Hindawi
Wireless Communications and Mobile Computing
Volume 2022, Article ID 5392474, 15 pages
https://doi.org/10.1155/2022/5392474

Research Article

A Load-Aware Multistripe Concurrent Update Scheme in Erasure-Coded Storage System

Junqi Chen,[1,2] **Yong Wang**,[1,2] **Miao Ye**,[3,4] **Qinghao Zhang**,[3] **and Wenlong Ke**[3,4]

[1]*School of Computer Science and Information Security, Guilin University of Electronic Technology, Guilin 541004, China*
[2]*Guangxi Engineering Technology Research Center of Cloud Security and Cloud Service, Guilin University of Electronic Technology, Guilin 541004, China*
[3]*School of Information and Communication, Guilin University of Electronic Technology, Guilin 541004, China*
[4]*Ministry of Education Key Lab. of Cognitive Radio and Information Processing, Guilin University of Electronic Technology, Guilin 541004, China*

Correspondence should be addressed to Yong Wang; ywang@guet.edu.cn

Received 24 January 2022; Accepted 7 May 2022; Published 19 May 2022

Academic Editor: Ashish Bagwari

Copyright © 2022 Junqi Chen et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Erasure coding has been widely deployed in today's data centers for it can significantly reduce extra storage costs while providing high storage reliability. However, erasure coding introduced more network traffic and computational overhead in the data update process. How to improve the efficiency and mitigate the system imbalance during the update process in erasure coding is still a challenging problem. Recently, most of the existing update schemes of erasure codes only focused on the single stripe update scenario and ignored the heterogeneity of the node and network status which cannot sufficiently deal with the problems of low update efficiency and load imbalance caused by the multistripe concurrent update. To solve this problem, this paper proposes a Load-Aware Multistripe concurrent Update (LAMU) scheme in erasure-coded storage systems. Notably, LAMU introduces the Software-Defined Network (SDN) mechanism to measure the node loads and network status in real time. It selects nonduplicated nodes with better performance such as CPU utilization, remaining memory, and I/O load as the computing nodes for multiple update stripes. Then, a multiattribute decision-making method is used to schedule the network traffic generated in the update process. This mechanism can improve the transmission efficiency of update traffic and make LAMU adapt to the multistripe concurrent update scenarios in heterogeneous network environments. Finally, we designed a prototype system of multistripe concurrent updates. The extensive experimental results show that LAMU could improve the update efficiency and provide better system load-balancing performance.

1. Introduction

The scale of the distributed storage system is rapidly expanding to deal with the proliferation of the global datasphere. Meanwhile, the node failures and data loss caused by various reasons are increasing, such as system crashes, natural disasters, hacker attach, and power outages [1–3]. To avoid irreversible losses caused by these threats and improve the reliability of the storage system, the redundancy mechanism is indispensable in data centers. The two most typical redundancy mechanisms are *replications* and *erasure coding*. Replications copy each chunk of original data to other storage devices to improve the system redundancy. However, it considerably incurs extra storage costs, especially in today's data scale explosion and growth. As another alternative, erasure coding can provide better storage efficiency via encoding computations, meeting the same degree of fault tolerance as replications [4]. Specifically, erasure coding divides the original data into several data chunks, and then, these data chunks are encoded into a few redundant chunks (also called parity chunks). These data chunks and parity chunks together form an erasure-coding stripe. When data failure occurs, as long

as the number of failed chunks does not exceed the recovery threshold, the lost chunks can still be recovered from the living chunks. Since the erasure coding can significantly reduce the extra storage cost while providing high storage reliability, it has been widely deployed in today's data centers, such as Facebook [5], Azure [6], and Google GFS [7].

However, while providing high reliability with less extra storage cost, erasure coding introduces more network traffic and computation overhead during the data update process. When the data chunk is updated, all the parity chunks in the same stripe should be updated simultaneously to maintain the consistency of the stripe, which boosts the disk I/O load and the update time. In addition, various real trace analyses show that over 90% of writing in the storage system is data update [8–10], indicating that data update is prevalent. If the data failure occurs during the update process, the system cannot recover the failure data correctly. Therefore, the update efficiency of erasure coding affects not only the performance but also the reliability of the distributed storage system.

There are two major challenging factors impacting the erasure-coding update. Challenge 1 is the heterogeneity of the storage node and network status. For example, the storage nodes purchased in different periods during the expansion of storage system have different performance [11, 12]. Meanwhile, these storage nodes may also be processing various tasks in real time, such as MapReduce [13] and system heartbeat and data migration [14], making the status of network links dynamic and heterogeneous. In this case, the computational load and traffic caused by the update may significantly impact the system performance and reduce the update efficiency. Challenge 2 is the multistripe concurrent update. Due to the potential correlation between the data of each stripe [15, 16], the update of one erasure-coding stripe will result in the contemporary update of multiple correlation stripes [15], which amplifies the node load and the update time. Therefore, how to improve the update efficiency of erasure code storage and guarantee the system load balance is still a critical problem. However, the existing update scheme ignores the node and network heterogeneity and only focuses on the single stripe update scenario, which cannot sufficiently deal with the problems of update efficiency declines and system load imbalance caused by the multistripe concurrent update.

This paper proposed a Load-Aware Multistripe concurrent Update (LAMU) scheme. As we will explain in Section 3, LAMU adopts a centralized update architecture in which the data update is divided into the data-delta convergence, parity-delta computation, and parity-delta divergence. The centralized update architecture can mitigate the system overhead by preventing the separate connection between the data node and the parity node. Firstly, we introduce Software-Defined Networking (SDN) to measure and collect node load information (such as CPU utilization, residual memory, disk I/O load, and node access bandwidth) and network status (such as network topology, link residual bandwidth, and link transmission delay) in real time. Secondly, based on the node load information, we select the nonrepetitive computing nodes with a lower load for each update stripe. Finally, the TOPSIS method is used to tailor the best path for data-delta convergence and parity-delta divergence for each update stripe. The decision-making factor uses diverse weights for different network load scenarios so that LAMU could be suitable for various environments.

The main contributions of this paper can be summarized as follows:

(1) Aiming to solve the problem that existing research cannot sufficiently deal with the efficiency decline of multistripe concurrent updates, this paper first establishes the optimization model of multistripe updates with multiple QoS constraints in the heterogeneous environment. The update efficiency can be improved by minimizing the cumulative weighted update delay of multistripe updates and balancing the link utilization. To the best of our knowledge, this is the first work attempt to improve the efficiency of multistripe concurrent updates with multiple QoS constraints

(2) This paper introduces SDN to perceive the node load status and network status of the erasure-coded storage system in real time and proposes a Load-Aware Multistripe concurrent Update (LAMU) scheme. LAMU selected the nonrepetitive computing nodes with better capacity for each update stripe. Then, the TOPSIS method is used to schedule the update traffic generated in the update process to improve the efficiency of multistripe updates. As far as we know, this is the first work that considers the heterogeneity of nodes and network status simultaneously during the erasure-coding update process

(3) We designed a prototype system of multistripe concurrent updates based on Containernet [17] to verify the effectiveness of LAMU. The extensive experimental results show that LAMU could improve the erasure-coding update efficiency and maintain better system load balancing

The rest of this paper is organized as follows: Section 2 presents the background and related work of the erasure-coding update. Section 3 describes the multistripe update problem in the erasure-coded system and provides the optimization model. Section 4 introduces the details of our LAMU scheme. We conduct extensive experiments to evaluate LAMU in Section 5. Section 6 concludes this paper.

2. Background and Related Work

2.1. Basics of Erasure Coding. In this paper, we concentrate on a well-known erasure code called the Reed-Solomon (RS) codes [18], which are widely used in today's commercial data centers [7]. To be precise, the system configures the RS codes with two parameters k and r and denote the code by RS $(k+r, k)$ codes. In RS $(k+r, r)$ codes, the original data D are divided into k data chunks $\{d_1, d_2, \cdots, d_k\}$, and these k data chunks are encoded to r parity chunk $s\{p_1, p_2, \cdots, p_r\}$ through the linear operation of equation (1).

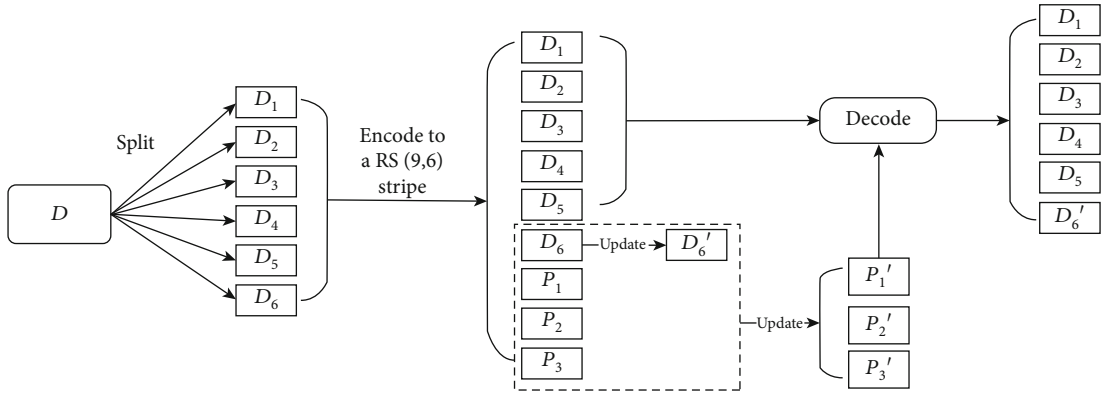

Figure 1: Example of RS (9, 6).

