Optimal Rack-Coordinated Updates in Erasure-Coded Data Centers

Guowen Gong*, Zhirong Shen*, Suzhen Wu*, Xiaolu Li†, Patrick P. C. Lee†
*Xiamen University, †The Chinese University of Hong Kong
23020201153743@stu.xmu.edu.cn, {shenzr,suzhen}@xmu.edu.cn, {lixl,pclee}@cse.cuhk.edu.hk

***Abstract*—Erasure coding has been extensively deployed in today's data centers to tackle prevalent failures, yet it is prone to give rise to substantial cross-rack traffic for parity update. In this paper, 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 finally conduct extensive evaluations, in terms of large-scale simulation and real-world data center experiments, showing that RackCU can reduce 22.1%-75.1% of the cross-rack update traffic and hence improve 34.2%-292.6% of the update throughput.**

I. Introduction

Data centers are often built atop of numerous storage nodes (also called *nodes*) to support a large number of services, including data storage, information retrieval, and MapReduce computation [10]. The large scale of data centers makes failures, which are originally accidental, become the norm [8], [9]. To tackle prevalent unexpected failures, production storage systems [3], [12], [22] often resort to maintaining additional data redundancy through replication [21] and erasure coding [12], 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 [34], and hence is preferable in practical storage systems [2], [3], [6], [19]. 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 suffices 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 update to the data chunks triggers additional updates to the corresponding parity chunks, thereby amplifying the storage and network I/O operations.

Corresponding author: Zhirong Shen (shenzr@xmu.edu.cn). This work is supported by NSFC (No. 62072381, U1705261, 61972325, 61872305), CCF-Tencent Open Fund WeBank Special Fund, the Natural Science Foundation of Fujian Province of China (2020J01002), Xiamen Youth Innovation Fund (3502Z20206052), and Research Grants Council of HKSAR (AoE/P-404/18), and Innovation and Technology Fund (ITS/315/18FX).

Enabling efficient updates of erasure coding in data centers is a challenging issue. Data centers usually organize nodes hierarchically, where a bunch of 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 [10]. 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 [4], [7], [10] and further fiercely consumed by various workloads (e.g., replication writes [7] and MapReduce shuffling [4]). 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 lessening disk seeks [5], [13], decreasing number of parity chunks being updated [27], [29], [30], and reducing update traffic [24], [32]. While CAU [28] can mitigate cross-rack update traffic, it degrades system reliability (by postponing parity update) 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 mitigates 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 implement a RackCU prototype and conduct extensive evaluation via both large-scale simulation and Alibaba Cloud Elastic Compute Service (ECS) [1] experiments, and show that RackCU reduces 22.1%-75.1% of cross-rack update traffic and hence increases 34.2%-292.6% of update throughput.

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

II. Background

We introduce the architecture of data center (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 [8], [19] and assumed in previous work [7], [11], [28], [31], [32]. Figure 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 [4], [7], [10]. Even worse, the cross-rack communication continues to grow dramatically, as the large analytics prevalently requires jobs across multiple racks [17].

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 [26]. 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 paper, to facilitate the understanding, we mainly use Reed-Solomon codes (RS codes) [26] as an instance, as they are popularly deployed in production systems [2], [3], [6], [19], [22]. Nevertheless, we also show that our approach can be readily extended to other codes like locally-repairable codes (LRCs) [12], [23] (see Section III-D). We use RS(k,m) to

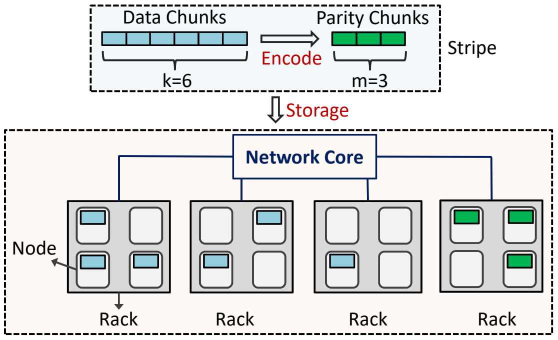


Fig. 1: Example of a data center deployed with RS(6,3).

denote the RS codes configured by the parameters k and m throughout the paper. Figure 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. Parity Update in Erasure Coding

In this paper, we mainly consider the delta-based update in erasure coding [5], [13], [28]. Suppose that $\{D_1, D_2, \cdots, D_k\}$ and $\{P_1, P_2, \cdots, P_m\}$ represent the k data chunks and the m parity chunks of a stripe, respectively. Each parity chunk P_j $(1 \le j \le m)$ can be calculated as a *linear combination* of the k data chunks via the Galois Field arithmetic [25], given by

