Cross-Rack-Aware Updates in Erasure-Coded Data Centers: Design and Evaluation

Zhirong Shen and Patrick P. C. Lee

Abstract—The update performance in erasure-coded data centers is often bottlenecked by the constrained cross-rack bandwidth. We propose CAU, a cross-rack-aware update mechanism that aims to mitigate the cross-rack update traffic in erasure-coded data centers. CAU builds on three design elements: (i) selective parity updates, which select the appropriate parity update approach based on the update pattern and the data layout to reduce the cross-rack update traffic; (ii) data grouping, which relocates and groups updated data chunks in the same rack to further reduce the cross-rack update traffic; and (iii) interim replication, which stores a specified number of temporary replicas for each newly updated data chunk. We evaluate CAU via trace-driven analysis, local cluster experiments, and Amazon EC2 experiments. We show that CAU enhances state-of-the-arts by mitigating the cross-rack update traffic as well as maintaining high update performance in both local cluster and geo-distributed environments.

1 Introduction

Modern data centers (DCs) (e.g., [5], [11], [22]) deploy thousands of storage nodes (or servers) in one or multiple geographic regions to provide large-scale storage services. It is critical for DCs to provide data reliability guarantees in the face of failures, which can be caused by unexpected factors from hardware (e.g., disk malfunctions) to software (e.g., file system errors). A common solution to addressing data reliability is to keep data with redundancy, in which *replication* and *erasure coding* are two most widely deployed approaches. Compared to replication, which creates multiple identical copies of data, erasure coding provably achieves the same degree of fault tolerance while incurring much less redundancy [40], and has been widely deployed in enterprise DCs [11], [15], [22]. At a high level, erasure coding takes a number of data chunks as input and produces additional redundant chunks called *parity chunks*, such that even if some data or parity chunks are lost due to failures, the lost chunks can still be reconstructed from the remaining available data and parity chunks.

Although erasure coding is storage-efficient, maintaining the consistency between data and parity chunks incurs high performance overhead under *update-intensive* workloads, since any update of a data chunk triggers *parity updates* for all other dependent parity chunks. We argue that updates become more common in today's DC storage workloads. For example, the proportion of updates in low-latency workloads in Yahoo!'s DCs reaches nearly 50% and continues to increase [32]. Deletes, which can be viewed as a special case of updates, are also common operations in Microsoft's erasure-coded DCs [7]. Furthermore, updates are mostly of small sizes (e.g., in online transactional processing [31] and enterprise server workloads [6]), and frequent small-size updates in turn lead to intensive parity updates in erasure-coded storage. Mitigating the update overhead in erasure-coded DCs is clearly a critical deployment issue.

The hierarchical topological nature of DCs further complicates the design of efficient updates in erasure-coded storage. Modern DCs organize nodes in *racks*, in which the cross-rack bandwidth is often oversubscribed [4] and much more scarce than the inner-rack bandwidth (typically 5-20× lower [3], [8]), yet it is heavily consumed by various types of workloads, such as replica writes [8], failure recovery [28], and data analytics [3], [16]. The same phenomenon is also found in geo-distributed DCs, in which nodes are located in multiple geographical regions and the cross-region bandwidth is much more scarce than the inner-region bandwidth [39]. Thus, enabling efficient updates with cross-rack (or cross-region) awareness is necessary, but is unfortunately largely unexplored by previous work on erasure-coded updates in the literature (see §6).

In this paper, we propose CAU, a novel cross-rack-aware update mechanism that mitigates the *cross-rack update traffic* (i.e., the cross-rack traffic triggered for maintaining the consistency of data and parity chunks in update operations) in erasure-coded DCs; note that CAU is also applicable for mitigating the cross-region update traffic in geo-distributed DCs. CAU builds on three design elements. First, CAU adopts *selective parity updates*, which selectively perform the appropriate parity update approach based on the update pattern and the data layout in a DC. Second, CAU can be extended to support *data grouping*, which relocates and groups updated data chunks into the same rack, so as to allow aggregate updates in the same rack and further reduce the cross-rack update traffic. Furthermore, CAU performs *interim replication*, which creates a specified number of short-

- *A preliminary version [33] of this paper was presented at the 47th International Conference on Parallel Processing (ICPP 2018). In this journal version, we extend CAU to provide fault tolerance against a general number of rack failures via interim replication. We also present more evaluation findings.*
- *Zhirong Shen is with the School of Informatics, Xiamen University, China. (E-mail: shenzr@xmu.edu.cn). He is also with the State Key Laboratory of Integrated Services Networks (Xidian University). This work is partially done when Zhirong Shen worked as a Postdoctoral Fellow at The Chinese University of Hong Kong.*
- *Patrick P. C. Lee is with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong. (E-mail:pclee@cse.cuhk.edu.hk)*
- *Corresponding author: Patrick P. C. Lee (pclee@cse.cuhk.edu.hk)*

lived replicas to maintain high data reliability against a number of rack failures, while limiting the addition of cross-rack traffic. Note that CAU is generic and can be applied to any practical erasure code that performs encoding based on linear combinations (see §2.3). Our contributions are summarized below:

- We present CAU, a novel cross-rack-aware update mechanism that mitigates the cross-rack update traffic through selective parity updates and data grouping.
- We show via reliability analysis that CAU maintains reliability guarantees through interim replication, as compared to traditional erasure coding that performs parity updates immediately for each updated data chunk.
- We implement a CAU prototype that is deployable in distributed environments, and evaluate CAU under real-world workloads from three perspectives: (i) trace-driven analysis, (ii) local cluster experiments, and (iii) Amazon EC2 experiments. Our trace-driven analysis shows that for some configurations, CAU saves 25.6-74.5% of cross-rack update traffic over the baseline approach and the recently proposed erasure-coded update scheme PARIX [18]. Also, our CAU prototype improves the update performance by 29.1-54.6% and 24.9-33.8% in local cluster and Amazon EC2 experiments, respectively.

The source code of our CAU prototype is available for download at **http://adslab.cse.cuhk.edu.hk/software/cau**.

2 Background

2.1 DC Architecture

We consider erasure-coded storage in a DC with a two-level hierarchical architecture. Specifically, a DC comprises multiple *nodes* (or servers) that provide storage space. It partitions nodes into different *racks*, such that multiple nodes within the same rack are connected via a top-of-rack (ToR) switch, while multiple racks are connected by the aggregation and core switches that collectively form the *network core*. Figure 1 depicts the DC architecture. Such a two-level hierarchical architecture is also employed in modern DC deployment [11], [22] and assumed by previous work [8], [14], [19], [36].

Our goal is to mitigate the cross-rack update traffic triggered by update operations in erasure-coded storage. We assume that the performance bottleneck of a DC lies in the cross-rack data transfer over the network core as in prior work [8], [14], [19], [36], as modern DCs are often oversubscribed and have constrained cross-rack bandwidth (see §1). Also, each node can be attached with multiple disks to achieve high I/O throughput [8], thereby further pushing the bottleneck to the network core. While our work focuses on rack-based DCs, we can also generalize our analysis to geo-distributed DCs, in which cross-region data transfer over the wide-area network is the performance bottleneck and the cross-region update traffic should be mitigated.

2.2 Erasure Coding

In this paper, we focus on a well-known family of erasure codes called Reed-Solomon (RS) codes [29], which are deployed in today's production DCs [11], [22], [24]. Specifically, we construct RS codes with two configurable integers n and k (where $0 < k < n$), and denote the code construction by RS(n,k). Suppose that the data is organized in fixed-size units called *chunks*. For every k (uncoded) chunks called *data chunks*, RS codes encode them into $n - k$ additional (coded) chunks called *parity chunks* via linear combinations (see §2.3 for details), such that any k out of the n data and parity chunks can reconstruct the original k data chunks. We call the set of the n data and parity chunks a *stripe*, which is distributed across n nodes to tolerate any $n - k$ node failures. In our discussion, we refer to the nodes that store data chunks and parity chunks as *data nodes* and *parity nodes*, respectively. In practice, a DC stores many stripes that are independently encoded and distributed across n different nodes, so each node can act as a data node or a parity node for different stripes. In this paper, we focus on the update operation for a single stripe.

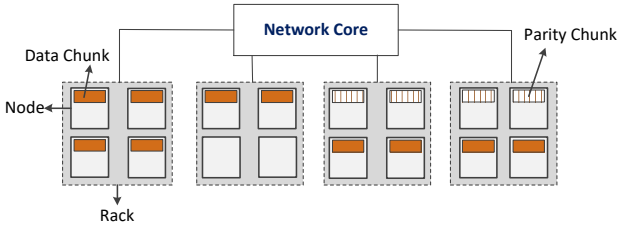


Fig. 1. A DC that comprises four racks with four nodes each. Suppose that the DC employs RS(14,10) for erasure coding. It may distribute the data and parity chunks of a stripe across 14 different nodes that reside in the four racks.

RS codes are both *storage-optimal* and *general*: by storage-optimal, we mean that the storage overhead (i.e., n/k) is the minimum to provide fault tolerance against any $n - k$ node failures (such storage-optimal fault tolerance is also called the *Maximum Distance Separable* property); by general, we mean that n and k can be arbitrary integers (provided that $0 < k < n$). The RS code construction that we consider is *systematic*, meaning that the k data chunks are included in a stripe after encoding.

To provide rack-level fault tolerance, existing erasure-coded DCs distribute each stripe across n nodes in n distinct racks [11], [15], [22]. Recent studies [14], [21], [36] propose to store each stripe in n nodes that reside in r racks, for some parameter $r < n$, to cut down the cross-rack traffic during failure repair at the expense of reduced rack-level fault tolerance. It is shown in [14] that the overall reliability can be improved under independent failures due to the reduction of cross-rack repair traffic, but drops when correlated failures become more common. For example, Figure 1 shows that the 14 chunks of a stripe coded by RS(14,10) are stored in $r = 4$ racks. Here, we assume that *each rack should store no more than $n - k$ chunks per stripe, so that an erasure-coded DC can tolerate at least a single rack failure*. In this work, we study how to mitigate the cross-rack update traffic by placing a stripe in $r < n$ racks.