These $k + r$ chunks distributed in different nodes of the storage system form an erasure code stripe S.

$$p_i = \sum_{j=1}^{k} c_{i,j} * d_j, \tag{1}$$

where $c_{i,j}$ is the conversion coefficient from d_j to p_i, $1 \le i \le r, 1 \le j \le k$. According to the linear characteristics of equation (1), as long as the number of surviving chunks in the stripe is larger than k, any k chunks can reconstruct the whole stripe.

Figure 1 depicts the process of encoding, updating, and decoding of RS (9, 6). First, the system divides the original data D into 6 data chunks $\{D_1, D_2, D_3, D_4, D_5, D_6\}$, and the data chunks are encoded by equation (1) to get 3 parity chunks $\{P_1, P_2, P_3,\}$; these 9 chunks form an erasure code stripe. When the data chunk D_6 is updated to $D_6{}'$, the 3 parity chunks will be synchronously updated to $\{P_1{}', P_2{}', P_3{}'\}$. In decoding, through the linear operation of equation (1), the whole stripe can be reconstructed from any 6 surviving chunks (such as $\{D_1, D_2, D_3, D_4, D_5, P_1{}'\}$).

As we can see from Figure 1, in the data update process of erasure coding, when the data chunk is updated, all the parity chunks in the same stripe must also be updated simultaneously to maintain the consistency of the stripe. Based on whether the complete data chunks need to be transmitted, there are 2 classes of update framework: *full-stripe* update and *delta-based* update. In the full-stripe update, the data chunk D_j (where $1 \le j \le k$) is updated to D_j', and then, equation (1) is used to calculate the new parity chunk P_i' (where $1 \le i \le r$), which needs to transmit the whole chunks and consumes significantly large network resources. In the delta-based update, we can update each parity chunk P_i into P_i' by equations (2a) and (2b):

$$\Delta P_i = \sum_{j=1}^{m} c_{i,j} * \left(D_j' - D_j\right), \tag{2a}$$

$$P_i' = P_i + \Delta P_i, \tag{2b}$$

$$1 \le i \le r, 1 \le j \le m. \tag{2c}$$

To elaborate further, when m data chunks $\{D_1 \cdots D_j \cdots D_m\}$ are updated to $\{D_1' \cdots D_j' \cdots D_m'\}$, each data chunk sends the data-delta $D_j' - D_j$ to the computing node, which calculates ΔP_i based on equation (2a) and then distributes ΔP_i to parity nodes to complete the whole update process. Clearly, the delta-based update can save network resources and improve the update efficiency compared with the full-stripe update.

2.2. Related Work. As mentioned above, the data update is prevalent in the storage system. It has a significant impact on the performance of the distributed storage system. Therefore, various update schemes have been proposed to improve the erasure-coding update efficiency in recent years. T-Update [19] builds the minimum update tree using the Prim [20] algorithm to deal with the single-node update problem, but T-Update neglects the network status during construction of the update topology, which is prone to cause network congestion when the system load is high. TA-Update [21] adds a rollback-based failure handle method based on T-Update, making the update process more adaptive. To cope with the problem of multiple-node update in erasure coding, PUM-P [22] first proposed the centralized update architecture that collects the data-delta Δd to the middle node close to the data nodes and distributes the Δp by the random choice route path. Although the PUM-P can reduce the connection number between the data node and the parity node, PUM-P ignores the heterogeneity of nodes when selecting the compute node and ignores the link status when scheduling the update traffic. In order to improve the data transmission efficiency of multiple-node updates, ACOUS [23] constructs an update tree that considers the link delay provided by the commercial cloud service provider, which reduces the multiple-node update time. But ACOUS also neglects the heterogeneity of nodes when selecting the compute node, and it is difficult to obtain the delay parameter from the service provider in common storage clusters.

The work mentioned above improves the update efficiency by optimizing the data update process. Shen et al. [15] proposed the CASO that solves this problem by organizing data chunks with high correlation into the same

stripes to reduce the update traffic. Specifically, CASO mines the correlation of different stripes from the real storage system work trace [16] and then organizes the highly correlated data into the same stripe to reduce the number of concurrent update stripes and improve the update efficiency. CASO can only mitigate the correlation between stripes, but it cannot entirely eliminate the correlation of stripes. Consequently, the multistripe concurrent updates are still frequently triggered by association stripes, especially in the storage system that the stripe is organized without consideration for data correlations. Therefore, improving the multistripe concurrent update efficiency and maintaining the system load balance are still very challenging tasks.

In summary, most of the existing update schemes of erasure codes only focus on the single stripe update scenario and ignore the heterogeneity of the node and network status, which cannot sufficiently deal with the problems of low update efficiency and load imbalance caused by the multistripe concurrent update. To solve these problems, this paper introduces SDN and multiattribute decision-making methods and proposes the Load-Aware Multistripe concurrent Update (LAMU) scheme in heterogeneous erasure-coded storage systems.

3. Model and Formulation of the Multistripe Concurrent Update Problem

In this section, we first state the multistripe concurrent update problem in the erasure-coded system. Our motivation is to find the best computing nodes, convergence path, and divergence path for each strip. These computing nodes and route paths are combined to form an update forest. Then, we give the optimization model of multistripe updates with multiple QoS constraints in the heterogeneous environment.

3.1. Problem Statement. Figure 2 shows a simple distributed erasure-coded storage system including several racks, in which each rack contains multiple storage nodes and each node stores many chunks from diverse erasure-coding stripes. For example, S_1d_1 denotes the first data chunk of stripe S_1 and S_1p_1 denotes the first parity chunk of stripe S_1. We can see from Figure 2 that the 4 data chunks and 4 parity chunks have been distributed in different racks or nodes in the system.

We use the centralized update architecture similar to PUM-P [22], which reduces the connection between the data and parity nodes by introducing middle computing nodes. We take the update process of 4 data chunks in stripe S_1 described in Figure 2 as an example: Firstly, stripe S_1 updates data chunks $S_1d_1, S_1d_2, S_1d_3, S_1d_4$ to $S_1d_1', S_1d_2', S_1d_3', S_1d_4'$ and then converges the data-delta $S_1\Delta d_j = (S_1d_j' - S_1d_j)$ to the computing node selected by the controller. Secondly, the computing node calculates the parity-delta ΔP_i by equation (2a). Finally, the computing node distributes the ΔP_i to corresponding parity nodes.

The detailed mathematical model of multistripe concurrent updates with multiple QoS constraints in a heterogeneous environment is introduced in Section 3.2. The network topology of an erasure-coded storage system can be modeled by a graph $G(V, E)$, in which V presents the set of switches and E denotes the set of links between adjacent switches. For easy reference, the notations used in this section are shown in Table 1.

3.2. Problem Formulation

3.2.1. Cumulative Weighted Update Delay for Multistripe Update. The first objective function is aimed at minimizing the cumulative update delay of M stripes defined as f_1 in equation (3a). Specifically, the update delay of stripe m is defined as d^m_{update} in equation (3b), which is composed of (a) the data-delta convergence delay $d^m_{\text{convergence}}$, (b) the parity-delta compute delay d^m_{compute}, and (c) the parity-delta divergence delay $d^m_{\text{divergence}}$.

$$\min f_1 = \sum_{m \in M} d^m_{\text{update}}, \tag{3a}$$

$$d^m_{\text{update}} = d^m_{\text{convergence}} + d^m_{\text{compute}} + d^m_{\text{divergence}}. \tag{3b}$$

(1) The Data-Delta Convergence Delay. The data-delta convergence delay $d^m_{\text{convergence}}$ is formulated as follows:

$$d^m_{\text{convergence}} = \sum_{i \in N_m} \max_{p \in i} \{d_{mp}\} x_{mi}, \tag{4a}$$

$$\text{s.t.} \sum_{m \in M} b_m x_{m\,i} \le \min_{e \in i} \{b_e\}, \quad \forall \text{i} \in N_m, \tag{4b}$$

$$\sum_{i \in N_m} x_{mi} = 1, \quad \forall m \in M, \tag{4c}$$

$$x_{mi} \in \{0, 1\}, \quad \forall m \in M, \forall i \in N_m, \tag{4d}$$

where N_m denotes the set of all possible convergence paths from the updated data nodes to the computing node of stripe m. i is an element in the set N_m, and it is composed of multiple point-to-point path p from data nodes to computing node. $\max_{p \in i}\{d_{mp}\}$ denotes the convergence delay of stripe m selecting i as the convergence path. The constraint in formula (4b) ensures that the total bandwidth requirement for all convergence paths through path i does not exceed the bottleneck bandwidth. The constraint (4c) is met to ensure that only one divergence path will be assigned to stripe m. In constraint (4d), x_{mi} denotes a binary variable: it is 1 if stripe m selects i as the convergence path and 0 otherwise.

