CPU: Cross-Rack-Aware Pipelining Update for Erasure-Coded Storage

Haiqiao Wu, Wan Du, *Member, IEEE,* Peng Gong, *Member, IEEE,* and Dapeng Oliver Wu, *Fellow, IEEE*

Abstract—Erasure coding is widely used in distributed storage systems (DSSs) to efficiently achieve fault tolerance. However, when the original data need to be updated, erasure coding must update every encoded block, resulting in long update time and high bandwidth consumption. Exiting solutions are mainly focused on coding schemes to minimize the size of transmitted update information, while ignoring more efficient utilization of bandwidth among update racks. In this paper, we propose a parallel Cross-rack Pipelining Update scheme (*CPU*), which divides the update information into small-size units and transmits these units in parallel along with an update pipeline path among multiple racks. The performance of *CPU* is mainly determined by slice size and update path. More slices bring finer-grained parallel transmissions over cross-rack links, but also introduces more overheads. An update path that traverses all racks with large-bandwidth links provide short update time. We formulate the proposed pipelining update scheme as an optimization problem, based on a new theoretical pipelining update model. We prove the optimization problem is NP-hard and develop a heuristic algorithm to solve it based on the features of practical DSSs and our implementations, including *Big chunk* and *Small overhead*. Specifically, we determine the best update path first by solving a max-min problem and then decide the slice size. We further simplify the slice size selection by offline learning a range of interesting (*RoI*), in which all slice sizes provide similar performance. We implement *CPU* and conduct experiments on Amazon EC2 under a variety of scenarios. The results show that *CPU* can reduce the average update time by 48.2%, compared with the state-of-the-art update schemes.

Index Terms—Distributed Storage System, Cross-rack-aware updates, Pipelining, Erasure coding.

1 Introduction

Distributed Storage System (DSSs) are the basics of many data-based applications [1], [2], [3], e.g., video streaming, data analysis, which require a stable and reliable data environment. However, DSSs suffer from highly dynamic storage nodes due to many unforeseen factors, such as node failure, link breaking, and disk malfunction [4], [5], [6], [7]. Replication and erasure coding are two mostly adopted solutions [8] to provide fault tolerance. In replication, multiple copies of the data are stored on different servers. However, erasure coding divides the original data into a set of fixed-size unit, called *data chunks*, and encodes data chunks into additional redundant chunks called *parity chunks*, such that any subset of a sufficient number of data and parity chunks can reconstruct the lost chunks due to failure. Compared with replication, erasure coding can save an order of magnitude of storage space under the same data reliability [9], [10], [11], [12], [13]. Moreover, although introducing higher network overhead (almost 8%) compared to replication, erasure coding improves the performance of data-intensive applications (up to 25% better than replication) [14]. These benefits motivate many enterprises to adopt erasure coding, e.g., Google [33], Facebook [34], and Microsoft [36].

- *H. Wu is with the School of Mechatronical Engineering, Beijing Institute of Technology, Beijing, China. He is also a visiting student in the Dept. of Computer science and Engineering, University of California, Merced, USA.E-mail: haiqiaowu@outlook.com, hwu62@ucmerced.edu.*
- *W. Du is with the Dept. of Computer science and Engineering, University of California, Merced, USA. E-mail: wdu3@ucmerced.edu.*
- *P. Gong is with the School of Mechatronical Engineering, Beijing Institute of Technology, Beijing, China.E-mail: penggong@bit.edu.cn.*
- *D. Wu is with the Dept. of Electrical and Computer Engineering, University of Florida, USA. E-mail: dpwu@ufl.edu.*

However, erasure coding consumes higher network traffic and takes longer time to complete the update process for maintaining the consistency between data and parity chunks. Specifically, to finish an update, replication only needs overwrite the replica by transmitting the modified data. While, erasure coding needs update both the data chunks and related parity chunks, involving data transmission and computation for updating parity chunks. We argue that data update is intensive for many real-world systems [17], [18], [19] [52]. For example, the update traffic occupies nearly 50% of the low-latency workloads in Yahoo's DSS and keeps growing [17]. Meanwhile, trace analysis in [52] illustrates that more than 90% writes are updates. Thus, improving the update efficiency is a critical issue of erasure coding.