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

where $\gamma_{i,j}$ $(1 \le i \le k$ and $1 \le j \le m)$ 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 $(1 \le h \le k)$. To promise the *encoding consistency* between the data and parity chunks, each parity chunk P_j (where $1 \le j \le m$) should be accordingly updated based on Equation (1) as below:

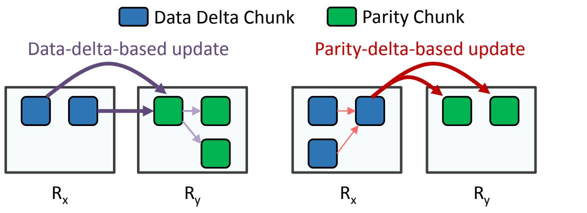
$$P'_j = P_j + \gamma_{h,j}(D'_h - D_h). \tag{2}$$

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* (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 [28]. Besides, as the encoding coefficients $\{\gamma_{i,j}\}_{1\le i\le k, 1\le j\le m}$ 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. Without loss of generality, suppose that the data chunks $\{D_1, D_2, \cdots, D_{u_x}\}$ in the rack R_x are updated to $\{D'_1, D'_2, \cdots, D'_{u_x}\}$, where u_x denotes the number of updated data chunks in R_x. Based on Equation (2), we can generalize the calculation of P'_j $(1 \le j \le m)$ as:

$$P'_j = P_j + \sum_{h=1}^{u_x} \gamma_{h,j} \Delta D_h = P_j + \Delta P_j \tag{3}$$



(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 $t_y = 2$.

where $\Delta D_h = D'_h - D_h$ denotes the data delta chunk of D_h and $\Delta P_j = \sum_{h=1}^{u_x} \gamma_{h,j} \Delta D_h$ represents the parity delta chunk of P_j.

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 Equation (3), there are two options to update the parity chunks in R_y, namely *data-delta-based update* and *parity-delta-based update* [28].

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 calculate the new parity chunk (i.e., P'_j) based on Equation (3). Figure 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 ($R_y \neq R_x$).

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* $\Delta P_j = \sum_{h=1}^{u_x} \gamma_{h,j} \Delta D_h$ 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 ΔP_j based on Equation (3). Figure 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 finally design RackCU that touches the lower bound of the cross-rack update traffic (Section III-D).

A. Design Overview

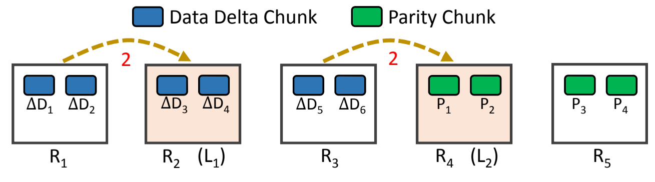
In principle, the rack-coordinated update is a synthesis of the data-delta-based update and parity-delta-based update approaches. The *main idea* is to allow racks to coordinate in parity update *immediately* after data chunks are updated, to reduce the cross-rack update traffic with system reliability 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 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 (Figure 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 (Figure 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 Figure 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.

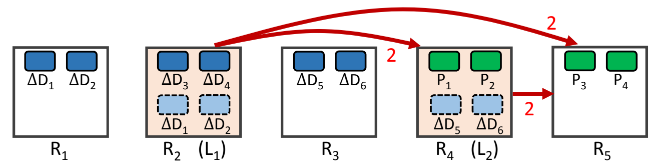
In the delta-collecting phase (Figure 3(a)), it selects two collector racks (R_2 and R_4), which fetch data delta chunks from R_1 and R_3, respectively. This phase transmits four chunks across racks.

In the selective parity update phase (Figure 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 paper. So this phase delivers six chunks across racks for parity update.

Finally, the rack-coordinated update in this example needs to transmit 10 chunks in total across racks for parity update. As a comparison, the delta-based update approach sends all data



(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 parity update.

delta chunks directly to every parity chunk (Section II-C), and hence needs to transmit 24 chunks across racks (calculated by multiplying 6 (i.e., number of data delta chunks) with 4 (i.e., number of parity chunks)). So our rack-coordinated update mechanism can vastly reduce the cross-rack update traffic.

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). Second, 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. Third, the placement of each stripe should ensure the rack-level fault tolerance [11], [32] (Section II-B).

