_2...e_j...e_ke_{k+1}$, our findings suggest that all of the n mutual neighbors of R in the dimension e_1 are the best substitutes if the receiver R will be replaced. If none of such substitutes are idle, the incast manager will select one idle substitute from the n mutual neighbors of R in the dimension e_2, and so on. In such a way, we can maximize the reuse gain of the existing incast tree and thus significantly reduce the computation overhead for generating a new incast tree after updating its receiver.

E. Robustness against failures in an incast tree

After deriving an efficient incast tree for an incast transfer, the appearance of the link, server, and switch failures in a data center may have a negative impact on the incast tree. For such reasons, we describe how can we make the resultant incast tree be robust against failures.

Recall that the fault-tolerant routing protocol of BCube or other network structures already provides some disjoint paths between any server pair. When a server at stage j fails to send packets to its parent server at stage $j-1$ due to the link failures or switch failures, it forwards its packets toward its parent along another path that routes around the failures at the default one. In case of server failures, the methods proposed in Sections IV-C and IV-D can handle the failures on the senders and the receiver. We now focus on the failures on other inner servers in an incast tree.

When a server cannot send packets to its failed parent server, it is useless to utilize other backup disjoint path to the failed server. If the failed one is not an aggregating server in the path from the current server to the receiver, the current server can reroute packets along another disjoint path toward the next aggregating server (if any) or the receiver. For example, if server v_3 in Fig.3(c) fails, the data flow from server v_{11} should be rerouted toward the receiver 0 along another disjoint path, $v_{11}\rightarrow w_2\rightarrow v_8\rightarrow w_4\rightarrow v_0$. If the failed server is an aggregating server, it may have received and stored intermediate data from some senders. Thus, each sender whose data flow passes through such a failed aggregating server needs to redeliver its data flow to the next aggregating server (if any) or the receiver from a disjoint backup path.

V. Discussion

While the aforementioned methods can achieve the benefits of managing and aggregating incast transfers in data centers, there exist some important issues that can be further handled.

A. Impact of task placement

For a MapReduce-like application, each server runs a TaskTracker that supports a fixed number tasks, two maps and two reduces by default [1], in the current practice. Consequently, some senders even the receiver of an incast transfer may be scheduled to the same server. Our incast tree building methods can support such a case after minimal modifications as follows.

If a server hosts multiply map tasks for an incast transfer, the server appears as a single sender since it can locally perform the inter-flow data aggregation and then delivers interest data to the receiver of the incast transfer. If a server hosts many map tasks as well as a reduce task and such tasks belong to the same incast transfer, the server appears as a receiver since it can locally aggregate the output of all its map tasks.

B. Extension to general incast transfers

To ease the presentation, we focus on an incast transfer in BCube(n,k), where the maximum hamming distance between the receiver and each sender is $k+1$. Our RS-based and ARS-based methods for building an incast tree can be extended to involve general incast transfers. Let d denote the maximum hamming distance between the receiver and each sender in an incast transfer. If $d<k+1$, there exist $k+1-d$ dimensions, among the total $k+1$ ones, in each of which the labels of all incast members are identical. A desired incast tree of the incast transfer should be $d+1$ stages with the receiver at stage 0.

In such a case, the definition about the routing sequence should be revised as follows. Let $e_1e_2...e_j...e_d$ denote a routing sequence with d routing symbols. It is one of $d!$ permutations of d dimensions in each of which the labels of all incast members are not identical. The stage j of an incast tree is associated with a routing symbol e_j for $1\leq j\leq d$. So far, our RS-based and ARS-based methods for building can be directly utilized to calculate an incast tree for the general incast transfer.

C. Extension to other network structures for data centers

Although we use the BCube, a server-centric network structure for data centers, as an example to design the incast tree building methods, such methods can also be applied to the FBFLY and HyperX, flat switch-centric network structures. The root cause is that the topology behind the BCube is an emulation of the generalized hypercube while the switch topologies of FBFLY and HyperX are just the generalized hypercube. For the FBFLY and HyperX, our building methods can be used directly after minimal modifications as follows.