2.3 Parity Updates in Erasure Coding

Most practical erasure codes perform encoding via *linear combinations*. We use RS codes as an example. Let $D_1, D_2, \cdots, D_k$ be the k data chunks, $P_1, P_2, \cdots, P_{n-k}$ be the $n - k$ parity chunks, and $\{\gamma_{i,j}\}_{1 \le i \le k, 1 \le j \le n-k}$ be the set of some encoding coefficients. Each parity chunk P_j, where

$1 \leq j \leq n-k$, can be computed based on Galois Field arithmetic [26] as follows:

$$P_j = \sum_{i=1}^{k} \gamma_{i,j} D_i. \quad (1)$$

From Equation (1), we can also efficiently update a parity chunk for any update of a data chunk. Suppose that a data chunk D_i (where $1 \leq i \leq k$) is updated to D'_i. Then we can update each parity chunk P_j (where $1 \leq j \leq n-k$) into P'_j as follows:

$$P'_j = P_j + \gamma_{i,j}(D'_i - D_i). \quad (2)$$

Equation (2) implies that a parity chunk can be updated directly from the *delta* of the data chunk $D'_i - D_i$, without accessing other unchanged data chunks of the same stripe. We call this type of parity updates *delta-based updates*. To elaborate, when a data node updates a data chunk D_i to a new data chunk D'_i, it sends the delta $D'_i - D_i$ to each of the $n-k$ parity nodes, which update their parity chunks based on Equation (2) (note that the coefficient $\gamma_{i,j}$ is known and determined by the erasure code construction). If we distribute a stripe across $r = n$ racks, the amount of cross-rack traffic for parity updates is equal to $n-k$ chunks, as the delta $D'_i - D_i$ has the same size as a data chunk. An open question is: if we distribute a stripe across $r < n$ racks, can we reduce the cross-rack update traffic?

3 Cross-Rack-Aware Updates

CAU is a *cross-rack-aware update* mechanism that aims to mitigate the cross-rack update traffic. It builds on three design elements: selective parity updates, data grouping, and interim replication.

3.1 Append-Commit Procedure

To avoid frequent parity updates, CAU adopts an iterative *append-commit* procedure to update data chunks. Each iteration consists of the *append* and *commit* phases (see Figure 2). In the append phase, when a data chunk is updated, CAU first identifies the data node where the original data chunk resides. It then appends the new data chunk to an *append-only* log that is co-located with and maintained by the data node, without immediately updating the associated parity chunks. The length of the append phase can be adjusted depending on the update frequency; for example, it lasts for a fixed time period if the update frequency is low, or until the append-only log reaches a size limit if the update frequency is high. Then CAU switches to the commit phase, in which it updates the parity chunks (via delta-based updates) based on the new data chunks in the append-only log of each data node. CAU performs the two phases of the append-commit procedure iteratively.

The append-commit procedure defers parity updates to exploit the opportunity of aggregating the updates of data or parity chunks in batch, and we use this property to design selective parity updates (see §3.2) and data grouping (see §3.3). However, it also degrades reliability as there is no redundancy to protect the updated data chunks until the commit phase. We address this issue via interim replication (see §3.4) and conduct reliability analysis (see §3.5) to justify that the fault tolerance is preserved.

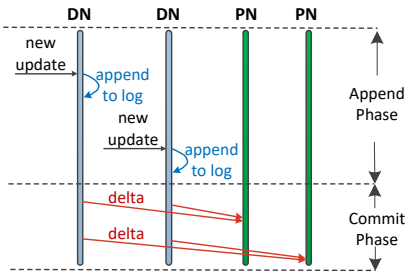


Fig. 2. Append-commit procedure (DN: data node; PN: parity node).

3.2 Selective Parity Updates

In the commit phase, parity updates incur cross-rack transfers when the data and parity chunks being updated reside in different racks. Here, we extend the delta-based updates (see §2.3) into *selective parity updates* so as to mitigate the cross-rack update traffic.

Problem: We first formalize the parity update problem as follows. Consider a stripe of n erasure-coded chunks, with k data chunks $\{D_1, D_2, \cdots, D_k\}$ and $n-k$ parity chunks $\{P_1, P_2, \cdots, P_{n-k}\}$ that are spread across r racks denoted by $\{R_1, R_2, \cdots, R_r\}$. Without loss of generality, suppose that rack R_i has i' data chunks being updated, denoted by $\{D_1, D_2, \cdots, D_{i'}\}$, and another rack R_j has j' parity chunks of the same stripe, denoted by $\{P_1, P_2, \cdots, P_{j'}\}$, where $1 \leq i \neq j \leq r$, $1 \leq i' \leq k$, and $1 \leq j' \leq n-k$. To update each parity chunk P_m into P'_m in R_j (where $1 \leq m \leq j'$), we can generalize Equation (2) as:

$$P'_m = P_m + \sum_{h=1}^{i'} \gamma_{h,m}(D'_h - D_h), \quad (3)$$

where $\gamma_{h,m}$ is the encoding coefficient used by D_h (where $1 \leq h \leq i'$) for the parity chunk P_m.

Based on Equation (3), we observe that there are two different ways to update a parity chunk in the commit phase. We call them *data-delta commit* and *parity-delta commit*. Figure 3 illustrates the two parity update approaches, as elaborated below.

Data-delta commit: A data-delta commit operation updates multiple parity chunks based on the change of each single data chunk (see Figure 3(a)). Specifically, for each data chunk D_h (where $1 \leq h \leq i'$) being updated, CAU computes a *data-delta chunk* $D'_h - D_h$. It then sends each of the i' data-delta chunks from R_i to one of the j' parity nodes in R_j, which then forwards a copy of each data-delta chunk to each of the remaining $j'-1$ parity nodes. To update each parity chunk P_m into P'_m (where $1 \leq m \leq j'$), the corresponding parity node adds all i' data-delta chunks to P_m as in Equation (3). We see that a data-delta commit operation incurs a cross-rack transfer of i' data-delta chunks. Figure 3(a) shows the data-delta commit operation with $i' = 2$ and $j' = 3$.

Parity-delta commit: A parity-delta commit operation updates each parity chunk by aggregating the changes of multiple data chunks (see Figure 3(b)). Specifically, to update each parity chunk P_m into P'_m (where $1 \leq m \leq j'$) in R_j, CAU collects all changes of data chunks in one of the

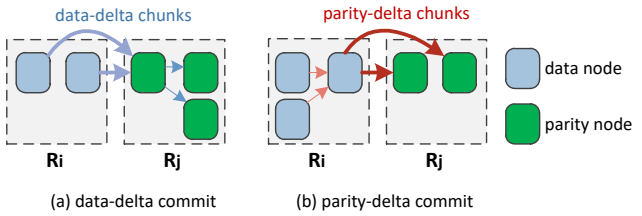


Fig. 3. Selective parity updates: (a) data-delta commit and (b) parity-delta commit. In (a), CAU sends $i' = 2$ data-delta chunks from R_i to R_j; in (b), CAU sends $j' = 2$ parity-delta chunks from R_i to R_j.

data nodes in R_i. The data node then computes a *parity-delta chunk* $\sum_{h=1}^{i'} \gamma_{h,m}(D'_h - D_h)$ and sends it to the parity node that stores P_m. The parity node adds the received parity delta chunk to P_m to form P'_m based on Equation (3). We see that a parity-delta commit operation incurs a cross-rack transfer of j' parity-delta chunks. Figure 3(b) shows the parity-delta commit operation with $i' = 3$ and $j' = 2$.

Discussion: The key difference between data-delta commit and parity-delta commit lies in where we compute the change of a parity chunk. In data-delta commit, we compute the change of a parity chunk in R_j, where the parity chunks are stored; in contrast, in parity-delta commit, we first compute the change of a parity chunk in R_i and then send the result to R_j. Both approaches incur different amounts of cross-rack update traffic. CAU performs the following decision: if $i' \leq j'$, CAU performs data-delta commit; otherwise, it performs parity-delta commit. Thus, the amount of cross-rack update traffic is $\min\{i', j'\}$.

Note that the current design of selective parity updates does not necessarily achieve the theoretically minimum cross-rack update traffic. For example, in data-delta commit, we treat all i' data-delta chunks different, but if they are identical, we may send only one data-delta chunk from R_i to R_j. Also, in parity-delta commit, we update each parity chunks independently. However, if the underlying erasure code allows one parity chunk to be computed from another parity chunk (e.g., RDP [10]), we may send only one parity-delta chunk (instead of j' parity-delta chunks) from R_i to R_j, and compute all parity chunks within R_j. How to find the theoretically minimum cross-rack update traffic is posed as future work.

3.3 Data Grouping

The effectiveness of selective parity updates is restricted by the underlying chunk placement. Here, we further reduce the cross-rack update traffic by relocating chunks to different nodes. Our observation is that the same group of data chunks is likely updated across several append-commit iterations due to high spatial locality in updates [37]. Thus, CAU performs *data grouping*, which relocates the data chunks that are updated in the current append-commit iteration to be stored in the same rack, so that they can be updated together within the same rack in the following append-commit iterations; meanwhile, the relocation should maintain the same degree of fault tolerance.

To limit parity recomputations, our current data grouping design processes each stripe independently, rather than

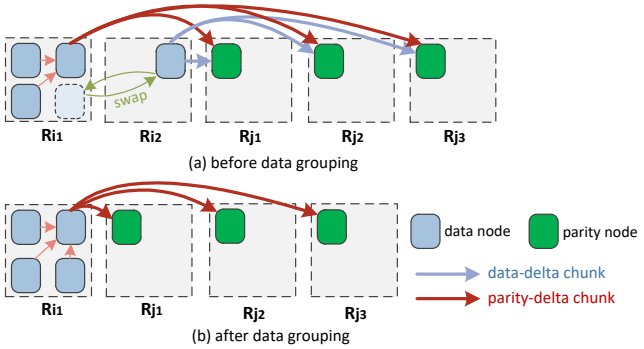


Fig. 4. Data grouping: we can swap the updated data chunk in R_{i2} with one of the chunks in R_{i1}, such that the four updated data chunks are now stored in R_{i1}.

multiple stripes. Also, to limit expensive data relocations, it only selects two racks for each stripe to perform data grouping, by relocating the data chunks of one rack into another rack. Such design choices are sufficient for reducing the cross-rack update traffic (see §5).

