Optimal Rack-Coordinated Updates in Erasure-Coded Data Centers: Design and Analysis

Guowen Gong, Zhirong Shen, *Member, IEEE*, Liang Chen, Suzhen Wu, *Member, IEEE*, Xiaolu Li, Patrick P. C. Lee, *Senior Member, IEEE*, Zhiguo Wan, and Jiwu Shu, *Fellow, IEEE*

***Abstract*—Erasure coding has been extensively deployed in today's data centers to tackle prevalent failures, yet it is prone to substantial cross-rack traffic for parity updates. In this article, we propose a new rack-coordinated update mechanism to suppress the cross-rack update traffic, which comprises two successive phases: a delta-collecting phase that collects data delta chunks, and another selective parity update phase that renews the parity chunks based on the update pattern and parity layout. We further design RackCU, an optimal rack-coordinated update solution that achieves the theoretical lower bound of the cross-rack update traffic. We also perform reliability analysis, demonstrating that RackCU can attain a lower data loss probability via shortening the update procedure. We conduct extensive evaluations, in terms of large-scale simulation and real-world data center experiments. We show that RackCU can reduce 16.5-77.1% of the cross-rack update traffic and hence improve 24.9-772.0% of the update throughput.**

***Index Terms*—Cross-rack update traffic, erasure codes, rack-coordinated updates.**

Manuscript received 17 March 2022; revised 21 November 2022; accepted 16 December 2022. Date of publication 5 January 2023; date of current version 9 June 2023. This work was supported in part by the National Key R&D Program of China under Grant 2021YFF0704001, in part by the Natural Science Foundation of China under Grants 62072381, 61832011, and 62272425, in part by the Science and Technology Projects of Innovation Laboratory for Sciences and Technologies of Energy Materials of Fujian Province under Grant (IKKEM)HRTP-[2022]-1. The work of Suzhen Wu was supported in part by the National Natural Science Foundation of China under Grants U22A2027, 61972325, and 61872305, in part by the Open Research Projects of Zhejiang Lab under Grant 2021DA0AM01/002, and in part by the Open Project Program of Wuhan National Laboratory for Optoelectronics under Grant 2021WNLOKF011. The work of Patrick P. C. Lee was supported by Research Grants Council of HKSAR under Grant AoE/P-404/18 and Research Matching Grant Scheme. Recommended for acceptance by H. Matsutani. *(Corresponding author: Zhirong Shen.)*

Guowen Gong, Liang Chen, Suzhen Wu, and Jiwu Shu are with the School of Informatics, Xiamen University, Xiamen 361005, China (e-mail: 23020201153743@stu.xmu.edu.cn; liang0_0chen@foxmail.com; suzhen@xmu.edu.cn; jwshu@xmu.edu.cn).

Zhirong Shen is with the School of Informatics, Xiamen University, Xiamen 361005, China, and also with the Innovation Laboratory for Sciences and Technologies of Energy Materials of Fujian Province (IKKEM), Xiamen 361005, China (e-mail: shenzr@xmu.edu.cn).

Xiaolu Li is with the School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China (e-mail: lixl666@hust.edu.cn).

Patrick P. C. Lee is with the Department of Computer Science and Engineering, Chinese University of Hong Kong,, Hong Kong (e-mail: pclee@cse.cuhk.edu.hk).

Zhiguo Wan is with the Zhejiang Lab, Hangzhou 311121, China (e-mail: wanzhiguo@zhejianglab.com).

This article has supplementary downloadable material available at https://doi.org/10.1109/TC.2023.3234215, provided by the authors.

Digital Object Identifier 10.1109/TC.2023.3234215

I. Introduction

Data centers are often built atop numerous storage nodes (also called *nodes*) to support a large number of services, including data storage, information retrieval, and MapReduce computation [14]. The large scale of data centers makes failures, which are originally accidental, become the norm [11], [12]. To tackle prevalent unexpected failures, production storage systems [4], [16], [27] often resort to maintaining additional data redundancy through replication [26] and erasure coding [16], such that the systems can leverage the pre-stored data redundancy to restore the lost data. Compared to replication, *erasure coding* can assuredly retain the same degree of fault tolerance with much less storage overhead [40], and hence is preferable in practical storage systems [2], [4], [7], [24]. In principle, erasure coding encodes a group of data chunks to generate a small number of redundant chunks (also called *parity chunks*), such that a subset of data and parity chunks still suffice to rebuild the original data chunks.

While being more storage-efficient, erasure coding incurs substantial *update traffic* (i.e., data transmitted over the network in update operations), making update performance unsatisfactory. The rationale is that to maintain encoding consistency, any change to the data chunks triggers additional updates to the corresponding parity chunks. Although we can perform the parity update in the background, it still triggers considerable storage and network I/Os, resulting in resource contention with foreground applications. For example, the conventional delta-based update approach (see Section II.C) requires to transmit m parity delta chunks for parity update whenever a data chunk is updated, implying that the storage and network I/Os are amplified for m times. Hence, realizing efficient parity update of erasure coding can not only improve the overall system reliability (for the newly updated data), but also mitigate the performance impact on the foreground applications.

The update problem of erasure coding in data centers becomes more complicated. Data centers usually organize nodes hierarchically, where multiple nodes are first organized into a *rack* and the racks are further interconnected via the *network core* — an abstraction of aggregation switches and core routers [14]. Such a hierarchical organization naturally results in the *bandwidth diversity* phenomenon, where the cross-rack bandwidth is often much more scarce than the intra-rack bandwidth [5], [8], [14] and further fiercely consumed by various workloads (e.g., replication writes [8] and MapReduce shuffling [5]). It is reported that the ratio of the available cross-rack bandwidth per node and the intra-rack bandwidth often ranges from $1/20$ to $1/5$ and may even drop down to $1/240$ in some extreme cases [14]. Hence, when deploying erasure coding in data centers to mitigate failures, suppressing the *cross-rack update traffic* (i.e., data transferred across racks for update operations) is clearly a crucial issue to be addressed.

Existing studies of erasure-coded updates mainly focus on mitigating disk seeks [6], [18], decreasing the number of parity chunks being updated [32], [34], [36], and reducing update traffic [29], [38]. While CAU [33] can mitigate cross-rack update traffic, it degrades system reliability (by postponing parity updates) and falls short on achieving the theoretically minimum cross-rack update traffic. How to minimize the cross-rack update traffic without compromising system reliability is unfortunately largely overlooked by existing studies.

We propose *rack-coordinated update*, a new parity update mechanism that comprises a *delta-collecting phase* and another *selective parity update phase* to renew the parity chunks *immediately* after data update, with the objective of minimizing the cross-rack update traffic with system reliability guaranteed. The main idea of the rack-coordinated update is to collect data delta (i.e., the difference between the old and new data chunks) in some dedicated racks (called *collector racks*), and update the parity chunks by selecting an appropriate update approach. We further design RackCU, the optimal *Rack*-**C**oordinated **U**pdate solution that reaches the lower bound of the cross-rack update traffic with linear computational complexity, by carefully selecting the collector racks based on the update pattern and parity layout. To summarize, our contributions include:

- We propose a new rack-coordinated update mechanism that aims to significantly mitigate the cross-rack update traffic.
- We design RackCU, an optimal rack-coordinated update solution that reaches the lower bound of the cross-rack update traffic. We also show that RackCU is a general design for different representative erasure codes.
- We carry out reliability analysis, showing that RackCU reduces the data loss probability during updates by up to 42.7% due to its optimized update procedure.
- We implement a RackCU prototype and conduct extensive evaluation via both large-scale simulation and Alibaba Cloud Elastic Compute Service (ECS) [1] experiments. We show that RackCU reduces 16.5-77.1% of cross-rack update traffic and hence increases 24.9-772.0% of update throughput.

Our RackCU prototype can be reached via https://github.com/ggw5/RackCU-code.

II. Background

We introduce the architecture of data centers (Section II.A) and elaborate erasure coding (Section II.B). We also describe the parity update in erasure coding (Section II.C) and erasure-coded data centers (Section II.D).

A. Data Center

We focus on a data center with a two-layer hierarchical architecture, in which a bunch of nodes are first organized into *a rack* and multiple racks are further interconnected by the network core (i.e., aggregation and core switches). Such an architecture has been applied in modern data centers [11], [24] and assumed in previous work [8], [15], [33], [37], [38]. Fig. 1 depicts a data center with four racks and each rack comprises four nodes. The hierarchical architecture results in the *bandwidth diversity* phenomenon. That is, as being shared and fiercely competed among the nodes within the same rack, the cross-rack bandwidth is often a small fraction of the intra-rack bandwidth [5], [8], [14]. Even worse, the cross-rack communication continues to grow dramatically, as large-scale analytic workloads prevalently distribute jobs across multiple racks [22].

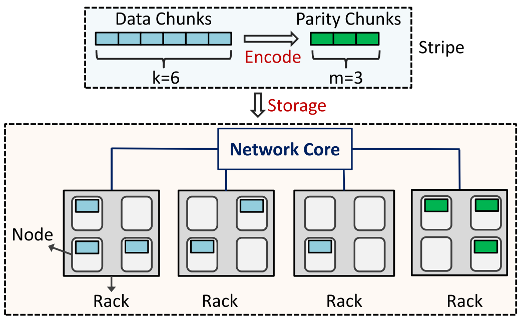


Fig. 1. Example of a data center deployed with RS(6,3).

B. Erasure Coding

Erasure codes are often configured by two parameters (namely k and m) to balance storage overhead and fault tolerance capability. At a high level, erasure codes operate using an *encoding* operation (to generate additional redundancy on the data) and another *decoding* operation (to recover the original data). In the encoding stage, erasure codes encode k *data chunks* to generate additional m parity chunks via arithmetics over Galois finite field [31]. These $k + m$ chunks that are encoded together collectively constitute a *stripe*, promising that any k out of the $k + m$ chunks within a stripe suffice to reproduce the original k data chunks. In other words, erasure codes can tolerate any m chunk failures within each stripe. Hence, by distributing the $k + m$ chunks of each stripe across $k + m$ nodes (one chunk per node), erasure codes can tolerate *any* m node failures. Further, we can tolerate *any single rack failure* by storing at most m chunks of any stripe in a rack, as we can always fetch at least k surviving chunks of the same stripe from other available racks (aside from the failed one).

In this article, to facilitate the understanding, we mainly use Reed-Solomon codes (RS codes) [31] as an instance, as they are popularly deployed in production systems [2], [4], [7], [24], [27]. Nevertheless, we also show that our approach can be readily extended to other codes like locally-repairable codes (LRCs) [16], [28] (see Sections III.D and V.B). We use RS(k, m) to denote the RS codes configured by the parameters k and m throughout the article. Fig. 1 shows the placement of a stripe encoded by RS(6,3) (i.e., $k = 6$ and $m = 3$) in a data center, which can tolerate any single rack failure, as at most three chunks (i.e., m chunks) of the same stripe are stored in a rack.

