SDCUP: Software-Defined-Control Based Erasure-Coded Collaborative Data Update Mechanism

WENJUAN PU, NINGJIANG CHEN, (Member, IEEE), AND QINGWEI ZHONG
School of Computer and Electronic Information, Guangxi University, Nanning 530004, China
Corresponding author: Ningjiang Chen (chnj@gxu.edu.cn)

This work was supported in part by the Natural Science Foundation of China under Grant 61762008, in part by the National Key Research and Development Project of China under Grant 2018YFB1404404, and in part by the Guangxi Natural Science Foundation Project under Grant 2017GXNSFAA198141.

ABSTRACT The storage system often uses erasure coding to provide the necessary fault tolerance. The erasure-coded update involves data transmission and data calculation of multiple nodes. Frequent updates will cause massive communication overhead. This paper mainly considers two issues: (1) In the scenario of frequent small-size updates, there are repetitive behaviors upon update, which causes bandwidth consumption to increase exponentially as the number of update nodes increases; (2) with the increase of data scale, there are local link busy phenomena caused by unbalanced use of links during update, which can prone to bottleneck links. In order to improve the inefficient update due to network bottlenecks. We propose SDCUP, a software-defined-control collaborative update mechanism that reduce update time for erasure-coded data with network load balance. Specially, SDCUP uses the software-defined control method to select the update transmission path according to the actual link load and adjust the data flow transmission rate by monitoring the degree of network load balance periodically. To further reduce the cross-rack update traffic, SDCUP unloads the calculation to the switch to realize the data aggregation in the rack, and it parallelizes sub-update operations to efficiently and cooperatively update. To evaluate the performance of SDCUP, we conduct simulation experiments on Mininet with real-world traces. The simulation results show that SDCUP can achieve a better load balance in multiple scenarios. Compared with the other data update schemes, the proposed method can improve the system throughput by up to 21% and reduce the update time by up to 47%.

INDEX TERMS Erasure coding, software-defined, data update, network load balance.

I. INTRODUCTION Based on the comprehensive trade-off of storage cost, bandwidth consumption, system load, and other factors, data center storage typically use erasure coding as a redundancy mechanism [1]. For example, Google applies RS erasure coding technology to a new generation of file system Colos-sus to achieve more economical reliability [2]. Facebook’s HDFS-RAID system saves storage overhead by introducing erasure coding into internal HDFS clusters [3]. Erasure coding provides higher storage efficiency for storage systems. Nevertheless, compared to replication, erasure-coded updates involve more data transmission, data calculation, and data write [4]. The fundamental reason is because of the complex nature of the data update mechanism of erasure coding. when the data block is updated, all the parity blocks associated with it must also be updated to maintain their consistency. As a result, erasure-coded updates need to consume CPU resources for encode calculations and network bandwidth for devices interaction, updated blocks download, and data blocks transfer. Therefore, the performance bottleneck of erasure-coded updates is mainly concentrated on network resources.

With the expansion of data volume and node scale in the data center, massive updates that occur frequently require higher bandwidth. The data updates become the primary consumers of network resources [1], [5]. The traffic in the erasure-coded data centers is vast and changes dynamically.
Because the access to each server node of the data update request is not entirely balanced, the link load is also unbalanced. The rich connection of partial links causes waste of resources, while the continuous overload of some links eventually evolves into network congestion, increasing the total latency of updates. Once link or node failure occurs during update, a large amount of cross-rack traffic will be generated upon data repair. Making matters even worse, the cross-rack bandwidth is typically oversubscribed and treated to be a scarce resource. Such tendency will have a great impact on system reliability and even cause permanent data loss. Therefore, the recent researches on erasure coding focus on mitigating data update overhead by proposing new codes with low update complexity [6], [7], data blocks layout strategy [8] and data update optimization algorithm [9]–[15].

A wide range of online applications require low-latency accesses under update-intensive workloads. Examples of these applications include Operational Data Store (ODS), Electronic Commerce (EC), and On-Line Transaction Processing (OLTP). However, the application scenarios with frequent small-size updates trigger intensive parity updates in erasure-coded storage. Two factors reduce the effectiveness of the prior work: (1) they do not fully excavate the correlation between update operations and optimize the update of each block in an independent fashion. and (2) they do not adequately consider the network state when assigning update blocks transfer tasks. Because of these reasons, with the number of update request increases, the bandwidth consumption increases significantly and the link load is imbalanced, which will lead to the degradation of update performance. Therefore, how to alleviate the network resource contention and load imbalance caused by frequent updates is a crucial issue for erasure-coded update optimization.

Recently, more and more data centers have deployed Software Defined Network (SDN) to improve the efficiency of processing data information. For example, Google and Microsoft have developed systems B4 [16] and SWAN [17] respectively to connect its data centers distributed around the world. This paper introduces SDN technology into the erasure-coded update, effectively and conveniently using the advantages of topology interconnection to improve the performance of update. SDN controller can automatically allocate and manage network resources and obtain information such as dynamic network topology changes and link busy state [18]–[21]. The SDN-based virtualization technology can speed up the calculation of routing information of data flow and implement the pre-judgment work to improve performance. The use of programmable switching technology to achieve the combination of data flow in transmission, which is helpful to reduce the amount of cross-rack traffic generated by the data updates.

In this paper, we introduce SDCUP, a Software-Defined-control based erasure-coded Collaborative Data Update Mechanism. It aims to reduce data update latency and mitigate the cross-rack update traffic. SDCUP uses the SDN controller to calculate the transmission path and dynamically adjust the speed of data flow injected into the network, which avoid link load imbalance and network congestion. Moreover, SDCUP advocates a concept called Collaborative Node Update, which groups the updated blocks and splits an update operation into intra-rack and cross-rack to mitigate the linear increase in update traffic with the expansion of data volume. Finally, we propose two general-purpose update schemes – SDCUP-R and SDCUP-I. Through theoretical analysis and experimental verification, SDCUP-R and SDCUP-I can achieve higher update performance and throughput than other update schemes. They have a good load balance under the different density of workload.

The main contributions of the paper are summarized as follows:

- Aiming at optimizing updates from both network transmission and encode calculation, we present a novel software-defined-control data update mechanism, SDCUP, which can mitigate cross-rack update traffic and maintain load balance of erasure-coded storage.
- As for system-wide balance, we design a grouping and path selection algorithm of updated blocks based on link load balance (GPUB), which is used to group updated blocks and dynamic adjust the transmission of updated flow by quickly determining the network status to maximize link sharing.
- In order to reduce cross-rack traffic, we propose two collaborative nodes data update implementations: collaborative reconstruct update (CRU) and collaborative incremental update (CIU). They aggregate blocks within the rack and split the update operation into multiple nodes for parallel collaborative computing.
- Applying SDCUP to the update operations of erasure coding, we implement two update schemes SDCUP-R and SDCUP-I. We then conduct extensive evaluations with them through simulation in Mininet and real-world traces to study performance benefits, efficiency, and throughput.