Given an incast transfer, the switch connecting one of its senders or the receiver acts an agent of such a server. Now, the incast manager can calculate an incast tree for the incast transfer at the switch level by using our ABS-building method. All senders first deliver data flows to their agents that further forward along the switch tree to the agent of the receiver. The receiver then collects data from its agent. For a switch in the resultant incast tree, if it has the responsibility of performing the inter-flow data aggregation, it needs to leverage its one server to realize the in-network cache.

VI. Performance evaluation

We start with the settings of our prototype implementation. We then evaluate our methods and related work under different data center sizes, incast transfer sizes, aggregation ratios, and distributions of incast members.

A. The prototype implementation

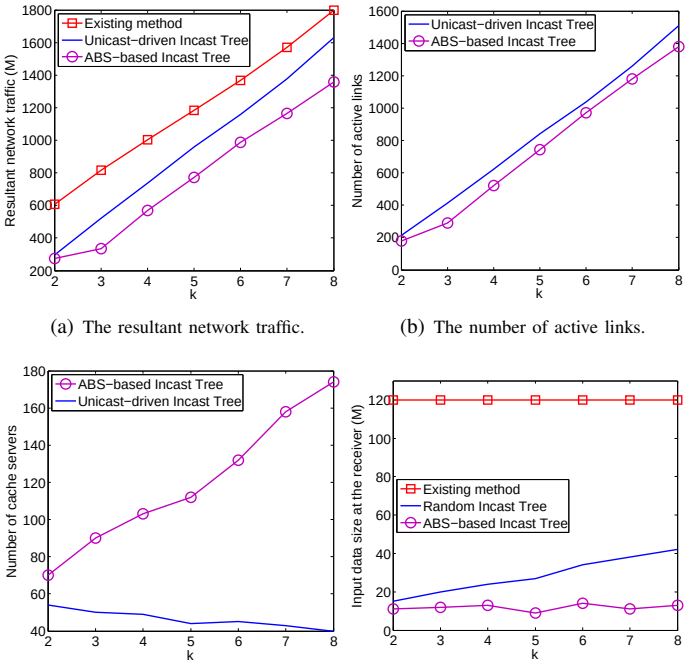
Our testbed consists of 31 virtual machines (VM) hosted by 3 physical servers connected together with switched Ethernet. Each server equips with two 8-core hyper-threaded Intel Xeon E5620 2.40 GHz processors, 24GB memory and a 1TB SATA disk, running an unmodified version of CentOS 5.6 with kernel version 2.6.18. Two of the servers run 10 virtual machines as the Hadoop slave nodes, the other one runs 10 virtual slave nodes and 1 master node. Each virtual slave node supports four map tasks and one reduce task. All the VMs run the same version of CentOS with modified Xen friendly kernels. Each server has a Intel 82567LM-2 NIC and a 82574L NIC, we only use the 82567LM-2 NIC during the evaluation. All VMs on a physic server share the hosts' NICs through a virtual switch as the default setting of the virtual hypervisor, Xen 3.1.2. Moreover, we improve the Hadoop implementation to embrace the in-network packet cache and inter-flow data aggregation under an incast tree for any incast transfer.

To achieve an incast transfer, we launch the built-in Word-Count job with the combiner in Hadoop 0.21.0. The number of senders (map tasks) and receivers (reduce tasks) for such a job are set to 120 and 1, respectively. We associate each map task ten input files each with 64M. In the shuffle phase of such a job, the average amount of data from a sender to the receiver is about 1M after performing the combiner at each sender. To evaluate such an incsat transfer in a data center, we need to generate an incast tree for completing it.

In this paper, we aim to deploy the incast transfer in BCube$(6,k)$ data centers for $2\leq k\leq 8$. To this end, we associate each sender and the only receiver with a $k+1$ dimensional BCube label in a random way. We thus conceptually achieve a random deployment of the incast transfer in the BCube data center and generate an ARS-based incast tree and a unicast-driven incast tree. As mentioned in Section III-B, our ARS-based incast tree omits the switch between two neighboring servers since it can be easily derived from the labels of servers.