C. Delta-Based Parity Update in Erasure Coding

In this article, we mainly consider the *delta-based update* in erasure coding [6], [18], [33]. Suppose that $\{D_1, D_2, \ldots, D_k\}$ and $\{P_1, P_2, \ldots, P_m\}$ represent the k data chunks and the m parity chunks of a stripe, respectively. Each parity chunk P_j ($1 \leq j \leq m$) can be calculated as a *linear combination* of the k data chunks via the Galois Field arithmetic [30], given by

$$P_j = \sum_{i=1}^{k} \gamma_{i,j} D_i, \tag{1}$$

where $\gamma_{i,j}$ ($1 \leq i \leq k$ and $1 \leq j \leq m$) is the encoding coefficient used by the data chunk D_i to calculate the parity chunk P_j.

Suppose that a data chunk D_h is updated to D'_h ($1 \leq h \leq k$). To promise the *encoding consistency* between the data and parity chunks, each parity chunk P_j (where $1 \leq j \leq m$) should be accordingly updated based on (1) as below:

$$P'_j = P_j + \gamma_{h,j}(D'_h - D_h) = P_j + \Delta P_j. \tag{2}$$

Equation (2) indicates that the new parity chunk P'_j can be obtained by leveraging the old parity chunk P_j and the *data delta chunk* (i.e., $D'_h - D_h$, the difference between the old and new data chunks) or the *parity delta chunk* ΔP_j (i.e., $\gamma_{h,j}(D'_h - D_h)$, the difference between the old and new parity chunks), without having to access the unchanged data chunks [33]. Besides, as the encoding coefficients $\{\gamma_{i,j}\}_{1 \leq i \leq k, 1 \leq j \leq m}$ can be derived once the parameters k and m are established, they are public to all the nodes without having to be re-transmitted.

D. Parity Update in Erasure-Coded Data Centers

We elaborate the parity update in erasure-coded data centers. There may be multiple racks containing the updated data chunks of a stripe. Without loss of generality, suppose that the data chunks $\{D_1, D_2, \ldots, D_{u_x}\}$ in the rack R_x are updated to $\{D'_1, D'_2, \ldots, D'_{u_x}\}$, where u_x denotes the number of updated data chunks in R_x and D_h denotes the h-th data chunk of the stripe (where $1 \leq h \leq u_x$). Based on (2), we can calculate the parity delta chunk derived from the u_x updated data chunks in R_x to update the parity chunk P_j ($1 \leq j \leq m$), given by

$$\Delta P_{x,j} = \sum_{h=1}^{u_x} \gamma_{h,j} \Delta D_h, \tag{3}$$

where $\Delta D_h = D'_h - D_h$ denotes the data delta chunk of D_h.

Let us consider another rack R_y ($R_y \neq R_x$) that stores t_y parity chunks, denoted by $\{P_1, P_2, \cdots, P_{t_y}\}$. Based on (3), there are two options to update the parity chunks in R_y, namely *data-delta-based update* and *parity-delta-based update* [33].

Data-Delta-Based Update: It updates the parity chunks of a rack *in batch* via transmitting data delta chunks directly. It first calculates u_x data delta chunks of the u_x data chunks updated in R_x (i.e., $\{\Delta D_h\}_{1 \leq h \leq u_x}$) and sends them to a relay node in R_y, which will then forward the u_x data delta chunks to the corresponding t_y nodes of R_y that store the parity chunks. For the node that keeps the parity chunk P_j ($1 \leq j \leq t_y$), it will read the old parity chunk (i.e., P_j) from local storage and renew it by adding the parity delta chunk (i.e., $\Delta P_{x,j}$) with the old parity chunk. Fig. 2(a) shows an example of the data-delta-based update approach (where $u_x = 2$ and $t_y = 3$), which transmits u_x (i.e., 2) data delta chunks from R_x to update the t_y parity chunks in R_y ($R_y \neq R_x$).

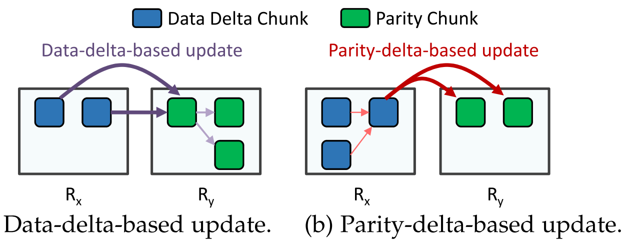

Data Delta Chunk
Parity Chunk
Data-delta-based update
Parity-delta-based update
Rx
Ry
Rx
Ry
(a) Data-delta-based update.
(b) Parity-delta-based update.

Fig. 2. Examples of the data-delta-based update and parity-delta-based update: (a) $u_x = 2$ and $t_y = 3$; (b) $u_x = 3$ and $t_y = 2$.

Parity-Delta-Based Update: It updates each parity chunk in another rack *individually* via transmitting the corresponding parity delta chunk. In particular, to update a parity chunk P_j in R_y ($1 \leq j \leq t_y$), the parity-delta-based update approach first calculates a *parity delta chunk* $\Delta P_{x,j}$ in R_x, and then sends it to the corresponding node in R_y. Finally, the new parity chunk P'_j can be generated based on the old parity chunk P_j and the received $\Delta P_{x,j}$. Fig. 2(b) shows an example of the parity-delta-based update approach (where $u_x = 3$ and $t_y = 2$), which needs to send t_y (i.e., 2) parity delta chunks from R_x to update the t_y parity chunks in R_y ($R_y \neq R_x$).

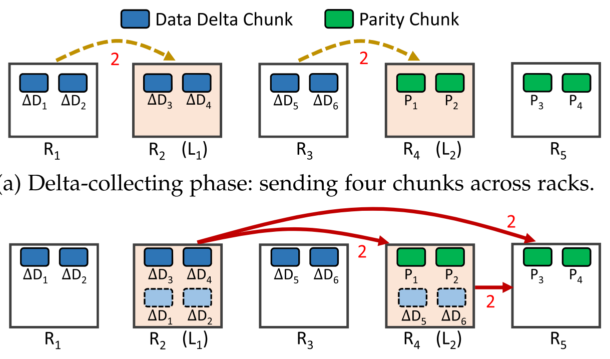
Difference: The two update approaches differ in which delta chunk is delivered across racks and hence induce different amounts of the cross-rack update traffic. To summarize, if there are u_x data chunks updated in the rack R_x, the data-delta-based update (resp. parity-delta-based update) transmits u_x data delta chunks (resp. t_y parity delta chunks) to renew the t_y parity chunks in another rack R_y ($R_y \neq R_x$).

III. Rack-Coordinated Updates

We elaborate the design overview of the rack-coordinated update (Section III.A) and present a rigorous formulation (Section III.B). We also perform in-depth theoretical analysis (Section III.C) and design RackCU that touches the lower bound of the cross-rack update traffic (Section III.D). We finally provide in-depth reliability analysis (Section III.E).

A. Design Overview

In principle, the rack-coordinated update is a synthesis of the data-delta-based update and parity-delta-based update approaches. The *main idea* is to allow racks to coordinate in the parity update *immediately* after data chunks are updated, to reduce the cross-rack update traffic with system reliability guaranteed. It breaks the whole parity update procedure into a *delta-collecting phase* and another *selective parity update phase* that are performed successively. Specifically, in the delta-collecting



(a) Delta-collecting phase: sending four chunks across racks.

(b) Selective parity update phase: sending six chunks across racks.

Fig. 3. Guiding example of the rack-coordinated update mechanism: select R_2 and R_4 as collector racks, and transmit 10 chunks in total for the parity update.

phase, the rack-coordinated update mechanism will elect several *collector racks* that are responsible for collecting data delta chunks from other racks. On the other hand, the selective parity update phase will choose either the data-delta-based update or the parity-delta-based update to renew the parity chunks based on the update pattern and parity layout of the data center, with the primary objective of suppressing the cross-rack update traffic. In particular, suppose that a rack R_x has u_x updated data chunks and another rack R_y ($R_y \neq R_x$) stores t_y parity chunks. The selective parity update performs the following actions: if $u_x \leq t_y$, it uses the data-delta-based update by sending u_x data delta chunks from R_x to R_y for updating the t_y parity chunks in batch (Fig. 2(a), where $u_x = 2 \leq t_y = 3$); otherwise, it resorts to the parity-delta-based update by transmitting the corresponding t_y parity delta chunks (Fig. 2(b), where $u_x = 3 > t_y = 2$). Hence, the selective parity update needs to transmit $\min\{u_x, t_y\}$ chunks across racks for renewing the t_y parity chunks of R_y based on the u_x updated data chunks in R_x.

Guiding Example: We show a guiding example via Fig. 3 to elaborate the rack-coordinated update mechanism. Suppose that a data center consists of five racks, namely $\{R_1, R_2, \cdots, R_5\}$, and each of the first three racks $\{R_1, R_2, R_3\}$ has two data chunks updated (marked in blue). The rack-coordinated update performs the following two phases to renew the corresponding parity chunks of the same stripe (marked in green) in the racks R_4 and R_5.

In the delta-collecting phase (Fig. 3(a)), it selects two collector racks (R_2 and R_4), which fetch data delta chunks from R_1 and R_3, respectively. This phase transmits four chunks across racks.

In the selective parity update phase (Fig. 3(b)), it updates the parity chunks in R_4 and R_5 using the data delta chunks in the collector racks (i.e., R_2 and R_4). For R_2, as its data delta chunks is more than the parity chunks in either R_4 or R_5, it employs the parity-delta-based update by sending four corresponding parity delta chunks. On the other hand, R_4 has two data-delta chunks whose number is equal to the number of parity chunks in R_5, it uses the data-delta-based update by sending two data delta chunks to R_5 for updating P_3 and P_4. Notice that R_4 will update P_1 and P_2 (also in R_4) through intra-rack transmission, which is not our concern in this article. So this phase delivers six chunks across racks for the parity update.

Finally, the rack-coordinated update in this example needs to transmit 10 chunks in total across racks for the parity update. As a comparison, the conventional delta-based parity update (Section II.C), which directly transmits m parity delta chunks for parity update whenever a data chunk is updated, calls for the delivery of 24 chunks across racks (calculated by multiplying 6 (i.e., number of updated data chunks) with 4 (i.e., number of parity chunks)); while the data-delta-based update (see Section II.D and the example in ;Fig. 2(a)) needs to transmit 12 chunks across racks.

B. Formulation

Assumptions: Our formulation is based on the following assumptions. First, we assume that a rack can only store either data chunks or parity chunks of a stripe (rather than a combination of them). This assumption has also been made in some previous studies [24], [42]. We try to seek the optimal solution with minimized cross-rack update traffic under this assumption. We pose seeking the optimal solution under the mixed storage of the data and parity chunks of a stripe in the same rack as our future work. Second, we assume that a rack can send the data deltas of a stripe to *only one* collector rack (rather than multiple racks) for renewing the parity chunks of the same stripe. This assumption is to simplify the problem formulation and save unnecessary cross-rack traffic. Third, the placement of each stripe should ensure the rack-level fault tolerance, which is commonly considered in extensive studies [15], [33], [37], [38] (Section II.B).

Preliminaries: Suppose that the k data chunks of a stripe are stored in d racks (denoted by $\{R_1, R_2, \ldots, R_d\}$) and the corresponding m parity chunks within the same stripe are distributed in another p racks (denoted by $\{R_{d+1}, R_{d+2}, \ldots, R_{d+p}\}$). For example, in Fig. 3, $d = 3$ and $p = 2$. For clarity, we call the d racks (storing data chunks) and the p racks (storing parity chunks) *data racks* and *parity racks* of this stripe, respectively. Consequently, each rack can serve as either the data rack or the parity rack for different stripes, just depending on the data and parity placement. In Fig. 3, R_1, R_2, and R_3 are data racks of this stripe, while R_4 and R_5 are both parity racks. In the rest of this article, we mainly discuss the parity update of a single stripe. We emphasize that the parity update of multiple stripes can be manipulated independently.