The rest of this paper is organized as the following: Section II briefly introduces the background and foundation of the erasure coding, as well as the work motivation; Section III describes the design of the SDCUP mechanism, then proposes a grouping and path selection algorithm of updated blocks based on link load balance, and two collaborative nodes data update schemes based on aggregate blocks in the rack; Section IV gives the experiments and the analysis of the experimental results; Section V introduces the related work; Section VI concludes the paper with future research directions.

II. BACKGROUND AND MOTIVATION

A. RS (N, K) CODE AND ITS DATA UPDATE MODE

RS(n, k) code is widely used in erasure coding technology [22]. RS(n, k) code divides the original data into k data blocks (denoted by d0, d1, · · · , dk−1), which are stored in the data nodes (denoted by D0, D1, · · · , Dk−1). k data blocks are encoded to form m(m = n − k) parity blocks (denoted by
\[ p_0, p_1, \ldots, p_{m-1} \]

which are stored in parity nodes (denoted by \( P_0, P_1, \ldots, P_{m-1} \)). The \( k \) original data blocks and the \( m \) dependent parity blocks form a stripe, which is distributed to \( n \) different storage nodes for node-level fault tolerance. When the number of temporary or permanent failure nodes does not exceed \( m \), the lost blocks can be reconstructed from any \( k \) remaining available data or parity blocks.

To deploy erasure coding in data centers, existing approaches [9], [23], [24], [32] mostly adopt hierarchical node placement by placing \( n \) nodes on \( r \) racks (\( r < n \)). We consider erasure-coded storage in a data center, and the bandwidth of its intra-rack and cross-rack links varies greatly. Multiple nodes within the same rack are connected to the top-of-rack switch (ToR) through the 10Gbps links, while multiple racks are connected through the network core composed of aggregation and core switches. Since other tasks in the system compete for network resources, the cross-rack bandwidth is usually oversubscribed and considered scarce, but the intra-rack links enjoy sufficient bandwidth [23]. Figure 1 describes an example of a data center architecture using RS (10,6) code. The encoding and decoding computation of erasure coding usually brings high cross-rack transfer overhead, which results in performance bottleneck of data update. Our goal is exploit the property of hierarchical node placement to minimize the cross-rack traffic triggered by erasure-coded update operations.

Compared with the direct coverage update mode of replication, the erasure-coded update is more complicated. It can be seen from Figure 1 that in order to ensure the consistency of data, when any data block in a stripe is updated, it is necessary to reacquire and calculate all the parity blocks associated with that data block, which adds other overhead such as calculation, network transmission, scheduling, disk I/O and so on. There are two main ways to update the parity blocks in \( RS(n, k) \) code: whole stripe update and partial stripe update. When the data block \( d_s \) is updated to \( d_s' \):

1) WHOLE STRIPE UPDATE
The update node \( N_{\text{update}} \) reads all the original data blocks \( d_i \) that are not updated in the same stripe and the updated block \( d_s' \). \( N_{\text{update}} \) re-encodes \( d_s' \) and \( d_i \) using Equation (1) to obtain new parity blocks.

\[
p_s' = a_{j,s}d_s' + \sum_{i=0, i \neq s}^{k-1} a_{j,i}d_i
\]

2) PARTIAL STRIPE UPDATE
The update node \( N_{\text{update}} \) reads all old parity blocks \( p_j \) and data blocks \( d_s \) in the same stripe. The increment is calculated by the difference between the original data block \( d_s \) and the updated block \( d_s' \). \( N_{\text{update}} \) obtains the new parity blocks through \( p_j \) and the increment, using Equation (2).

\[
p_s' = p_j + a_{j,s} (d_s' - d_s)
\]

The encoding and decoding calculation of the RS code is defined in the Galois field arithmetic. The addition in the Galois field is XOR operation, so the result of the addition of multiple data blocks is the same size as that of one data block. In the actual update process, the number of the updated block in each stripe may be single or multiple. There may be more than one stripe that need to be updated. Nevertheless, note that a single update operation is within the stripe; the updates between the different stripes are independent. That is to say, the update operations to access different data blocks in the same stripe can be performed simultaneously. Multiple updates to access different stripes can be executed in parallel, and they do not affect each other. This provides the basis for the optimization method in this paper.

B. MOTIVATION

The erasure-coded updates require a lot of communication overhead and complex computing processing under the update-intensive workloads. At the same time, in sequentially performing update operations, there are data transmission overlaps, calculation redundancy, and disk repeated reading, resulting in an exponential growth of network transmission traffic and disk I/O. As a result, long update latency is hard to be avoided. However, this did not attract enough attention in previous work. We illustrate this problem using a simple example in Figure 2. Update requests \( R_1, R_2, R_3 \) and \( R_4 \) access four different data blocks on the same stripe, all arriving time within \( 0 - T \), and each update takes \( T \). In Figure 2(a), it takes a total of \( 4T \) to complete the four requests in sequence, and the parity nodes on the stripe are updated four times. Since \( R_1, R_2, R_3 \) and \( R_4 \) all access the same stripe, according to Section II-A, they can be executed in parallel. As shown in Figure 2(b), the client waits for \( T \) to receive \( R_1, R_2, R_3 \) and \( R_4 \). Then client sends them in the next \( T \). The process is carried out in parallel by synthesizing four small updates into one large update. Finally, it takes \( 2T \) to complete four update requests. The parity nodes on the stripe are updated only once. In fact, the data update is more intensive than the example in practice. The first way will further deteriorate the situation with long latency and heavy I/O overhead. The second way will improve the situation. Thus, a new way of sending updated blocks to related nodes in parallel would have great audience.

Besides, regardless of whole stripe update or partial stripe update, two common problems will occur during the update process in the application scenario where small update requests are frequent: single-node bottleneck and link load imbalance. Our solution to this problem is motivated by one key observation: during the data update process, network traffic is aggregated on the uplink of the client and the data node with massive computation, forming link overload; otherwise, when the link is busy, there are more data flows related to the data node at the lower end of the link, which may cause single node overheated. The updated data transmission between nodes is not only affected by the nodes load but also limited by the bandwidth of the link. However, most of the current research work focuses on the optimization of static algorithms for data transmission, so there are the following two deficiencies: (1) The dynamic adjustment of possible.
network state changes is not efficient. The data transmission can only wait or be retransmitted when the bottleneck link is busy. (2) The transmission path selection method based on distance vector does not consider the actual problems such as network state and link cost. Due to the reasons above, the previous work has not completely solved the problem of single node overheat and link overload in data update of erasure coding.

To address the aforementioned issues, we consider the update problem in a typical hierarchical network architecture and propose SDCUP that fuses software-defined network technology with erasure coding for efficient data update (see Section III.A). More specifically, we exploit the SDN centralized controller to predict the busy link in advance and periodically detect the network load balance. The usage of SDN addresses the transmission inefficiencies associated with erasure-coded update (see Section III.B). The other hand, the programmable switch performs partial update calculations to aggregate the data blocks within the rack, which can reduce cross-rack traffic through parallel collaborative updates (see Section III.C). It should be pointed out that the main focus of this paper is to improve the throughput of the whole system during the update process, not to increase the execution speed of a single task.