Our insight is that we can exploit the idle bandwidth between parity nodes that store parity chunks to improve the update efficiency of erasure coding. Traditionally, the update process is performed on a star structure [16], which means that the data node with an update computes the update information and transmits it to all the related parity nodes respectively. The data node will be burdened with the data computation and transmission under the star scheme when updates are intensive. However, the links between parity nodes remains idle. Although a tree update [24] is proposed to mitigate the load on data nodes and exploit the idle links between parity nodes by organizing the transmission of update data in a tree structure, there is still significant space left to reduce the update time further by utilizing the idle links more efficiently.

Moreover, the bandwidth between racks is much scarcer (about 5-20x lower [25] [26]) than the inner-rack bandwidth. Moreover, for geo-distributed storage systems (Geo-DSSs),

in which nodes are located in multiple geographical regions, the cross-region bandwidth is also much smaller than the inner-region bandwidth [27]. Thus, in this paper, we propose a cross-rack (cross-region) pipelining update scheme for erasure-coded DSSs, called *CPU*. The main idea of *CPU* is to pipeline the update information in smaller slices along a path. The update path composes of the data rack (the rack with the data node to be updated represented as the source of an update path) and dependent parity racks (the racks with the parity nodes to be updated within the same stripe of the data node). Through this method, the burden on the data rack can be released, and the bandwidth resources between parity racks are fully utilized.

Based on our implementations (§5), the update time of *CPU* mainly comes from network transmission. We formulate the optimization problem of the pipelining update model. And we further prove that the problem is an NP-hard problem. To address the problem, we propose a heuristic algorithm based on the two observations: 1) DSSs encapsulate the data into large chunk (default 64MB for Hadoop.1 [31] and QFS [32]) to reduce the metadata size; 2) The traffic overhead to transfer a slice is small (less than 200 bytes) in *CPU*. These observations indicate that the chunk can be divided into thousands of slices before introducing significant traffic overhead.

Thus, as stated in §4-E, the performance of *CPU* is mainly bottlenecked by the worst link in the update path when exploiting enough parallelisms. *CPU* can first find a path with max-minimum link bandwidth to do pipelining update. However, the computation for finding the optimal path is time-consuming because it is still NP-hard problem, which will increase the update time in *CPU*. To achieve a good balance between accuracy and computation, we adopt tabu search algorithm to obtain the update path.

Given the update path, the slice size selection can be obtained because it is a concave problem. For simplification, instead of obtaining the optimal slice size, which is related to the bandwidth distribution of links in the update path, we can select a slice size as default size from the *RoI*, where *CPU* performs almost same as the slice size changes. We can obtain *RoI* under the worst case that bandwidths of all the links in the found path are same because it still ensures the good performance of *CPU* due to *Big chunk* and *Small overhead*. Thus, the *RoI* can be profiled offline before deploying *CPU* on practical DSSs under homogeneous bandwidth distribution.

We implement a *CPU* prototype that can be deployed in distributed environments and applicable for general erasure codes, including Reed-Solomon codes [30] and Local Reconstruction Codes [36]. RS codes are implemented as an example in *CPU*. The experiments on Amazon EC2 under various parameter settings show that *CPU* can significantly improve the update efficiency over star update and tree update, e.g., reducing the update time by 67% and 48.2% under three parity racks and 64MB chunk compared with the two update schemes, respectively. The contributions of this work can be summarized as follows:

- We propose a rack-aware pipelining update technique to guide the transmission of update information in slices along a path to distribute update traffic and fully utilize bandwidths across racks.
- We formulate the optimization problem based on pipelining update model, and prove the problem is NP-hard.
- We propose a heuristic algorithm to solve the problem through determining the update first through tabu search algorithm and then obtaining the slice size based on the given update path. *RoI* is further proposed to simply the slice size selection.
- We implement a *CPU* prototype and conduct experiments on Amazon EC2 under various parameter settings to evaluate the update efficiency of *CPU*.