Preliminary: Suppose that the k data chunks of a stripe are stored in d racks (denoted by $\{R_1, R_2, \cdots, R_d\}$) and the corresponding m parity chunks within the same stripe are distributed in another p racks (denoted by $\{R_{d+1}, R_{d+2}, \cdots, R_{d+p}\}$). For example, in Figure 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 Figure 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 paper, 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 ($d_c \le d$) and another p_c parity racks ($p_c \le p$) 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 $1 \le i \le d_c$), 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 $1 \le j \le p_c$). For example, in Figure 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 $1 \le i \le d_c + p_c$). In the motivating example (Figure 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 Figure 3(b)) and $l_1 = 2$ (see Figure 3(a)). Besides, we can deduce that $l_{d_c+j} = 0$ ($1 \le j \le p_c$), 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 Figure 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 $\{L_i\}_{1 \le i \le d_c}$, it can update the corresponding t_{d+j} parity chunks for each parity rack R_{d+j} ($1 \le j \le p$) 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 Figure 3(b), $p = 2$ and $t_{d+j} = 2$ for $1 \le j \le 2$, 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} ($1 \le i \le p_c$) that is also a parity rack (e.g., $L_2 = R_4$ in Figure 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 $\sum_{i=1}^{p_c} \sum_{j=1, R_{d+j} \ne L_{d_c+i}}^{p} \min\{l'_{d_c+i}, t_{d+j}\}$. Consequently, the number of delta chunks to be transmitted across racks in the selective parity update phase is

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

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 $\{u_1, u_2, \cdots, u_d\}$ (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. □

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. □

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.*

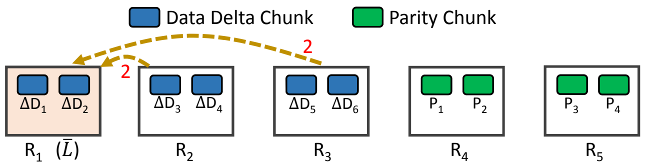
D. Design of RackCU

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).

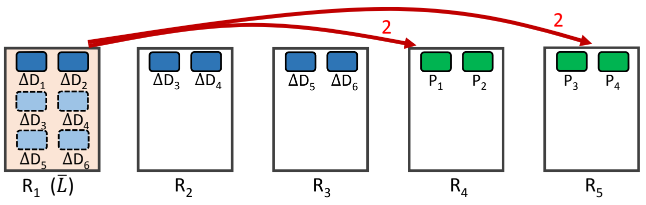
Algorithm details: We first find the data rack R_{d^*} with the most updated data chunks and the parity rack R_{p^*} with the most parity chunks (Lines 1-2). If the number of updated data chunks in R_{d^*} is no smaller than that of parity chunks in R_{p^*}, we select R_{d^*} as the sole collector rack $\overline{L}$; 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