III. DESIGN OF SDCUP

A. THE SDCUP FRAMEWORK

The update process of the RS code mainly includes two parts: data transmission and encode calculation. The design idea of SDCUP is to group update requests that frequently come, select idle paths to transmit the update groups, and schedule the update groups according to the degree of network load balance. By splitting a single update calculation, multiple nodes in the update group are updated concurrently, and different update groups are updated in parallel.

The SDCUP framework is mainly composed of two parts: Storage System with RS(n,k) code and SDN Network. The RS code storage system is built on an SDN network. The SDN network connects its nodes of various roles through OpenFlow switches. The data flow generated by data updates is monitored and managed by the SDN controller.

1. The storage nodes of the RS code storage system are used to store data blocks and parity blocks. They are deployed in units of the rack. The storage location of the data and the redundant backup are stored in the metadata node.

2. The SDN network consists of two parts: SDN Controller and Network of Switches. The network of switches includes a hierarchical switch cluster that implements the OpenFlow protocol standard and the links between them. The programmable switches on the top of the rack support simple XOR operation. The SDN controller controls and monitors the switch cluster through the application module based on the OpenFlow protocol.

Figure 3 demonstrates the updated workflow of the SDCUP mechanism. The SDCUP mechanism is essentially composed of two consecutive stages.
Updated Blocks Transfer Stage. After receiving the updated blocks, the client initiates a remote procedure call (RPC) request to the metadata node (step 1), and the metadata node feeds back the corresponding information to the client after receiving the request (step 2). The client groups the updated blocks and requests updates from the controller (step 3). The controller calculates the transmission path and schedules the updated blocks by monitoring the network link load to achieve multi-path traffic balance. The controller installs the flow table item indicating the forwarding status on the OpenFlow switches and establishes a connection with them (step 4). The OpenFlow switches return successful connection (step 5). Finally, the controller returns status to the client (step 6).

Update Calculation Stage. The client sends the updated block (in update group) to the network of switches in parallel. The switches transmit updated blocks to the corresponding node according to the forwarding path stored in the flow table (step 7). In the rack, ToR performs partial update calculation—sum (XOR) the received updated blocks; outside the rack, ToRs exchange data through the network of switches (step 8-9). Finally, the storage nodes send ACK to the client to complete the update request (step 10).

The SDCUP mechanism includes two key algorithms to be designed: Grouping and Path Selection of Updated Blocks Based on Link Load Balance for updated blocks transfer and Collaborative Nodes Data Update Based on Aggregate Blocks in Rack for update calculation, which will be respectively introduced in Sections III-B and III-C. The parameters and symbols used in the SDCUP mechanism are illustrated in Table I.

### Table 1. Parameters and symbols.

| Parameters & Symbols | Description |
|----------------------|-------------|
| $W_c$               | Receive window on the Client (Byte) |
| $W_s$               | Send window on the Client (Byte) |
| $G(V, E)$           | The dynamic topological flow graph of network, $V$ consists of network nodes, $E$ consists of all links between nodes |
| $path$              | The available transfer path |
| $path. bw$          | The bandwidth of the path (Mbps) |
| $path. load$        | The background load in a link on path (Mbps) |
| $RTT$               | Round Trip Time (ms) |
| $\varphi_{path}(t)$| The congestion coefficient of path at time $t$ |
| $N$                 | The number of overall links in the network |
| $MSS$               | Maximum Segment Size (Byte) |
| $\delta(t)$         | The variance of $N$ links load at time $t$ (Mbps) |
| $load_{ij}(t)$      | The link load of link $(i, j)$ at time $t$ (Mbps) |
| $g_g$               | The $g$th update group |
| $d_i$               | The $i$th date node, $i \in \{0, 1, \ldots, k - 1\}$ |
| $p_j$               | The $j$th parity node, $j \in \{0, 1, \ldots, m - 1\}$ |
| $r_x$               | The $x$th top-of-rack switch |
| $d_i$               | The $i$th original data block, $i \in \{0, 1, \ldots, k - 1\}$ |
| $d'_i$              | The $i$th updated block, $i \in \{0, 1, \ldots, k - 1\}$ |
| $p'_j$              | The $j$th updated parity block, $j \in \{0, 1, \ldots, m - 1\}$ |
| $a_{ij}$            | The encoding coefficient from $d_i$ to $p_j$ |
| $p_x^*$, $p_y^*$    | The partially update intermediate block |

### B. GROUPING AND PATH SELECTION OF UPDATED BLOCKS

The data transmission path has great impact on update performance. In the case of transmitting the same amount of data, the path that can make full use of the network bandwidth is likely to complete the data transmission faster. In this part, we propose a Grouping and Path Selection Algorithm of Updated Blocks Based on Link Load Balance, referred to as GPUB. By mining the association between multiple updates,
we group the updated blocks and compute the best path for them. Specifically, GPUB processes each updated block by the following steps.

First, after receiving the update requests, the client interacts with the metadata node to obtain the metadata information such as IP address and parity location of the updated blocks that arrives within the receive window time $W_r$. According to the grouping rule (i.e., the update operations that access different nodes in the same stripe are grouped into an update group), the updated blocks are divided into $g$ groups ($G_1, G_2, \cdots, G_g$). The updated blocks that are not grouped will be prioritized in the next $W_r$.

Second, according to the link load information collected by the SDN controller (these are maintained in the background of the controller with the network dynamic topology flow graph $G(V, E)$, where $V$ is mainly composed of the switch nodes, $E$ is all links between the nodes, and link load is the edge weight), the controller calculates the best path for update flow, and installs the flow entries in the OpenFlow switches along the path.

Finally, the update groups are sent to network in parallel by the client within the send window time $W_s$. The size of $W_s$ is dynamically adjusted by the degree of network load balance during update.

Figure 4 shows the flowchart for the execution process of the GPUB algorithm. To achieve isolation from traditional networks, the GPUB uses the Virtual Local Area Network (VLAN) flag in the OpenFlow protocol. By matching the two fields of VLAN id and VLAN priority, specific flow (data flow) is transmitted according to the specified path (the path calculated by the controller) without affecting the existing network functions.

![FIGURE 4. The flowchart of GPUB algorithm.](image)

Algorithm 1 shows the overall procedure of GPUB, using the updated blocks as input. A structure $update\_block\_list$ stores the updated blocks received within $W_r$. As shown in line 3 of Algorithm 1, the updated blocks in the list are grouped to form the $updatingGroupSet$. We first need function $DynamicAdjustmentOfW_s$ periodically calculates $newW_s$ (line 5). Subsequently, function $BestPathSelect$ uses topology flow graph $G(V, E)$ to calculate the transmission path of the update flow (line 8). Finally, the update groups are transmitted in parallel within $W_s$ (line 10). If all groups send destination successfully, the update operation will return true, otherwise it will return false and roll back.