Formulation: We now formalize the rack-coordinated update problem. We first analyze the cross-rack traffic incurred in the delta-collecting phase. We define a *rack-coordinated update solution* $\mathbb{S} = \{L_1, L_2, \cdots, L_{d_c+p_c}\}$, which comprises d_c data racks ($d_c \leq d$) and another p_c parity racks ($p_c \leq p$) to act as the collector racks. We use $\{L_1, L_2, \cdots, L_{d_c}\}$ to denote the d_c selected collector racks that are data racks (i.e., $L_i \in \{R_1, R_2, \cdots, R_d\}$ for $1 \leq i \leq d_c$), and employ $\{L_{d_c+1}, L_{d_c+2}, \cdots, L_{d_c+p_c}\}$ to represent the p_c collector racks that are actually parity racks (i.e., $L_{d_c+j} \in \{R_{d+1}, R_{d+2}, \cdots, R_{d+p}\}$ for $1 \leq j \leq p_c$). For example, in Fig. 3, we select two collector racks, including one data rack (i.e.,

$d_c = 1$ and $L_1 = R_2$) and another parity rack (i.e., $p_c = 1$ and $L_2 = R_4$), and hence the solution $\mathbb{S} = \{L_1 = R_2, L_2 = R_4\}$.

Each collector rack retrieves data delta chunks from the specified data racks in the delta-collecting phase. Let l_i and l'_i (where $l_i < l'_i$) be the number of data delta chunks that the collector rack L_i possesses before and after the delta-collecting phase, respectively. Therefore, a collector rack L_i will receive $l'_i - l_i$ data delta chunks from other data racks in total (where $1 \leq i \leq d_c + p_c$). In the motivating example (Fig. 3(a)), we can identify that the collector rack L_1 (i.e., R_2) receives two chunks across racks, as $l'_1 = 4$ (see Fig. 3(b)) and $l_1 = 2$ (see Fig. 3(a)). Besides, we can deduce that $l_{d_c+j} = 0$ $(1 \leq j \leq p_c)$, as any parity rack solely stores parity chunks before the delta-collecting phase (see assumptions of Section III.B). For example, for the collector rack L_2 (i.e., R_4) in Fig. 3(a), it is a parity rack that does not store any data delta chunk before, so $l_2 = 0$. Consequently, the number of data delta chunks that the $d_c + p_c$ collector racks receive across racks in the delta-collecting phase is

$$T_{\text{collect}} = \sum_{i=1}^{d_c+p_c} (l'_i - l_i) = \sum_{i=1}^{d_c+p_c} l'_i - \sum_{i=1}^{d_c} l_i.$$

We then calculate the cross-rack traffic in the selective parity update phase. For the first d_c collector racks $\{L_i\}_{1\leq i\leq d_c}$, it can update the corresponding t_{d+j} parity chunks for each parity rack R_{d+j} $(1 \leq j \leq p)$ using the selective parity update approach, and hence the cross-rack traffic of the first d_c collector racks is $\sum_{i=1}^{d_c}\sum_{j=1}^{p}\min\{l'_i, t_{d+j}\}$. In Fig. 3(b), $p = 2$ and $t_{d+j} = 2$ for $1 \leq j \leq 2$, so the cross-rack traffic of L_1 is $\sum_{i=1}^{1}\sum_{j=1}^{2}\min\{4, 2\} = 4$. For each of the p_c collector racks L_{d_c+i} $(1 \leq i \leq p_c)$ that is also a parity rack (e.g., $L_2 = R_4$ in Fig. 3(b)), it will perform the selective parity update approach to renew the parity chunks of the other $p - 1$ parity racks (aside from L_{d_c+i} itself). Therefore, the cross-rack traffic caused by the last p_c collector racks is $\sum_{i=1}^{p_c}\sum_{j=1, R_{d+j}\neq L_{d_c+i}}^{p}\min\{l'_{d_c+i}, t_{d+j}\}$. Consequently, the number of delta chunks to be transmitted across racks in the selective parity update phase is

$$T_{\text{update}} = \sum_{i=1}^{d_c}\sum_{j=1}^{p}\min\{l'_i, t_{d+j}\} + \sum_{i=1}^{p_c}\sum_{\substack{j=1\\ R_{d+j}\neq L_{d_c+i}}}^{p}\min\{l'_{d_c+i}, t_{d+j}\}.$$

Finally, the total number of chunks transmitted across racks of the rack-coordinated update solution $\mathbb{S}$ is

$$T_{\mathbb{S}} = T_{\text{collect}} + T_{\text{update}}. \tag{4}$$

Objective: Our objective is to seek the optimal rack-coordinated update solution that minimizes the amount of the cross-rack update traffic (i.e., minimizing $T_{\mathbb{S}}$).

C. Theoretical Analysis

Given a stripe, suppose that the numbers of the updated data chunks in the d data racks are $\{u_1, u_2, \ldots, u_d\}$ (where $u_i \leq m$ for rack-level fault tolerance, see Section II.B) and the numbers of the corresponding m parity chunks in the p parity racks are $\{t_{d+1}, t_{d+2}, \cdots, t_{d+p}\}$ (where $\sum_{j=1}^{p} t_{d+j} = m$). We use R_{d^*} and R_{p^*} to denote the data rack and the parity rack that have the most updated data chunks and parity chunks, respectively. We determine a rack $\overline{L}$ based on the following rule: if the updated data chunks in R_{d^*} is no less than the parity chunks in R_{p^*}, then we set $\overline{L} = R_{d^*}$; otherwise, we set $\overline{L} = R_{p^*}$. We first have Theorem 1 about the efficacy of selecting $\overline{L}$ as a collector rack.

Theorem 1: For any rack-coordinated update solution $\mathbb{S}$ that does not select $\overline{L}$ as a collector rack, we can always find another solution $\mathbb{S}'$ that chooses $\overline{L}$ as a collector rack and introduces no more cross-rack update traffic than $\mathbb{S}$.

Proof: The proof sketch is that we can always find $\mathbb{S}'$ by opportunistically replacing a collector rack in $\mathbb{S}$ by $\overline{L}$. The detailed proof is shown in the appendix of the supplementary file, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TC.2023.3234215.

Theorem 1 implies that even for an optimal rack-coordinated update solution $\mathbb{S}_{\text{opt}}$, we can also construct another optimal one $\mathbb{S}'_{\text{opt}}$ that includes $\overline{L}$ to serve as a collector rack. Therefore, we can have the following corollary.

Corollary 1: We can always find an optimal rack-coordinated update solution that includes $\overline{L}$ as a collector rack.

Given any rack-coordinated update solution $\mathbb{S}'$ that selects $\overline{L}$ as a collector rack, we further deduce that selecting $\overline{L}$ as the *sole* collector rack will introduce no more cross-rack update traffic than $\mathbb{S}'$. Therefore, we have Theorem 2.

Theorem 2: For any rack-coordinated update solution $\mathbb{S}'$ that comprises $\overline{L}$ as a collector rack, we can find another solution $\mathbb{S}^*$ that selects $\overline{L}$ as the sole collector rack and incurs no more cross-rack update traffic than $\mathbb{S}'$.

Proof: The detailed proof is presented in the appendix of the supplementary file, available online.

Based on Corollary 1 and Theorem 2, we can readily deduce the following corollary.

Corollary 2: The solution $\mathbb{S}^*$ minimizes the cross-rack update traffic for the rack-coordinated update mechanism.

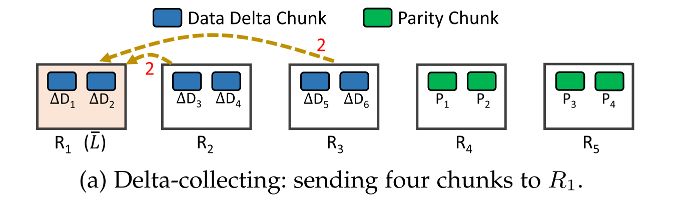
D. Design of RackCU

Based on Corollary 2, we design RackCU, an optimal rack-coordinated update solution that touches the lower bound of the cross-rack update traffic. Algorithm 1 elaborates the main procedure to find the collector rack $\overline{L}$ (Lines 1-8) and update the parity chunks (Lines 9-21).

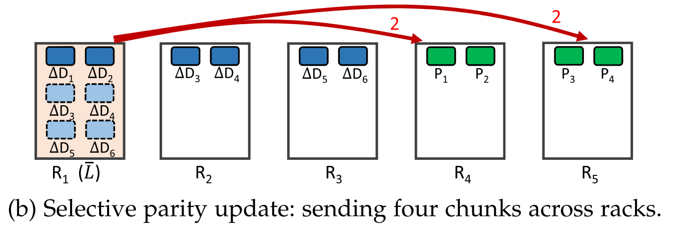
Algorithm Details: We first find the data rack R_{d^*} with the most updated data chunks and the parity rack R_{p^*} with the most parity chunks (Lines 1-2). If the number of updated data chunks in R_{d^*} is no smaller than that of parity chunks in R_{p^*}, we select R_{d^*} as the sole collector rack $\overline{L}$; otherwise, we choose R_{p^*} to be $\overline{L}$ (Lines 4-8). In the delta-collecting phase, each data rack first calculates the data delta chunk for each updated data chunk and sends it to the collector rack (Lines 9-12). In the selective parity update phase, for each parity rack R_{d+j} (where $1 \leq j \leq p$), if the parity chunks that R_{d+j} stores is fewer than the data delta chunks (the number is $\overline{l}'$) that the collector rack possesses now,

Algorithm 1: Procedure of RackCU.

```
Input: $\{R_1, R_2, \ldots, R_d\}$ (data racks)  $\{u_1, u_2, \ldots, u_d\}$
       (distribution of updated data chunks)
       $\{R_{d+1}, R_{d+2}, \ldots, R_{d+p}\}$ (parity racks)
       $\{t_{d+1}, t_{d+2}, \ldots, t_{d+p}\}$ (distribution of parity
  chunks)
Output: The new $n-k$ parity chunks of the same stripe
1:  Find the data rack $R_{d^*}$, where
    $u_{d^*} = \max\{u_i | 1 \leq i \leq d\}$
2:  Find the parity rack $R_{p^*}$, where
    $t_{p^*} = \max\{t_{d+j} | 1 \leq j \leq p\}$
3:  // Determine the sole collector rack
4:  if $u_{d^*} \geq t_{p^*}$ then
5:      $\overline{L} = R_{d^*}$
6:  else
7:      $\overline{L} = R_{p^*}$
8:  end if
9:  // Delta-collecting phase
10:  for $1 \leq i \leq d$ do
11:      Send the $u_i$ data delta chunks from $R_i$ to $\overline{L}$
12:  end for
13:  // Selective parity update phase
14:  for $1 \leq j \leq p$ do
15:      if $\overline{l}' > t_{d+j}$ then
16:          Send the $t_{d+j}$ parity delta chunks to $R_{d+j}$
17:      else
18:          Send the $\overline{l}'$ data delta chunks to $R_{d+j}$
19:      end if
20:      Update the $t_{d+j}$ parity chunks
21:  end for
```



(a) Delta-collecting: sending four chunks to R_1.