(2) The Parity-Delta Computing Delay. This section adopts the definition of node computing capacity in the erasure-

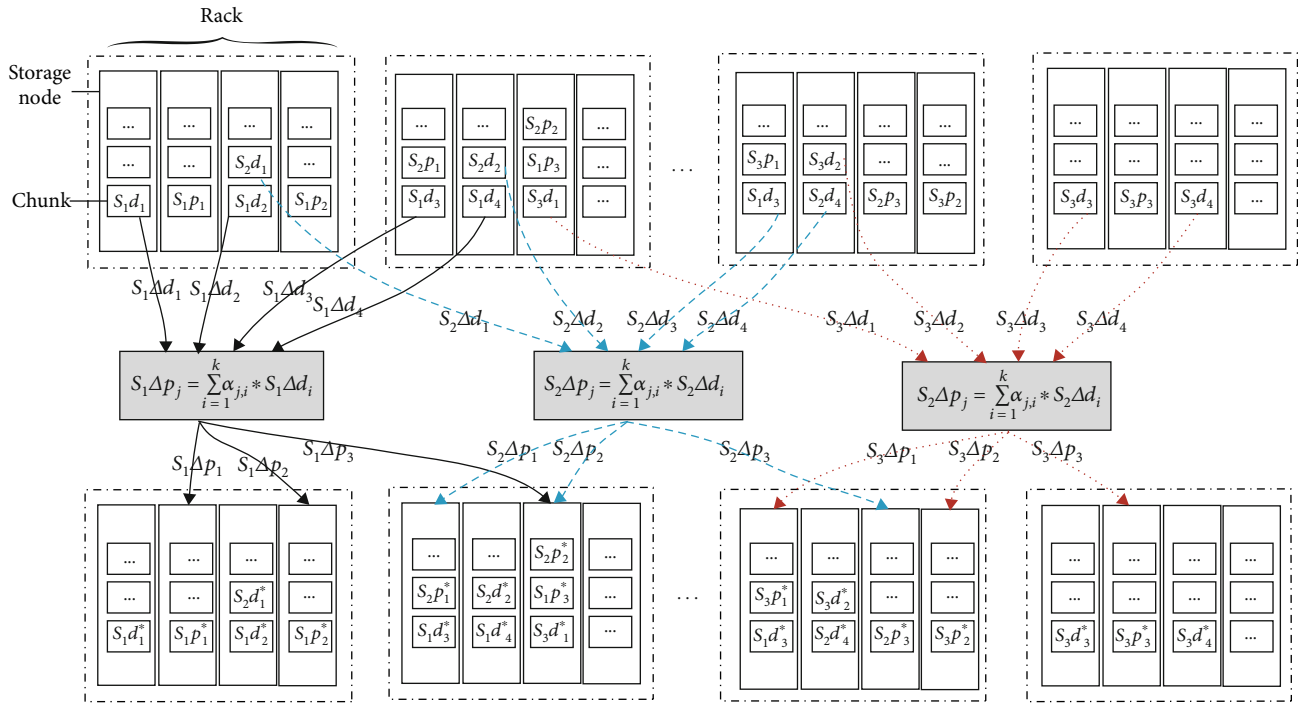

Figure 2: Example of multistripe concurrent update in erasure-coded storage system.

Table 1: Symbols in the problem formulation.

Symbol	Description	Symbol	Description
G	Network topology	E	Link set
V	Switch set	M	Number of update stripes
m	Update stripe index	e	Link between two adjacent switches
N_m	Convergence paths set of update stripe m	L_m	Divergence paths set of update stripe m
i	Convergence path	j	Divergence path
p	Unicast path in convergence path i	q	Unicast path in divergence path j
d_{mp}	Convergence delay of stripe m via path p	d_{mq}	Divergence delay of stripe m via path q
b_m	Bandwidth cost of update stripe m	b_e	Bandwidth capacity of link e
x_{mi}	Binary variable	x_{mj}	Binary variable
N	Number of node load decision factors	D_m	Update data volume of stripe m

coded system proposed by Fenglin et al. [24]. It uses the sequence $c_1, c_2, \cdots, c_k, \cdots, c_n$ to denote the factors which affect the processing ability of a node, such as CPU utilization, remaining memory, and disk I/O, and the corresponding weight factors are $\varphi_{1,}, \varphi_2, \cdots, \varphi_k, \cdots, \varphi_n$. Therefore, the computing capacity of each node in the erasure-coding update can be expressed as

$$\text{Capacity}_m = \sum_{k=1}^{N} c_k \varphi_k. \tag{5a}$$

It is assumed that D_m represents the update volume of stripe m; the parity-delta computing delay d^m_{compute} is formulated as follows:

$$d^m_{\text{compute}} = \gamma \frac{D_m}{\text{Capacity}_m} = \gamma \frac{D_m}{\sum_{k=1}^{N} c_k \varphi_k}, \tag{5b}$$

where γ indicates the capacity conversion coefficient.

(3) The Parity-Delta Divergence Delay.

$$d^{m}_{\text{divergence}} = \sum_{j \in L_m} \max_{q \in j} \{d_{mq}\} x_{mj}, \tag{6a}$$

$$\text{s.t.} \sum_{m \in M} b_m x_{m\,j} \le \min\{b_e\}, \quad \forall j \in L_m, \tag{6b}$$

$$\sum_{j \in L_m} x_{mj} = 1, \quad \forall m \in M, \tag{6c}$$

$$x_{m\,j} \in \{0, 1\}, \quad \forall m \in M, \forall j \in L_m, \tag{6d}$$

where L_m denotes the set of all possible divergence paths from the computing node to the parity nodes of stripe m. j is an element in the set L_m, and it is composed of multiple point-to-point path q from the computing node to parity nodes. $\max_{q \in j}\{d_{mq}\}$ denotes the divergence delay of stripe m selecting j as the divergence path. The constraint in formula (6b) ensures that the total bandwidth requirement for all convergence paths through path j does not exceed the bottleneck bandwidth. The constraint (6c) is met to ensure that only one divergence path will be assigned to the stripe m. In constraint (6d), x_{mj} denotes a binary variable: it is 1 if stripe m selects j as the convergence path and 0 otherwise.

(4) The Proposed Objective Function. According to formulas (3)–(6), the objective function of the cumulative weighted delay of the multistripe update can represent as

$$\min \quad f_1 = \sum_{m \in M} \left(\sum_{i \in N_m} \max_{p \in i} \{d_{mp}\} x_{mi} + \gamma \frac{D_m}{\sum_{k=1}^{N} c_k \varphi_k} + \sum_{j \in L_m} \max_{q \in j} \{d_{mq}\} x_{mj} \right). \tag{7}$$

3.2.2. Network Load-Balancing Performance for Multistripe Update. While improving the update efficiency, the load balance of the network is also critical. The objective function of minimizing the maximum link bandwidth utilization is defined as follows:

$$\min \quad f_2 = \max_{e \in E} \sum_{m \in M} \frac{b_m x_{me}}{b_e}, \tag{8a}$$

$$\text{s.t.} \sum_{m \in M} b_m x_{me} \le b_e, \quad \forall e \in E, \tag{8b}$$

$$x_{me} \in \{0, 1\}, \quad \forall m \in M, \forall e \in E. \tag{8c}$$

The constraint (8b) ensures that the used bandwidth of the link e cannot be in excess of the link capacity; x_{me} is a binary variable for the link selection of the update traffic.