2 Background

Erasure coding: In this paper, we mainly focus on Reed-Solomon (RS) codes, which are deployed in today's DSSs [33], [34], [35]. The source node divides the data object into fixed-size units called *chunks* and sends them to the nodes after encoding operations. Each node in the system could reconstruct the required data by accessing data from the nodes. Specifically, an erasure code is typically configured with two integer parameters $(n; k)$, where $k < n$. For every k original chunks called *data chunks*, it encodes them into $n \quad k$ coded chunks called *parity chunks*. The set of n coded chunks is called a *stripe*, which are distributed across n storage nodes to tolerate any $n \quad k$ node failure, and any k out of n coded chunks can be decoded to the original k uncoded chunks. A large-scale storage system stores data of multiple stripes, all of which are independently encoded. All additions and multiplications of practical erasure codes are based on Galois Field arithmetic over w-bit units called *words*. Each chunk is partitioned into multiple w-bit words, and *words* at the same offset of each chunk within a stripe are encoded together.

Hierarchy of DSSs: The two-level hierarchical architecture of DSSs is showed in Figure 1. Specifically, the minimum storage devices to provide storage space for a DSS are *nodes*, which are located in different racks. The communication among the nodes in the same rack is via a top-of-rack (ToR) switch, while multiple racks are connected by the core switches that collectively form the *core network*.

Existing erasure-coded DSSs distribute each stripe across n nodes in n distinct racks [33] [34] [36]. For some parameter $m < n$, m is the number of racks, recent studies [37] [38] propose to store each stripe in n nodes that reside in m racks to minimize the cross rack traffic during failure repair and update at the expense of reduced rack-level fault tolerance. For example, if all the $n \quad k$ *parity chunks* of a stripe are stored in one parity rack, the updates can be completed with only one cross-rack data transmission from the data racks, while the DSS can only tolerant one rack failure. Thus, we consider the situation that the DSSs place parity chunks in a stripe to $m \ 6 \ n$ racks, in which any updates will cause cross-rack traffic. The constrained bandwidths between racks are the main reason that weakens the performance of a DDS [37] [38] [41] [50]. In this paper, we study how to exploit the cross-rack bandwidth more efficiently to speed up update in DSSs.

Parity Updates: Most practical erasure codes are linear codes, where each parity chunk could be represented by

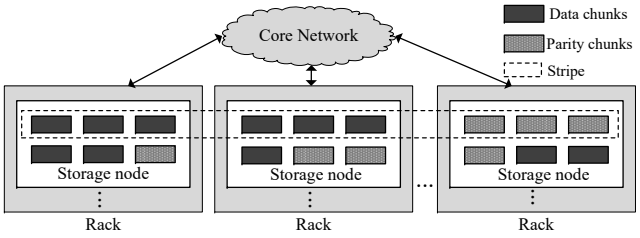


Fig. 1. The storage hierarchy of DSSs.

the linear combination of the k data chunks with Equation (1). $_{ij}$ is the coefficient for p_j from d_i. Since the encoding and update operation between stripes are independent with each other, we discuss the update process within a stripe.

$$p_j = \sum_{i=1}^{k} {}_{ij}\, d_i; \quad 1 \quad j \quad n \quad k \tag{1}$$

From Equation (1), we can also efficiently update a parity chunk for any update of a data chunk. Specifically, if a data chunk $d_i; 1 \quad i \quad k$, is modified to d_i^0, each parity chunk p_j (where $1 \quad j \quad n \quad k$) can be updated into p_j^0 as follows:

$$p_j^0 = p_j + {}_{ij}(d_i^0 \quad d_i) \tag{2}$$

Based on equation (2), we can find that the new parity chunk could be updated by accessing the *delta* of the data chunk (d_i^0-d_i), called *data-delta chunk*, without accessing the other unchanged data chunks within the same stripe. The volume of a data-delta chunk is the same as the size of a data chunk. This type of parity updates is called *delta-based update*. Specifically, the update processes consist of two steps: First, the data node with an update overwrites the original data chunk d_i with the updated data chunk d_i^0, and computes and sends the data-delta chunk to all related parity nodes; Then, each parity node gets the updated parity chunk p_j^0 by combining the data-delta chunk with the original parity chunk p_j according to equation (2).