### Algorithm 1 GPUB Algorithm

**Input:** The updated blocks $d'_1, d'_2, \cdots, d'_f, G(V, E)$.

**Output:** $flag$ //a boolean value indicating whether the update transmission is successful or not.

1. $update\_block\_list \leftarrow \{d'_1, d'_2, \cdots, d'_f\}, flag = false$
2. $foreach W_r do$
3. $ updatingGroupSet = DivideUpdatingGroup(update\_block\_list)$
4. $end for$
5. $newW_s = DynamicAdjustmentOfW_s(W_s, \delta^*)$
6. $foreach newW_s do$
7. $ foreach G_i in updatingGroupSet do$
8. $ best_path = BestPathSelect(G(V, E), G_i)$
9. $ end for$
10. $ ParallelSendBlocks(updatingGroupSet, best_path)$
11. $ get flag = true$
12. $end for$

The GPUB algorithm contains two key functions: (1) $BestPathSelect$ that selects the optimal transmission path to achieve the purpose of load balance, and (2) $DynamicAdjustmentOfW_s$ that dynamically adjusts the size of $W_s$ to adapt to the changes of network load actively. Next, the two key functions are described as follows in detail.

1. $function BestPathSelect(G(V, E), G_i)$
2. $ foreach f \in G_i do$
3. $ R_f \leftarrow GetAvailablePathSet(G(V, E), f)$
4. $ foreach pathR_f do$
5. $ calculate \phi_{path}(t) of path$
6. $ end for$
7. $ \phi_{min}(t) = min(\phi_{path}(t))$
8. $ best_path \leftarrow utilization\_info(\phi_{min}(t))$
9. $end for$
10. $ return best_path$

The function $BestPathSelect$ provides the best paths for transfer update groups. We define the best path from source $s$ to destination $d$ as the least congested of all paths from $s$ to $d$. Once the client sends a request to the SDN controller to trigger the update. The controller finds the set of available paths for each update group separately through LLDP polling and OpenFlow statistic polling (line 3). The different SDN controllers can use link discovery protocols like [19] to create and maintain their topology flow graph $G(V, E)$. This method is very efficient for the SDN controllers to discover the underlying topology. All the optional paths are polled to calculate the path’s congestion coefficient $\phi(t)$ using equation (3) (line 5). After comparing the congestion coefficients of these paths, the controller selects path with the smallest congestion coefficient $\phi(t)$ as the best path (lines 7-8). The above action is done with the idea of offline increasing first
The function `DynamicAdjustmentOfW_s(W_s, δ*)` is used to dynamically adjust the size of `W_s` according to the change of network state. This function consists of three parts: data collection, network state judgment and window size adjustment. The controller collects the real-time links load through periodic polling of all switches in the network with OpenFlow `ofp_port_stats` queries, similar to the approaches presented in [26]. The controller uses the equation (4) to calculate `δ(t)`. Let `δ(t)` be the variance of all link loads at time `t` (line 2). By comparing with the threshold `δ*` (the value of `δ*` in a given network structure is determined by the actual environment measurement), the controller evaluates the current network load balance degree (lines 3, 7); finally, according to the multiplicative decrease and additive increase of the congestion avoidance mechanism, dynamically increase or decrease the `W_s` to control the sending rate of the update flow (lines 5, 8).

\[
δ(t) = \frac{\sum_i^{N} [load_{i,j}(t) - load_{i,j}(t)]^2}{N}
\]

where `load_{i,j}(t)` represents the load on the link `(i, j)` at time `t`, `N` is the total number of all links in the network, `load_{i,j}(t)` represents the average load of `N` links at time `t`. In the function, `MSS` represents maximum segment size.

In the above function, the value of `δ(t)` is inversely proportional to the network load balance, i.e., the larger `δ(t)` value is, the more unbalanced the link load is. `δ(t) ≥ δ*` indicates network congestion. The controller sends a message to the client to reduce the send window. Client decreases `W_s` to reduce the update flow rate injected into the network. `δ(t) < δ*` indicates that the network load is balanced. Client increases `W_s`, so that updated blocks are transmitted faster.

The GPUB algorithm chooses the actual link bandwidth as the equalization object when selecting the transmission path. Also, the transmission rate of the update flow is dynamically adjusted to avoid the link from being too heavy or too light, which can effectively reduce the network delay. The GPUB algorithm preprocesses updated blocks, which provides the basis for the parallelization of the next update calculation. However, how to encode multiple nodes in same update group in parallel? In section III-C, we propose a Collaborative Nodes Data Update Based on Aggregate Blocks in Rack to effectively handle the simultaneous update of multiple nodes within an update group.

C. COLLABORATIVE NODES DATA UPDATE BASED ON AGGREGATE BLOCKS IN RACK

In the update calculation stage, when the updated data nodes and parity nodes are located in different racks, a large amount of cross-rack traffic will be generated. One solution is to replace repeated cross-rack paths with intra-rack transmissions. A ToR is ideally positioned to aggregate data as the data must be forwarded to the TOR before being transmitted to the cross-rack link. Therefore, we use two mechanisms of programmable switches in SDN – the read/write of the specified memory address and XOR operation to aggregate the blocks in rack. We propose a Collaborative Nodes Data Update Based on Aggregate Blocks in Rack method, which aims to make full use of the bandwidth in the rack and reduce the amount of data transmission across the rack.

- The calculation of update group.

Considering that the updating processes between stripes are independent of each other, this section focuses on the analysis of updates on one stripe without loss of generality. First, the update of the update group is described as: Suppose that a stripe has `k` data nodes `{D_0, D_1, \cdots, D_{k-1}}` distributed on rack `R_k`, and `m` parity nodes `{P_0, P_1, \cdots, P_{m-1}}` distributed on rack `R_m`. There are `h` updated blocks `{d_0, d_1, \cdots, d_{h-1}}` (where `0 ≤ h ≤ k`) in group update `G` waiting for update, and the updated blocks are distributed on rack `R_h`. The remaining `k - h` data blocks `{d_h, d_{h+1}, \cdots, d_{k-1}}` are not updated.

To update each parity block `p_j` into `p_j'` (where `0 ≤ j ≤ m`), we can generalize Equation (1) as Equation (5):

\[
p_j' = \sum_{i=0}^{k-1} a_{j,i}d_i + \sum_{i=0}^{h-1} a_{j,i}d_i'
\]

We can generalize Equation (2) as Equation (6):

\[
p_j' = p_j + \sum_{i=0}^{h-1} a_{j,i} (d_i' - d_i)
\]

where `a_{j,i}(0 ≤ i ≤ k − 1, 0 ≤ j ≤ m − 1)` is the encoding coefficient used by `d_i'` for the new parity block `p_j'`, `r_k`, `r_m` and `r_h` are the top switches of racks `R_k, R_m, R_h` respectively.

Based on Equations (5) and (6), we propose two implementations of Collaborative Nodes Data Update Based on Aggregate Blocks in Rack: Collaborative Reconstruct Update (CRU) and Collaborative Incremental Update (CIU). For each update group: In the CRU scheme, `D_t` computes the linear combination of the data block and the encoding
coefficient to get encoded block, and then transmits it to $r_k$. $r_k$ sums the encoded blocks in the same rack to obtain the intermediate calculation result (partially update), and then transmits them to $r_m$ in parallel. $r_m$ sums the intermediate calculation results received from other racks to get new parity blocks, and then transmits them to $P_j$ for update; In the CIU scheme, $D_i$ computes increment, $r_j$ sums the increments (partially update) to get the intermediate calculation results, and then transfers them to $r_m$ in parallel. $r_m$ sums the intermediate calculation results sent by other racks to get the encoded blocks, and then sends them to $P_j$. $P_j$ sums the old parity block and encoded block to get a new parity block and updates itself.