3.3. The Proposed Multiobjective Optimal Model of Multistripe Update. Our goal is to minimize the cumulative weighted delay of multistripe updates denoted as f_1 and minimize the maximum link bandwidth utilization represented as f_2. However, it is difficult to achieve the minimum values of f_1 and f_2 at the same time. The overall objective function of this paper is defined as follows:

$$\text{minimize } Z = [f_1, f_2], \tag{9a}$$

$$\text{s.t.} \sum_{m \in M} b_m x_{m\,i} \le \min_{e \in i}\{b_e\}, \quad \forall i \in N_m, \tag{9b}$$

$$\sum_{m \in M} b_m x_{m\,j} \le \min_{e \in j}\{b_e\}, \quad \forall j \in L_m, \tag{9c}$$

$$\sum_{m \in M} b_m x_{me} \le b_e, \quad \forall e \in E, \tag{9d}$$

$$\sum_{i \in N_m} x_{mi} = 1, \quad \forall m \in M, \tag{9e}$$

$$\sum_{j \in L_m} x_{mj} = 1, \quad \forall m \in M, \tag{9f}$$

$$x_{mi} \in \{0, 1\}, \quad \forall m \in M, \forall i \in N_m, \tag{9g}$$

$$x_{mj} \in \{0, 1\}, \quad \forall m \in M, \forall j \in L_m, \tag{9h}$$

$$x_{me} \in \{0, 1\}, \quad \forall m \in M, \forall e \in E. \tag{9i}$$

4. SDN-Based Load-Aware Multistripe Concurrent Update Scheme

To solve the objective functions (9), we propose the LAMU scheme. Figure 3 presents the system architecture of LAMU, which includes four main modules: the Node Monitor (NodeM) module, Network Monitor (NetM) module, Compute Node Selection (CNS) module, and Path Selection (PS) module. In the process of LAMU, seeking the best computing node and transmission path for the multistripe erasure code data update is briefly described as follows: Firstly, the NodeM and NetM modules update the real-time node load information and network information. Then, the CNS module is used to select the computing nodes for update stripes according to the network and node status. Last, LAMU employs the PS module to find an appropriate convergence path between data nodes and the compute node and an appropriate divergence path between the compute node and parity nodes. The combination of the computing node, convergence path, and divergence path forms the update tree. Multiple update trees constitute an update forest.

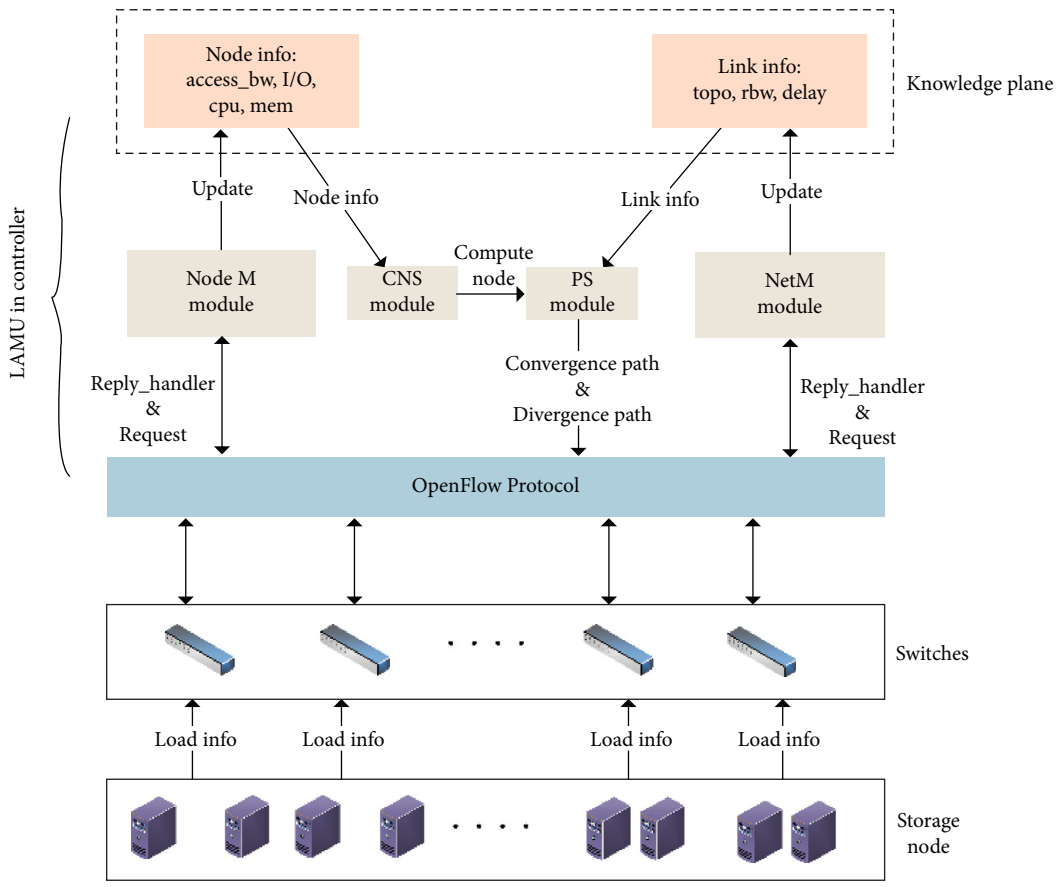

Figure 3: The architecture of LAMU.

4.1. The NodeM Module and NetM Module. The Software-Defined Network (SDN) can significantly simplify the network configuration and alleviate the measurement overhead compared with the traditional network architecture. For example, SDN can provide a flexible and efficient monitor strategy through the centralized control plane. In the LAMU scheme, the NodeM and NetM modules interact with switches through the OpenFlow protocol of SDN to discover the global network topology. It updates the load information of the storage node and the network link status in real time to provide a knowledge plane for LAMU. The node load information recorded by the NodeM is as follows:

$$C_n = \{c_1, c_2, \cdots, c_k\}, \tag{10}$$

$$M_n = \{m_1, m_2, \cdots, m_k\}, \tag{11}$$

where C_n denotes the set of CPU utilization of each storage node and M_n denotes the set of the residual memory capacity of each storage node. Both the basic functions of compute nodes and the calculation of parity increment require CPU and memory resources.

$$L_n = \{l_1, l_2, \cdots, l_k\}, \tag{12}$$

where L_n denotes the set of the I/O load of each node. The I/O load performance represents the reading and writing performance of the storage node. Since computing nodes receive and forward data involving data reads and writes, a more accurate node selection weighting factor can be obtained by considering the I/O load.

$$A_n = \{a_1, a_2, \cdots, a_k\}, \tag{13}$$

where A_n denotes the set of access bandwidths of each storage node. In the scenario of the multistripe concurrent update, the compute node, as the convergence node of Δd and the divergence node of p, has a relatively larger demand for access bandwidth. Therefore, larger access bandwidth means less possibility of congestion, thus improving the overall update efficiency.

The set of CPU utilization C_n, residual memory capacity M_n, and I/O load L_n can be obtained by periodically requesting status information from storage nodes. The set of the node access bandwidth can be calculated by using the SDN-based network measurement method in our previous work [25].

The NetM module follows the OpenFlow protocol of SDN to obtain the global network topology and update the real-time network information. The network information obtained by the NetM module is as follows:

$$P = \{p_1, p_2, \cdots, p_n\}, \tag{14}$$

where P is the set of point-to-point paths between all nodes of each stripe. This path set can be obtained by the Dijkstra [26] algorithm. P is calculated during LAMU initialization, and P can be accessed directly in the subsequent path calculation, without repeated path calculation, which reduces the cost of the algorithm.

$$B_p = \{b_1, b_2, \cdots, b_n\}, \tag{15}$$

where B_p represents the set of residual bandwidths for each path P. It can be calculated using the SDN-based network measurement method in our previous work [27].

$$D_p = \{d_1, d_2, \cdots, d_n\}, \tag{16}$$

where D_p represents the set of transmission delays for each path. It can be obtained by using the SDN-based network measurement method [27].

$$H_p = \{h_1, h_2, \cdots, h_n\}, \tag{17}$$

where H_p denotes the hops of data from the start node to the end node. It can be calculated from the length of each path P.

We use NodeM and NetM to obtain the storage node load and discover the network global status information mentioned above (10)–(17), which provides data support for the following computing node selection, aggregation path, and divergence path selection.

4.2. The CNS Module. As shown in Figure 3, LAMU selects nonduplicate computing nodes with better performance for multiple stripes by the CNS module. Firstly, when computing nodes are assigned to multiple update stripes, it is necessary to avoid numerous stripes selecting the duplicated computing node. Otherwise, the efficiency of parity-delta computing will be reduced and network congestion will occur. Secondly, according to Section 3.2, the parity-delta computing efficiency is positively correlated with the computing capacity of nodes, so the load status of heterogeneous nodes should be considered when selecting the computing node. Specifically, the CNS module uses the node load information obtained by the NodeM module to select computing nodes with better capacity by equation (20). Then, it deletes the selected nodes from the candidate computing node set to avoid concurrent update stripes from selecting the duplicate computing node. The entire process of the CNS module is as follows.

4.2.1. Normalizing the Load Attributes. To eliminate the dimension of each node load factor, a min–max normalization method is used. Equation (18) is used for the node CPU utilization and disk I/O load factor, which can achieve better performance with smaller values. Equation (19) is used for the node residual memory and node access bandwidth factor, achieving better performance with larger values.

$$u(x) = \frac{x^{\max} - x}{x^{\max} - x^{\min}}, \tag{18}$$

$$v(x) = \frac{x - x^{\min}}{x^{\max} - x^{\min}}. \tag{19}$$

Then, the normalization decision factor vector can be obtained as follows: $C_j^* = u(C_j)$, $M_j^* = v(M_j)$, $L_j^* = u(L_j)$, $A_j^* = v(A_j)$, and $j \in \{1, 2, \cdots, n\}$; j represents the sequence number of the candidate computing node.