Algorithm 1 shows how data grouping works. CAU performs data grouping on a per-stripe basis at the end of each append-commit iteration. For each stripe that has data chunks updated in an append-commit iteration, CAU first identifies rack R_i that has the highest number of updated data chunks in the stripe in the last append phase (step 2). Suppose that R_i stores c_i data chunks including the i' updated data chunks, where we require that $c_i \leq n - k$ for single-rack fault tolerance (see §2.2). Then CAU checks the remaining $r-1$ racks. For each rack R_l (where $1 \leq i \neq l \leq r$) that has l' updated data chunks of the same stripe, CAU first checks if $i' + l' \leq c_i$ (step 4). The rationale is that if we swap all the l' updated chunks from R_l with l' non-updated data chunks in R_i, and the next append phase only updates the $i' + l'$ chunks, then we can eliminate the cross-rack update traffic from R_l in the future commit phases. Specifically, we calculate b_l and b_l^*, which correspond to the amounts of cross-rack update traffic (in units of chunks) before and after relocating l' chunks from R_l to R_i, based on selective parity updates in §3.2. Since the relocation will swap the l' updated data chunks in R_l and another l' non-updated data chunks in R_i, it also incurs a cross-rack traffic of $2l'$ chunks. Thus, the gain of such data grouping is $b_l - (b_l^* + 2l')$ (step 5). Finally, CAU finds the rack R_l that has the maximum gain, and swaps its l' updated data chunks with the l' non-updated chunks in R_i (steps 8-9). The complexity of Algorithm 1 is $O(tr)$, where t is the number of stripes that have data chunks updated and r is the number of racks.

Figure 4 depicts the idea of data grouping. Before data grouping, rack R_{i_1} has $i'_1 = 3$ updated data chunks and rack R_{i_2} has $i'_2 = 1$ updated data chunk. Suppose that we want to relocate the updated data chunk in R_{i_2} to R_{i_1} (which has the most updated data chunks). Before data grouping (see Figure 4(a)), in order to update the three parity chunks in racks R_{j_1}, R_{j_2}, and R_{j_3}, CAU needs to send one parity-delta chunk from R_{i_1} and one data-delta chunk from R_{i_2} to each of the three racks (i.e., $b_{i_2} = 6$ chunks of cross-rack update traffic). Now we swap $i'_2 = 1$ updated

Algorithm 1: Data Grouping

```
for each stripe do
    Identify rack R_i (1 ≤ i ≤ r) with the highest
      number of updated data chunks in the stripe in
      the last append phase
    for each rack R_l (1 ≤ l ≠ i ≤ r) do
        if l' + i' < c_i then
            Compute the gain G_l = b_l − (b_l^* + 2l')
        else
            Set the G_l = 0
    Find R_l where G_l is maximum among all racks
    Swap l' data chunks in R_l with l' non-updated ones in R_i
```

data chunk from R_{i_2} with a non-updated data chunk in R_{i_1} (which incurs two chunks of cross-rack traffic). If the four data chunks in R_{i_1} are updated again, CAU only needs to send $b^*_{i_2} = 3$ parity-delta chunks from R_{i_1} to R_{j_1}, R_{j_2}, and R_{j_3} (see Figure 4(b)). Thus, the gain of data grouping is $b_{i_2} - (b^*_{i_2} + 2 \times i'_2) = 1$ chunk.

3.4 Interim Replication

To prevent any data loss of updated data chunks in the append phase, CAU performs *interim replication* by storing replicas *temporarily* for the updated data chunks until we perform parity updates in the commit phase. Such replicas will be removed afterwards, so that they do not incur additional storage overhead in the long run.

Tolerance of a single rack failure: To balance between fault tolerance and the amount of cross-rack update traffic, CAU can store *one* replica for each newly updated data chunk in a different rack (i.e., not in the same rack where the data node with the newly updated data chunk resides), so as to tolerate any single-node or single-rack failure. For example, since each rack stores no more than $n-k$ chunks of a stripe (see §2.2), there must exist one of the $n-k$ parity nodes of the same stripe residing in a different rack, and we may choose the parity node to store the replica. We argue that providing temporary protection against a single-node or single-rack failure is sufficient in short term, as single failures are the most common failure pattern in production [15], [28]. Our reliability analysis also shows that CAU preserves fault tolerance (see §3.5).

Tolerance of multiple rack failures: We also study how to provide rack-level fault tolerance against a general number of rack failures via interim replication. Suppose that r^* denotes the number of rack failures to be tolerated by interim replication (where $r^* < r$). To tolerate any r^* rack failures, we emphasize that the number of chunks per stripe in any r^* racks should be no more than $n-k$ by default, which is also the number of parity chunks in a stripe of RS(n,k) (see §2.2).

To protect the reliability of a newly updated data chunk even in the presence of any r^* rack failures, we propose to find the parity nodes of the same stripe from another r^* racks (apart from the rack where the data node storing the newly updated data chunk resides) and store additional r^* replicas of the newly updated data chunk in them. Thus, by keeping r^*+1 replicas of each newly updated data chunk in a total of r^*+1 racks, even any r^* rack failures happen, we can always find at least one surviving replica to restore the newly updated data.

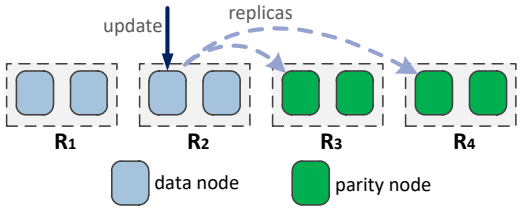


Fig. 5. Interim replication: we store two replicas of a newly updated data chunk (resided in R_2) in two racks (i.e., R_3 and R_4), in order to guarantee the reliability of the newly updated data chunk against any double rack failures.

We can easily prove via contradiction that for any newly updated data chunk, we can always find parity nodes from another r^* racks to store the r^* replicas. Specifically, suppose that we can only find parity nodes from another r' racks (where $r' \leq r^*-1$) to keep the replicas of a newly updated data chunk. That is to say, the r' racks found plus the rack that the newly updated chunk resides have all the $n-k$ parity nodes and at least a data node (i.e., the node storing the newly updated chunk). Consequently, these $r'+1 \leq r^*$ racks have at least $n-k+1$ chunks of a stripe, thereby violating the premier requirement that any r^* racks have no more than $n-k$ chunks of a stripe for RS(n,k).

We assess the impact of r^* selected in interim replication on the induced cross-rack update traffic (see Experiment A.5 in §5.1).

Figure 5 presents an example, where we set the number of replicas as two in interim replication to tolerate any double rack failures. When a data chunk in R_2 is updated, we transmit another two replicas of the newly updated data chunk to two parity nodes of the same stripe selected from R_3 and R_4, such that the reliability of the newly updated data chunk can be still guaranteed even in the face of any double rack failures.

3.5 Reliability Analysis

We now analyze the reliability of CAU. We show that even though CAU uses the append-commit procedure to update data chunks, if interim replication is enabled, then it still achieves the same level of reliability as the *baseline* erasure coding approach, which updates all parity chunks of a stripe immediately for each data chunk update. Our analysis studies the reliability of CAU during the append phase; once all parity chunks are updated in the commit phase, CAU has the same reliability as the baseline approach.

Setting: We consider both node failures and rack failures. Let θ_1 and θ_2 be the expected lifetimes of a node and a rack, respectively. Suppose that nodes and racks are independent and their lifetimes are exponentially distributed; such assumptions provide useful approximations [17]. The probability that a node fails (denoted by f_1) and the probability that a rack fails (denoted by f_2) for a duration of time τ can be computed by:

$$f_1 = 1 - e^{-\frac{\tau}{\theta_1}}, \quad f_2 = 1 - e^{-\frac{\tau}{\theta_2}}. \tag{4}$$

For node failures, we set $\theta_1 = 10$ years [9]. For rack failures, we focus on top-of-rack (ToR) switch failures. We

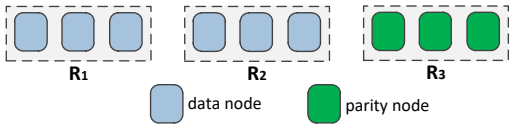


Fig. 6. A stripe with RS(9,6) used for reliability analysis. It can tolerate any triple-node failures or any single-rack failure.

take the average probability of a ToR switch failure in one year as 0.0278 [13, Figure 4]. From Equation (4), we estimate that $\theta_2 = 36$ years (by setting $f_2 = 0.0278$ and $\tau = 1$ year).

Failure events and probabilities: Our objective is to calculate the *data loss probabilities* for the baseline erasure coding approach as well as CAU in the append phase. For CAU, we consider two variants: (i) CAU-0, which keeps no replica for each newly updated data chunk, and (ii) CAU-1, which enables interim replication and keeps one replica for each newly updated data chunk in a parity node residing in a different rack. To simplify our analysis, we assume that the $n-k$ parity nodes are organized in the same rack, and the replicas are distributed across all parity nodes in CAU-1.

We first analyze the probability for a general number of node failures, while there is no rack failure. Let $E_{i,j}$ denote the event that i data nodes and j parity nodes fail concurrently, while all r racks are still available, where $0 \le i \le k$ and $0 \le j \le n-k$. We can compute the probability of $E_{i,j}$ (denoted by $\Pr(E_{i,j})$) as:

$$\Pr(E_{i,j}) = \underbrace{\Pr(A_i)}_{i \text{ data node failures}} \cdot \underbrace{\Pr(Y_j)}_{j \text{ parity node failures}} \cdot \underbrace{(1-f_2)^r}_{\text{no rack failure}} \quad (5)$$

where $\Pr(A_i)$ and $\Pr(Y_j)$ denote the probabilities when there are only i data nodes (where $0 \le i \le k$) and j parity nodes (where $0 \le j \le n-k$) fail, respectively. Then they can be calculated as follows:

$$\Pr(A_i) = \binom{k}{i} \cdot f_1^i \cdot (1-f_1)^{k-i}$$

$$\Pr(Y_j) = \binom{n-k}{j} \cdot f_1^j \cdot (1-f_1)^{n-k-j}.$$

We next analyze the probability for a general number of rack failures, while the remaining nodes in other surviving racks are accessible. Let F_l denote the event that l racks fail, where $0 \le l \le r$, while the nodes in the remaining $r-l$ racks are all available. Each rack consists of n/r nodes (assuming that n/r is an integer), so there are $(r-l)n/r$ remaining nodes in other surviving racks. We compute the probability of F_l (denoted by $\Pr(F_l)$) as:

$$\Pr(F_l) = \underbrace{\binom{r}{l} \cdot f_2^l \cdot (1-f_2)^{r-l}}_{l \text{ rack failures}} \cdot \underbrace{(1-f_1)^{(r-l)n/r}}_{\text{remaining nodes are available}} . \quad (6)$$

Using Equations (5) and (6), we compute the data loss probabilities for baseline erasure coding and CAU as follows.