Through the data interaction between the racks and the cooperation between the nodes, the update of multiple nodes is completed synchronously, which can effectively reduce the update overhead. Note that the size of the intermediate calculation result obtained from the partial update is the same as that of one data block, so that it can reduce the amount of data transmission across the rack. We use RS (10,6) as an example to describe their update process, as shown in Figure 5.

![Collaborative Reconstruct Update](image1)

![Collaborative Incremental Update](image2)

**FIGURE 5.** An example of Collaborative Reconstruct Update and Collaborative Incremental Update with RS (10,6).

In Figure 5, it is a RS (10,6) storage system, in which 10 storage nodes $\{D_0, \cdots, D_5, P_0, \cdots, P_3\}$ placed on 4 racks. The nodes in each rack are connected through programmable switches $\{r_0, r_1, r_2, r_3\}$ on the top of the rack. 6 data nodes $\{D_0, \cdots, D_5\}$ are placed in the rack of $\{r_0, r_1\}$ on average. 4 parity nodes $\{P_0, \cdots, P_3\}$ are placed in the rack of $\{r_2, r_3\}$. Below, we specify the detailed steps of updating the update group composed of updated blocks $\{d'_1, \cdots, d'_5\}$, whereas the same methodology applies to update group composed of other updated blocks.

- **The update process of CRU scheme**

CRU scheme uses Equation (5) to update (see Figure 5(a)).

1. Each data node to be updated $\{D_1, D_2, \cdots, D_5\}$ receives the updated blocks $\{d'_1, d'_2, \cdots, d'_5\}$ and completes its own data update. Each of the node $\{D_0, D_1, \cdots, D_5\}$ computes the linear combination of the data block and the encoding coefficients to get encoded block $da_{ij}(0 \leq j \leq 3, 1 \leq i \leq 5)$, and then sends the encoded blocks to the ToRs $r_0$ and $r_1$.

2. $r_0$ and $r_1$ sum the received encoded blocks (e.g., $r_0$ computes $p^*_0 = p^*_{0,0} + p^*_{0,1}$) to get new parity blocks $p^*_0, p^*_1, p^*_2$ and $p^*_3$. The parity nodes $P_0, P_1, P_2$ and $P_3$ update parity blocks.

- **The update process of CIU scheme**

CIU scheme uses Equation (6) to update (see Figure 5(b)).

1. Each data node to be updated $\{D_1, D_2, \cdots, D_5\}$ receives the updated blocks $\{d'_1, d'_2, \cdots, d'_5\}$ and completes its own data update. Each of the node $\{D_1, D_2, \cdots, D_5\}$ computes increment $a_{ij}(d'_j - d_j)(0 \leq j \leq 3, 1 \leq i \leq 5)$, and then sends the increment to the ToRs $r_0$ and $r_1$.

2. $r_0$ and $r_1$ sum the received increments (e.g., $r_0$ computes $p^*_{0,0} = a_{0,1}(d'_1 - d_1) + a_{0,2}(d'_2 - d_2)$) to get partially-updated intermediate calculation results $p^*_{i,j}(0 \leq j \leq 3, 0 \leq x \leq 1)$. Then $r_0$ and $r_1$ send the intermediate calculation results across the racks to the ToRs $r_2$ and $r_3$ of the rack where the parity nodes $P_0, P_1, P_2$ and $P_3$ are located.

3. $r_2$ and $r_3$ sum the intermediate calculation results received from other racks (e.g., $p^*_{0,0} = p^*_{0,0} + p^*_{0,1}$) to get new parity blocks $p^*_{0,0}, p^*_{1,0}, p^*_{2,0}$ and $p^*_{3,0}$, and then sends the encoded blocks to the parity nodes $P_0, P_1, P_2$ and $P_3$.

4. $P_0, P_1, P_2$ and $P_3$ sum the old parity blocks $p_{0,0}, p_{1,0}, p_{2,0}$ and $p_{3,0}$ and the corresponding encoded block $p^*_{0,0}, p^*_{1,0}, p^*_{2,0}$ and $p^*_{3,0}$ to get the new parity blocks $p_0, p_1, p_2$ and $p_3$ and update themselves.

CRU and CIU disassemble the update operation into multiple sub-update operations. The sub-update operations perform data transmission and data calculation in parallel to realize the efficient multi-node collaborative update in the update group and speed up the data update process. Using the XOR operation mechanism of the programmable switch, the partially-updated intermediate calculation results are aggregated in the rack to reduce the amount of data transmission across the rack. We next analyze the performance gain of CRU and CIU schemes.
Performance analysis

The critical difference between CRU and CIU is that the parity block’s update method is different, so the division of sub-updates is also different. CRU adopts whole stripe update mode. CIU adopts partial stripe update mode, so the cross-rack traffic generated by the CIU depends on the location of the updated blocks in the update group. When $R_k$ and $R_h$ are the same, the cross-rack traffic generated by the two schemes is the same. When the number of $R_k$ is greater than the number of $R_h$, CRU generates more cross-rack traffic than CIU.

By effectively organizing data transmission and calculation between racks, CRU and CIU can achieve better update performance in two aspects:

1. **Reduce cross-rack update traffic.** We use RS (10,6) example in Figure 5 for comparison. In the traditional RS update (see Figure 6(a)), the data nodes send data blocks to update node. The update node calculates new parity blocks and sends them to the parity nodes, and then the parity nodes update. If the size of a data block is $M$, the cross-rack traffic it generates is $(3 + 3 + 2 + 2) \times M = 10M$.

2. **Eliminate single-point overheating.** In Figure 6 (a), the update node has six blocks transmitted in the uplink and four blocks in the downlink. As a result, the network bandwidth of the update node is overloaded, and the network resources of other nodes are idle. Therefore, the network of the update node becomes the bottleneck of the system.

In Figure 6(b), the scheme in this paper decentralizes the update operation to data nodes, switch nodes, and parity nodes for parallel execution. The same number of blocks are transmitted on each link, which not only eliminates single point overheating but also relieves the network pressure on the bottleneck link.

In fact, when the number of racks is fixed, the larger $k$ (i.e., the more data nodes in the same stripe stored in each rack) is, the more obvious the optimization effect of the collaborative nodes data update based on aggregate blocks in the rack is.