3 Motivation

In this section, we use a simple example to show the limitations of the existing update schemes and the potential performance the proposed pipelining scheme can provide. A DSS example system is shown in figure 2(a). The numbers on the edges denote the seconds for a data-delta chunk transmission between storage nodes.

For star update [16], a data node with an update is responsible for sending a data-delta chunk to each related parity node $PN_1; PN_2; :::; PN_{n \quad k}$. Figure 2(b) shows the star update for $n \quad k = 4$. Since the data node can only communicate with one parity node at one time, which means the data transmissions between the data node and parity nodes are done in sequence, an update process in star structure will take *18s*. In general, for star update, an update takes R timeslots. In addition, the data nodes need send four data-delta chunks, which will burden the data node with update traffic, so as to bottleneck the update time under limited link resource. However, the links between parity nodes are always idle during the update process.

Tree update [24] constructs an update tree (the data node as the root and parity nodes as the children) to organize the transmission of update data. Figure 2(c) shows how T-Update completes an update with $n \quad k = 4$. At the beginning, the DN_0 sends a data-delta chunk to PN_1. Second, DN_0 sends a data-delta chunk to PN_2, at the same time, PN_1 forwards the received chunk to PN_3 after receiving the whole data-delta chunk. Finally, PN_2 forwards the received chunk to PN_4. Thus, the tree update reduces the update time to *11s*. Overall, tree update only takes $log_2 d(R + 1)e$ timeslots. Although T-Update improves the update efficiency, the bandwidths among data nodes and parity nodes still remain unexploited fully due to that the links among the nodes with a deeper position in the update tree remain unutilized more time. Although tree update reduces the number of transferred data-delta chunks in the data node to 2, the data node still need send more than one data-delta chunk.

Besides, to tolerate rack failures, DSSs often distribute the chunks in a stripe to distinct racks [10] [29] [36]. It is inevitable to generate cross-rack traffic for updating the parity chunks. However, the network bandwidth in modern DSSs is shared among many other application workloads [28]. Meanwhile, the cross-rack links of modern DSSs are oversubscribed [40]. So, bandwidth, especially for the cross-rack (cross-region) bandwidth, left for update tasks is limited in DSSs. Therefore, utilizing the limited bandwidth fully is critical for improving the update efficiency.

These motivate us to design a new update scheme that fully utilizes available cross-rack bandwidth resource to reduce update time. In this paper, we design the *CPU* to parallel the update process for erasure coding, which pipelines the update data in slices. Figure 2(c) shows the update with *CPU* in the path ($DR_0 \; ! \; PN_1 \; ! \; PN_2 \; ! \; PN_3 \; ! \; PN_4$). To be simple, we slice the data-delta chunks into three slices. Through this method, an update only takes *7.3s*. The update time can be further reduced by chopping the data-delta chunks into more slices. Ideally, *CPU* can reduce the update time to only $1 + \frac{R \quad 1}{g}$ timeslots. However, more slices mean more overheads. With different update path, the time for an update in *CPU* varies (e.g., *10.3s* in the path $DR_0 \; ! \; PN_2 \; ! \; PN_3 \; ! \; PN_4 \; ! \; PN_1$). Thus, slice size selection and update path selection are the key issues for *CPU* to obtain better performance. Besides, only one data-delta chunk is transferred between racks, which achieves a good traffic balance among updated racks.

4 The design of CPU

In this section, we first introduce the design of *CPU*, and demonstrate how *CPU* improves the update efficiency. Then, we formulate the optimization problem based on the pipelining update model, and propose a heuristic algorithm to solve the problem based on the features of DSSs and our implementations. The notations used in this paper are illustrated in Table I.