Reliability analysis for RS(9,6): We first consider RS(9,6) for erasure coding in the analysis, as it is also used in production (e.g., QFS [24]). For RS(9,6), we assume that the $n = 9$ chunks of a stripe are stored in $n = 9$ distinct nodes organized in $r = 3$ racks with $n/r = 3$ nodes each. For simplicity, we organize all the $n-k = 3$ parity nodes within the same rack. This configuration can tolerate any triple-node failure or any single-rack failure (as shown in Figure 6).

• *Baseline erasure coding:* The baseline erasure coding approach under RS(9,6) ensures data availability in the following cases: (i) no more than three nodes fail while there is no rack failure (i.e., $\bigcup_{0 \le i+j \le 3} E_{i,j}$); and (ii) only one rack fails while the nodes in the surviving racks are available (i.e., F_1). The data loss probability (denoted by $\Pr_{ec}$) is given by:

$$\Pr_{ec} = 1 - \Big[\Big(\sum_{0 \le i+j \le 3} \Pr(E_{i,j}) \Big) + \Pr(F_1) \Big].$$

• *CAU-0:* Since there is no redundancy to protect newly updated data chunks in CAU-0, any data node failure will result in data loss. Thus, CAU-0 only ensures data availability in the following cases: (i) no failure happens (i.e., $E_{0,0}$); (ii) only parity nodes fail (i.e., $\bigcup_{1 \le j \le 3} E_{0,j}$); (iii) only the rack in which the parity nodes reside fails. The data loss probability (denoted by $\Pr_{cau0}$) is

$$\Pr_{cau0} = 1 - \Big[\Pr(E_{0,0}) + \Big(\sum_{1 \le j \le 3} \Pr(E_{0,j}) \Big) + \frac{\Pr(F_1)}{r} \Big].$$

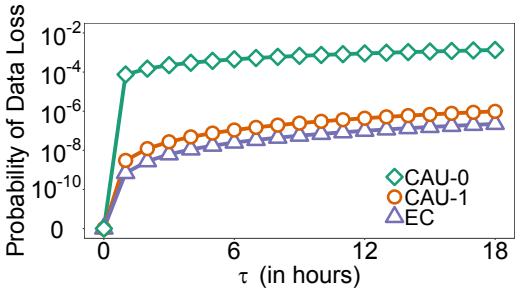
• *CAU-1:* Since CAU-1 replicates a new data chunk to another parity node, a pair of data node and parity node failures will result in data loss (assuming that each parity node holds the replicas of some data chunks). Thus, CAU-1 ensures data availability in the following cases: (i) no failure happens (i.e., $E_{0,0}$); (ii) only a single node fails (i.e., $E_{0,1} \cup E_{1,0}$); (iii) only two data nodes fail (i.e., $E_{2,0}$); (iv) only two parity nodes fail (i.e., $E_{0,2}$); (v) only three data nodes fail (i.e., $E_{3,0}$); (vi) only three parity nodes fail (i.e., $E_{0,3}$); and (vii) a single rack fails while the nodes in the surviving racks are available (i.e., F_1). Thus, the data loss probability (denoted by $\Pr_{cau1}$) is

$$\begin{aligned} \Pr_{cau1} = & 1 - [\Pr(E_{0,0}) + \Pr(E_{0,1}) + \Pr(E_{1,0}) + \Pr(E_{2,0}) \\ & + \Pr(E_{0,2}) + \Pr(E_{3,0}) + \Pr(E_{0,3}) + \Pr(F_1)]. \end{aligned}$$

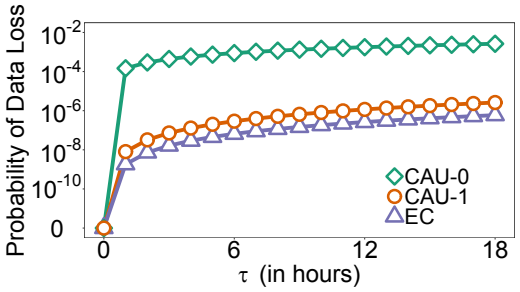
Reliability analysis for RS(16,12): We also consider RS(16,12), as it is also considered in production systems (e.g., Windows Azure Storage [15]). Like RS(9,6), we assume that all the $n = 16$ chunks are distributed in $n = 16$ nodes, which are further organized into $r = 4$ racks with $n/r = 4$ nodes per rack. In addition, we assume that the $n-k = 4$ parity nodes are in the same rack. This configuration can tolerate any four-node failure or any single rack failure.

The analysis of RS(16,12) is similar to that of RS(9,6), except the following differences. For the baseline erasure coding, it can tolerate any four-node failure under RS(16,12) (i.e., $\bigcup_{0 \le i+j \le 4} E_{i,j}$). For the CAU-0 approach, it can tolerate up to four parity node failures (i.e., $\bigcup_{1 \le j \le 4} E_{0,j}$). For the CAU-1 approach, it can tolerate two additional node failures: (i) only four data node failures (i.e., $E_{4,0}$), and (ii) any four parity node failures (i.e., $E_{0,4}$).

Figure 7 plots the data loss probabilities for $\Pr_{ec}$, $\Pr_{cau0}$, and $\Pr_{cau1}$ for a duration τ from 0 to 18 hours; we can view this as a duration of the append phase before the parity chunks are updated in the commit phase. As both f_1 and f_2 increase with τ, the data loss probabilities increase with τ



(a) Reliability analysis: RS(9,6) with $r = 3$



(b) Reliability analysis: RS(16,12) with $r = 4$.

Fig. 7. Data loss probabilities for baseline erasure coding, CAU-0 (no interim replication), and CAU-1 (with one replica in interim replication).

as well. CAU-0 has the highest data loss probability without any redundancy, so adding redundancy for the append phase is critical. CAU-1 has higher data loss probability than the baseline erasure coding approach, but it maintains the same order of magnitude for the data loss probability. For example, when $\tau = 18$, we have $\Pr_{cau1} = 9.83 \times 10^{-7}$ and $P_{ec} = 2.24 \times 10^{-7}$ under RS(9,6).

4 Implementation

We have implemented a CAU prototype. Figure 8 shows the CAU architecture, which comprises a metadata server and multiple storage nodes. The metadata server manages the metadata information of every chunk being stored, including the chunk ID, the stripe ID that the chunk belongs to, the data node ID where the chunk is stored, and the parity node IDs. It also records the chunk IDs of the updated data chunks as well as the stripe IDs that have data chunk updates during the append phase.

Append phase: We first describe the workflow of the append phase when a client issues an update request to a data chunk (see Figure 8). The client first sends the updated request, with the chunk ID of the updated data chunk, to the metadata server (step 1). The metadata server returns an *access ticket* (step 2), which states the data node ID where the data chunk is stored, the parity node ID where the replica of the data chunk is stored for interim replication, and the parity node IDs where the parity chunks will be stored in the commit phase. The client attaches the access ticket to the new data chunk and sends the data chunk to the corresponding data node (step 3). The data node appends the updated data chunk to its append-only log (step 4), and also forwards a replica of the updated data chunk to a parity node in another rack (step 5). The parity node stores the replica (step 6) and returns an ACK to the data node

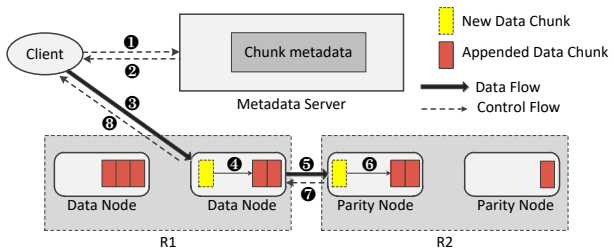


Fig. 8. System architecture of CAU.

(step 7). Finally, the data node sends an ACK to the client to complete the update request (step 8).

Commit phase: The metadata server triggers the commit phase to update parity chunks. It first identifies all stripes that have updated data chunks from its recorded information. For each stripe, it sends a commit request to the involved data nodes and specifies whether data-delta commit or parity-delta commit should be used, and the data nodes send the data-delta or parity-delta chunks accordingly. Each parity node returns an ACK to the metadata server upon completing the parity updates. When the metadata server receives the ACKs from all $n - k$ parity nodes, it ensures that the stripe is correctly committed.

Implementation details: Our CAU prototype is written in C on Linux. We implement the erasure coding operations using the Jerasure Library v1.2 [27]. To speed up performance, we also leverage multi-threading to parallelize data transmissions; for example, a node may send (receive) chunks to (from) multiple nodes via multiple threads, and the metadata server issues commit requests to multiple nodes via multiple threads as well.

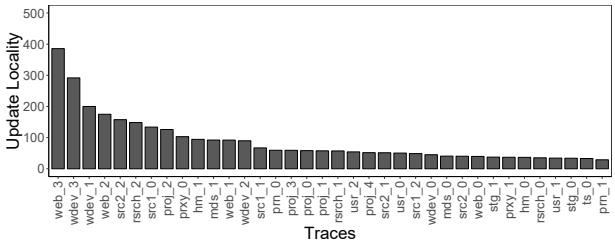
5 Evaluation

We evaluate CAU from three aspects: (i) trace-driven analysis, which shows that CAU significantly saves cross-rack update traffic under real-world workloads with different access characteristics; (ii) local cluster experiments, which show that CAU achieves high update performance in various cluster configurations; and (iii) Amazon EC2 experiments, which show that CAU achieves high update performance in real-world geo-distributed environments.