4.2.2. Calculating the Capacity of the Node. The capacity of each candidate node can be calculated using the following equation:

$$\text{Capacity}_j = w_C C_j + w_M M_j + w_L L_j + w_A A_j, \tag{20}$$

where $W = [w_C, w_M, w_L, w_A]$ is the vector of weighted coefficients for the node CPU utilization, residual memory, disk I/O load, and node access bandwidth. Capacity_j represents the weighted summation of the normalized factor of candidate node j. A node j with a larger Capacity_j value is a better computing node.

4.2.3. Selecting the Computing Node. To prevent severe network congestion and excessive node load, we need to avoid multiple update stripes selecting the same computing nodes. The entire process of the computing node selection is summarized in Algorithm 1.

4.3. The Path Selection (PS) Module. As described in Figure 3, when processing the multistripe concurrent update request, after LAMU selects the computing node with the CNS module, the system uses the PS module to schedule the update traffic, which includes the convergence traffic between the data nodes and computing node and the divergence traffic between the computing node and parity nodes. Specifically, LAMU uses the real-time network status and the multiattribute decision-making method based on TOPSIS to schedule the update traffic. In order to improve the update efficiency and maintain better system load balancing, we adjust the weight of decision factors under different network loads. The entire process of the PS module is as follows:

Step 1. Obtain candidate path.

The PS module firstly filters the existing path set P according to the network bandwidth requirements of the

1. Inputs:
   N: candidate computing nodes set
   S: concurrent update stripes set
   C_n: CPU utilization
   M_n: remaining memory
   L_n: I/O load
   A_n: access bandwidth
   Output: best computing nodes for concurrent update stripes
2. **For***s* in stripe set S**do**
3. **For***j* in node set N**do**
4. Obtain the load parameters C_n, M_n, L_n, A_n
5. Normalize load parameter to $C_n^*, M_n^*, L_n^*, A_n^*$ according to (18) and (19)
6. Calculate the capacity of node j according to (20)
7. Set the node has largest capacity as computing node R for stripe s
8. Delete R from N to ensure that the computing nodes selected by multiple stripes are not duplicated
9. **End for**
10. **End for**

Algorithm 1: Computing node selection module.

update traffic and then obtains the candidate path set P^*, where b_ϕ is the network bandwidth requirements of the update traffic ϕ.

$$P^* = \{p_1, p_2, \cdots, p_n\}, \tag{21a}$$

$$b_e \geq b_\phi, \quad \forall e \in P^*, \tag{21b}$$

$$P^* \subseteq P. \tag{21c}$$

$$\mathbf{M} = \begin{bmatrix} b_1 & b_2 & \cdots & b_n \\ d_1 & d_2 & \cdots & d_n \\ h_1 & h_2 & \cdots & h_n \end{bmatrix}, \tag{22}$$

where $\mathbf{M}$ is the decision-making matrix for finding the best path for each point-to-point update traffic; each column in $\mathbf{M}$ presents a candidate path. The symbols b, d, and h in each column denote each path's residual bandwidth, transmission delay, and network hops, respectively. These network attributes are obtained by the NetM module.

Step 2. Construct and normalize the decision-making matrix.

To eliminate the influence of dimensions between each network attribute, the min–max normalization method is used, as shown in equations (18) and (19) in Section 4.2. Equation (18) is used for the path delay and network hop attribute, which can achieve better performance with smaller values. Equation (19) is used for the residual bandwidth, achieving better performance with larger values. Then, the normalization decision-making matrix $\mathbf{M}^*$ is described as follows:

$$\mathbf{M}^* = \begin{bmatrix} {b_1}^* & {b_2}^* & \cdots & {b_n}^* \\ {d_1}^* & {d_2}^* & \cdots & {d_n}^* \\ {h_1}^* & {h_2}^* & \cdots & {h_n}^* \end{bmatrix}, \tag{23}$$

where ${b_j}^* = v(b_j)$, ${d_j}^* = u(d_j)$, ${h_j}^* = u(h_j)$, and $j \in \{1, 2, \cdots, n\}$; j is the sequence number of the matrix column, corresponding to the sequence number of the candidate path of the update traffic.

$$W = [w_b, w_d, w_h]^T, \tag{24}$$

where W is the vector of weighted coefficients; w_b, w_d, and w_h are the weight coefficients of the residual bandwidth, path delay, and network hops, respectively. The value of the weight coefficient set is usually determined through experiment [28] and will be introduced in Section 5. The weighted decision matrix can be obtained using the following equation:

$$Z_{ij} = W_i \times {M^*}_{ij}, \tag{25}$$

where $i \in \{1, 2, 3\}$ and $j \in \{1, 2, \cdots, n\}$.

$$P_i^+ = \max_j \{Z_{ij} | i = 1, 2, 3\}, \quad j \in \{1, 2, \cdots, n\}, \tag{26a}$$

$$P_i^- = \min_j \{Z_{ij} | i = 1, 2, 3\}, \quad j \in \{1, 2, \cdots, n\}, \tag{26b}$$

Step 3. Construct the weighted decision matrix.

Step 4. Obtain the positive and negative ideal solutions.

1. Inputs:
   Candidate path set P; path residual bandwidth set B_p; path delay set D_p; path hop set H_p; source node n_{src} and destination node n_{dst} of convergence flow or divergence flow; bandwidth requirement $b_{\varnothing}$; of update flow; vector of weighted coefficients for the residual bandwidth, end-to-end delay, and network hops $W = [w_b, w_d, w_h]^T$
   Output: the best path from update traffic source n_{src} to update traffic destination n_{dst}
2. **for** path p in path set P**do**
3. **if**p is from n_{src} to n_{dst} and $b_p > b_{\varnothing}$**do**
4. add p to path set P_n
5. **end if**
6. **end for**
7. Build the decision matrix M based on P_n according to Equation (22)
8.Normalize M to M^* according to Equation (23)
9. Construct the weighted decision matrix Z based on M^* according to Equation (25)
10. Calculate the positive P^+ and negative ideal solution P^- of weight matrix Z according to (26)
11. Calculate the Euclidean distance D^+, D^- from each candidate path to the positive and negative ideal solutions according to (27)
12. Calculate the relative closeness C_j^+ between each candidate path and the best candidate path according to (28)
13. **Return** the candidate path p with the largest relative closeness as the route path

Algorithm 2: Path Selection module.

where P_i^+ represents the positive ideal solution of the i_{th} attribute value, which is composed of the maximum value of each decision factor, and P_i^- represents the negative ideal solution of the i_{th} attribute value, which is composed of the minimum value of each decision factor.

$$D_j^+ = \sqrt{\sum_{i=1}^{3} \left(Z_{ij} - P_i^+\right)^2}, \quad j \in \{1, 2, \cdots, n\}, \tag{27a}$$

$$D_j^- = \sqrt{\sum_{i=1}^{3} \left(Z_{ij} - P_i^-\right)^2}, \quad j \in \{1, 2, \cdots, n\}, \tag{27b}$$

where Z_{ij} is an element in the candidate path $[Z_{1j}, Z_{2j}, Z_{3j}]^T$ and D_j^+ and D_j^- are the distances from each candidate path to the positive and negative ideal solutions, respectively.

$$C_j^+ = \frac{D_j^-}{D_j^+ + D_j^-}, \quad j \in \{1, 2, \cdots, n\}. \tag{28}$$

Step 5. Calculate the distance from the candidate path to the positive and negative ideal solutions.

Step 6. Calculate the relative closeness C_j^+ between each candidate path and the optimal candidate path.

When the relative closeness C_j^+ is larger, the path is more suitable for the update traffic.

The entire process of the PS module is summarized in Algorithm 2.

5. Implementation and Evaluation

5.1. Experiment Environment. The performance of the proposed erasure-coding update scheme is evaluated in this section. We implement the prototype of LAMU on Containernet [17], a fork of the famous Mininet [29] network emulator. Different from Mininet, Containernet uses the Docker [30] containers as hosts in emulated network topologies. This feature allows Containernet to better simulate distributed storage systems. Ryu [31] is used as the SDN controller that supports the OpenFlow protocol. The entire experimental environment is deployed on an Ubuntu 18.04 system on a Sugon A840r-G, which has $64 * 2.1$ GHz AMD processors and 128 GB of memory. In terms of the experimental topology, we use Containernet 3.1.0 to simulate the fat-tree topology [32]. As shown in Figure 4, the bandwidth capacity of each link in the fat tree is set to 200 Mbps because the simulation experiment assumes limited resources. Storage nodes in the fat-tree topology are heterogeneous; when selecting the computing node for each update stripe, we set the weight of the access bandwidth, CPU utilization, residual memory, and I/O load to [0.6, 0.1, 0.1,0.2].