Algorithm 1 Procedure of RackCU

```
Input: {R_1, R_2, ..., R_d} (data racks)
       {u_1, u_2, ..., u_d} (distribution of updated data chunks)
       {R_{d+1}, R_{d+2}, ..., R_{d+p}} (parity racks)
       {t_{d+1}, t_{d+2}, ..., 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 phase: sending four chunks to R_1.



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

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

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, then RackCU generates the parity delta chunks for parity update (Lines 14-16). Otherwise, RackCU sends the data delta chunks for 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 Figure 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

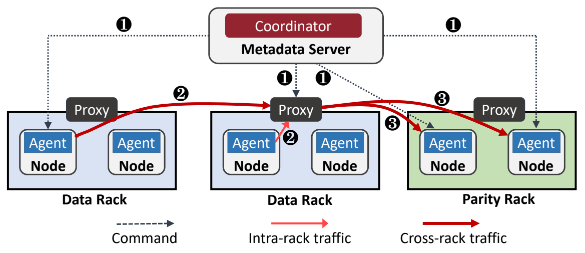


Fig. 5: System architecture of RackCU.

$t_4 = t_5 = 2$. 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 (Figure 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 (Figure 4(b)). Hence, RackCU transmits eight chunks in total across racks, which is fewer than the example (shown in Figure 3) that selects two collector racks and sends 10 chunks across racks for parity update.

Extension: Though RackCU mainly focuses on RS codes, we show that it can be effortlessly extended for other representative codes like LRCs [12], [23]. LRCs keep *a local parity chunk* in each rack and maintain a number of *global parity chunks* in other racks. Therefore, when a data chunk is updated, its corresponding local parity chunk can be renewed via intra-rack traffic. To minimize the cross-rack traffic in updating global parity chunks, we can also employ RackCU to select the sole collector rack based on the footprints of the updated data chunks and the layout of global parity chunks.

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 parity update and the computation complexity is $O(p)$. So the overall computation complexity of Algorithm 1 is $O(d + p)$.

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 [25].

System architecture: Figure 5 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.

Figure 5 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 (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.

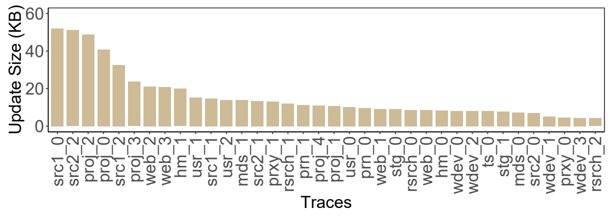


Fig. 6: Update sizes of MSR Cambridge Traces [20].

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 22.1%-75.1% of cross-rack update traffic (Section V-B); (2) RackCU increases 34.2%-292.6% of update throughput (Section V-C).

A. Preliminaries

Traces: We assess the update performance via trace-driven evaluation. We employ MSR Cambridge Traces (MSR) [20], 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. Figure 6 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) [28], (ii) the baseline delta-based update approach, and (iii) Parix [13]. We summarize these three approaches as below.

- **CAU [28]:** CAU updates parity chunks simply through the selective parity update [1]: if the updated data chunks

[1]We remove the data grouping and interim replication from CAU [28] and let CAU merely perform the selective parity update. We emphasize that RackCU can achieve higher reliability than the original CAU [28].

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.

- **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 Equation (2).
- **Parix [13]:** 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 Equation (2).

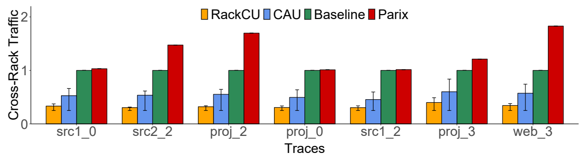
We summarize that Parix incurs additional network traffic (for transmitting the old data chunk updated for the first time), but avoids frequent storage I/O operations (for reading the old parity chunk) to generate the new parity chunk.

B. Large-Scale Simulation

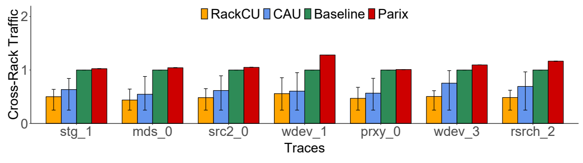