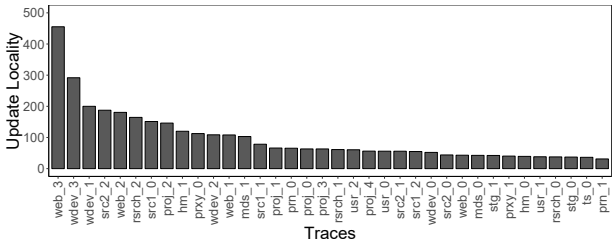
5.1 Trace-Driven Analysis

Preliminary trace analysis: We conduct trace-driven analysis on Microsoft Cambridge Traces [23], which record the access characteristics of enterprise storage servers. The traces span 36 volumes of 179 disks from 13 servers in one week. Each trace lists the read/write requests, including the timestamps, request addresses, request sizes, etc. We further classify the volumes based on a new metric called *update locality*. We sequentially partition the access requests of a volume into a collection of non-overlapped *update request sets*, each of which includes e consecutive update requests. Suppose that for a volume, the requests of an update request set are issued to u stripes on average. Then its update locality (denoted by l) can be calculated as

$$l = \frac{e}{u}. \quad (7)$$



(a) Update locality in RS(9,6)



(b) Update locality in RS(16,12)

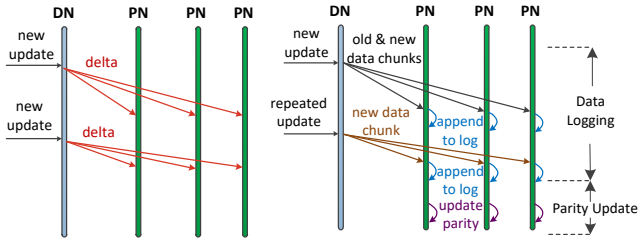
Fig. 9. An analysis of the update locality for MSR Cambridge Traces.

A lower u implies higher update locality, as the updates are more clustered in fewer stripes. To demonstrate, we conduct a preliminary analysis of the update locality for all the 36 volumes of the MSR Cambridge Traces, where e is set as 1,000. We pay special attention to two erasure codes: RS(9,6) and RS(16,12), as they are extensively considered in today's commodity storage systems [15], [24]. We set the size of a chunk as 1 MB.

Figure 9 shows the results. We make two observations. First, the update locality varies across different volumes. For example, the update locality of `web_3` is 385.6 in RS(9,6) (see Figure 9(a)), indicating that every 1,000 write requests in `web_3` only operate on the data chunks across 2.6 stripes on average (i.e., $u = \frac{e}{l} = \frac{1{,}000}{385.6} = 2.6$). The volume `prn_1` has the lowest update locality (e.g., 28.3 in Figure 9(a)), implying that every 1,000 write requests operate on the data chunks across 35.3 stripes on average (i.e., $u = \frac{e}{l} = \frac{1{,}000}{28.3} = 35.3$). Second, for a certain volume, its update locality increases when being deployed with an erasure code with a larger k. For example, the update locality of the volume `web_3` is 385.6 in RS(9,6), and reaches 455.3 in RS(16,12). The reason lies in that an erasure code with a larger k includes more data chunks in a stripe and therefore has a broader access range to receive more update requests.

In the interest of space, we select 20 volumes for our analysis: 10 of them have the highest update locality and another 10 of them have the lowest update locality among all 36 volumes. We consider two configurations of erasure-coding deployment: (i) RS(9,6) over $n = 9$ nodes and $r = 3$ racks; and (ii) RS(16,12) over $n = 16$ nodes and $r = 4$ racks[1]. We partition the address space of the trace for each volume into units of chunks, which we select 1 MB by default in our analysis. Our analysis assumes that the chunks are stored in a DC based on each of the above two configurations. For each volume of traces, we replay the write requests, which are treated as updated requests and will trigger parity updates.

1. Recall that in practice, a DC contains much more than n nodes (see §2.2). Our analysis can be viewed as focusing on the stripes stored in the same n nodes.



(a) Baseline (b) PARIX

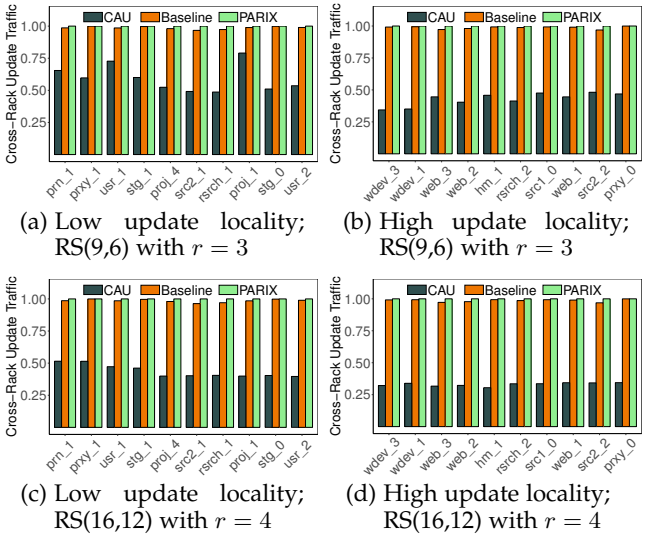
Fig. 10. Update flow of the baseline and PARIX.

In CAU, when the metadata server finds that the number of updated stripes (i.e., the stripes with updated data chunks) exceeds some threshold (denoted by u_s), it triggers the commit phase; by default, we set $u_s = 100$. We also enable both data grouping and interim replication (with one replica), so our analysis includes the cross-rack transfer overhead due to both features.

We compare CAU with two approaches (see Figure 10): the baseline delta-based update approach and PARIX [18]. The baseline transmits $n-k$ delta chunks to update all $n-k$ parity chunks immediately for each data chunk update. On the other hand, PARIX handles updates in two stages. If a data chunk is updated for the first time, PARIX sends the new data chunk and the old data chunk to all $n-k$ parity nodes. If the same data chunk is updated again, PARIX only sends the new data chunk to the parity nodes, each of which appends the received data chunk to a log. Later when the metadata server finds that the number of updated stripes exceeds u_s (by default, we set $u_s = 100$ as in CAU), it notifies each parity node to fetch the old and new data chunks from the log to update the parity chunk. Compared to the baseline, PARIX incurs slightly more network traffic (for sending the old data chunk), but saves I/Os for reading parity chunks to perform individual parity updates (each parity chunk can now be computed from multiple updated data chunks in batch). Note that both the baseline and PARIX provide the same degree of reliability protection (see the reliability analysis of the baseline erasure coding in §3.5).

When comparing CAU with PARIX and the baseline, we plot the average results over five runs, as well as the error bars that show the maximum and minimum across the five runs throughout the trace-driven analysis (some may be invisible).

Experiment A.1 (Comparisons of cross-rack update traffic): Figure 11 shows the amounts of cross-rack update traffic of the baseline, PARIX, and CAU, in which the results are normalized to that of PARIX. Overall, CAU significantly saves the cross-rack update traffic. For example, among all 20 volumes, CAU saves 48.4% and 51.4% of cross-rack update traffic on average compared to the baseline and PARIX, respectively, in the first configuration (i.e., RS(9,6) with $r = 3$ racks) (see Figures 11(a) and 11(b)), while the savings further increase to 60.9% and 63.4%, respectively, in the second configuration (i.e., RS(16,12) with $r = 4$ racks) (see Figures 11(c) and 11(d)). The second configuration comprises more racks and includes more parity chunks for fault tolerance, in which case the cross-rack update overhead in



(a) Low update locality; RS(9,6) with $r = 3$

(b) High update locality; RS(9,6) with $r = 3$

(c) Low update locality; RS(16,12) with $r = 4$

(d) High update locality; RS(16,12) with $r = 4$

Fig. 11. Experiment A.1 (Comparisons of cross-rack update traffic). The less cross-rack update traffic is better.

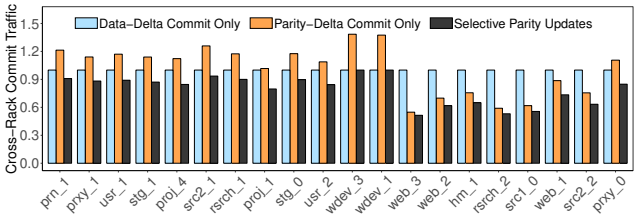


Fig. 12. Experiment A.2 (Analysis on selective parity updates). The less cross-rack update traffic is better.

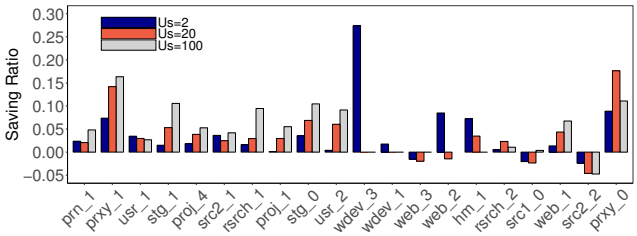


Fig. 13. Experiment A.3 (Analysis of data grouping). The larger saving ratio is better.

both the baseline and PARIX is higher.

Also, CAU generally saves more cross-rack update traffic when the update locality is high. For example, in RS(16,12) with four racks, CAU saves 56.3% of cross-rack update traffic over PARIX for the volumes with low update locality (see Figure 11(c)), while the saving increases to 66.9% for the volumes with high update locality (see Figure 11(d)). The reason is that the volumes with high update locality have more update requests clustered, thereby allowing CAU to be more likely to aggregate update requests within a rack in selective parity updates.

Experiment A.2 (Analysis on selective parity updates): We next analyze the performance gain of selective parity updates. We reconfigure the append-commit phase of our CAU prototype to perform different parity update approaches in the commit phase (see §3.2 for details): (i) data-delta commit only, which always performs data-delta commit for cross-rack parity updates, (ii) parity-delta commit only, which always performs parity-delta commit for cross-rack parity updates, and (iii) selective parity updates, in which we select the minimum of data-delta commit and parity-delta commit for each stripe to mitigate the cross-rack update traffic. We focus on the configuration RS(16,12) with $r = 4$ racks.

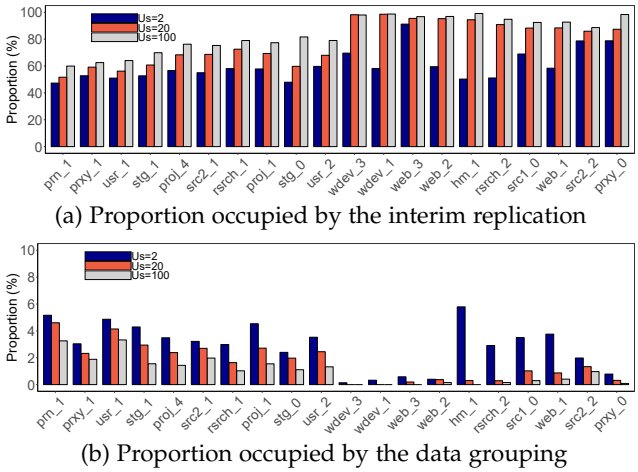
Figure 12 shows the results for all 20 volumes, in which we normalize the results with respect to data-delta commit only. Both data-delta commit only and parity-delta commit only may outperform each other for different volumes, yet selective parity updates achieve the least cross-rack update traffic in all volumes. Overall, selective parity updates reduce 20.7% and 20.0% of cross-rack update traffic on average compared to data-delta commit only and parity-delta commit only, respectively.