To really deploy an incast transfer into BCube-based data centers, we design an injection mapping from all vertices in the incast tree to our testbed as follows. The master VM acts as the incast manager. Four senders, one possible receiver, and some inner vertices of the incast tree are mapped to each of the 30 slave VMs. In such a way, we abstract each VM as multiple VMAs (virtual machine agent) each of which corresponds to a vertex in the incast tree. We require that all vertices mapped to the same VM should not contain any pair of neighbors across successive stages in the incast tree. For example, vertices v_5 and v_9 in Fig.3(c) can appear at an identical VM that cannot accommodate the vertex v_1. Thus, no local communication will happen between VMAs inside a VM when performing the incast transfer. Actually, for any VMA its neighbor at the next stage in the incast tree appears at different VMs on the same even different physical servers. Thus, each edge in the incast tree is mapped to a virtual link between two VMs or a physical link across two servers in our testbed.

We then compare our ARS-based incast tree against the unicast-driven incast tree and the existing method in terms of four performance metrics. They are the resultant network traffic, the number of active links, the number of cache servers, and the input data size at the receiver for completing an incast transfer. The network traffic denotes the sum of network traffic over all edges in the tree. The existing method differentiate itself by not performing the data aggregation except at the receiver. In our experiments and large-scale simulations, we regulate the following factors to evaluate their impacts on the performance metrics. They are the data center size, the incast transfer size, the aggregation ratio among inter-flows, and the distribution of incast members.



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

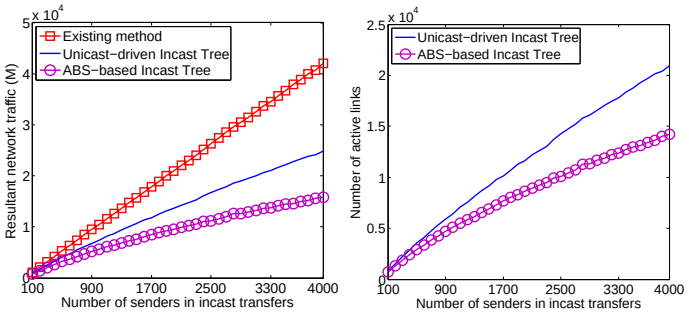
(c) Number of cache servers. (d) Input data size at the receiver.

Fig. 5. The change trends of some metrics for incast transfers with 120 senders in BCube(6,k), where k ranges from 2 to 9.

B. Impact of the data center size

Conceptually, we deploy an incast transfer with 120 senders to 1 receiver over a subnet of BCube(6,k) on our testbed given the ARS-based incast tree. We conduct experiments and then collect the performance metrics after completing such an incast transfer along the incast tree. We carry out similar experiments for the same incast transfer under the unicast-driven incast tree and the incast tree resulting from the existing method. Fig.5 shows the changing trends of the metrics on average, among 30 rounds of experiments, under different methods and settings of data centers. Note that a BCube(6,k) has 6^{k+1} servers.

We can see that both our ARS-based incast and the unicast-driven trees can considerably save the resultant network traffic by 39% and 18% on average compared to the existing method as expected. Such results demonstrate the gain of inter-flow data aggregation even in a small incast transfer. The root cause is that our method manages the data transmissions at the level of the incast transfer while the unicast-driven method just does



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

Fig. 6. The change trends of some metrics along with the increasing number of senders in an incast transfer in a BCube(8,5) with 262144 servers.

it at the level of individual flows. Consequently, the ARS-based incast tree achieves much more gain of the inter-flow data aggregation than the unicast-driven incast tree. Actually, the number of aggregating servers in the former one increases while that in the latter one decreases along with the increase of k, as shown in Fig.5(c). Besides such benefits, the ARS-based incast tree utilizes less links, and hence less servers and network devices, than the unicast-driven incast tree, as shown in Fig.5(b). Moreover, the ARS-based incast tree significantly reduces the size of input data at the receiver, as shown in Fig.5(d). Thus, it considerably reduces the delay of the shuffle and reduce phases of a job, as shown in Table I. Note that the delay during the map phase is the same for each method.

In summary, our ARS-based method supports a small size incast transfer with 120 senders well with less data center resources and network traffic than the other two methods, irrespective of the size of the data center.