4.1 Pipelining update model

CPU decomposes a data-delta chunk into a set of slices S_1, S_2, ..., S_s, and pipelines the slices along a selected path

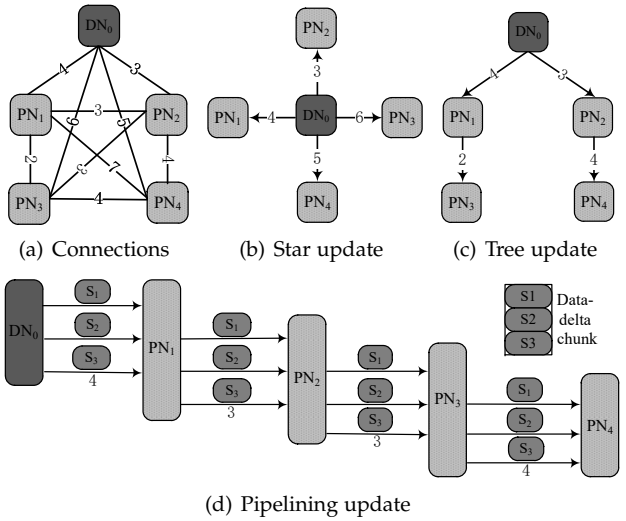


Fig. 2. Example for star update, tree update, and pipelining update with a data node DN_0 and four related parity nodes (PN_1; PN_2; PN_3; PN_4). The update time for each scheme is *18s*, *11s*, *7.3s*, respectively.

TABLE 1
Notations of the parameters

Notation	Representation
(n, k)	k is the number of data chunks, and n is the total number of chunks in a stripe
l	Chunk size
c	Slice size
g	Number of slices
m	Number of racks within a stripe
R	Number of parity racks within a stripe
d_i	Original data chunks
d_i^0	Updated data chunks
p_j	Original parity chunks
p_j^0	Updated parity chunks
b_{ij}	Bandwidth between rack i and rack j
T`	Total transmission time
$_{ij}$	Coefficient for p_j from d_i
DR_0	The source rack with a data node to be updated in the update path
PR_r	The r-th parity racks with related parity nodes to be updated in the update path

$DR_0 ! PR_1 ! ::: ! PR_{n\ k}$. To speed up the update process, we take racks as the receiving unit in the path instead of storage nodes due to cross-rack transmission consuming more time than inner-rack transmission. To further improve the update efficiency, *CPU* adopts tabu search based path selection algorithm to find a good update path because the links in a practical DSS own different bandwidth [4] [15]. The update process is triggered by a data node located in a data rack, where the original data chunk d_i is modified to d_i^0. The update data is delta-data chunks ($d_i \quad d_i^0$) calculated and sent by the data node. In each parity rack, the parity node receiving slices from the other rack, called *gateway node*, first forwards the slices to the next rack in the update path, then sends the slices to other related parity nodes within the same rack. And, each parity node does the read and write operations in chunk level, but encoding in slice level, which are parallel with the update data transmission via multi-thread. To sum up, the performance of *CPU* mainly comes from the following three features: 1) *CPU* achieves a good balance in bandwidth usage (e.g., each link in the path only need to transfer one data-delta chunk for an update), which mitigates the bottleneck caused by the link with the intensive update traffic; 2) *CPU* parallels transmission and encoding of the update data by pipelining the update data in slice level; 3) *CPU* further shortens the update time through selecting a good update path.

4.2 Problem analysis

Let T_{upt} be the time to finish an update, which includes network transmission time (T_{trans}), computation and disk I/O (T_{compu_io}), and *protocol overhead* (T_{pro_head}). Protocol overhead mainly comes from TCP connection establishment.

$$T_{upt} = T_{trans} + T_{compu\_io} + T_{pro\_head} \tag{3}$$

In general, the overhead of computation and disk I/O is less than the time of network transmission and can be executed in parallel with network transmission via multi-thread [28], which is implemented in *CPU* (§5). Thus, we can neglect the overhead coming from computation and disk I/O. The overhead of slicing comes from additional traffic (traffic overhead) and protocol overhead. Specifically, more additional traffic coming from the header of a slice (e.g., chunk ID, slice ID, etc.) as the slice size decreases. Besides, more slices will issue more TCP connection so as to increase the protocol overhead. Through keeping the TCP connection alive while transmitting slices from a same delta-data chunk, the protocol overhead can be mitigated, which is also implemented in §5.

Therefore, the key issue of *CPU* is to minimize the the network transmission time by choosing the slice size and update path carefully.