To evaluate the performance of LAMU in a more realistic environment, we use the real distributed storage system background traffic pattern, which was measured in our previous work [33], to reproduce the realistic network condition, as shown in Table 2. According to [33], the speed of the heartbeat traffic is set to 1 Mbps to reduce the packet loss rate in the experimental environment; all the background traffic is maintained for a long time to ensure that the background traffic exists throughout the whole update process. To further evaluate the efficiency of our LAMU method under different network loads, three kinds of traffic load scenarios are set in the evaluation, as follows:

(i) *Low-load (LL) scenario*: 10 heart beating flows, 10 user data flows, and 10 migration flows

(ii) *Middle-load (ML) scenario*: 20 heart beating flows, 20 user data flows, and 20 migration flows

(iii) *High-load (HL) scenario*: 30 heart beating flows, 30 user data flows, and 30 migration flows

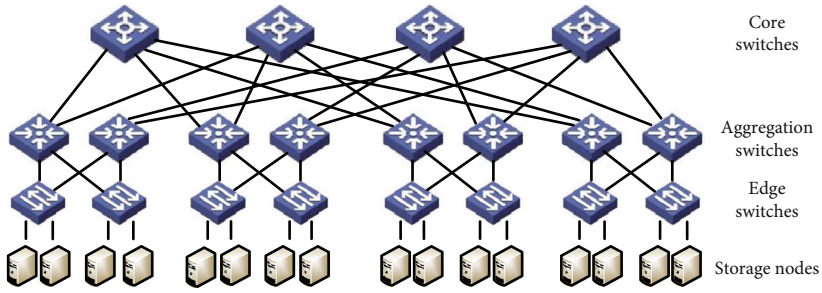

Figure 4: Experimental fat-tree topology.

Table 2: The statistical information of the different types of traffic in the distributed storage system.

Traffic type	Speed (Mbps)	Duration (s)	Packet number	Avg. packet size (bytes)
Heartbeat traffic	92.87	0.006554	54	1477.42
User data traffic	12.93	39.31	67073	1508.65
Migration traffic	4.36	654.12	340354	1505.56

The value of the weight coefficient set is usually determined through experiments [28]. As the system load increase, the network bandwidth resources become more limited. Therefore, we increase the bandwidth weight with the load increase. We set $W = [w_b, w_d, w_h]^T$ mentioned in equation (24) to [0.25, 0.6,0.15], [0.5, 0.4,0.1], and [0.7, 0.2,0.1] for the LL, ML, and HL scenarios, respectively.

In the evaluation, we compare LAMU with PUM-P [22] and DelaySelect. For PUM-P, it also improves the update efficiency by introducing the computing node. Yet, PUM-P ignores the heterogeneity of computing nodes and network status; all nodes and route paths have an equal probability of being selected. DelaySelect is extended from [23], which also adopts a centralized update framework and improves the update efficiency by selecting the path with the least delay as the routing path for update traffic. The experimental comparison items include the average update time, the standard deviation of bandwidth, and the link maximum bandwidth utilization of the system.

We focus on the update performance between different update schemes under various system load scenarios. In terms of experimental parameters, we use the parameters that may impact the update performance, including the number of update data nodes, the number of parity nodes, the size of data-delta, and the number of update stripes. The range of these parameters is listed in Table 3. To get a more convincing experiment result, each experiment was done 10 times, and the average value of these experiments was taken as the result.

5.2. Update Efficiency

5.2.1. Average Update Time with Varying Numbers of Parity Nodes in Different Load Scenarios. This subsection presents extensive comparisons of the average update time of three update schemes under different experimental parameters and load scenarios. Figure 5 shows the average update time increase along with p when $d = 8$. As the number of parity nodes increases, more parity-delta needs to be transmitted, increasing the average update time. While the load becomes higher, the update time between different schemes begins to present differences. As we can see, in the high-load (HL) scenario, LAMU reduces the average update time by 17.9% and 43.1% compared with DelaySelect and PUM-P, respectively.

5.2.2. Average Update Time with Varying Numbers of Update Data Nodes in Different Load Scenarios. Figure 6 presents that the average update time is generally stable with the increase of update data nodes in different load scenarios. This is because, on the premise that the data volume is constant, the increasing number of update data nodes will reduce the average data-delta sent by each data node. Therefore, the extra time caused by connecting more data nodes is offset. While the load becomes higher, the update time between different schemes begins to show more significant differences. As we can notice, in the low-load (LL) scenario, the three update schemes achieve a comparable average update time. In the middle load (ML) scenario, LAMU starts to show better update efficiency. In the HL scenario, LAMU reduces the average update time by 18.8% and 49.5% compared with DelaySelect and PUM-P, respectively.

5.2.3. Average Update Time with Varying Sizes of Update Data Volume in Different Load Scenarios. Figure 7 illustrates how the average update time increases along with the update data volume in different load scenarios. The three update schemes achieve comparable average update times in the LL scenario. In the ML scenario, LAMU starts to show better update efficiency. Compared with DelaySelect and PUM-P, LAMU reduces the average update time by 12.1% and 26.5% under the ML scenario, respectively, and 19.7% and 43.4% under the HL scenario, respectively.

5.2.4. Average Update Time with Varying Update Stripes in Different Load Scenarios. Figure 8 illustrates the average update time variation with the number of concurrent update stripes. As we can see, with the increase of the number of concurrent update stripes in all three scenarios, the average update time of LAMU is only increasing a little. It illustrates that LAMU is more efficient in dealing with the multistripe concurrent update. However, the update time increases significantly when adopting the DelaySelect and PUM-P schemes with the increase of the number of update stripes. Specifically,

Table 3: Experiment parameters.

Parameter	Ranges	Default
The number of data node for update of each stripe, d	6, 7, 8, 9, 10	8 update data nodes
The number of parity node of each stripe, p	3, 4, 5, 6, 7	6 parity nodes
The size of update data volume (MB)	1, 2, 4, 8, 16	8 MB
The number of update stripe	3, 4, 5, 6, 7	5 update stripes

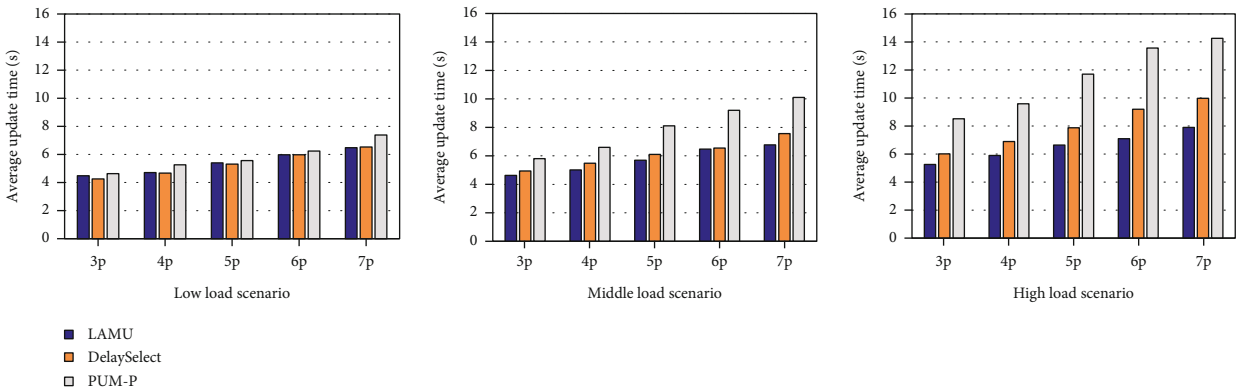

Figure 5: Average update time comparison with the variation of the number of parity nodes in different load scenarios.

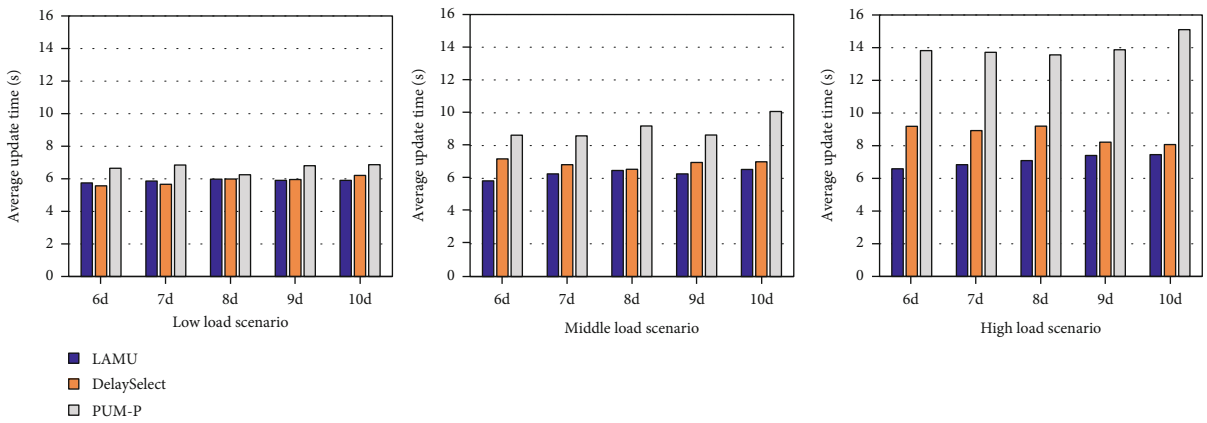

Figure 6: Average update time comparison with the variation of the number of data nodes in different load scenarios.