C. Impact of the incast transfer size

Consider that a MapReduce-like job often involves several hundreds even thousands map tasks in the current practice. We should evaluate the performance of our method for an incast transfer with varying number of senders. It is difficult for our testbed to perform a large scale wordcount job since it is only equipped with limited resources. To address such a problem, we conduct extensively simulations to demonstrate the scaling properties of our method. To conduct simulations of incast transfers with m senders for $m \in \{100, 200, ..., 3900, 4000\}$, we created a synthetic wordcount job. The wordcount job provides the input data of 64M for each sender, randomly generated by the built-in RandomTextWriter of Hadoop based on a 1000-word dictionary. The data transmission from each sender to the receiver is controlled 1M on average. Fig.6 shows the network traffic and the number of active links on average under varying sizes of incast transfers in BCube (8,5). The number of servers accommodated by a data center with BCube(8,5) is 262144 and is large enough for production data centers.

TABLE I
TOTAL DELAY DURING THE SHUFFLE AND REDUCE PHASES (MINUTES).

BCube$(6,k)$	k=2	k=3	k=4	k=5	k=6
Existing method	15.2055	15.2073	15.2092	15.2110	15.2128
Our method	2.6655	3.0473	3.1758	2.6710	3.3062

Results in Fig.6(a) indicate that both our ARS-based and the unicast-driven incast trees considerably save the network traffic by 59% and 27% on average compared to the existing method as m ranges from 100 to 4000. Additionally, for any incast transfer our method always incurs much less network traffic than the other two methods, irrespective of the size of the incast transfer. Such results demonstrate the gain of inter-flow data aggregation even in large incast transfers. Although an incast transfer with 4000 senders is large enough in production data centers, our ARS-based method still outperforms other two methods in even larger incast transfers. Besides such a benefit, we can see from Fig.6(b) that the ARS-based incast tree utilizes less links (hence less servers and network devices) than the unicast-driven incast tree.

In summary, our ARS-based method can support an incast transfer of any size with less data center resources and network traffic than the other two methods, irrespective of the size of the data center.

D. Impact of the aggregation ratio

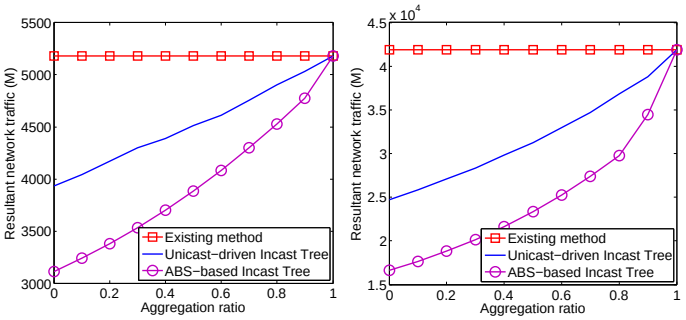
To ease the explanation, our analysis in Section III-B and the simulations for large-scale incast transfers make an assumption as follows. Data flows consisting of key-value pairs can be aggregated at a rendezvous server as a new one whose size is equal to that of the largest among such participants. In other words, the set of keys in each involved data flow is the subset of that in the largest data flow. In this section, we evaluate the performance of our ARS-based method under more general large-sale incast transfers.

Given β data flows in a general incast transfer, let c_i denote the size of the i^{th} data flow for $1 \le i \le \beta$. Let α denote the aggregation ratio among any number of data flows, where $0 \le \alpha \le 1$. Thus, the size of the resultant new data flow after aggregating such β ones is given by

$$\max\{c_1, c_2, ..., c_\beta\} + \alpha \times \left(\sum_{i=1}^{\beta} c_i - \max\{c_1, c_2, ..., c_\beta\}\right).$$