4.3 Problem formulation

Assume parity chunks within a stripe are allocated to $R \quad m$ distinct racks. When a data chunk d_i gets modified, the related parity racks fR$_1$, R$_2$, ..., R$_R$$g$ need to receive data-delta chunks from the data rack to keep data consistency. The chunk size and slice size are denoted as l and c respectively. Thus, the slice number of a chunk is $g = dl/ce$. Figure 3 shows the timeline of the pipelining update. PR$_r$ means the *r-th* parity rack in the update path for a stripe, which could be any parity rack. For different stripes, the set and number of parity racks may be different due to a random distribution of data chunks and parity chunks.

The network transmission of an update is finished on the condition that the last parity rack in the path receives all the slices. Thus, the network transmission time for a pipelining update under path ` is defined as (4):

$$T_{trans}(`) = \sum_{r=1}^{R\backslash 1} t_r + T_R \tag{4}$$

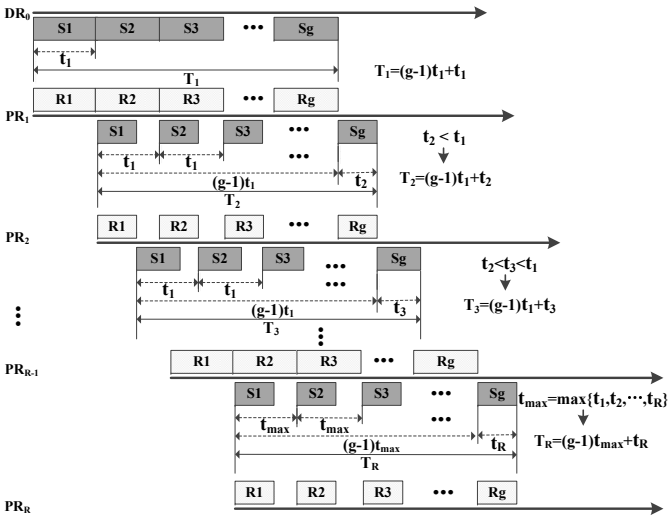


Fig. 3. Timeline of pipelining update. The slice receiving interval of r-th parity rack is determined by the worst link of the present r links in the pipelining line. For example, the slice receiving interval of third parity rack is the time for the link between DR_0 and PR_1 to send a slice since the link is the worst of the present three links in the pipelining line.

T_r means the duration for *r-th* parity racks (PR_r) to receive all the slices of a chunk from the prior rack (PR_{r-1}). t_r denotes the time for the prior rack to send a slice to the r-th rack in the path.

In pipelining update, each parity rack can only forward the slice to the next parity rack when it receives the whole slice from the prior rack. So, the receiving interval of slices for the last parity rack is the time for the worst link in the path to transmit a slice, denoted as t_{max}. Then, the duration from the first slice sent by the PR_{R-1} to the last slice received by the PR_R can be expressed as follows:

$$T_R = (g-1)t_{max} + t_R; t_{max} = max\{t_1; t_2; :::; t_R\} \quad (5)$$

From the equations (4)(5), we can draw equation (6):

$$T_{trans}(`) = (g-1)\ t_{max} + \sum_{r=1}^{R} t_r \quad (6)$$

Recall that:

$$g = \lceil l/c \rceil \quad (7)$$

$$t_r = \frac{+c}{b_r}; \forall r \in [1; R] \quad (8)$$

b_r means the bandwidth between the *r-th* parity rack and related prior rack. denotes the header size of a slice.

Substitute (7), (8) in (6):

$$T_{trans}(`; c) = (\lceil \frac{l}{c} \rceil - 1)\ \frac{+c}{b_{min}} + \sum_{r=1}^{R} \frac{+c}{b_r}; \quad (9)$$
$$b_{min} = min\{b_1; b_2; :::; b_R\}$$

Therefore, the optimization problem is formulated as follow:

$$P: \min_{(`;c)} T_{trans}(`; c) \quad (10a)$$

$$s.t.: c = aw; a \in \{1; 2; :::; l/w\}; \quad (10b)$$
$$` \in L: \quad (10c)$$

The constraints in the formulation above can be explained as follows: constraint (10b) implies that the slice size must be a positive integral multiple of the *word* size to achieve in-time computation in slice level, and cannot exceed the size of a chunk. In (10c), ` means a pipelining update path, which must contain all the related parity racks and starts from the updated data chunks. The set of all path is denoted as L, the size of which is $R!$.