We first carry out large-scale simulation. We remove the storage and network operations of the RackCU prototype, and keep eyes on the amount of induced cross-rack traffic.

Experimental setup: We use the following default configurations in this simulation. We deploy RS(12,4) (also considered in Windows Azure Storge [12]) in a data center, which is built atop of 200 nodes with 10 racks (i.e., 20 nodes per rack). 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 (some may be invisible as they are very small).

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, and another seven traces with smaller update sizes. Figure 7 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 Parix for the seven traces with larger update size (Figure 7(a)); the reductions shrink to 22.1%, 50.9%, and 55.1% (Figure 7(b)), respectively.

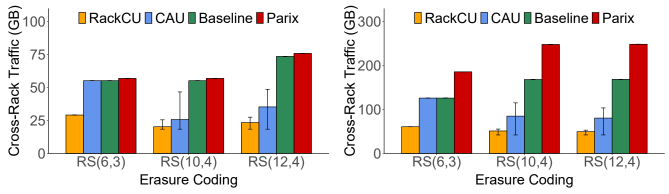


(a) Comparison on the traces with larger update sizes.

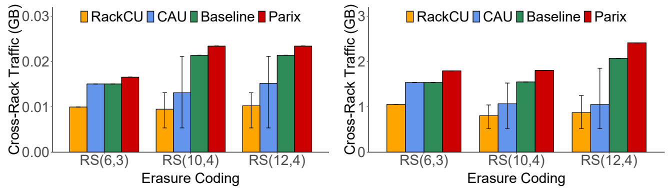


(b) Comparison on the traces with smaller update sizes.

Fig. 7: Experiment A.1 (Impact of update size). The smaller value is better.



(a) src1_0 (b) src2_2

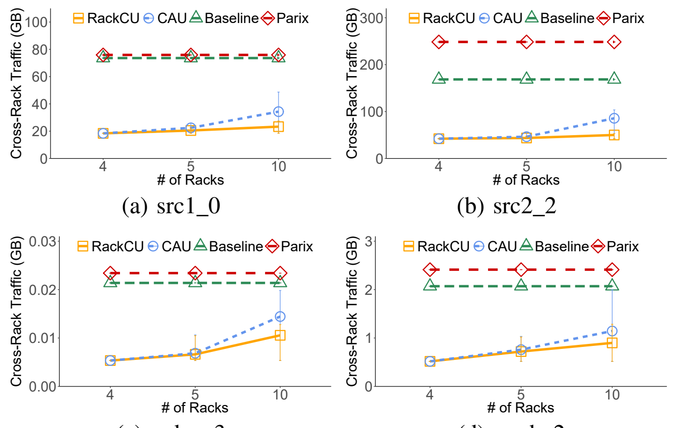


(c) wdev_3 (d) rsrch_2

Fig. 8: Experiment A.2 (Impact of erasure coding)

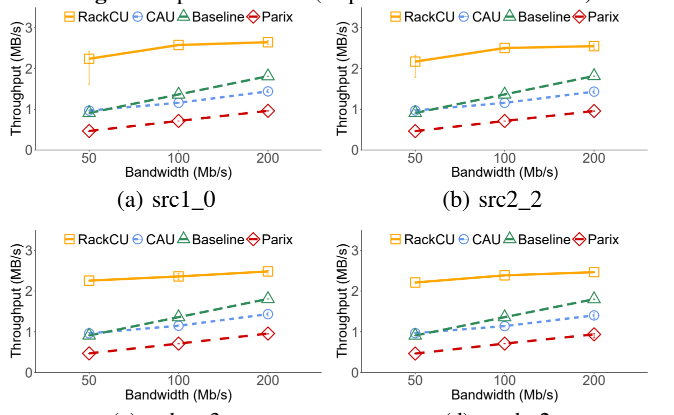
Experiment A.2 (Impact of erasure coding): We evaluate the impact of erasure coding parameters via choosing two traces (i.e., `src1_0` and `src2_2`) with larger update sizes and another two traces (i.e., `wdev_3` and `rsrch_2`) with smaller update sizes. We focus on the following three erasure coding schemes: RS(6,3) (selected in QFS [22] and Hadoop HDFS [3]), RS(10,4) (deployed in Facebook f4 [19]), and RS(12,4) (considered in Windows Azure Storage [12]). Figure 8 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. Figure 9 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 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



(a) src1_0 (b) src2_2 (c) wdev_3 (d) rsrch_2

Fig. 9: Experiment A.3 (Impact of number of racks)



(a) src1_0 (b) src2_2 (c) wdev_3 (d) rsrch_2

Fig. 10: Experiment B.1 (Impact of cross-rack bandwidth). The larger value is better.

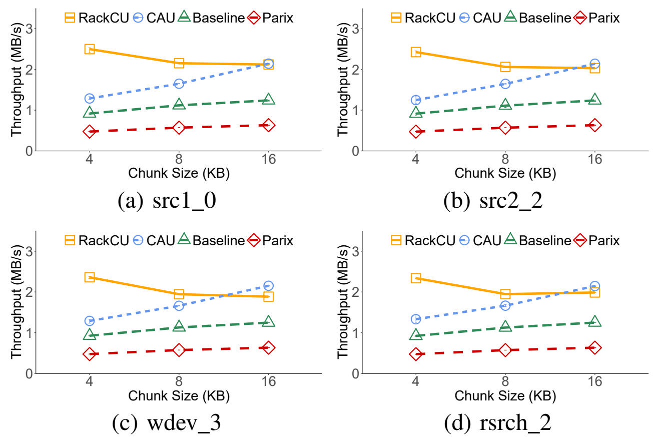
parity chunks across different racks. As the baseline and Parix directly update each parity chunk in other racks, the cross-rack update traffic depends on the number of parity chunks.

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 throughput by varying the cross-rack bandwidth from 50 Mb/s to 200 Mb/s. Figure 10 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% 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.



(a) src1_0 (b) src2_2 (c) wdev_3 (d) rsrch_2

Fig. 11: Experiment B.2 (Impact of chunk size)

Experiment B.2 (Impact of chunk size): We study the update throughput under different chunk sizes. Figure 11 shows that RackCU improves the update throughput by 34.2%, 101.1%, and 292.6%, 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, degrading the efficacy of RackCU. Based on this experiment, we suggest deploying RackCU in the scenario with multiple chunks updated per update request.

VI. Related Work

Delta-based updates: Erasure-coded systems often manipulate parity update via delta-based update approaches. Parity logging [33] appends parity deltas to a dedicated log device for avoiding random small writes. CodFS [5] couples in-place data update and log-based parity update to tailor update performance and repair performance. To avoid frequent disk seeks in parity update, Parix [13] appends the old and latest data chunks, and only calculates the delta of them in parity update. UCODR [27] selects the combination of appropriate data and parity chunks to mitigate the storage I/O in parity update. T-Update [24] 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 [28] appends new data chunks and