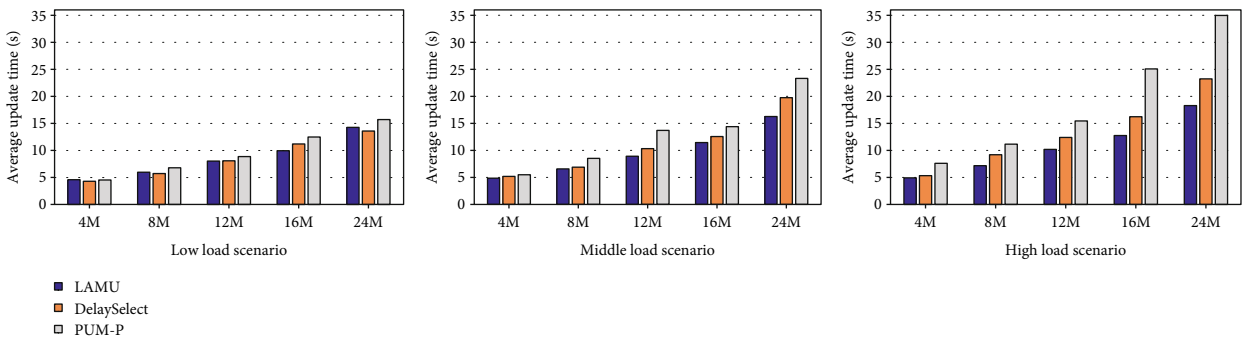

Figure 7: Average update time comparison with the variation of the update data volume in different load scenarios.

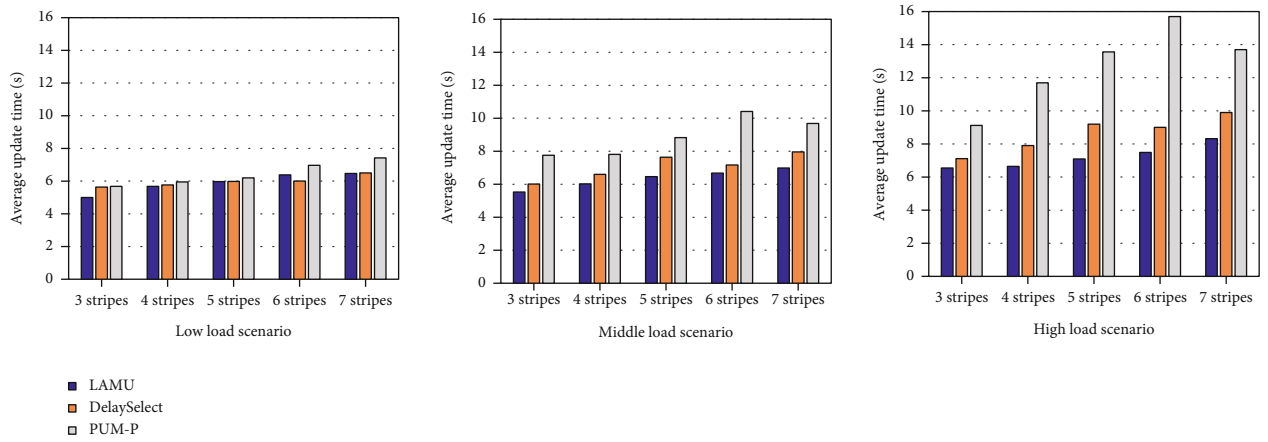

Figure 8: Average update time comparison with the variation of the update stripe number in different load scenarios.

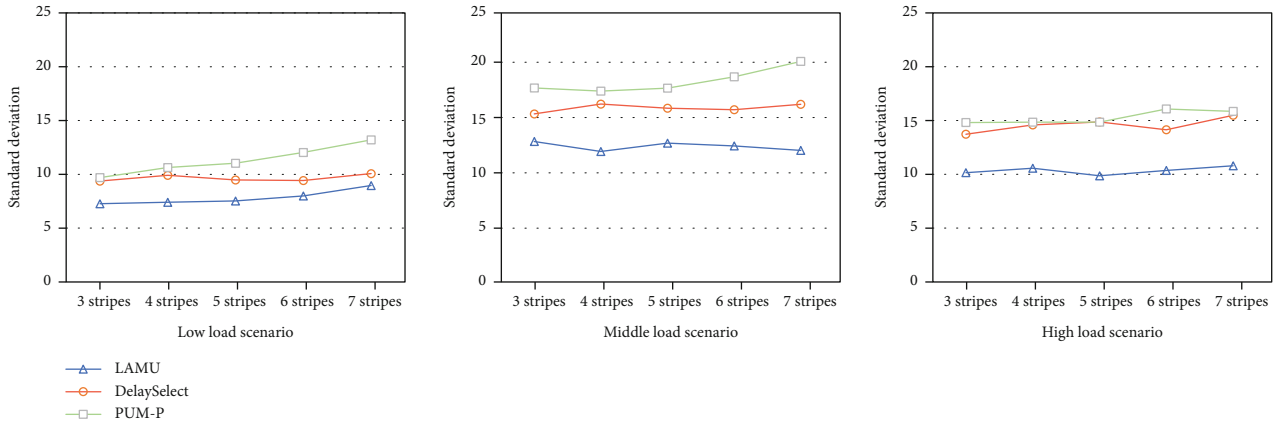

Figure 9: Standard deviation of link utilization with the variation of the number of update stripes in different load scenarios.

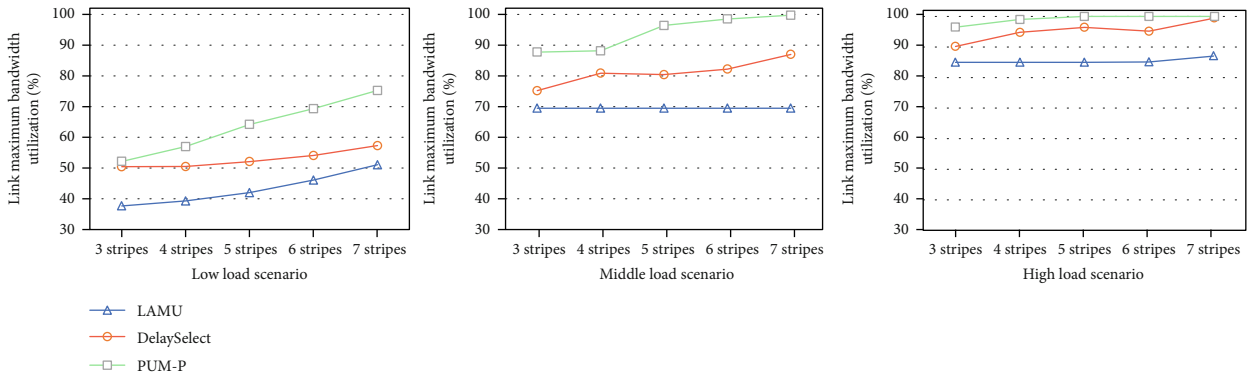

Figure 10: Maximum link bandwidth utilization with the variation of the number of update stripes in different load scenarios.

compared with DelaySelect and PUM, LAMU reduces the average update time by 10.4% and 28.8% under the ML scenario, respectively, and reduces the average update time by 16.3% and 43.4% under the HL scenario, respectively.

5.3. Network Load-Balancing Performance

5.3.1. Standard Deviation of Link Bandwidth Utilization with Varying Update Stripes in Different Load Scenarios. To verify the load-balancing performance of the three update schemes, we evaluate the standard deviation of link utilization, as presented in Figure 9. The lower the standard deviation of link utilization, the more balanced the link loads are. PUM-P has the largest standard deviation in all three scenarios. This finding is because PUM-P does not consider the network status when scheduling the update traffics, which easily leads to load imbalance, while we can notice that in the HL scenario, the standard deviation of PUM-P

is a little lower than that of the ML scenario. The reason is that, with the load increasing, PUM-P makes more and more links saturated. Thus, the standard deviation is decreased. The DelaySelect has a lower standard deviation than PUM-P for DelaySelect uses the link delay to schedule the update traffic, which has better load-balancing performance. The LAMU has the lowest standard deviation in all three scenarios. This result is because LAMU comprehensively considers link bandwidth, delay, and path hop when scheduling the update traffic; LAMU achieves better load balancing and avoids network congestion caused by several links reaching the full load.