### IV. EXPERIMENTS

#### A. EXPERIMENTAL ENVIRONMENT AND EXPERIMENTAL DESIGN

To verify the effectiveness of the software-defined-control based erasure-coded collaborative data update mechanism proposed in this paper, we conduct simulation experiments of the data update process on the Mininet simulation platform. The SDN controller is Open Network Operating System (ONOS) 1.13.1. ONOS is a software-based OpenFlow controller used for developing java-based software-defined networking control applications. It is modular and provides rapid development and prototyping. The Open vSwitch (OVS) 2.9.5 is used to simulate the OpenFlow switch. To ensure the unity of the experimental environment, the servers are also generated through simulation. We generate a batch of Docker container instances on the physical machine to build a distributed storage system HDFS-RAID [3]. HDFS-RAID internally implements RS code. The operating system running on the server is Ubuntu 18.04. The version of Docker is 19.03.8. Each container acts as a virtual server node, including a client node, a central control node, and the rest are storage nodes (datanodes/paritynodes). The system running on the physical server is Ubuntu 18.04, with two quad-core 2.4GHz Intel Xeon E5-260 CPU, 16G RAM. Figure 7 shows the experimental environment architecture.

- We implement the metadata node (NameNode) of HDFS-RAID on the central control node as an application module on the ONOS controller in SDN. The NameNode obtains the network parameters by calling the Restful API provided by ONOS.
- ONOS controls the route and forward of the underlying OVS. The exchange of information takes place via Http and OpenFlow protocols. In addition to the service modules that come with the ONOS such as the packet management, path discovery, flow rule management, and topology management, we expand the functions of the ONOS by adding routing control module and load balance module to realize the functions of data collection, load comparison, and traffic statistics of the GPUB algorithm.
- The middle part is responsible for the transmission of the underlying network data. The OVS connect Client and DataNodes/ParityNodes of HDFS-RAID. It mainly provides the route forwarding and flow monitoring functions of the virtual switch, both of which are natively integrated by OVS. In the experiment, we use the method...
in XORInc [27] to simulate the programmable switches’ XOR calculation function.

- Because the support function of the Host simulated by Mininet is relatively simple, we map it to the Docker cluster through NAT, such as DataNodes/ParityNodes and Client. Except for communicating with NameNode, the Client side implements the UpdatingGroup module to implement the function of GPUB algorithm. We implement multi-threading on the DataNodes/ParityNodes side to parallelize encoding calculation and network transmission to realize CRU and CIU schemes.

The scheme in this paper is compared with RCW and RMW to measure the optimization effect in terms of I/O load and update efficiency. Depending on the update calculation location, the scheme in this paper can be divided into two categories:

- **SDCUP-R**: the update scheme using the SDCUP framework performs update calculations through Collaborative Reconstruct Update (CRU).
- **SDCUP-I**: the update scheme using the SDCUP framework performs update calculations through Collaborative Incremental Update (CIU).

In the updated blocks transmission stage, an appropriate network load balance threshold \( \delta^* \) can accurately determine the network traffic status and adjust the rate of update flow. But if \( \delta^* \) is not set properly, not only does it not improve the update performance, but also causes performance degradation. Hence, we analyze the effect of \( \delta^* \).

**B. EXPERIMENTAL RESULTS AND ANALYSIS**

1) **I/O LOAD ASSESSMENT**

The I/O load is evaluated through the following two performance metrics.

- **Load Balancing Factor**: Let \( L(i) \) as the number of out-bound connections on switch \( i \) measured under each workload. It symbolizes the workload of forwarding data from the switch. \( L_{\text{max}} \) represents the maximum number of \( L(i) \), and \( L_{\text{min}} \) represents the minimum number of \( L(i) \). The load balancing factor \( LBF \) is defined as the ratio between \( L_{\text{max}} \) and \( L_{\text{min}} \), as shown in Equation (7).

\[
LBF = \frac{L_{\text{max}}}{L_{\text{min}}} \tag{7}
\]

The lower bound of \( LBF \) is 1 (i.e., \( L_{\text{max}} = L_{\text{min}} \)), which means that the system has reached the best load balance.

- **Throughput**: it is calculated by the amount of data updated divided by the cumulative update time.

The experiments are driven by real block-level tracking of MSR Cambridge Traces [30]. Because this paper only focuses on write requests, read requests do not involve the update process. In the experiments, we select a volume \( wdev_{-1} \) with a write rate of more than 99%. In order to measure the sensitivity of the update scheme to the workload density, the density of the update load is changed by increasing or decreasing the timestamp of the \( wdev_{-1} \), e.g., \( \times 0.5, \times 1.0, \times 1.5, \times 2.0, \times 2.5 \). Other workloads in the real production environment can be derived from their combination. In addition, we measure the impact of the updated block size on the I/O load of different update schemes under the update load density \( \times 1.0 \).

Figure 8(a) shows the relationship between the load balancing degree of RCW, RMW, SDCUP-R, SDCUP-I and the updated block size. We observe that as the block size decreases, the significant increase in the LBFs of RCW and RMW indicates that the network load is not balanced. This is because the amount of updated data is constant, the smaller the updated block, the more update requests are generated.

![Image of experimental system architecture diagram]
The increase in sequential writes has increased the load of data nodes and exacerbated the imbalance of network links. The LBFs of SDCUP-R and SDCUP-I at different updated block sizes are very close to 1, so they are balanced, which is in line with our expectations.

![Figure 8](image1.png)

**FIGURE 8.** Load Balancing Factor of RCW, RMW, SDCUP-R and SDCUP-I under different scenarios.

Figure 8(b) shows the relationship between the load balancing degree of RCW, RMW, SDCUP-R, SDCUP-I and the update load density. We find that the difference between the four schemes is not significant at 0.5X. Because the update injection is slow, the system resources are sufficient. As the load density doubles, due to severe contention for network resources and the bandwidth limitation of the bottleneck link, the LBFs of RCW and RMW continued to increase. The SDCUP-R and SDCUP-I benefit from the intelligent scheduling of the update groups and maintain a good balance under high load density.

In general, SDCUP-R and SDCUP-I achieve a good load balance in different scenarios.

![Figure 9](image2.png)

**FIGURE 9.** Throughput of RCW, RMW, SDCUP-R and SDCUP-I under different scenarios.

Figure 9(b) describes the relationship between the system throughput of RCW, RMW, SDCUP-R, SDCUP-I and the update load density. When the load density is small, the throughput of the four schemes is not much different. As the load density continues to increase, the difference between the various schemes becomes obvious. We see that SDCUP-R and SDCUP-I can well adapt to the gradual intensive update load. The increase of throughput is attributed to the controller’s routing planning based on the network state and parallel sub-update operations. However, RCW and RMW do not consider the link load status when transmitting data blocks. Therefore, under heavy update workload, the increase of throughput caused by the low link utilization is not much and even has a downward trend at 2.5X. The experiment is consistent with the theoretical analysis.
In general, the throughput of SDCUP-R and SDCUP-I is better than the other two schemes in different scenarios.

2) UPDATE EFFICIENCY ASSESSMENT

We evaluate the update efficiency by the **Update Time**, which is defined as the total duration from the update request to the end of the update. We consider three standard configurations of RS\((n,k)\) code in HDFS-RAID: (1) RS \((9,6)\) \([n = 9, k = 6, m = 3]\) (2) RS \((10,6)\) \([n = 10, k = 6, m = 4]\) (3) RS \((14,10)\) \([n = 14, k = 10, m = 4]\) and test the update efficiency of the four schemes under different cross-rack bandwidths. In actual production, the intra-rack bandwidth is 10Gb/s, the cross-rack bandwidth is 1Gb/s [24].