Experiment A.3 (Analysis of data grouping): As data grouping triggers cross-rack data reallocation (see §3.3), we also analyze its overhead and justify that the cross-rack update traffic saving brought by data grouping outweighs the data allocation overhead. We compare the saving ratio of the cross-rack update traffic with data grouping compared to that without data grouping. We focus on RS(16,12) with $r = 4$ racks. We also examine how the number of updated stripes u_s kept in the append phase affects the results.

Figure 13 shows the saving ratio of data grouping for all 20 volumes, where $u_s = 2$, 20, and 100. A positive saving ratio means that data grouping reduces the cross-rack update traffic. We see that 85% of cases (51 out of 60) have a positive saving ratio. The savings reach up to 27.4%, 17.6%, and 16.3% for $u_s = 2$, 20, and 100, respectively. For the other cases with a negative saving ratio, data group may incur up to 5.8% more cross-rack update traffic (`src2_2` when $u_s = 100$). We further examine the effect of u_s on the update performance in §5.2.

Experiment A.4 (Impact of interim replication and data grouping): As interim replication and data grouping both introduce additional cross-rack update traffic, we investigate their impact on the total cross-rack update traffic in CAU. We consider the configuration RS(16,12) with $r = 4$ racks, and calculate the proportions of cross-rack update traffic in CAU that are taken up by interim replication and data grouping.

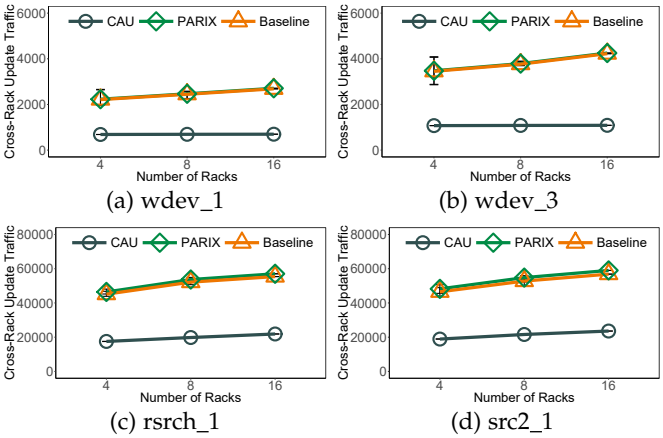
Figure 14 shows the results for all 20 volumes, where u_s is varied from 2 to 100. We first notice that interim replication and data grouping incur 74.0% and 1.8% of the cross-rack update traffic in CAU on average, respectively. The reason is that CAU performs interim replication for each updated data chunk for reliability, while data grouping is carried out in the commit operation only. As the commit operation is performed infrequently, the proportion of the cross-rack update traffic taken up by data grouping is marginal. In addition, the proportion of interim replication increase with u_s, while that of the data grouping exhibits an opposite tendency. The reason is that a larger u_s leads to fewer commit operations and hence less cross-rack update traffic in CAU. Thus, the proportion of the cross-rack update traffic caused by data grouping decreases with the increase of u_s. On the other hand, as the cross-rack update traffic induced by interim replication is still unchanged even if u_s varies, its proportion in the overall cross-rack update traffic increases with u_s instead.



(a) Proportion occupied by the interim replication

(b) Proportion occupied by the data grouping

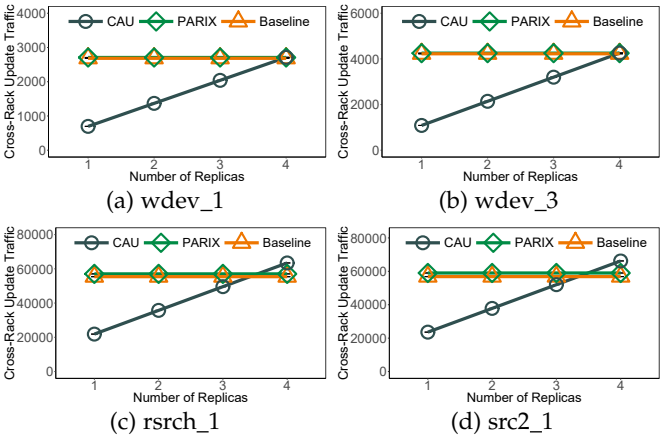
Fig. 14. Experiment A.4 (Impact of interim replication and data grouping).



(a) wdev_1 (b) wdev_3

(c) rsrch_1 (d) src2_1

Fig. 15. Experiment A.5 (Impact of number of racks). The less cross-rack update traffic is better.



(a) wdev_1 (b) wdev_3

(c) rsrch_1 (d) src2_1

Fig. 16. Experiment A.6 (Impact of general rack-level fault tolerance). The less cross-rack update traffic is better.

Experiment A.5 (Impact of number of racks): We evaluate the impact of the number of racks on the induced cross-rack update traffic. We select four volumes from the MSR Cambridge Traces: two volumes with high update locality (i.e., `wdev_1` and `wdev_3`), and another two volumes with low update locality (i.e., `rsrch_1` and `src2_1`). We consider a data center with 16 nodes and select the erasure code RS(16,12). We set $u_s = 100$, meaning that the parity commit occurs only when there are 100 stripes updated in the append phase. We set the number of replica in the interim replication as one, and then measure the cross-rack update traffic for the four volumes when the 16 nodes of the data center are organized into 4 racks (i.e., four nodes per rack), 8 racks (i.e., two nodes per rack), and 16 racks (i.e., one node only per rack), respectively.

Figure 15 shows the results. We observe that CAU always introduces the least cross-rack update traffic under different numbers of racks. In particular, CAU can reduce the cross-rack update traffic of the baseline and PARIX by 58.5-74.1% and 60.0-74.5% under different numbers of racks. Besides, even when the number of racks varies, CAU can still save a significant amount of cross-rack update traffic for the volumes with different update localities. For example, when the number of racks is configured as 16, CAU can reduce 74.2% (resp. 74.4%) of the cross-rack update traffic on average for the two volumes with high update locality (i.e., `wdev_1` and `wdev_3`) when compared to the baseline (resp. PARIX). The traffic reductions are 59.5% and 60.8% for the volumes with low update locality (i.e., `rsrch_1` and `src2_1`), respectively.

Experiment A.6 (Impact of general rack-level fault tolerance): We evaluate the induced cross-rack update traffic when CAU allocates different numbers of replicas in the interim replication (see §3.4 for details). To allow the wide-range selection for the number of replicas in the interim replication, we consider a data center with 16 nodes, which are organized into 16 racks (i.e., one node per rack). We then select RS(16,12) as the erasure coding scheme, such that each rack stores exactly one chunk of a stripe. We set $u_s = 100$, vary the number of replicas in the interim replication from 1 to 4, and measure the amount of data transferred across racks for parity updates.

Figure 16 depicts the results. We make three observations. First, the amount of cross-rack update traffic caused by CAU linearly increases with the number of replicas in the interim replication. The reason is that for each update request, CAU will store the specified number of replicas for each newly updated data chunk in other racks, to promise that these newly updated data chunks can still be available even in the presence of a certain number of rack failures. Second, CAU may even require more cross-rack update traffic than the baseline and PARIX when the number of replicas is set as four (i.e., $n - k$ in RS(16,12)) in the interim replication, especially for the volumes with low update locality (e.g., `rsrch_1` and `src2_1` in Figures 16(c) and 16(d)). The reason is that when CAU keeps four replicas (i.e., $n - k$ in RS(16,12)) in other racks for each newly updated data chunk, the amount of cross-rack update traffic of CAU in the append phase is equal to that of the baseline approach. As CAU needs additional cross-rack traffic in both selective parity updates (see §3.2) and data grouping (see §3.3), it finally introduces more cross-rack update traffic than the baseline approach. Third, as the baseline and PARIX do not rely on interim replication for protecting data reliability, their cross-rack update traffics are constant when the number of replicas in the interim replication changes.

Experiment A.7 (Impact of threshold number of updated

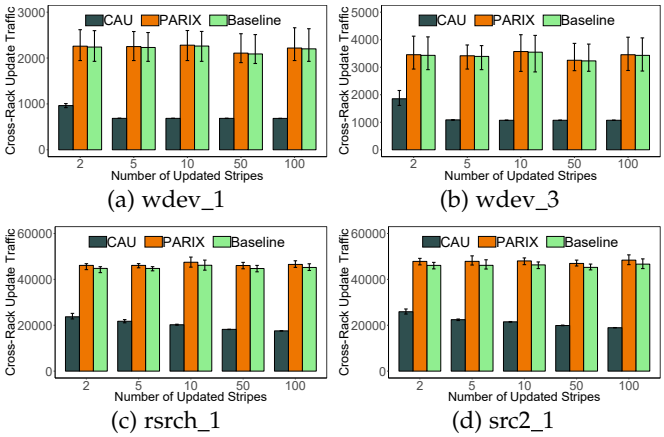


Fig. 17. Experiment A.7 (Impact of threshold number of updated stripes). The less cross-rack update traffic is better.

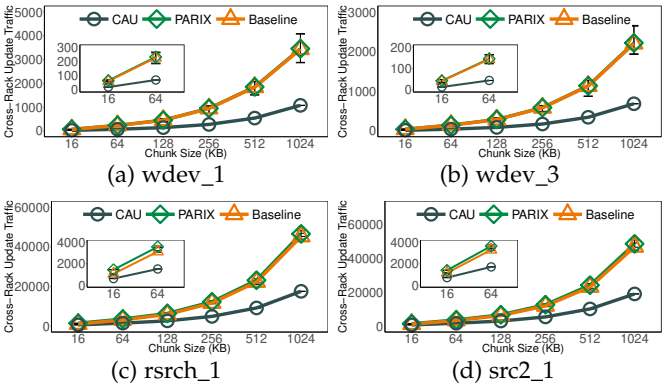


Fig. 18. Experiment A.8 (Impact of chunk sizes). The less cross-rack update traffic is better.

stripes): We also assess the cross-rack update traffic when the threshold number of updated stripes (i.e., u_s) that is allowed in the append phase changes. We deploy RS(16,12) in a data center with 16 nodes organized into $r = 4$ racks (i.e., four nodes per rack). We require the interim replication storing one replica for each newly updated data chunk. We vary the value of u_s from 2 to 100.