defers 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 parity update. PDP [30] arranges sequential data chunks to generate the same parity chunk for reducing parity update of sequential writes. CASO [29] organizes correlated data chunks that are likely to be updated together into the same stripe. CAU [28] relocates updated data chunks within the same rack to reduce cross-rack traffic in 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 [12], [23] keep a local parity chunk within a rack to avoid cross-rack data transfers in single chunk's repair. Some studies [11], [15], [16], [18], [31], [32] 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 [14], [35] 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. Large-scale simulation and extensive testbed experiments both show that RackCU can vastly reduce the cross-rack update traffic and improve the update throughput.

Appendix

Proof of Theorem 1. Without loss of generality, suppose that the numbers of updated data chunks in the d data racks are $\{u_1, u_2, \cdots, u_d\}$ (where $u_1 \geq u_2 \geq \cdots \geq u_d$) and those of parity chunks in the p parity racks are $\{t_{d+1}, t_{d+2}, \cdots, t_{d+p}\}$ (where $t_{d+1} \geq t_{d+2} \geq \cdots \geq t_{d+p}$). Hence, $u_{d^*} = u_1$ and $t_{p^*} = t_{p+1}$.

We prove that the theorem establishes when $\overline{L} = R_1$ (i.e., $u_1 \geq t_{d+1}$) and the case when $\overline{L} = R_{d+1}$ (i.e., $u_1 < t_{d+1}$) is similar. Suppose that a rack-coordinated update solution $\mathbb{S}$ selects $d_c + p_c$ racks (aside from R_1) to be the collector racks. Our main idea is to replace a collector rack in $\mathbb{S}$ by R_1 and form another solution $\mathbb{S}'$, such that $\mathbb{S}'$ introduces no more cross-rack update traffic than $\mathbb{S}$. Suppose that the u_1 data delta chunks of the rack R_1 are originally sent to a collector rack $L_i = R_x$ in $\mathbb{S}$ (where $1 \leq i \leq d_c + p_c$ and $1 \leq x \leq d + p$). We can let $L_i = R_1$ in $\mathbb{S}'$ (i.e., let R_1 replace R_x as the collector rack L_i) and collect the data delta chunks from the same data racks as in $\mathbb{S}$. We now calculate the gains of this replacement. When $L_i = R_1$, we do not need to send the u_1 data delta chunks from R_1 to L_i (i.e., avoid sending u_1 chunks across racks), but have to retrieve u_x data delta chunks from R_x to L_i (i.e., increase additional u_x chunks of cross-rack transmission). For other data racks (i.e., aside from R_1) that are required to send data delta chunks to $L_i = R_x$ in $\mathbb{S}$, they will submit their data delta chunks to $L_i = R_1$ in $\mathbb{S}'$ and their cross-rack traffic in this phase remains unchanged. Hence, this replacement can save $u_1 - u_x$ chunks for cross-rack transmission in the delta-collecting phase.

Let us consider the selective parity update phase. Suppose that R_x has t_x parity chunks ($t_x = 0$ if R_x is a data rack). For the original solution $\mathbb{S}$ that sets $L_i = R_x$, the corresponding t_x parity chunks can be updated within R_x directly; conversely, setting $L_i = R_1$ in $\mathbb{S}'$ should update the t_x parity chunks (in the rack $R_x \neq R_1$) using the selective parity update, resulting in additional $\min\{l'_i, t_x\} = t_x$ chunks (as $l'_i \geq l_i = u_1 \geq t_x$) of cross-rack transmission after replacement.

Finally, we deduce that $\mathbb{S}'$ saves $u_1 - u_x - t_x$ chunks for cross-rack transfers. If R_x is a data rack (i.e., $t_x = 0$ and $1 \leq x \leq d$), then $u_1 - u_x - t_x = u_1 - u_x \geq 0$. On the other hand, if R_x is a parity rack (i.e., $u_x = 0$ and $d+1 \leq x \leq d+p$), then $u_1 - u_x - t_x = u_1 - t_x \geq 0$ (as $u_1 \geq t_{d+1} \geq t_x$). Hence, the conclusion holds.