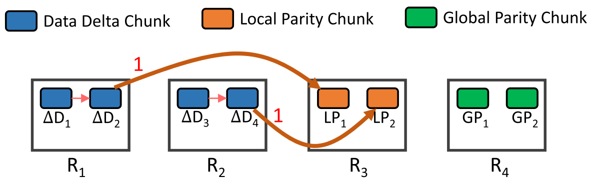


(b) Selective parity update: sending four chunks across racks.

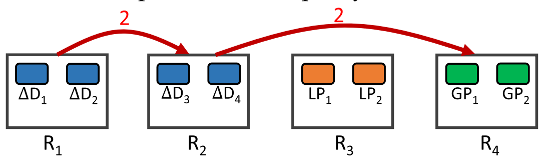
Fig. 4. Example of RackCU, which only needs to transmit eight chunks for the parity update.

then RackCU generates the parity delta chunks for parity update (Lines 14-16). Otherwise, RackCU sends the data delta chunks for the parity update (Lines 17-19). RackCU finally generates the t_{d+j} new parity chunks for R_{d+j} (Line 20).

Example: We show an example via Fig. 4 to clarify the workflow of Algorithm 1. In this example, there are three data racks (i.e., $\{R_1, R_2, R_3\}$) storing updated data chunks (i.e., $d = 3$) and two parity racks (i.e., $\{R_4, R_5\}$ and $p = 2$). All the three data racks have the same number of updated data chunks (i.e., $u_1 = u_2 = u_3 = 2$), so $u_{d^*} = 2$; similarly, we can get $t_{p^*} = 2$, as $t_4 = t_5 = 2$. We select $\overline{L} = R_1$ to serve as the sole collector rack. In the delta-collecting phase, R_1 collects four data delta chunks from R_2 and R_3 (Fig. 4(a)). In the selective parity update phase, as R_1 possesses six data delta chunks (which is more than the parity chunks in any parity rack), it simply performs the parity-delta-based update by sending four corresponding parity delta chunks (Fig. 4(b)). Hence, RackCU transmits eight chunks in total across racks, which is fewer than the example (shown in Fig. 3) that selects two collector racks and sends 10 chunks across racks for the parity update.



(a) Update the local parity chunks.



(b) Update the global parity chunks.

Fig. 5. Example of extension to LRCs.

Extensions to LRCs: Though RackCU mainly focuses on RS codes, we show that it can be extended for another representative family of erasure codes called LRCs [16], [28], which are also used in today's commodity storage systems [3], [16]. Formally, LRCs can be configured via three parameters, namely k, l, and g. LRC(k, l, g) further divides k data chunks of a stripe into l groups with $\frac{k}{l}$ data chunks per group (suppose that k is divisible by l). The $\frac{k}{l}$ data chunks of each group are encoded to generate a local parity chunk. In addition, since the g global parity chunks are all generated from the k data chunks as in RS codes, we can directly apply RackCU to minimize the cross-rack update traffic to the global parity chunks (where the number of data chunks in a rack should be no more than g): it selects a sole collector rack based on the footprints of the updated data chunks and the layout of global parity chunks, and then performs the selective parity update. However, as a local group only has one local parity chunk, RackCU requires to store at most one chunk of a local group in a rack to minimize the cross-rack update traffic to the local parity chunk, which will violate the placement requirement for the updates to the global parity chunks. Hence, we choose to use the selective parity update approach (Section II.D) that can achieve less cross-rack update traffic to the local parity chunk.

Fig. 5 shows an example of the updating process of RackCU for LRC(4, 2, 2). The four data chunks of a stripe are divided into two groups, where D_1 and D_2 are in the first group to generate the local parity chunk LP_1, while D_3 and D_4 are in the second group to calculate the local parity chunk LP_2. The two local parity chunks are stored in the rack R_3. In addition, the two global parity chunks (i.e., $\{GP_1, GP_2\}$) are generated from the four data chunks and stored in R_4. In this example, there are two racks (i.e., $\{R_1, R_2\}$) storing updated data chunks. Fig. 5(a) shows that RackCU updates the two local parity chunks via parity-delta-based update. It first calculates the parity delta chunks and then sends them to the corresponding nodes. Fig. 5(b) shows the process of RackCU to update the global parity chunks. It chooses a collector rack with the most updated data chunks (i.e., R_2) to collect all the data delta chunks. The collector rack then updates all the global parity chunks. Hence, it transmits six chunks across racks for parity update for LRCs in this example. We further show the effectiveness of RackCU on LRCs in Experiment A.4 of Section V.B.

Complexity Analysis: To find the sole collector rack, Algorithm 1 needs to scan the corresponding $d + p$ racks of a stripe and the computation complexity is $O(d + p)$. In the selective parity update phase, Algorithm 1 scans each parity rack for the parity update and the computation complexity is $O(p)$. So the overall computation complexity of Algorithm 1 is $O(d + p)$.

Impact on Intra-Rack Communication Patterns: The proposed RackCU mainly considers the reduction on the cross-rack update traffic, yet it will also change the intra-rack communication patterns. Specifically, it will increase the intra-rack communications for the collector rack, since the collector rack that gathers all the data deltas will perform the selective parity update to update all the parity chunks.

E. Reliability Analysis

We now analyze the reliability improvement gained by RackCU. We choose the metric termed *data loss probability* [33], [35] for reliability analysis, which measures the average likelihood that the stored data are permanently lost in the presence of unexpected rack and node failures. Our main idea is to demonstrate that by minimizing the cross-rack update traffic, RackCU can quickly guarantee the encoding consistency of the updated stripes, thereby improving the overall system reliability.

Settings: Like previous studies [33], [35], this analysis also assesses the data loss probability under the combination of both rack and node failures. Let θ_1 and θ_2 denote the expected lifespan of a node and a rack, respectively. We use f_1 and f_2 to represent the failure probabilities of a node and a rack for a duration of time τ, respectively, calculated by

$$f_1 = 1 - e^{-\frac{\tau}{\theta_1}}, \quad f_2 = 1 - e^{-\frac{\tau}{\theta_2}}, \tag{5}$$

where e is Euler's number. We estimate the values of θ_1 and θ_2 as follows. We first get $\theta_1 = 10$ years from a filed study [9], which assumes that a node fails every 10 years. For the rack failure, we mainly focus on the top-of-rack (ToR) failure. A field study [12] has measured the failure probabilities of five ToRs across tens of geographically distributed data centers (Fig. 4 in [12]), showing that the average failure probability of ToR in one year is 0.0278. Hence, by setting $f_2 = 0.0278$ and $\tau = 1$ year, we can deduce $\theta_2 = 36$ years. Hence, the values of f_1 and f_2 only vary with that of τ. In what follows, we will calculate the loss probabilities of the newly updated data during the update phases for different approaches (the data loss probabilities of the non-updated data in the baseline and RackCU are the same).

Comparison Approaches: We mainly compare RackCU against the baseline delta-based update approach (also called "the baseline" for short), which suggests renewing all the m parity chunks by sending m parity delta chunks whenever a data chunk is updated. Note that although we compare RackCU with three other approaches in the performance evaluation (i.e., the baseline, Parix [18], and CAU [33], see Section V.A), CAU is demonstrated to have a higher data loss probability than the baseline [33] and Parix has the same data loss probability as the baseline. Hence, we believe that the comparison between RackCU and the baseline on the data reliability is reasonable and persuasive.

Assumptions: To simplify our reliability analysis, we make the following assumptions. First, we suppose that the node failure and the rack failure (i.e., the top-of-rack failure in our consideration) happen independently. Second, for an updated data chunk, we assume that its associated m parity chunks are all updated once the corresponding parity delta chunks are successfully transmitted to the associated m parity nodes (i.e., we ignore the time for storage I/Os as the network transmission is commonly considered as the performance bottleneck in repair [23], [38]). Third, we ignore the case when the parity node fails at the time of the parity update, as it does not directly cause the loss of the newly updated data chunks; in this case, we can relocate the parity delta chunks to other interim nodes at first and then move them back to the associated parity nodes (after repair), so as to ensure the reliability of the newly update data chunks during the parity update.

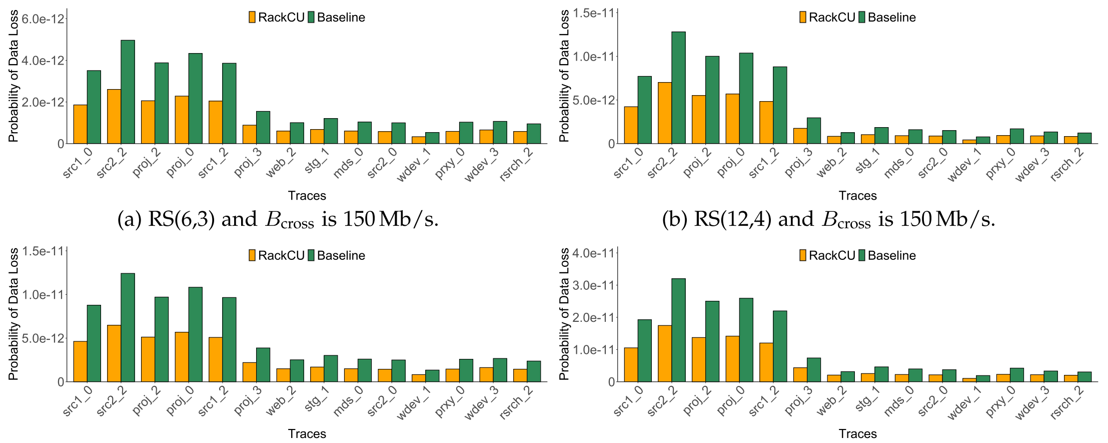
Failure Events and Probabilities: We then calculate the data loss probabilities of the baseline and RackCU based on the following analysis. We consider the deployment of RS(k, m) in a storage system, which comprises n nodes ($n \geq k + m$) and r racks with $\frac{n}{r}$ nodes per rack (suppose that n is divisible by r). Suppose that u data chunks ($1 \leq u \leq k$) within the same stripe are updated (without updating the corresponding parity chunks at this time), whose footprints are across a racks ($1 \leq a \leq r$). Hence, the newly updated data will be pertinently lost, as long as any one of the u nodes (where the u newly updated data chunks reside) fails before the associated m parity chunks are successfully renewed. Let E_{intact} denote the event that all the u nodes with data updated and the associated a racks are intact. Hence, the probability of E_{intact} (denoted by $\Pr(E_{\text{intact}})$) can be given by:

$$\Pr(E_{\text{intact}}) = (1 - f_1)^u \cdot (1 - f_2)^a. \tag{6}$$

We use E_{dl} to represent the event that the newly updated data are lost. Hence, its probability $\Pr(E_{\text{dl}})$ can be computed as:

$$\Pr(E_{\text{dl}}) = 1 - \Pr(E_{\text{intact}}) = 1 - (1 - f_1)^u \cdot (1 - f_2)^a. \tag{7}$$

Results: In this analysis, we consider two practical erasure codes: (i) RS(6,3), which has been used in Hadoop HDFS [4] and QFS [27], and (ii) RS(12,4), which is considered in Windows Azure Storage [16]. We deploy the two erasure codes in a storage system with 100 nodes, which are further organized into 10 racks (i.e., 10 nodes per rack). We select 14 representative traces from MSR Cambridge Traces (MSR) [25] with different average update sizes, where the first seven traces (i.e., `src1_0`,



(a) RS(6,3) and B_{cross} is 150 Mb/s. (b) RS(12,4) and B_{cross} is 150 Mb/s.