Figure 10 describes the relationship between the update time and cross-rack bandwidth of the RS codes of the above three fault-tolerant parameters using RCW, RMW, SDCUP-R, and SDCUP-I. Compared with RS \((9,6)\) and RS \((10,6)\), RS \((9,6)\) using SDCUP-I at 1Gb/s has 31.3% less update time than RS \((9,6)\) using RCW, RS \((10,6)\) using SDCUP-I has 37.5% less update time than RS \((10,6)\) using RCW. Because the increase in \(m\) value lead to increase the data transmission. However, the schemes in this paper are superior to other schemes because the data blocks are aggregated in the rack, which reduces the amount of data transferred. Comparing RS \((10,6)\) and RS \((14,10)\), at 0.5Gb/s, RS \((10,6)\) uses SDCUP-I to reduce the update time by 35.5% compared to RCW. RS \((14,10)\) uses SDCUP-I to reduce the update time by 47.2% compared to RCW. This is because the \(k\) value increases and fewer update groups make the update time shorter. The results are consistent with the analysis in Section III-C.

By comparing the update time of different schemes under 0.5Gb/s, 1Gb/s, and 2Gb/s, we find that the effect gains of SDCUP-I and SDCUP-R increase as the cross-rack bandwidth decreases (i.e., it is more limited by the cross-rack bandwidth). For example, for RS \((14,10)\), when the cross-rack bandwidth is 2Gb/s, the update time of SDCUP-I is 27.5% and 26.7% less than that of RCW and RMW respectively; when the cross-rack bandwidth is reduced to 0.5Gb/s, the improvement is increased to 47.2% and 46.5%. When the cross-rack bandwidth is limited, the reduction of cross-rack update traffic is more conducive to improve the update performance of the proposed schemes in this paper.

Since the timestamp of wdev_1 was changed in the experiments, the average update time can better measure the impact of update load density on update efficiency. Figure 11 describes the relationship between the average update time of RCW, RMW, SDCUP-R, SDCUP-I and the update load density. It is observed in the figure that the four schemes work well under light load and the average update time is short. As the load density increases, the network link becomes crowded. The update efficiency of the four schemes decreases. Compared with RCW and RMW, the average update time of SDCUP-R and SDCUP-I increases slowly. This is because the actual link load is used as the balancing object when selecting the transmission path, which effectively uses idle links to improve the network link utilization.

The average update time of SDCUP-R and SDCUP-I under high load increases slowly, indicating that they are more effective than other schemes in balance link traffic.

Figure 12(a) shows the relationship between the update time of RCW, RMW, SDCUP-R, SDCUP-I and the data block size. When the data block in the stripe is too large or too small, the performance of SDCUP-I and SDCUP-R will decrease slightly. When the data block is too small (less than 4MB), accessing a block with a small amount of data on the stripe requires calling a large number of programs, which increases the "stripe overhead". When the data block is too large (larger
than 256MB), the parallelism of the single block update in the proposed scheme is reduced. Therefore, when the data block size is between the two sizes, the update efficiency is optimal.

3) EFFECT OF THE NETWORK LOAD BALANCE THRESHOLD

Finally, we analyze how the network load balance threshold $\delta^{*}$ affects the scheme’s update performance in the updated blocks transmission phase. In Figure 13, only SDCUP-I is shown because the key difference between SDCUP-R and SDCUP-I is the location of the update calculation. In the updated blocks transmission stage, their operations are the same, so the SDCUP-I with a better effect is selected to analyze the effect of $\delta^{*}$ on the update performance. We set 5 values for $\delta^{*}$: 0.01, 0.1, 1.0, 2.0, 5.0, and 10.0, and fix the cross-rack bandwidth to 0.5Gb/s. For comparison, the results of the RCW scheme are included as the baseline in the figure. For different values of $\delta^{*}$, the results of the RCW scheme remain unchanged. Figure 13 shows the error bars of the maximum and minimum values in the experiments.

We observe from Figure 13 that when the threshold $\delta^{*}$ is small (smaller than 0.1), the client may send too few update groups, resulting in insufficient utilization of the bottleneck link. The update time of the scheme in this paper is higher than the baseline; otherwise, when the threshold $\delta^{*}$ is large (larger than 5.0), the utilization rate of bottleneck link is higher, but the accumulation of the update groups causes the update time to increase accordingly; when the threshold $\delta^{*}$ is between the two values, the update efficiency is the highest. Therefore, we make a trade-off between the update time and link utilization, choose the appropriate threshold $\delta^{*}$ to ensure the system’s quality of service.
V. RELATED WORK

The relevant work on application software-defined network technology to optimize storage networks and improve data update performance in erasure-coded storage mainly includes the following different aspects of work.

The SWAN architecture proposed by Microsoft [17] uses software-defined network technology to connect data centers, effectively improving network utilization and reducing end-to-end transmission delay and packet loss rate. By deploying OpenFlow-based centralized traffic engineering services in the data center WAN, Google B4 [16] increase link utilization to nearly 100%. Mayflower [28] uses the network measurement information collected by software-defined network technology for copying and network path selection to improve read performance. The CAU [24] relocates updated blocks together into the same stripe to reduce the data update overhead. Some researches focus on reducing the consumption of cross-rack bandwidth. Hu et al. [23] propose a hierarchical block placement strategy for placing multiple data blocks per rack in the repair framework DoubleR. By searching for suitable relay nodes within the rack to aggregate the data blocks, cross-rack traffic can be minimized.

DR-Update [9] is a two-layer relay update scheme, following a similar idea, which introduces rack-level and intra-rack relays to minimize cross-rack update traffic. T-Update [4] transfers traffic from a bottleneck link to a link with sufficient bandwidth to reduce update delay. In contrast, SDCUP maps the update operation to multiple nodes, and the sub-update operations are parallel. SDCUP completes data aggregation operations at ToR, which reduces cross-rack bandwidth consumption.

In summary, SDCUP differs from the current work of using static algorithm to select the transmission path, which can calculate the transmission path and adjust the transmission rate in time according to dynamic feedback of network load. Therefore, the proposed scheme improves link utilization and avoids congestion caused by frequent updates. In addition, the current schemes reduce cross-rack traffic mainly by finding a relay node in the rack for data block aggregation. SDCUP uses the nature of hierarchical topology to implement data block aggregation at the ToR, which can reduce network core communication and save the process of finding a relay node. Therefore, SDCUP has a faster update speed. To the best of our knowledge, we believe SDCUP is the first work that uses software defined network technology to optimize erasure-coded data transmission structure.