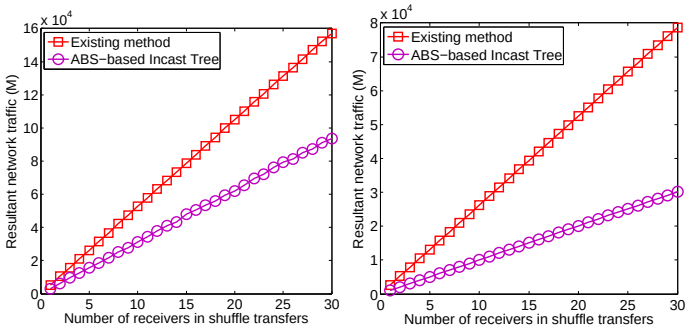
It is clear that our analysis in Section III-B and the simulations for large-scale incast in Section VI-B just tackle a special scenario when α=0. Another special scenario is that each key in one of such β data flows does not appear at other data flows, i.e., α=1. In such a special case, the data aggregation among the β data flows does not bring any gain. Such two special scenarios rarely happen in the practice. Therefore, we conduct extensive simulations to evaluate our ARS-based method in a more general scenario where $0 \le \alpha \le 1$.

Fig.7 reports the resultant network traffic for various settings of α under the different incast tree designing methods and settings of incast transfers in BCube(8,5). For each of the two incast transfers, our method always incurs much less network traffic than the other two methods when $\alpha < 1$, especially for small values of α. Such results demonstrate the gain of inter-flow data aggregation for more general incast transfers. Assume that the value of the random variable α follows a uniform distribution (if any). In such a setting, our ARS-based incast tree saves the resultant network traffic by 24% and 40%, compared to the existing method, when the incast



(a) An incast transfer with 500 senders. (b) An incast transfer with 4000 senders.

Fig. 7. The change trends of resultant network traffic along with the increase of aggregation factor under different settings of incast transfers in BCube(8,5).



(a) Random distribution of senders. (b) Managed distribution of senders.

Fig. 8. The resultant network traffic varies as the increasing number of receivers in shuffle transfers with 500 senders in BCube(8,5).

transfer involves 500 and 4000 senders, respectively. Thus, our ARS-based method outperforms other two methods in more general scenario of the aggregation ratio, irrespective of the size of incast transfer.

E. Impact of the distribution of incast members

In the state-of-the-practice, all members of an incast transfer are scheduled to idle servers in the scope of the entire data center. In such a case, the senders and receiver are somehow randomly distributed in a data center. The hamming distance between the receiver and each sender of the incast transfer in BCube(n,k) may reach $k+1$. One can image that such a random distribution makes the incast transfer occupy more data center resources and generate more network throughput. Although our method can significantly improve the performance of such an incast transfer, we extend it to involve incast transfers whose members are distributed in a managed manner.

The key idea is to find the smallest subnet BCube(n,k_1) in BCube(n,k) such that all members of an incast transfer satisfy their schedule constraints in such a subnet. In such a case, the hamming distance between the receiver and each sender of the incast transfer is at most $k_1+1\leq k+1$. The ARS-based incast tree in the scope of BCube(n,k_1) for such an incast transfer has at most k_1+1 stages. In theory, it thus occupies less data center resources and incurs less network traffic than prior ARS-based incast tree in the scope of BCube(n,k). We evaluate the impact of the two distribution schemes of all members in an incast transfer even a shuffle transfer with multiple incast transfers.

More precisely, we generate 30 shuffle transfers each of which has 500 senders but different number of receivers, ranging from 1 to 30. For each shuffle transfer, we apply the two difference distribution schemes to it's all members in a data center BCube$(8,5)$. Fig.8 reports the resultant network traffic of each such shuffle transfer under two different deployment schemes. We can see that both our method and the existing method result in less network traffic in the managed distribution scheme than that in the random distribution scheme. Moreover, our ARS-based method saves the network traffic by 62% and 24% on average compared to the existing method in the managed and random distribution schemes, respectively. Such results indicate that our ARS-based method can achieve more gains in the managed distribution scheme.

VII. CONCLUSION

Massively distributed computing frameworks like MapReduce transfer massive amount of data between their successive processing stages in data centers. In this paper, we aim to perform the inter-flow data aggregation during the transfer phase so as to decrease the resultant network traffic. To this end, such data transfers are normalized as the incast transfer that is modeled as building an incast minimal tree problem, an NP-hard problem. We propose the ARS-based method to build an efficient incast tree for any incast transfer so as to occupy less data center resources and incur lowest network traffic. We then present incremental methods to address the dynamic behaviors of incast members and the fault-tolerant issue of the incast tree. We evaluate the performance of our method and related work using a prototype implementation and large-scale simulations. The results demonstrate that our method can significantly reduce the network traffic and save data center resources compared to others. Moreover, our proposals for BCube and FBFLY can also be applied to other novel data centers after minimal modifications.