(c) RS(6,3) and B_{cross} is 60 Mb/s. (d) RS(12,4) and B_{cross} is 60 Mb/s.

Fig. 6. Reliability analysis: data loss probabilities of RackCU and the baseline.

src2_2, proj_2, proj_0, src1_2, proj_3, and web_3) have larger update sizes (see Fig. 8). We configure the chunk size to 4 KB and set the intra-rack bandwidth as 3 Gb/s (which is consistent with the value in our testbed evaluation, see Section V.C). We then vary the cross-rack bandwidth (denoted by B_{cross}) from 60 Mb/s (which is the cross-region bandwidth measured in our testbed experiments, see Experiment B.4 of Section V.C) to 150 Mb/s (i.e., one twentieth of the intra-rack bandwidth), respectively. We then calculate the update time of each stripe to serve as the duration of τ (see (5)) and get the data loss probabilities for each updated stripe under RackCU and the baseline. For each trace, we finally calculate the average data loss probabilities of the two update approaches and show the results in Fig. 6. We can obtain two findings.

First, RackCU significantly reduces the data loss probability compared to the baseline for a board spectrum of real-world traces. The underlying reason is that RackCU effectively reduces the cross-rack update traffic and leads to shorter update time (i.e., a smaller value of τ). Specifically, when the cross-rack bandwidth is 150 Mb/s, for RS(6,3), the average data loss probabilities of RackCU and the baseline are 1.2×10^{-12} and 2.1×10^{-12} (Fig. 6(a)), respectively; while for RS(12,4), the average data loss probabilities of RackCU and the baseline are 2.6×10^{-12} and 4.6×10^{-12} (Fig. 6(b)), respectively. When the cross-rack bandwidth is more stringent and drops to 60 Mb/s, the update procedure lengthens and hence the average data loss probabilities of RackCU and the baseline both increase. For RS(6,3), the average data loss probabilities of RackCU and the baseline increase to 2.9×10^{-12} and 5.4×10^{-12} (Fig. 6(c)), respectively; while for RS(12,4), the average data loss probabilities of RackCU and the baseline increase to 6.4×10^{-12} and 1.1×10^{-11} (Fig. 6(d)), respectively.

Second, RackCU is more advantageous in the environments with larger update sizes. In particular, compared to the baseline, RackCU reduces the data loss probability by 44.3% for the first seven traces. The reduction shrinks to 41.1% for the last seven traces. This is because RackCU can gain more traffic reduction for the traces with larger update sizes (Experiment A.1, Section V.B).

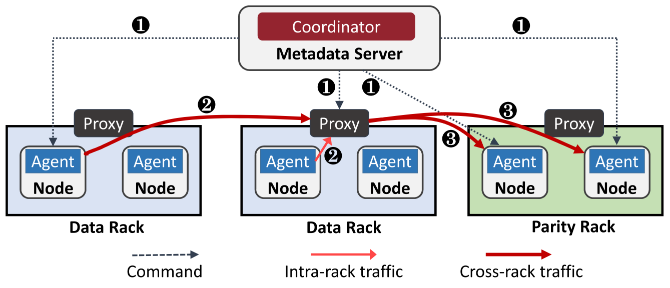


Fig. 7. System architecture of RackCU.

IV. Implementation

We implement a RackCU prototype in C with around 2,800 lines of codes (LoC), and realize the encoding functionality via Jerasure v1.2 [30].

System Architecture: Fig. 7 presents the architecture of our RackCU prototype, which comprises three components: a *coordinator* sitting on the metadata server, a *proxy* in each rack, and an *agent* on every node. The coordinator manages each chunk's metadata, including the stripe identity to which the chunk belongs and the node where a chunk resides. The proxy is responsible for receiving the data delta chunks once the rack it resides serves as a collector rack, while the agent is in charge of interacting with the coordinator, sending the data delta chunks, and calculating the new parity chunks.

Operating Flow: To update data chunks, the client first sends an update request with the corresponding chunk ID to the coordinator. The coordinator then seals the stripe identity and the node associated to this chunk into an *access token*, and returns it to the client. Instructed by the access token, the client writes new data chunks to the target nodes and returns an ACK to imply the completeness of the update operation.

Fig. 7 then illustrates the parity update procedure. The coordinator first determines the collector rack based on the footprints of the updated data chunks and the associated parity chunks, and launches commands to the agents of the involved nodes as well as the proxy of the collector rack for instructing the parity update

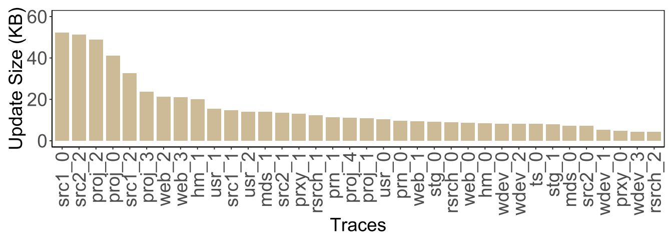


Fig. 8. Update sizes of MSR Cambridge Traces [25].

(step ❶). Upon receiving the command, the agent calculates the data delta chunk and sends it to the proxy of the collector rack (step ❷). After collecting enough data delta chunks, the proxy then performs the selective parity update to update the parity chunks. Once generating the new parity chunk, the agent of the *parity node* (i.e., the node storing parity chunks) commits an ACK to the coordinator. The coordinator understands the completeness of the parity update of a stripe once successfully collecting ACKs from all the m parity nodes of this stripe.

V. Performance Evaluation

We conduct extensive performance evaluation via both of large-scale simulation and real-world cloud data center experiments to study the real performance of RackCU. We summarize our major findings below: compared to the state-of-the-art algorithms, (1) RackCU saves 16.5-77.1% of cross-rack update traffic (Section V.B); (2) RackCU increases 24.9-772.0% of update throughput (Section V.C).

A. Preliminaries

Traces: We assess the update performance via trace-driven evaluation. We employ MSR Cambridge Traces (MSR) [25], which record the I/O patterns from 13 core servers of a data center. Every trace consists of successive read/write requests, each of which records the request type (read or write), the start position of the requested data, and the request size, etc. We first classify the 36 traces based on the *update size* by averaging the operating sizes of all the update requests in a trace. Fig. 8 shows that the update sizes dramatically vary across different traces, ranging from 4.3 KB to 52.0 KB.

Counterparts: We compare RackCU to another three state-of-the-art approaches: (i) cross-rack-aware update (CAU) [33], (ii) the baseline delta-based update approach, and (iii) Parix [18]. We summarize these three approaches as below.

- *CAU [33]:* CAU updates parity chunks simply through the selective parity update [1]: if the updated data chunks of a data rack are more than the parity chunks of a parity rack, CAU updates those parity chunks via transmitting parity delta chunks; otherwise, CAU updates them through delivering data delta chunks.

1.We remove the data grouping and interim replication from CAU [33] and let CAU merely perform the selective parity update. We emphasize that RackCU can achieve higher reliability than the original CAU [33].

- *The baseline:* When a data chunk is updated, the baseline will send the m corresponding parity delta chunks to generate the new parity chunks based on (2).
- *Parix [18]:* Parix updates parity chunks via two phases: (1) for a data chunk that is updated for the first time, Parix sends both the old and the new data chunks to all the m parity nodes and keeps them in an append-only log; (2) for the data chunk that has been updated before, Parix solely transmits the new data chunk to all the m parity nodes. To update a parity chunk, each parity node reads the old and the newest data chunks from local storage to derive the new parity chunk based on (2).

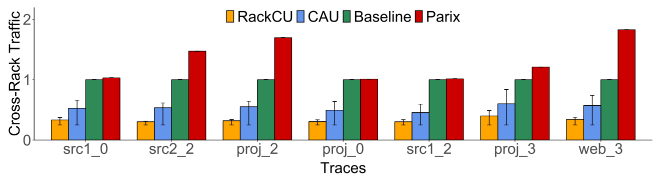
We summarize that Parix incurs additional network traffic (for transmitting the old data chunk updated for the first time), but avoids frequent storage I/O operations (for reading the old parity chunk) to generate the new parity chunk.

B. Large-Scale Simulation

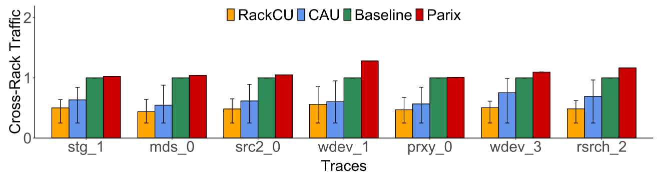
We first carry out large-scale simulation. We remove the storage and network operations of the RackCU prototype, and keep eyes on the amount of induced cross-rack traffic.

Experimental Setup: We use the following default configurations in this simulation. We deploy RS(12,4) (also considered in Windows Azure Storge [16]) in a data center, which is built atop of 200 nodes with 10 racks (i.e., 20 nodes per rack). The data chunks and parity chunks within the same stripe are stored in different racks. If the number of racks is greater than $k+m$ (i.e., number of chunks of a stripe), we place a stripe across $k+m$ racks for maximizing rack-level fault tolerance. We then partition the address space of each trace into units of chunks and set the chunk size as 4 KB. When replaying a trace, we extract the start address and the operating size in each update request, and identify the chunk IDs to be updated. We then update the data chunks as well as the corresponding parity chunks by using the four parity update approaches, and measure the introduced cross-rack update traffic. We repeat each experiment for ten runs and show the average results as well as the error bars indicating the maximum and minimum values across the test. Note that the error bars of the baseline and Parix in the simulation are all zeros, since we separate data and parity chunks of a stripe across different racks. Both the baseline and Parix send m parity delta chunks to update the m parity chunks (resided in other racks) whenever a data chunk is updated, so their cross-rack update traffic is constant under a given trace.

Experiment A.1 (Impact of Update Size): We first study the impact of the update size by selecting 14 traces: seven traces with larger update sizes (i.e., 38.6 KB on average) and another seven traces with smaller update sizes (i.e., 5.8 KB on average). Fig. 9 shows the results, which are normalized by that of the baseline for clarity. Among all the 14 traces, RackCU reduces 29.8%, 58.9%, and 64.4% of the cross-rack update traffic on average compared to CAU, the baseline, and Parix, respectively. In addition, RackCU is more advantageous on saving the cross-rack update traffic for the traces with larger update sizes. Statistically, RackCU saves 38.2%, 67.0%, and 75.1% of the cross-rack update traffic on average compared to CAU, the baseline, and

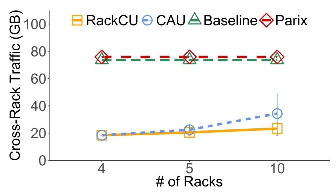


(a) Comparison on the traces with larger update sizes.

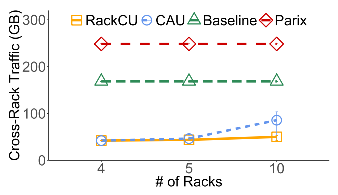


(b) Comparison on the traces with smaller update sizes.