Figure 17 presents the results. The required cross-rack update traffic in CAU is less if CAU can wait for more stripes being updated in the append phase. For example, for the volume `rsrch_1`, CAU can reduce 25.9% of the cross-rack update traffic by setting u_s from 2 to 100. The reason is that by keeping more updated stripes before commit, CAU can avoid the frequent parity update and data grouping operations, thereby eliminating additional cross-rack update traffic. Also, for the volumes with high update locality (e.g., `wdev_1` and `wdev_3` in Figures 17(a) and 17(b)), the reduction of the cross-rack update traffic is prone to be less significant even for a larger u_s.

Experiment A.8 (Impact of chunk size): We analyze the impact of the chunk size on the cross-rack update traffic. We consider RS(16,12) in a data center with 16 nodes organized into four racks (i.e., four nodes per rack). We set $u_s = 100$ and store one replica in the interim replication. We vary the chunk size from 16 KB to 1 MB and measure the induced cross-rack update traffic (in units of MBs).

Figure 18 depicts the results. We make two observations.

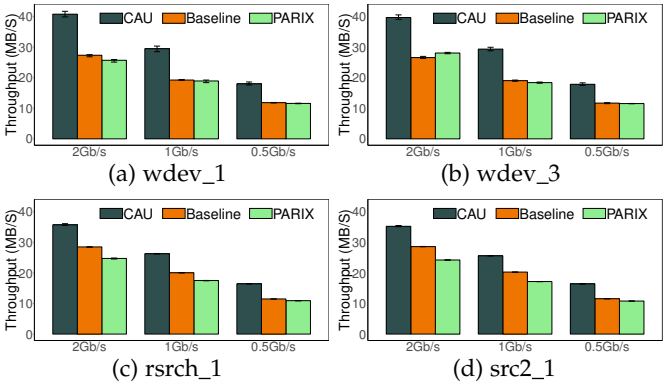


Fig. 19. Experiment B.1 (Impact of gateway bandwidth). The higher update throughput is better.

First, the amount of cross-rack update traffic drastically increases with the chunk size. Second, CAU preserves the savings of cross-rack update traffic under different chunk sizes. For example, for the volume `wdev_1` (Figure 18(a)), when the chunk size is 16 KB, CAU reduces 67.4% and 69.8% of the cross-rack update traffic compared to the baseline and PARIX, respectively. The reductions are still 68.7% and 68.9% when the chunk size increases to 1 MB.

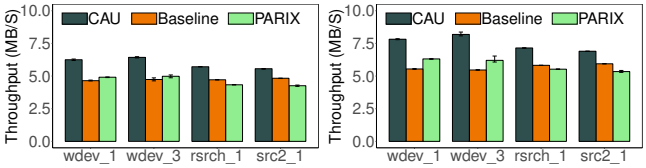
5.2 Local Cluster Experiments

We evaluate our CAU prototype on a local cluster with 12 machines to study its update performance under various cluster settings. Each machine runs Ubuntu 16.04.3 LTS, and has a quad-core 3.4 GHz Intel Core i5-7500 CPU, 32 GB RAM, and 1 TB TOSHIBA DT01ACA100 SATA disk. All nodes are connected via a 10 Gb/s Ethernet switch.

We consider RS(9,6) with $r = 3$ racks for erasure coding deployment. Among the 12 machines, we assign nine of them as storage nodes, one as the client, one as the metadata server, and the remaining one as the *gateway* that resembles the network core (see Figure 1). To simulate a hierarchical DC, we partition the nine storage nodes into three logical racks with three storage nodes each. Any inner-rack transfer can go through the 10 Gb/s switch directly, while any cross-rack transfer is redirected to the gateway, which relays the traffic to the destination node. We use the Linux traffic control command `tc` to limit the gateway bandwidth, so as to mimic the over-subscription scenario (see §1) where the cross-rack bandwidth is constrained and less than the inner-rack bandwidth. Also, unless otherwise specified, our CAU prototype issues buffered I/Os (the default I/O mode in Linux), in which read/write requests may be served by the buffer cache of each storage node.

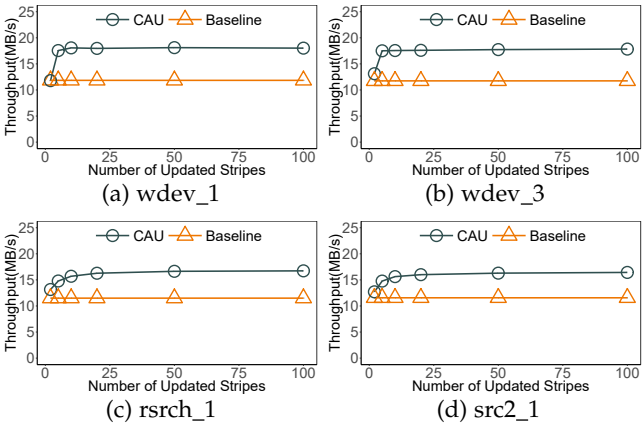
We again compare CAU with the baseline and PARIX as in §5.1. We focus on four volumes: `wdev_1`, `wdev_3`, `rsrch_1`, and `src2_1`. Both `wdev_1` and `wdev_3` have high update locality, while both `rsrch_1` and `src2_1` have low update locality. We plot the average results over five runs, as well as the error bars that show the maximum and minimum across the five runs.

Experiment B.1 (Impact of gateway bandwidth): We first evaluate the update performance for different values of gateway bandwidth. We vary the gateway bandwidth (i.e., the cross-rack bandwidth) as 0.5 Gb/s, 1 Gb/s, 2 Gb/s; note



(a) GW bandwidth: 0.5Gb/s (b) GW bandwidth: 2Gb/s

Fig. 20. Experiment B.2 (Impact of non-buffered I/O). The higher update throughput is better.



(a) wdev_1 (b) wdev_3 (c) rsrch_1 (d) src2_1

Fig. 21. Experiment B.3 (Impact of u_s). The higher update throughput is better.

that the cross-rack bandwidth in production is 1 Gb/s [30], while the inner-rack bandwidth is 10 Gb/s.

Figure 19 shows the results in terms of update throughput (i.e., the amount of updated data chunks per second). Overall, CAU significantly improves the update throughput by 41.8% and 51.4% on average compared to the baseline and PARIX, respectively. Also, the performance gain of CAU increases as the gateway bandwidth decreases (i.e., more constrained cross-rack bandwidth). For example, for `wdev_3`, when the gateway bandwidth is 2 Gb/s, CAU increases the update throughput of the baseline and PARIX by 49.3% and 41.4%, respectively; when the gateway bandwidth decreases to 0.5 Gb/s, the improvements increase to 52.6% and 54.6%, respectively. When the cross-rack bandwidth is more constrained, the reduction of cross-rack update traffic in CAU is more beneficial for high update performance.

Note that the baseline generally outperforms (slightly) than PARIX in most cases, as PARIX is designed to reduce I/Os in parity updates at the expense of incurring more cross-rack transfers [18]. Since buffer I/Os are used here and I/O requests may be served by the buffer cache, the cross-rack bandwidth plays a more critical role in determining the update performance.

Experiment B.2 (Impact of non-buffered I/O): We now study the impact of non-buffered I/O (i.e., the buffered cache for I/O requests is disabled) on update throughput. Specifically, we enable the flag `O_SYNC` in write requests to flush all data to disk, and also enable the flag `O_DIRECT` in read requests to directly retrieve data from disk without accessing the buffer cache. We consider two settings of the gateway bandwidth: 0.5 Gb/s and 2 Gb/s.

Figure 20 shows the results. Clearly, compared to the case with buffered I/O, the update throughput drops when non-buffered I/O is used and the I/O overhead also plays a role in determining the update performance. Nevertheless, CAU still improves the updated throughput by 29.6% and 29.1% compared to the baseline and PARIX, respectively. CAU not only reduces the cross-rack update traffic, but also reduces the I/O overhead by aggregating the updates of data and parity chunks.

We note that PARIX achieves higher update throughput than the baseline for `wdev_1` and `wdev_3`, both of which have high update locality. In both volumes, the updates are more clustered and have less cross-rack traffic, so the reduction of the I/O overhead in PARIX is more advantageous in improving the update performance.

Experiment B.3 (Impact of u_s): We also study how the number of updated stripes u_s kept in the append phase affects the update performance of CAU. We vary u_s as 2, 5, 10, 20, 50, and 100, and fix the gateway bandwidth as 0.5 Gb/s. For comparisons, we also include the results of the baseline, which remain fixed for different values of u_s.

Figure 21 shows the results. When u_s is small, CAU has similar performance to the baseline as it triggers parity updates frequently. The update throughput of CAU increases with u_s at the beginning since it has more opportunity to aggregate updates in the append phase, but becomes stable when u_s exceeds 10.

TABLE 1
Measured bandwidth among regions (Unit: Mb/s)

Regions	Seoul	Singapore	Sydney	Tokyo
Seoul	919.0	46.4	43.0	118.0
Singapore	58.0	560.4	43.6	43.6
Sydney	44.8	37.0	840.3	53.9
Tokyo	108.9	53.7	62.6	493.5

5.3 Amazon EC2 Experiments

We further evaluate CAU on Amazon EC2 in geo-distributed settings. We create a set of virtual machine (VM) instances across four regions, namely Tokyo, Seoul, Sydney, and Singapore. We select VM instance type `t2.small`, in which each VM instance runs Ubuntu 14.04.5 LTS and has a 2.40GHz Intel Xeon E5-2627 CPU, 2GB memory, and 70GB storage capacity. Before running our experiments, we first measure the inner-region and cross-region bandwidth across the four regions using `iperf`. Table 1 presents the results from one of our measurements, in which each number denotes the measured bandwidth from the region in the row to the region in the column. It shows that the cross-region bandwidth is much more scarce than the inner-region bandwidth, such that the inner-region bandwidth is 11.3× the cross-region bandwidth on average.