VI. CONCLUSION

This paper introduces SDCUP, a collaborative data update mechanism that improves the data update performance of erasure coding in a hierarchical data center. Based on SDN technology to control the global view of the network, SDCUP makes use of redundant paths in the network to transmit updated blocks. It dynamically adjusts the update flow rate based on network load. Accordingly, this paper designs the GPUB algorithm to improve the bottleneck link workload caused by the update request access. In addition, SDCUP effectively maps update calculations to multiple different nodes for parallel execution. Through the data interaction between nodes to reduce data transmission and encode pressure, this paper proposes two effective collaborative nodes data update calculation schemes – CRU and CIU, that effectively reduce cross-rack update traffic. The simulation experiments show that the proposed method is effective in improving system update throughput and reducing update time. In the future work, since SDCUP only considers the link load when selecting the transmission path, the calculation of link weights under multiple constraints will be studied in order to establish a dynamic cost optimization path. Another future work is to consider maintaining a low-load path pool for each rack to reduce the computational cost of the controller.

REFERENCES

[1] J. C. Chan, Q. Ding, P. P. Lee, and H. H. Chan, “Parity logging with reserved space: Towards efficient updates and recovery in erasure-coded clustered storage,” in Proc. 12th USENIX Conf. File Storage Technol., 2014, pp. 163–176.
[2] J. C. Corbett, J. Dean, and M. Epstein, “Spanner: Google’s globally-distributed database,” in Proc. 10th USENIX Conf. Operating Syst. Design Implement. (OSDI), New York, NY, USA, 2012, pp. 251–264.
[3] GitHub. (2017). Facebook Archive/Hadoop-20. Accessed. Oct. 12, 2017. [Online]. Available: https://github.com/facebookarchive/hadoop-20
[4] X. Pei, Y. Wang, X. Ma, and F. Xu, “T-update: A tree-structured update scheme with top-down transmission in erasure-coded systems,” in Proc. 35th Annu. IEEE Int. Conf. Comput. Commun. (INFOCOM), San Francisco, CA, USA, Apr. 2016, pp. 1–9.

[5] Y. Hu, Q. Li, W. Xie, and Z. Ye, “An ant colony optimization based data update scheme for erasure-coded storage systems,” IEEE Access, vol. 8, pp. 118696–118706, 2020.

[6] Y. Fu and J. Shu, “D-code: An efficient RAID-6 code to optimize I/O loads and read performance,” in Proc. IEEE Int. Parallel Distrib. Process. Symp., Hyderabad, India, May 2015, pp. 603–612.

[7] Y. Zhang, C. Wu, J. Li, and M. Guo, “TIP-code: A three independent parity code to tolerate triple disk failures with optimal update complexity,” in Proc. 45th Annu. IEEE/ACM Int. Conf. Dependable Syst. Netw., Río de Janeiro, Brazil, Jun. 2015, pp. 136–147.

[8] P. Subedi, P. Huang, B. Young, and X. He, “FINGER: A novel erasure coding scheme using fine granularity blocks to improve Hadoop write and update performance,” in Proc. IEEE Int. Conf. Netw., Archit. Storage (NAS), Boston, MA, USA, Aug. 2015, pp. 255–264.

[9] M. Deng, S. Yu, X. Xiao, F. Liu, and Z. Chen, “DR-update: A dual-level relay scheme in erasure-coded storage systems for balanced updates,” in Proc. IEEE 15th Int. Conf. Dependable, Autonomic Secure Comput., 15th Int. Conf. Pervasive, Intell. Comput., 3rd Int. Conf. Big Data Intell. Comput. Cyber Sci. Technol. Cong. (DASPC/Con/DataCom/CyberSciTech), Orlando, FL, USA, Nov. 2017, pp. 1150–1159.

[10] Y. Li, J. Zhou, W. Wang, and Y. Chen, “RE-store: Reliable and efficient KV-store with erasure coding and replication,” in Proc. IEEE Int. Conf. Cluster Comput. (CLUSTER), Albuquerque, NM, USA, Sep. 2019, pp. 1–12.

[11] J. Shen, K. Zhang, J. Gu, Y. Zhou, and X. Wang, “Efficient scheduling for multi-block updates in erasure coding based storage systems,” IEEE Trans. Comput., vol. 67, no. 4, pp. 573–581, Apr. 2018.

[12] F. Zhang, J. Huang, and C. Xie, “Two efficient partial-updating schemes for erasure-coded storage clusters,” in Proc. IEEE 7th Int. Conf. Netw., Archit. Storage, Xiamen, Fujian, Jun. 2012, pp. 21–30.

[13] K. S. Esmaili, A. Chinhia, and A. Datta, “Efficient updates in cross-object erasure-coded storage systems,” in Proc. IEEE Int. Conf. Big Data, Silicon Valley, CA, Oct. 2013, pp. 28–32.

[14] Q. Luo and Y. Wang, “UHUM: An efficient hybrid update mechanism in distributed storage systems with erasure coding,” in Proc. IEEE 23rd Int. Conf. Comput. Supported Cooperat. Work Design (CSCWD), May 2019, pp. 158–163.

[15] Y. Wang, X. Pei, X. Ma, and F. Xu, “TA-update: An adaptive update scheme with tree-structured transmission in erasure-coded storage systems,” IEEE Trans. Parallel Distrib. Syst., vol. 29, no. 8, pp. 1893–1906, Aug. 2018.

[16] S. Jin, A. Kumar, S. Mandal, J. Ong, and L. Poutievski, “B4: Experience with a globally-deployed software defined WAN,” in Proc. SIGCOMM, 2013, pp. 3–14.

[17] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, “Achieving high utilization with software-driven WAN,” in Proc. ACM SIGCOMM Conf. SIGCOMM (SIGCOMM), 2013, pp. 15–26.

[18] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Malzi, “ZL-update: Updating data center networks with zero loss,” in Proc. ACM SIGCOMM Conf. SIGCOMM (SIGCOMM), 2013, pp. 411–422.

[19] A. Nehra, M. Tripathi, M. S. Gaur, R. B. Battula, and C. Lal, “TILAK: A token-based prevention approach for topology discovery threats in SDN,” Int. J. Commun. Syst., vol. 32, no. 17, p. e3781, Nov. 2019.

[20] T. Holterbach, E. C. Molero, and M. Apostolaki, “Blink: Fast connectivity recovery entirely in the data plane,” in Proc. 16th USENIX Symp. Networked Syst. Design Implement. (NSDI), Boston, MA, USA, 2019, pp. 161–176.

[21] I. L. Awan, N. Shah, and M. Imran, “WITHDRAWN: An improved mechanism for flow rule installation in In-band SDN,” J. Syst. Archit., vol. 96, pp. 32–51, Jun. 2019.

[22] A. Chinhia and A. Mangur, “Dynamic erasure coding policy allocation (DECPA) in Hadoop 3.0,” in Proc. 6th IEEE Int. Conf. Cyber Secur. Cloud Comput. (CSCloud/6th IEEE Int. Conf. Edge Comput. Scalable Cloud (EdgeCom)), Paris, France, Jun. 2019, pp. 29–33.

[23] F. Zhang, J. Huang, and C. Xie, “Optimal repair layering for erasure-coded data centers: From theory to practice,” ACM Trans. Storage, vol. 13, no. 4, p. 33, 2017.

[24] Z. Shen and P. C. C. Lee, “Cross-Rack-Aware updates in erasure-coded data centers,” in Proc. 47th Int. Conf. Parallel Process., New York, NY, USA, Aug. 2018, pp. 1–10.