Proof of Theorem 2. Suppose the numbers of updated data chunks of the d data racks follow $u_1 \geq u_2 \geq \cdots \geq u_d$ and those of parity chunks of the p parity racks obey $t_{d+1} \geq t_{d+2} \geq \cdots \geq t_{d+p}$. We have $u_i \leq m$ (rack-level tolerance) and $\sum_{j=1}^{p} t_{d+j} = m$. Without loss of generality, let $L_1 = R_1$ in $\mathbb{S}'$ (i.e., $u_1 = l_1$).

We first analyze the cross-rack traffic when setting $\overline{L} = R_1$ (i.e., $u_1 \geq t_{d+1}$) in $\mathbb{S}^*$, and the case when setting $\overline{L} = R_{d+1}$ (i.e., $u_1 < t_{d+1}$) is similar. In the delta-collecting phase, R_1 collects $\sum_{i=2}^{d} u_i$ data delta chunks from other $d-1$ data racks and possesses $\sum_{i=1}^{d} u_i$ data delta chunks in total. In the selective parity update phase, as $\sum_{i=1}^{d} u_i \geq u_1 \geq t_{d+1} \geq t_{d+j}$ for any parity rack R_{d+j} (where $1 \leq j \leq p$), we use the parity-delta-based update by sending the corresponding t_{d+j} parity delta chunks to R_{d+j}. The cross-rack update traffic in this phase is $\sum_{j=1}^{p} t_{d+j} = m$. Hence, the total cross-rack update traffic is

$$T_{\mathbb{S}^*} = \sum_{i=2}^{d} u_i + m$$

Based on Equation (4), we deduce the following inequation for $T_{\mathbb{S}'}$, where $\mathbb{S}'$ selects $d_c + p_c$ racks as the collector racks.

$$T_{\mathbb{S}'} = T_{\text{collect}} + T_{\text{selective}} \geq \sum_{i=1}^{d} u_i - \sum_{i=1}^{d_c} l_i + \sum_{i=1}^{d_c} \sum_{j=1}^{p} \min\{l'_i, t_{d+j}\}$$

Therefore, we have $T_{\mathbb{S}'} - T_{\mathbb{S}^*}$

$$\geq u_1 - \sum_{i=1}^{d_c} l_i + \sum_{i=1}^{d_c} \sum_{j=1}^{p} \min\{l'_i, t_{d+j}\} - m$$

$$= u_1 - \sum_{i=1}^{d_c} l_i + \sum_{j=1}^{p} \min\{l'_1, t_{d+j}\} + \sum_{i=2}^{d_c} \sum_{j=1}^{p} \min\{l'_i, t_{d+j}\} - m$$

Since $L_1 = R_1$ in $\mathbb{S}'$ and $l'_1 \geq l_1 = u_1 \geq t_{d+j}$ for $1 \leq j \leq p$, we can deduce that $\sum_{j=1}^{p} \min\{l'_1, t_{d+j}\} = \sum_{j=1}^{p} t_{d+j} = m$. We can also readily have: if $l'_i \geq t_{d+j}$ for $1 \leq j \leq p$, then $\sum_{j=1}^{p} \min\{l'_i, t_{d+j}\} \geq \sum_{j=1}^{p} t_{d+j} = m \geq l_i$ (for rack-level fault tolerance); otherwise, $\sum_{j=1}^{p} \min\{l'_i, t_{d+j}\} \geq l'_i \geq l_i$. Hence, $\sum_{i=2}^{d_c} \sum_{j=1}^{p} \min\{l'_i, t_{d+j}\} \geq \sum_{i=2}^{d_c} l_i$. Finally, we have:

$$T_{\mathbb{S}'} - T_{\mathbb{S}^*} \geq u_1 - \sum_{i=1}^{d_c} l_i + m + \sum_{i=2}^{d_c} l_i - m \geq u_1 - l_1 = 0.$$

Therefore, the theorem holds.

References

[1] Alibaba Cloud Elastic Compute Service. https://www.alibabacloud.com/product/ecs.

[2] Erasure Coding in Ceph. https://ceph.com/planet/erasure-coding-in-ceph/, 2014.

[3] Apache Hadoop 3.0.0. https://hadoop.apache.org/docs/r3.0.0/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html, 2017.

[4] F. Ahmad, S. Chakradhar, A. Raghunathan, and T. Vijaykumar. Shufflewatcher: Shuffle-aware Scheduling in Multi-Tenant Mapreduce Clusters. In *Proc. of USENIX ATC*, 2014.

[5] 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. of USENIX FAST*, 2014.

[6] H. Chen, H. Zhang, M. Dong, Z. Wang, Y. Xia, H. Guan, and B. Zang. Efficient and Available In-Memory KV-Store with Hybrid Erasure Coding and Replication. *ACM Transactions on Storage*, 13(3):1–30, 2017.

[7] M. Chowdhury, S. Kandula, and I. Stoica. Leveraging Endpoint Flexibility in Data-Intensive Clusters. In *Proc. of ACM SIGCOMM*, 2013.

[8] D. Ford, F. Labelle, F. Popovici, M. Stokely, V. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in Globally Distributed Storage Systems. In *Proc. of USENIX OSDI*, 2010.

[9] P. Gill, N. Jain, and N. Nagappan. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. In *Proc. of ACM SIGCOMM*, 2011.

[10] A. Greenberg, J. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *Proc. of ACM SIGCOMM*, 2009.