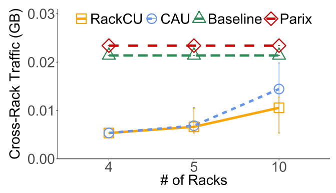
Fig. 9. Experiment A.1 (Impact of update size). Here, we normalize the results by that of the baseline for clear presentation (as the cross-rack traffic varies dramatically across different traces). The smaller value is better.



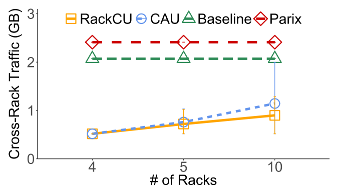
(a) src1_0



(b) src2_2

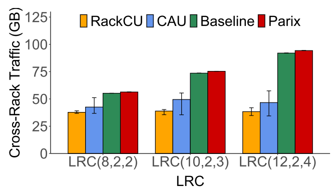


(c) wdev_3

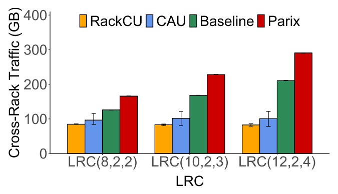


(d) rsrch_2

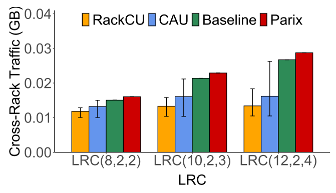
Fig. 11. Experiment A.3 (Impact of number of racks).



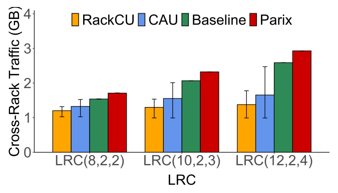
(a) src1_0



(b) src2_2

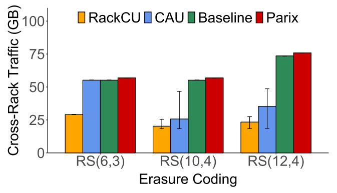


(c) wdev_3

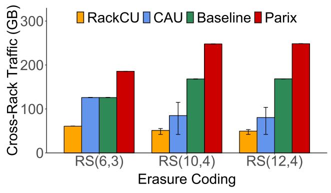


(d) rsrch_2

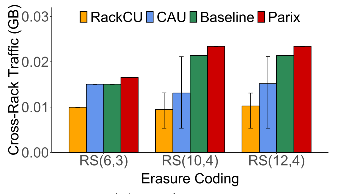
Fig. 12. Experiment A.4 (Extension to LRCs).



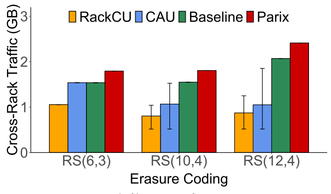
(a) src1_0



(b) src2_2



(c) wdev_3



(d) rsrch_2

Fig. 10. Experiment A.2 (Impact of erasure coding).

Parix for the seven traces with larger update sizes (Fig. 9(a)); the reductions shrink to 22.1%, 50.9%, and 55.1% for other seven traces with small update sizes (Fig. 9(b)), respectively.

Experiment A.2 (Impact of Erasure Coding): We evaluate the impact of erasure coding parameters via choosing two traces (i.e., `src1_0` and `src2_2`) with larger update sizes and another two traces (i.e., `wdev_3` and `rsrch_2`) with smaller update sizes. We focus on the following three erasure coding schemes: RS(6,3) (selected in QFS [27] and Hadoop HDFS [4]), RS(10,4) (deployed in Facebook f4 [24]), and RS(12,4) (considered in Windows Azure Storage [16]). Fig. 10 implies that RackCU retains its efficacy across different erasure coding schemes. In a nutshell, RackCU can reduce 33.3%, 54.1%, and 60.4% of the cross-rack update traffic on average compared to CAU, the baseline, and Parix, respectively.

Experiment A.3 (Impact of Number of Racks): We assess the impact of the number of racks. We organize the 200 nodes into four racks (i.e., 50 nodes per rack), five racks (i.e., 40 nodes per rack), and 10 racks (i.e., 20 nodes per rack), respectively. Fig. 11 indicates that the amounts of the cross-rack update traffic incurred by RackCU and CAU both increase with the number of racks. The rationale is that when a data center comprises more racks, each rack is more likely to store fewer chunks of a stripe, and hence RackCU and CAU have to access more racks to accomplish the parity update. Besides, the amounts of the cross-rack update traffic caused by the baseline and Parix stay constant even when the number of racks varies. The reason is that we separate the storage of data chunks and parity chunks across different racks. As the baseline and Parix directly update each parity chunk in other racks, the cross-rack update traffic depends on the number of parity chunks.

Experiment A.4 (Extension to LRCs): To demonstrate the generality, we also evaluate the performance of RackCU when being deployed atop LRCs. In this experiment, we choose LRC(8, 2, 2), LRC(10, 2, 3) and LRC(12, 2, 4) for evaluation, where all the three codes are considered in previous work [17]. We organize 200 nodes in ten racks (i.e., with 20 nodes per rack), and randomly select a rack to place the local parity chunk without violating the rack-level fault tolerance guarantee. Fig. 12 shows that RackCU can still dramatically reduce the cross-rack update traffic for different LRCs; in particular, it reduces the cross-rack update traffic by 16.5%, 49.5% and 59.8% compared with CAU, the baseline and Parix, respectively. Besides, we identify that the cross-rack update traffic of the baseline and Parix dramatically increases with the value of the g (i.e., the number of global parity chunks configured in LRCs), while that of RackCU and CAU

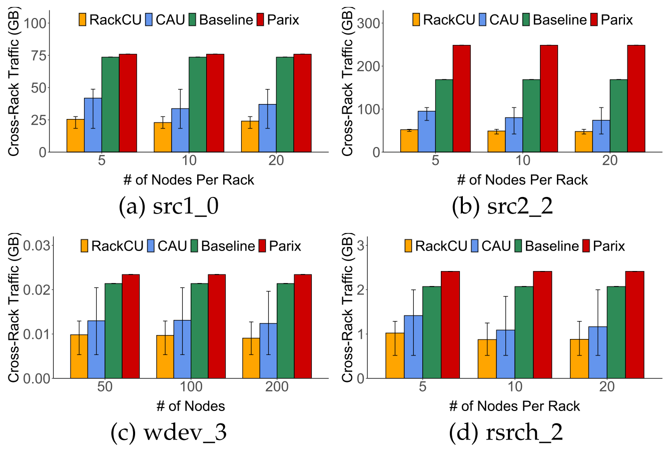


Fig. 13. Experiment A.5 (Impact of number of nodes per rack).

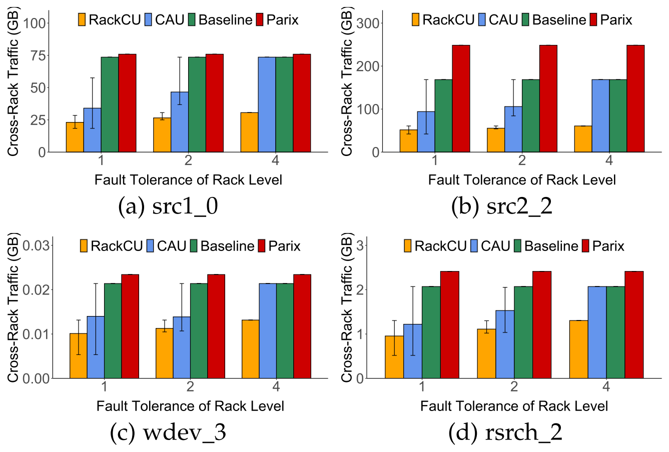


Fig. 14. Experiment A.6 (Impact of rack-level fault tolerance degrees).

behaves to be much more stable, since both the baseline and Parix transmit g parity delta chunks whenever a data chunk is updated, while RackCU and CAU employ the selective parity update to reduce the cross-rack update traffic.

Experiment A.5 (Impact of Number of Nodes per Rack): We then investigate the impact of number of nodes per rack on the cross-rack update traffic. We fix the number of racks to 10 and measure the cross-rack update traffic when the number of nodes per rack is changed from 5 to 20. Fig. 13 shows the results. We can observe that the cross-rack update traffic of all the four methods changes marginally under different numbers of nodes per rack. This is because the change in the number of nodes per rack actually does not affect the number of racks that a stripe spans (which is determined by the number of chunks stored per rack instead), so it does not change the resulting cross-rack update traffic. We can also observe that RackCU still achieves the least cross-rack update traffic; it reduces the cross-rack update traffic by 38.7%, 69.4% and 77.1% on average compared with CAU, the baseline and Parix, respectively.

Experiment A.6 (Impact of Rack-Level Fault Tolerance Degrees): We finally evaluate the cross-rack update traffic under different rack-level fault tolerance degrees. We select RS(12,4) and deploy each stripe according to the following conditions to reach different degrees of rack-level fault tolerance: 1) we place at most one chunk of a stripe in each rack to tolerate any four rack failures; 2) we store at most two chunks of a stripe in a rack to tolerate any double rack failures; and 3) we keep at most four chunks of a stripe in a rack to tolerate any single rack failure. We also separate the data and parity chunks of the same stripe into different racks. We then measure the resulting cross-rack update traffic and show the results in Fig. 14. We have the following two observations.

First, RackCU always achieves the least update traffic under different rack-level fault tolerance degrees. Specifically, it reduces the cross-rack update traffic of CAU, the baseline, and Parix by 52.3%, 65.7%, and 74.3% on average under different rack-level fault tolerance degrees, respectively.

Second, the cross-rack update traffic of RackCU and CAU both increases with the rack-level fault tolerance degrees, while those of the baseline and Parix keep unchanged. This is because for the baseline and Parix, whenever a data chunk is updated, they always transmit m parity delta chunks ($m = 4$ in this experiment) to update the corresponding m parity chunks, which are stored in another dedicated rack (racks). Hence, their cross-rack update traffic is directly determined by the value of m, rather than the number of parity chunks within a rack. On the other hand, the update approaches selected in RackCU and CAU depend on the number of parity chunks within a rack, and therefore their cross-rack update traffic is affected by the number of parity chunks within a rack.

C. Testbed Experiments

We further assess RackCU on Alibaba Cloud ECS [1] to unveil its performance in a real-world cloud data center. We set up 18 virtual machine instances with the type of `ecs.g6.large`. Each instance is equipped with 2vCPU (2.5GHz Intel Xeon Platinum 8269CY) and 8 GB memory. The operating system is Ubuntu 18.04 and the network bandwidth is around 3 Gb/s (measured by `iperf`).

Experimental Setup: Among the 18 instances, we deploy the RackCU coordinator on one instance to serve as the metadata server, and use anther one to act as the client. We then organize the remaining 16 instances into eight racks (two instances per rack) and run both RackCU proxy and agent on each instance. We choose RS(12,4) (i.e., each rack stores two chunks of a stripe) and set the chunk size as 4 KB. We use the Linux tool `tc` to throttle the cross-rack bandwidth, and evaluate the *update throughput* (i.e., the size of data updated per unit time) by replaying the first 1,000 update requests of each trace.