REFERENCES

[1] T. Condie, N. Conway, P. Alvaro, and J. M. Hellerstein, "Mapreduce online," in *Proc. 7th USENIX NSDI*, 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] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica., "Spark: Cluster computing with working sets," in *Proc. 2nd HotCloud*, Boston, MA, USA, 2010.

[6] 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.

[7] 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.

[8] 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. SIGCOMM*, Seattle, Washington, USA, 2008.

[9] 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. SIGCOMM*, Barcelona, Spain, 2009.

[10] 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.

[11] 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.

[12] 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.

[13] M. A. Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. SIGCOMM*, Seattle, Washington, USA, 2008.

[14] 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. SIGCOMM*, Barcelona, Spain, 2009.

[15] R. Mysore, A. Pamboris, and N. Farrington, "Portland: A scalable fault-tolerant layer 2 data center network fabric," in *Proc. SIGCOMM*, 2009.

[16] D. Li, C. Guo, H. Wu, K. Tan, Y. Zhang, S. Lu, and J. Wu, "Scalable and cost-effective interconnection of data-center servers using dual server ports," *IEEE/ACM Transactions on Networking*, vol. 19, no. 1, pp. 102–114, 2011.

[17] H. Abu-Libdeh, P. Costa, A. I. T. Rowstron, G. O'Shea, and A. Donnelly, "Symbiotic routing in future data centers," in *Proc. ACM SIGCOMM*, New Delhi, India, 2010, pp. 51–62.

[18] J. Kim, W. J. Dally, S. Scott, and D. Abts, "Technology-driven, highly-scalable dragonfly topology," in *Proc. 35th IEEE ISCA*, 2008, pp. 77–88.

[19] 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.

[20] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, J. Gerth, J. Talbot, K. Elmeleegy, and R. Sears, "Online aggregation and continuous query support in mapreduce," in *Proc. ACM SIGMOD*, Indianapolis, Indiana, 2010, pp. 1115–1118.

[21] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie, "Online aggregation for large mapreduce jobs," *PVLDB*, vol. 4, no. 11, pp. 1135–1145, 2011.

[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] C. Guo, G. Lu, Y. Xiong, J. Cao, Y. Zhu, C. Chen, and Y. Zhang, "Datacast: A scalable and efficient group data delivery service for data centers," Microsoft Research Asia, Haidian District, Beijing, China, Tech. Rep. MSR-TR-2011-76, Jun. 2011.

[24] S. Han, K. Jang, K. Park, and S. B. Moon, "Packetshader: a gpu-accelerated software router," in *Proc. of SIGCOMM*, New Delhi, India, 2010, pp. 195–206.

[25] J. Naous, G. Gibb, S. Bolouki, and N. McKeown, "Netfpga: reusable router architecture for experimental research," in *Proc. of PRESTO*, Seattle, WA, USA, 2008, pp. 1–7.

[26] D. Li, Y. Li, J. Wu, S. Su, and J. Yu, "Esm: Efficient and scalable data center multicast routing," *IEEE/ACM Transactions on Networking*, 2012.

[27] Y. Kwon, M. Balazinska, B. Howe, and J. A. Rolia, "Skew-resistant parallel processing of feature-extracting scientific user-defined functions," in *Proc. ACM SoCC*, 2010, pp. 75–86.

[28] E. Gonina, A. Kannan, J. Shafer, and M. Budiu, "Parallelizing large-scale data processing applications with data skew: a case study in product-offer matching," in *International Workshop on MapReduce and its Applications (MAPREDUCE)*, San Jose, CA, Jun 2011.

[29] B. H. J. R. YongChul Kwon, Magdalena Balazinska, "Skewtune: Mitigating skew in mapreduce applications," in *Proc. ACM SIGMOD*, May 2012.