We deploy RS(16,12) and store four chunks of each stripe at four different VM instances in each region. We also create two additional VM instances as the metadata server and the client. We compare the baseline, PARIX, and CAU, and set the chunk size as 512KB. We present the average results over five runs, and also show the error bars indicating the maximum and minimum across the five runs.

Figure 22 plots the results. Note that the network bandwidth among the VM instances fluctuates across time, so

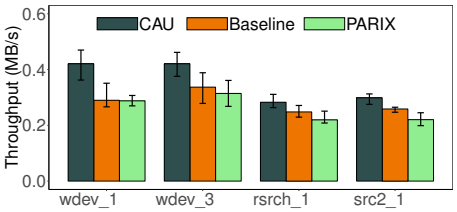


Fig. 22. Update throughput on Amazon EC2. The higher update throughput is better.

the variance of each result is higher than that in local cluster experiments. Again, CAU outperforms both the baseline and PARIX, and its performance gain is higher in `wdev_1` and `wdev_3` with high update locality. In `wdev_1`, the average update throughput of CAU is 31.5% and 32.4% higher than those of the baseline and PARIX, respectively, while in `wdev_3`, the average update throughput of CAU is 24.9% and 33.8% higher than those of the baseline and PARIX, respectively.

6 Related Work

We review related work on improving parity update performance in erasure-coded storage.

Delta-based updates: Existing parity update solutions mostly build on delta-based updates for partial-stripe writes (see §2.3). Parity logging [38] is a well-known approach of mitigating parity update overhead in RAID-5 by eliminating the reads of parity chunks and appending parity deltas to a log device. CodFS [6] realizes parity logging in clustered storage, by placing parity deltas next to the original parity chunks to limit disk seeks during recovery. PARIX [18] eliminates the reads of old data chunks for parity computations by directly logging data deltas (i.e., changes of data chunks), at the expense of extra network transmissions for reconstructing parity chunks from the original data chunk. Other studies enhance delta-based updates in different aspects. FAB [12] proposes quorum-based algorithms for decentralized erasure coding operations. Aguilera *et al.* [2] propose distributed protocols for lightweight concurrent updates. T-Update [25] finds a minimum spanning tree to propagate parity updates across nodes; while T-Update constructs the minimum spanning tree given a rack-based DC topology, it does not reduce the amount of cross-rack update traffic. CAU also builds on delta-based updates. In contrast to previous work, CAU mitigates the cross-rack update traffic in erasure-coded DCs by taking into account the hierarchical nature of DCs.

Full-stripe updates: To eliminate the reads of parity chunks in partial-stripe writes, some approaches directly form new stripes using new data chunks and issue full-stripe updates in a log-structured manner. They also mark the old data chunks as invalid and reclaim their space via garbage collection. Full-stripe updates are commonly used in systems that treat stored data as immutable, such as HDFS-RAID [1], QFS [24], BCStore [20], and Giza [7]. However, full-stripe updates not only incur garbage collection overhead to reclaim the space of stale data chunks, but also require parity re-computations for the remaining active data chunks.

Data placement: Some approaches (e.g., [34], [35]) propose new data placement strategies that group the data chunks that are likely accessed together into the same stripe, so as to mitigate parity update overhead. CAU also addresses data placement via data grouping, but is tailored for mitigating the cross-rack update traffic.

7 Conclusion

Erasure coding provides a storage-efficient means for modern DCs to achieve data reliability. However, it incurs high update penalty in maintaining the consistency between data and parity chunks. CAU is a cross-rack-aware update mechanism that addresses the hierarchical nature of DCs. It mitigates the cross-rack update traffic through selective parity updates and data grouping, and further maintains data reliability through interim replication. Trace-driven analysis, local cluster experiments, and Amazon EC2 experiments show that CAU reduces a significantly amount of cross-rack update traffic and achieves high update throughput.

Acknowledgments

This work is supported in part by the Research Grants Council of the Hong Kong Special Administrative Region, China (GRF 14216316 and AoE/P-404/18), the National Natural Science Foundation of China (Grant No. 61602120), and the Open Research Fund of the State Key Laboratory of Integrated Services Networks (Xidian University) (Grant No. ISN21-19).

References

[1] HDFS RAID. http://wiki.apache.org/hadoop/HDFS-RAID, 2011.

[2] M. Aguilera, R. Janakiraman, and L. Xu. Using Erasure Codes Efficiently for Storage in a Distributed System. In *Proc. of IEEE/IFIP DSN*, 2005.

[3] F. Ahmad, S. Chakradhar, A. Raghunathan, and T. Vijaykumar. ShuffleWatcher: Shuffle-aware Scheduling in Multi-tenant MapReduce Clusters. In *Proc. of USENIX ATC*, 2014.

[4] T. Benson, A. Akella, and D. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proc. of ACM IMC*, 2010.

[5] B. Calder, J. Wang, A. Ogus, et al. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proc. of ACM SOSP*, 2011.

[6] J. Chan, Q. Ding, P. 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.

[7] Y. Chen, S. Mu, J. Li, C. Huang, J. Li, A. Ogus, and D. Phillips. Giza: Erasure Coding Objects across Global Data Centers. In *Proc. of USENIX ATC*, 2017.

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

[9] A. Cidon, R. Escriva, S. Katti, M. Rosenblum, and E. Sirer. Tiered Replication: A Cost-effective Alternative to Full Cluster Geo-replication. In *Proc. of USENIX ATC*, 2015.

[10] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-diagonal Parity for Double Disk Failure Correction. In *Proc. of USENIX FAST*, 2004.

[11] 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.

[12] S. Frolund, A. Merchant, Y. Saito, S. Spence, and A. Veitch. A Decentralized Algorithm for Erasure-Coded Virtual Disks. In *Proc. of IEEE/IFIP DSN*, 2004.

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

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

[15] 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.

[16] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar. Network-Aware Scheduling for Data-Parallel Jobs: Plan When You Can. In *Proc. of ACM SIGCOMM*, 2015.

[17] G. Lefebvre and M. J. Feeley. Separating Durability and Availability in Self-Managed Storage. In *Proc. of ACM SIGOPS European Workshop*, 2004.

[18] 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.

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

[20] S. Li, Q. Zhang, Z. Yang, and Y. Dai. BCStore: Bandwidth-Efficient In-memory KV-Store with Batch Coding. In *Proc. of IEEE MSST*, 2017.

[21] X. Li, R. Li, P. P. Lee, and Y. Hu. OpenEC: Toward Unified and Configurable Erasure Coding Management in Distributed Storage Systems. In *Proc. of USENIX FAST*, 2019.

[22] S. Muralidhar, W. Lloyd, S. Roy, et al. F4: Facebook's Warm Blob Storage System. In *Proc. of USENIX OSDI*, 2014.

[23] D. Narayanan, A. Donnelly, and A. Rowstron. Write Off-loading: Practical Power Management for Enterprise Storage. *ACM Trans. on Storage*, 4:1–23, 2008.

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

[25] 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.

[26] J. Plank. A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems. *Software - Practice & Experience*, 27(9):995–1012, 1997.

[27] J. Plank, S. Simmerman, and C. 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.

[28] K. Rashmi, N. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A Solution to the Network Challenges of Data Recovery in Erasure-coded Distributed Storage Systems: A Study on the Facebook Warehouse Cluster. In *USENIX Workshop on HotStorage*, 2013.

[29] I. Reed and G. Solomon. Polynomial Codes over Certain Finite Fields. *Journal of the Society for Industrial & Applied Mathematics*, 8(2):300–304, 1960.

[30] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. Xoring Elephants: Novel Erasure Codes for Big Data. In *Proc. of the VLDB Endowment*, volume 6, pages 325–336, 2013.

[31] J. Schindler, S. Shete, and K. Smith. Improving Throughput for Small Disk Requests with Proximal I/O. In *Proc. of USENIX FAST*, 2011.

[32] R. Sears and R. Ramakrishnan. bLSM: A General Purpose Log Structured Merge Tree. In *Proc. of ACM SIGMOD*, 2012.

[33] Z. Shen and P. P. Lee. Cross-Rack-Aware Updates in Erasure-Coded Data Centers. In *Proc. of ACM ICPP*, page 80, 2018.

[34] Z. Shen, P. 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.

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

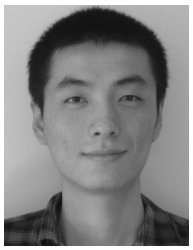
[36] Z. Shen, J. Shu, and P. P. Lee. Reconsidering Single Failure Recovery in Clustered File Systems. In *Proc. of IEEE/IFIP DSN*, 2016.

[37] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber. Extending SSD Lifetimes with Disk-Based Write Caches. In *Proc. of USENIX FAST*, 2010.

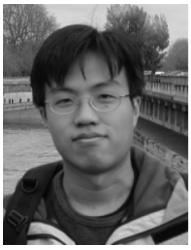
[38] D. Stodolsky, G. Gibson, and M. Holland. Parity Logging Overcoming the Small Write Problem in Redundant Disk Arrays. In *Proc. of ISCA*, 1993.

[39] A. Vulimiri, C. Curino, P. Godfrey, T. Jungblut, J. Padhye, and G. Varghese. Global Analytics in the Face of Bandwidth and Regulatory Constraints. In *Proc. of USENIX NSDI*, 2015.

[40] H. Weatherspoon and J. D. Kubiatowicz. Erasure Coding vs. Replication: A Quantitative Comparison. In *Proc. of IPTPS*, 2002.



Zhirong Shen received the B.S. degree from University of Electronic Science and Technology of China in 2010, and the Ph.D. degree in Computer Science from Tsinghua University in 2016. He is now an associate professor at Xiamen University. His current research interests include storage reliability and storage security. He is a member of the IEEE.



Patrick P. C. Lee received the B.Eng. degree (first-class honors) in Information Engineering from the Chinese University of Hong Kong in 2001, the M.Phil. degree in Computer Science and Engineering from the Chinese University of Hong Kong in 2003, and the Ph.D. degree in Computer Science from Columbia University in 2008. He is now an Associate Professor of the Department of Computer Science and Engineering at the Chinese University of Hong Kong. His research interests are in various applied/systems topics including storage systems, distributed systems and networks, dependability, and security. He is a senior member of the IEEE.