5.3.2. Network Maximum Link Bandwidth Utilization with Varying Update Stripes in Different Load Scenarios. The maximum link bandwidth utilization represents the utilization of the most congested link in the system, and the larger it is, the more unbalanced the system is. As shown in Figure 10, in the ML scenario, full load links have already appeared in PUM-P and have also appeared in DelaySelect in the HL scenario. It means that some links in the system are highly congested. The LAMU method has the lowest maximum link bandwidth utilization in all three scenarios, which means LAMU can achieve better load balancing and avoids network congestion caused by links reaching the full load.

6. Conclusions

Erasure coding has become an indispensable redundancy mechanism in today's large-scale distributed storage system. However, the data update of erasure coding introduces additional computing load and network traffic, reducing the efficiency of data updates and affecting the system load balancing. Most of the existing erasure-coding update schemes ignore the heterogeneity of node and network status and the multistripe concurrent update caused by data correlation. To solve this problem, this paper establishes the optimization model of multistripe updates with multiple QoS constraints in the heterogeneous environment and then proposes LAMU, a load-aware multistripe concurrent update scheme. Firstly, LAMU introduces SDN to measure the node load and network status in real time, and then, the obtained nodes and network information are used to select nonduplicated computing nodes with better capacity for multiple update stripes. Finally, the multiattribute decision-making method is used to schedule the network traffic between data nodes, computing nodes, and parity nodes. Extensive experimental results show that LAMU can reduce the average update time while providing better load-balancing performance.

Moreover, we'll consider implementing LAMU in a real erasure-coded storage system in the future. Another direction for future work is to use reinforcement learning to adjust the decision parameter weight when scheduling the update traffic and making a trade-off between the number and the location of the computing nodes to achieve better results.

Data Availability

The data used to support the findings of this study are included within the article.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This research work obtained the subsidization of the National Natural Science Foundation of China (Nos. 61861013 and 62161006), the Science and Technology Major Project of Guangxi (No. AA18118031), and the Innovation Project of Guangxi Graduate Education (No. YCSW2022271).

References

[1] D. Ford, F. Labelle, F. I. Popovici et al., "Availability in globally distributed storage systems," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pp. 61–74, Vancouver, BC, Canada, 2010.

[2] C. A. Rincón, J.-F. Pâris, R. Vilalta, A. M. Cheng, and D. D. Long, "Disk failure prediction in heterogeneous environments," in *International Symposium on Performance Evaluation of Computer and Telecommunication Systems*, pp. 1–7, Seattle, WA, July 2017.

[3] S. S. Arslan and E. Zeydan, "On the distribution modeling of heavy-tailed disk failure lifetime in big data centers," *IEEE Transactions on Reliability*, vol. 70, no. 2, pp. 507–524, 2021.

[4] H. Weatherspoon and J. D. Kubiatowicz, "Erasure coding vs. replication: a quantitative comparison," in *International Workshop on Peer-to-Peer Systems*, pp. 328–337, Springer, Cambridge, MA, USA, 2002.

[5] S. Muralidhar, W. Lloyd, S. Roy et al., "f4: Facebook's warm BLOB storage system," in *11th USENIX Symposium on Operating Systems Design and Implementation*, pp. 383–398, Broomfield, CO, USA, 2014.

[6] C. Huang, H. Simitci, Y. Xu et al., "Erasure coding in windows Azure storage," in *Proceedings of the USENIX conference on Annual Technical Conference*, p. 2, Boston, MA, 2012.

[7] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos et al., "XORing elephants," *Proceedings of the VLDB Endowment*, vol. 6, no. 5, pp. 325–336, 2013.

[8] D. Narayanan, A. Donnelly, A. Rowstron, and Usenix, "Write off-loading: practical power management for enterprise storage," in *6th USENIX Conference on File and Storage Technologies*, pp. 253–267, San Jose, CA, 2008.

[9] 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 *12th {USENIX} Conference on File and Storage Technologies*, pp. 163–176, Santa Clara, CA, USA, 2014.

[10] D. J. Ellard, *Trace-Based Analyses and Optimizations for Network Storage Servers*, Harvard University, 2004.

[11] Z. Fengyan, W. Yan, and L. Nianshuang, "Survey of heterogeneous-based data repair strategies for erasure codes," *Application Research of Computers*, vol. 36, no. 8, pp. 2249–2255, 2019.

[12] M. Ye, R. Wei, W. Guo, Q. Jiang, H. Qiu, and Y. Wang, "A new method for reconstructing data on a single failure node in the distributed storage system based on the MSR code," *Wireless Communications & Mobile Computing*, vol. 2021, pp. 1–14, 2021.

[13] S. Maitrey and C. K. Jha, "MapReduce: simplified data analysis of big data," in *3rd International Conference on Recent Trends in Computing*, pp. 563–571, Delhi, India, 2015.

[14] Z. Yuan, X. You, X. Lv, and P. Xie, "SS6: online short-code RAID-6 scaling by optimizing new disk location and data migration," *The Computer Journal*, vol. 64, no. 10, pp. 1600–1616, 2021.

[15] Z. R. Shen, P. P. C. Lee, J. W. Shu, and W. Z. Guo, "Correlation-aware stripe organization for efficient writes in erasure-coded storage: algorithms and evaluation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 7, pp. 1552–1564, 2019.

[16] Z. Li, Z. Chen, S. M. Srinivasan, and Y. Zhou, "C-Miner: mining block correlations in storage systems," in *3rd Conference on File and Storage Technologies*, pp. 173–186, Usenix Assoc, San Francisco, CA, 2004.

[17] "Contanernet," 2022, https://containernet.github.io/.

[18] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the society for industrial applied mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[19] 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 *IEEE INFOCOM -The 35th Annual IEEE International Conference on Computer Communications*, pp. 1–9, San Francisco, CA, USA, April 2016.

[20] S. Manen, M. Guillaumin, and L. Van Gool, "Prime object proposals with randomized Prim's algorithm," in *IEEE International Conference on Computer Vision*, pp. 2536–2543, Sydney, Australia, 2013.

[21] 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 Transactions on Parallel Distributed Systems*, vol. 29, no. 8, pp. 1893–1906, 2017.

[22] F. Zhang, J. Huang, and C. Xie, "Two efficient partial-updating schemes for erasure-coded storage clusters," *IEEE Seventh International Conference on Networking, Architecture, and Storage*, 2012, pp. 21–30, Xiamen, China, 2012.

[23] L. Qian, H. Yupeng, Y. Zhenyu, X. Ye, and Q. Zheng, "An ant colony optimization algorithms based data update scheme for erasure-coded storage systems," *Journal of Computer Research and Development*, vol. 58, no. 2, p. 305, 2021.

[24] Q. Fenglin, G. Qingyuan, Z. Yangfan, and X. Wang, "Heterogeneity-aware node selection for data repair in distributed storage systems," *Journal of Computer Research and Development*, vol. 52, no. 2, pp. 68–74, 2015.

[25] Y. Wang, M. Ye, Q. He, Y. Huan, and W. Kang, "A new node selecting approach in Ceph storage system based on software defined network and multi-attributes decision-making model," *Chinese Journal of Computers*, vol. 42, no. 2, pp. 93–108, 2019.

[26] M. Barbehenn, "A note on the complexity of Dijkstra's algorithm for graphs with weighted vertices," *IEEE Transactions on Computers*, vol. 47, no. 2, pp. 263–263, 1998.

[27] W. Ke, Y. Wang, M. Ye, and J. Chen, "A priority-based multicast flow scheduling method for a collaborative edge storage datacenter network," *IEEE Access*, vol. 9, pp. 79793–79805, 2021.

[28] Y. Luo, J. Xia, and T.-p. Chen, "Comparison of objective weight determination methods in network performance evaluation," *Journal of Computer Applications*, vol. 29, no. 10, pp. 2624–2626, 2009.

[29] "Mininet," 2022, http://mininet.org/.

[30] "Docker," 2022, https://www.docker.com/get-started.

[31] "Ryu," 2022, https://osrg.github.io/ryu/.

[32] C. Zhang, S. Zhang, B. Jin, W. Li, Z. Wang, and Y. Wang, "A3: an automatic malfunction detection and fixation system in FatTree data center networks," in *Conference of the ACM-Special-Interest-Group-on-Data-Communication*, pp. 24–26, Beijing, China, 2019.

[33] W. Ke, Y. Wang, and M. Ye, "GRSA: service-aware flow scheduling for cloud storage datacenter networks," *China Communications*, vol. 17, no. 6, pp. 164–179, 2020.