Experiment B.1 (Impact of Cross-Rack Bandwidth): We measure the update throughputs by varying the cross-rack bandwidth from 50 Mb/s to 200 Mb/s. Fig. 15 indicates that the update throughput of the four approaches all increase with the cross-rack bandwidth. The baseline outperforms CAU when the cross-rack bandwidth is larger than 100 Mb/s as the storage bandwidth becomes the bottleneck at this time. Overall, RackCU improves the update throughput by 106.8%, 88.2%, and 262.2% on average across different cross-rack bandwidth and traces,

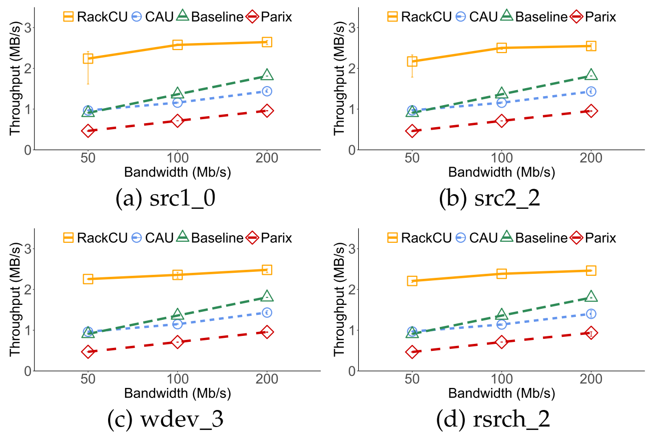


Fig. 15. Experiment B.1 (Impact of cross-rack bandwidth). The larger value is better.

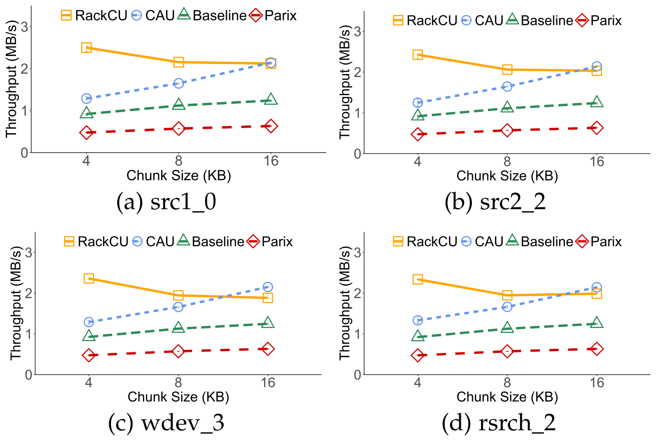


Fig. 16. Experiment B.2 (Impact of chunk size with MSR traces).

when compared to CAU, the baseline, and Parix, respectively. In addition, the update throughput is small (around several MB/s) as it is restricted by both the cross-rack bandwidth and storage bandwidth of small accesses.

In addition, the four traces showcase the similar trend due to the following two reasons. First, we replay the first 1,000 write requests for each trace in testbed experiments. The average update sizes of the four traces (i.e., `src1_0`, `src2_2`, `wdev_3`, and `rsrch_2`) are 10.6 KB, 5.6 KB, 4.4 KB, and 4.3 KB, respectively. Since the chunk size is set from 4 KB to 16 KB, the difference on the number of updated chunks among the four traces is small (with at most two chunks). Second, we concern the resulting update throughput, calculated as the ratio of the size of the updated data chunks and the time duration of the update process. For a given update approach, its update throughputs under the four traces are similar, as the larger update size introduces more cross-rack update traffic and also the longer update process.

Experiment B.2 (Impact of Chunk Size With MSR Traces): We study the update throughput under different chunk sizes. Fig. 16 shows that RackCU improves the update throughput by 34.2%, 101.1%, and 292.6% on average across different chunk sizes and

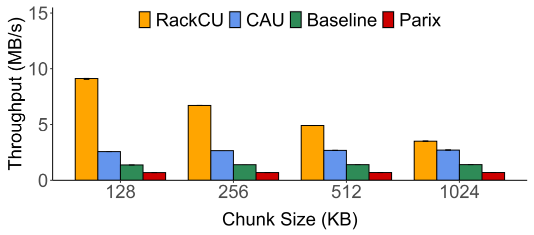


Fig. 17. Experiment B.3 (Impact of chunk size with YCSB workloads).

traces, compared to CAU, the baseline, and Parix, respectively. Besides, when the chunk size is 16 KB, the efficacy of RackCU recedes. The major cause is that the average chunk sizes of the 1,000 update requests of the four traces range from 2.9 KB to 10.6 KB. Hence, when the chunk size is 16 KB, each update request is likely to manipulate only one chunk, hence degrading the efficacy of RackCU. Hence, we suggest deploying RackCU in the scenario with multiple chunks updated per update request.

Experiment B.3 (Impact of Chunk Size With YCSB Workloads): To demonstrate the generality of RackCU, we also evaluate the effectiveness of RackCU for the traces selected from different repositories. Here, we generate a representative YCSB workload [10] with 50% of reads and another 50% of writes, which follow the Zipfian distribution (with the alpha value of 0.99 by default). We then set the object size to 10 MB and configure the update size as 1 MB. We partition an object into multiple fixed-size chunks and distribute the chunks uniformly across racks. We vary the chunk size from 128 KB to 1 MB and measure the resulting update throughput.

Fig. 17 implies that RackCU outperforms CAU, the baseline, and Parix under different chunk sizes. Specifically, RackCU improves the update throughput by 130.4%, 338.6% and 772.0% on average when compared with CAU, the baseline, and Parix, respectively. Besides, we also observe that the update throughput gradually decreases when the chunk size is enlarged, since fewer data chunks are updated in an update request, which decays the effectiveness on suppressing the cross-rack update traffic (see Experiment B.2).

Experiment B.4 (Performance in Geo-Distributed Environments): We finally investigate the update performance of RackCU in geo-distributed environments. We deploy RackCU across eight different geo-distributed regions, namely Hangzhou (HZ), Shanghai (SH), Qingdao (QD), Beijing (BJ), Hohhot (HH), Wulanchabu (WL), Shenzhen (SZ), and Heyuan (HY), where each region comprises two instances with the type of `ecs.g6.large` (hence there are 16 instances in total). We measure via `iperf` that the average bandwidth of any two different regions is 60.85 Mb/s and the ratio of the intra-region and the cross-region is 52.6 on average. Table I lists the bandwidths among the regions (in unit of Mb/s). We deploy RS(12,4) across the 16 instances (i.e., two chunks per region) and set the chunk size to 256 KB. We generate three YCSB workloads following the Zipfian distribution (with the alpha value of 0.99): (i) the read-heavy workload with 75% of reads and another 25% of writes; (ii) the read-write-balanced workload with 50% of reads and another 50% of writes; and (iii) the write-heavy workload

TABLE I
THE BANDWIDTH AMONG REGIONS (RGS) (UNIT: MB/S)

RGs	HZ	SH	QD	BJ	HH	WL	SZ	HY
HZ	3,239.25	62.53	61.40	61.60	60.07	61.57	61.43	60.97
SH	62.40	3,112.96	62.23	61.83	61.13	61.23	61.97	60.93
QD	61.37	62.17	3,112.96	62.37	56.93	61.87	56.90	59.80
BJ	61.50	60.30	62.57	3,235.84	62.27	62.53	56.50	58.73
HH	61.10	61.43	59.03	62.33	3,246.08	62.30	57.97	58.57
WL	61.37	61.87	62.30	62.63	62.13	3,269.97	59.67	58.97
SZ	61.17	61.53	58.57	61.40	59.60	60.27	3,112.96	62.60
HY	61.00	61.83	61.13	58.53	57.20	61.17	62.63	3,256.32

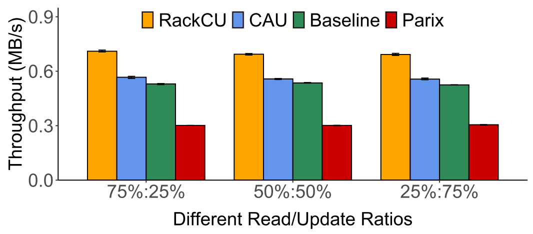


Fig. 18. Experiment B.4 (Performance in geo-distributed environments).

with 25% of reads and another 75% of writes. We then measure the update throughputs of the four approaches under the three different workloads.

Fig. 18 shows that RackCU improves the update throughputs by 24.9%, 29.5% and 130.1% when compared to CAU, the baseline, and Parix, respectively. This experiment also demonstrates that RackCU still preserves its effectiveness on accelerating the update procedure for the geo-distributed environment.

VI. RELATED WORK

Delta-Based Updates: Erasure-coded systems often manipulate parity update via delta-based update approaches. Parity logging [39] appends parity deltas to a dedicated log device for avoiding random small writes. CodFS [6] couples in-place data update and log-based parity update to tailor update performance and repair performance. To avoid frequent disk seeks in the parity update, Parix [18] appends the old and latest data chunks, and only calculates the delta of them in the parity update. UCODR [32] selects the combination of appropriate data and parity chunks to mitigate the storage I/O in parity update. T-Update [29] constructs a minimum spanning tree to guide the prorogation of the parity update. All the above studies do not consider the reduction of the cross-rack update traffic. CAU [33] appends new data chunks and defers the parity update to reduce the cross-rack update traffic, at the cost of system reliability degradation. Compared to CAU, RackCU achieves higher reliability by immediately updating parity chunks and theoretically minimizes the cross-rack update traffic.

Data Placement: Some studies utilize access characteristics to mitigate the parity update. PDP [36] arranges sequential data chunks to generate the same parity chunk for reducing the parity update of sequential writes. CASO [34] organizes correlated data chunks that are likely to be updated together into the same stripe. CAU [33] relocates updated data chunks within the same rack to reduce cross-rack traffic in the parity update. RackCU is orthogonal and complementary to these studies for further mitigating the cross-rack update traffic.

Rack-Aware Operations: Previous studies also notice the scarcity of cross-rack bandwidth in some system operations. LRCs [16], [28] keep a local parity chunk within a rack to avoid cross-rack data transfers in single chunk's repair. Some studies [15], [20], [21], [23], [37], [38] decompose a chunk's repair into many sub-stages that are performed within racks in parallel, such that the cross-rack repair traffic can be reduced. In addition, some studies [19], [41] consider the rack-aware transition. As a comparison, our RackCU pays close attention to the reduction of the cross-rack update traffic in data centers.

VII. CONCLUSION

We study how to reduce cross-rack update traffic in erasure-coded data centers. We propose a rack-coordinated update mechanism that comprises two phases: (i) a delta-collecting phase that carefully chooses collector racks for retrieving data delta chunks, and (ii) another selective parity update phase that renews the parity chunks through selecting the appropriate parity update approach. We then design RackCU, an optimal rack-coordinated update solution that minimizes the cross-rack update traffic. We finally conduct in-depth reliability analysis, large-scale simulation, and extensive testbed experiments, showing that RackCU can vastly reduce the cross-rack update traffic and improve the update throughput, hence gaining higher data reliability.

ACKNOWLEDGMENTS

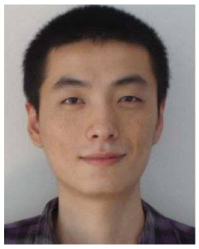
A preliminary version [13] of this article was presented at the IEEE International Conference on Computer Communications (INFOCOM'21). In this version, we further extend RackCU for LRCs, analyze RackCU on the data reliability, and uncover more experimental findings.