[11] Y. Hu, X. Li, M. Zhang, P. P. C. Lee, X. Zhang, P. Zhou, and D. Feng. Optimal Repair Layering for Erasure-Coded Data Centers: From Theory to Practice. *ACM Transactions on Storage*, 13(4):33, 2017.

[12] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure Coding in Windows Azure Storage. In *Proc. of USENIX ATC*, 2012.

[13] H. Li, Y. Zhang, Z. Zhang, S. Liu, D. Li, X. Liu, and Y. Peng. PARIX: Speculative Partial Writes in Erasure-Coded Systems. In *Proc. of USENIX ATC*, 2017.

[14] R. Li, Y. Hu, and P. P. C. Lee. Enabling Efficient and Reliable Transition from Replication to Erasure Coding for Clustered File Systems. *IEEE Transactions on Parallel and Distributed Systems*, 28(9):2500–2513, 2017.

[15] R. Li, X. Li, P. P. C. Lee, and Q. Huang. Repair Pipelining for Erasure-Coded Storage. In *Proc. of USENIX ATC*, 2017.

[16] 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. of USENIX FAST*, 2019.

[17] V. Liu, D. Zhuo, S. Peter, A. Krishnamurthy, and T. Anderson. Subways: A Case for Redundant, Inexpensive Data Center Edge Links. In *Proc. of ACM CoNEXT*, 2015.

[18] S. Mitra, R. Panta, M.-R. Ra, and S. Bagchi. Partial-Parallel-Repair (PPR): A Distributed Technique for Repairing Erasure Coded Storage. In *Proc. of EuroSys*, 2016.

[19] S. Muralidhar, W. Lloyd, S. Roy, et al. f4: Facebook's Warm Blob Storage System. In *Proc. of USENIX OSDI*, 2014.

[20] D. Narayanan, A. Donnelly, and A. Rowstron. Write Off-Loading: Practical Power Management for Enterprise Storage. *ACM Transactions on Storage*, 4(3):1–23, 2008.

[21] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *Proc. of ACM SOSP*, 2011.

[22] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly. The Quantcast File System. *Proceedings of the VLDB Endowment*, 6(11):1092–1101, 2013.

[23] D. S. Papailiopoulos and A. G. Dimakis. Locally Repairable Codes. *IEEE Transactions on Information Theory*, 60(10):5843–5855, 2014.

[24] 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. of IEEE INFOCOM*, 2016.

[25] J. S. Plank, S. Simmerman, and C. D. Schuman. Jerasure: A library in C/C++ Facilitating Erasure Coding for Storage Applications-Version 1.2. *University of Tennessee, Tech. Rep. CS-08-627*, 23, 2008.

[26] I. Reed and G. Solomon. Polynomial Codes over Certain Finite Fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

[27] J. Shen, K. Zhang, J. Gu, Y. Zhou, and X. Wang. Efficient Scheduling for Multi-Block Updates in Erasure Coding based Storage Systems. *IEEE Transactions on Computers*, 67(4):573–581, 2017.

[28] Z. Shen and P. Lee. Cross-Rack-Aware Updates in Erasure-Coded Data Centers. In *Proc. of ACM ICPP*, 2018.

[29] Z. Shen, P. Lee, J. Shu, and W. Guo. Correlation-Aware Stripe Organization for Efficient Writes in Erasure-Coded Storage Systems. In *Proc. of IEEE SRDS*, 2017.

[30] Z. Shen, J. Shu, and Y. Fu. Parity-Switched Data Placement: Optimizing Partial Stripe Writes in XOR-Coded Storage Systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(11):3311–3322, 2016.

[31] Z. Shen, J. Shu, Z. Huang, and Y. Fu. ClusterSR: Cluster-Aware Scattered Repair in Erasure-Coded Storage. In *Proc. of IEEE IPDPS*, 2020.

[32] Z. Shen, J. Shu, and P. P. C. Lee. Reconsidering Single Failure Recovery in Clustered File Systems. In *Proc. of IEEE/IFIP DSN*, 2016.

[33] D. Stodolsky, G. Gibson, and M. Holland. Parity Logging Overcoming the Small Write Problem in Redundant Disk Arrays. *ACM SIGARCH Computer Architecture News*, 21(2):64–75, 1993.

[34] H. Weatherspoon and J. D. Kubiatowicz. Erasure Coding vs. Replication: A Quantitative Comparison. In *International Workshop on Peer-to-Peer Systems*, 2002.

[35] S. Wei, Y. Li, Y. Xu, and S. Wu. DSC: Dynamic Stripe Construction for Asynchronous Encoding in Clustered File System. In *Proc. of IEEE INFOCOM*, 2017.