REFERENCES

[1] Alibaba Cloud Elastic Compute Service. [Online]. Available: https://www.alibabacloud.com/product/ecs
[2] Erasure Coding in Ceph, 2016. [Online]. Available: https://docs.ceph.com/en/latest/rados/operations/erasure-code
[3] Locally Repairable Codes in Ceph, 2016. [Online]. Available: https://docs.ceph.com/en/latest/rados/operations/erasure-code-lrc
[4] Apache Hadoop 3.0.0, 2017. [Online]. Available: https://hadoop.apache.org/docs/r3.0.0/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html
[5] F. Ahmad, S. Chakradhar, A. Raghunathan, and T. Vijaykumar, "Shuffle-Watcher: Shuffle-aware scheduling in multi-tenant MapReduce clusters," in *Proc. USENIX Annu. Tech. Conf.*, 2014, pp. 1–12.
[6] J. Chan, Q. Ding, P. Lee, and H. Chan, "Parity logging with reserved space: Towards efficient updates and recovery in erasure-coded clustered storage," in *Proc. USENIX Conf. File Storage Technol.*, 2014, pp. 163–176.
[7] H. Chen et al., "Efficient and available in-memory KV-store with hybrid erasure coding and replication," *ACM Trans. Storage*, vol. 13, no. 3, pp. 1–30, 2017.
[8] M. Chowdhury, S. Kandula, and I. Stoica, "Leveraging endpoint flexibility in data-intensive clusters," in *Proc. ACM SIGCOMM Conf.*, 2013, pp. 231–242.

[9] A. Cidon, R. Escriva, S. Katti, M. Rosenblum, and E. G. Sirer, “Tiered replication: A cost-effective alternative to full cluster geo-replication,” in *Proc. USENIX Annu. Tech. Conf.*, 2015, pp. 31–43.
[10] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with YCSB,” in *Proc. ACM Symp. Cloud Comput.*, 2010, pp. 143–154.
[11] D. Ford et al., “Availability in globally distributed storage systems,” in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2010, pp. 61–74.
[12] P. Gill, N. Jain, and N. Nagappan, “Understanding network failures in data centers: Measurement, analysis, and implications,” in *Proc. ACM SIGCOMM Conf.*, 2011, pp. 350–361.
[13] G. Gong, Z. Shen, S. Wu, X. Li, and P. P. C. Lee, “Optimal rack-coordinated updates in erasure-coded data centers,” in *Proc. IEEE Conf. Comput. Commun.*, 2021, pp. 1–10.
[14] A. Greenberg et al., “VL2: A scalable and flexible data center network,” in *Proc. ACM SIGCOMM Conf. Data Commun.*, 2009, pp. 51–62.
[15] Y. Hu et al., “Optimal repair layering for erasure-coded data centers: From theory to practice,” *ACM Trans. Storage*, vol. 13, no. 4, 2017, Art. no. 33.
[16] C. Huang et al., “Erasure coding in windows Azure storage,” in *Proc. USENIX Annu. Tech. Conf.*, 2012, Art. no. 2.
[17] O. Kolosov, G. Yadgar, M. Liram, I. Tamo, and A. Barg, “On fault tolerance, locality, and optimality in locally repairable codes,” *ACM Trans. Storage*, vol. 16, no. 2, pp. 1–32, 2020.
[18] H. Li et al., “PARIX: Speculative partial writes in erasure-coded systems,” in *Proc. USENIX Annu. Tech. Conf.*, 2017, pp. 581–587.
[19] R. Li, Y. Hu, and P. P. C. Lee, “Enabling efficient and reliable transition from replication to erasure coding for clustered file systems,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 9, pp. 2500–2513, Sep. 2017.
[20] R. Li, X. Li, P. P. C. Lee, and Q. Huang, “Repair pipelining for erasure-coded storage,” in *Proc. USENIX Annu. Tech. Conf.*, 2017, pp. 567–579.
[21] X. Li, R. Li, P. P. C. Lee, and Y. Hu, “OpenEC: Toward unified and configurable erasure coding management in distributed storage systems,” in *Proc. USENIX Conf. File Storage Technol.*, 2019, pp. 331–344.
[22] V. Liu, D. Zhuo, S. Peter, A. Krishnamurthy, and T. Anderson, “Subways: A case for redundant, inexpensive data center edge links,” in *Proc. ACM Conf. Emerg. Netw. Experiments Technol.*, 2015, pp. 27:1–27:13.
[23] S. Mitra, R. Panta, M.-R. Ra, and S. Bagchi, “Partial-parallel-repair (PPR): A distributed technique for repairing erasure coded storage,” in *Proc. Eur. Conf. Comput. Syst.*, 2016, Art. no. 30.
[24] S. Muralidhar et al., “f4: Facebook’s warm BLOB storage system,” in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2014, pp. 383–398.
[25] D. Narayanan, A. Donnelly, and A. Rowstron, “Write off-loading: Practical power management for enterprise storage,” *ACM Trans. Storage*, vol. 4, no. 3, pp. 1–23, 2008.
[26] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum, “Fast crash recovery in RAMCloud,” in *Proc. ACM Symp. Operating Syst. Princ.*, 2011, pp. 29–41.
[27] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly, “The quantcast file system,” *Proc. VLDB Endowment*, vol. 6, no. 11, pp. 1092–1101, 2013.
[28] D. S. Papailiopoulos and A. G. Dimakis, “Locally repairable codes,” *IEEE Trans. Inf. Theory*, vol. 60, no. 10, pp. 5843–5855, Oct. 2014.
[29] 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. IEEE 35th Annu. Int. Conf. Comput. Commun.*, 2016, pp. 1–9.
[30] J. S. Plank, S. Simmerman, and C. D. Schuman, “Jerasure: A library in C/C++ facilitating erasure coding for storage applications-version 1.2,” Univ. Tennessee, Knoxville, TN, USA, Tech. Rep. CS-08–627, vol. 23, 2008.
[31] I. Reed and G. Solomon, “Polynomial codes over certain finite fields,” *J. Soc. Ind. Appl. Math.*, vol. 8, no. 2, pp. 300–304, 1960.
[32] 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.
[33] Z. Shen and P. Lee, “Cross-rack-aware updates in erasure-coded data centers,” in *Proc. Int. Conf. Parallel Process.*, 2018, pp. 1–10.
[34] Z. Shen, P. Lee, J. Shu, and W. Guo, “Correlation-aware stripe organization for efficient writes in erasure-coded storage systems,” in *Proc. IEEE 36th Symp. Reliable Distrib. Syst.*, 2017, pp. 134–143.
[35] Z. Shen, S. Lin, J. Shu, C. Xie, Z. Huang, and Y. Fu, “Cluster-aware scattered repair in erasure-coded storage: Design and analysis,” *IEEE Trans. Comput.*, vol. 70, no. 11, pp. 1861–1874, Nov. 2021.
[36] Z. Shen, J. Shu, and Y. Fu, “Parity-switched data placement: Optimizing partial stripe writes in XOR-coded storage systems,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 11, pp. 3311–3322, Nov. 2016.
[37] Z. Shen, J. Shu, Z. Huang, and Y. Fu, “ClusterSR: Cluster-aware scattered repair in erasure-coded storage,” in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2020, pp. 42–51.
[38] Z. Shen, J. Shu, and P. P. C. Lee, “Reconsidering single failure recovery in clustered file systems,” in *Proc. IEEE/IFIP 46th Annu. Int. Conf. Dependable Syst. Netw.*, 2016, pp. 323–334.
[39] D. Stodolsky, G. Gibson, and M. Holland, “Parity logging overcoming the small write problem in redundant disk arrays,” *ACM SIGARCH Comput. Archit. News*, vol. 21, no. 2, pp. 64–75, 1993.
[40] H. Weatherspoon and J. D. Kubiatowicz, “Erasure coding vs. replication: A quantitative comparison,” in *Proc. Int. Workshop Peer-to-Peer Syst.*, 2002, pp. 328–338.
[41] S. Wei, Y. Li, Y. Xu, and S. Wu, “DSC: Dynamic stripe construction for asynchronous encoding in clustered file system,” in *Proc. IEEE Conf. Comput. Commun.*, 2017, pp. 1–9.
[42] S. Wu, Q. Du, P. P. C. Lee, Y. Li, and Y. Xu, “Optimal data placement for stripe merging in locally repairable codes,” in *Proc. IEEE Conf. Comput. Commun.*, 2022, pp. 1669–1678.

Guowen Gong received the BS degree from Xiamen University, in 2020. He is currently working toward the master degree with Xiamen University majoring in computer science and technology. His current research interests include storage reliability and security.



Zhirong Shen (Member, IEEE) received the BS degree from the University of Electronic Science and Technology of China, in 2010, and the PhD degree in computer science from Tsinghua University, in 2016. He is now an associate professor with Xiamen University. His current research interests include storage reliability and storage security.

Liang Chen received the BS degree from Xiamen University, in 2021. He is currently working toward the master degree with Xiamen University majoring in computer science and technology. His current research interests include programmable networks and storage reliability.

Suzhen Wu (Member, IEEE) received the BE and PhD degrees in computer science and technology and computer architecture from the Huazhong University of Science and Technology, Wuhan, China, in 2005 and 2010, respectively. She is an associate professor of Computer Science Department, Xiamen University. Her research interests include computer architecture and storage system. She has more than 40 publications in journal and international conferences, including the *IEEE Transactions on Computers*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *IEEE Transactions on Parallel and Distributed Systems*, *ACM Transactions on Storage*, USENIX FAST, USENIX LISA, ICS, ICDCS, ICCD, MSST, DATE, and IPDPS.

Xiaolu Li received the BEng degree from the University of Science and Technology of China, in 2016, and the PhD degree in computer science and engineering from the Chinese University of Hong Kong, in 2020. She is now a lecturer with the School of Computer Science and Technology, Huazhong University of Science and Technology. Her current research interests include distributed storage system, erasure-coding, and container storage.

Patrick P. C. Lee (Senior Member, IEEE) received the BEng (first-class honors) degree in information engineering from the Chinese University of Hong Kong, in 2001, the MPhil degree in computer science and engineering from the Chinese University of Hong Kong, in 2003, and the PhD degree in computer science from Columbia University, in 2008. He is now a full professor with the Department of Computer Science and Engineering, Chinese University of Hong Kong. His research interests include various applied/systems topics including storage systems, distributed systems and networks, dependability, and security.

Zhiguo Wan received the BS degree in computer science from Tsinghua University, in 2002, and the PhD degree in information security from the National University of Singapore, in 2007. He is a principal investigator with the Zhejiang Lab, Hangzhou, Zhejiang, China. His main research interests include security and privacy for cloud computing, Internet-of-Things, and blockchain. He was a postdoc with Katholieke University of Leuven, Belgium and an Assistant Professor with the School of Software, Tsinghua University, Beijing, China.

Jiwu Shu (Fellow, IEEE) received the PhD degree in computer science from Nanjing University, in 1998, and finished the postdoctoral position research with Tsinghua University, in 2000. Since then, he has been teaching with Tsinghua University. His current research interests include storage security and reliability, non-volatile memory based storage systems, and parallel and distributed computing.