4.4 Problem complexity

Proposition: The problem P is a NP-hard problem.

Proof: Let $V = \{v_0; v_1; :::; v_R\}$ be a set of racks, v_0 denotes the data rack, $\{v_1; :::; v_R\}$ denote the parity racks within the same stripe as the data rack, $A = \{(i; j) : i; j \in V; i \neq j\}$ be the edge set, d_{ij} be the link weight (the time to transmit a slice from rack i to rack j) associated with edge $(i; j) \in A$, and $d_{i0} = 0$ due to a transmitted slice need not return the data rack. Thus, the problem is to find a path in the asymmetric graph $G = (V; A)$ that visits each rack once to minimize the total transmission time, which can be reduced to an asymmetric traveling salesman problem (aTSP) when we set $c = l$. Note that the aTSP is a well-known NP-complete problem. Consequently, the decision problem of P is NP-hard.

A naive approach for the problem is to perform a brute-force search on all possible candidate paths under all possible slice size. However, there is a total of $R!$ permutations for each slice size, and the brute-force search becomes computationally expensive even for moderate sizes of R because of large selection set for slice size. Since the link weights vary over time, the path selection and slice selection should be done quickly on-the-fly based the measured link weights.

4.5 Heuristic algorithm

In high level, the heuristic algorithm works by determining the update path first, and then calculating the slice size based on the selected update path. We begin with the features of today's DSS and *CPU*, which are the basis of the heuristic algorithm.

Big chunk: DSSs encapsulate the data into large chunk to reduce the metadata size. For HDFS, the default chunk size for Hadoop.1 and Hadoop.2 is 64MB and 128MB, respectively. Moreover, For QFS, 64MB is the default chunk size.

Small overhead: In the implementation of *CPU*, the overhead to transfer a slice is no more than 200 bytes.

Assume that the traffic overhead introduced by slicing is negligible compared with slice size when slice size is larger than c_{low}. Thus, we can rewrite equation (6) as follow:

$$T_{trans}(`; g) = T^0_{max} + \frac{1}{g}(\sum_{r=1}^{R} T^0_r - T^0_{max}); g \in [1; \lceil \frac{l}{c_{low}} \rceil] \quad (11)$$

T^0_r means the time for r-th parity rack in the path ` to receive a delta-data chunk from the prior rack. $T^0_{max}$ means the time for the worst link in the path ` to transfer a delta-data chunk. Thus, before the slice size reaches the c_{low}, more slices mean less update time. Due to *Big chunk* and *Small overhead*, in order to exploit enough parallelisms,

the number of slice is greater than the number of parity chunks in practical DSSs before the slice size toughing c_{low}. For instance, splitting a 64MB chunk into over 2000 slices only introduces 6‰ traffic overhead. Thus, based on equation (11), we can draw that the performance of update pipelining is mainly bottlenecked by the link with the minimum available bandwidth in the update path. The path selection problem can be converted to a max-min problem. Specifically, we should find a path consisting of $R+1$ racks with a fixed start point (data rack) that owns the maximum-minimum link bandwidth compared with all possible paths.

Algorithm 1: Tabu Search based Path Selection

Input: The data rack ID D with updated chunks; The set of related parity racks $\aleph$; The set of link weights between each rack W;
Output: The optimal path $\grave{}$;

1 **Function** TSPS($\aleph; D; W$)
2 Initialization: generate a starting current update path $\grave{}$ randomly, tabu table $tabu[]$, $w^{*} = w(\grave{}), \grave{}^{*} = \grave{}$.
3 **for** $iteration < MaxItera$ **do**
4 Generate the set $N(\grave{})$ of K neighbors of current update path through *swap* operations, and calculate the maximum link weight for each neighbor to annotate every neighbor;
5 Sort the elements in $N(\grave{})$ in ascending order based the corresponding annotation.
6 **for** $i = 1$ *to* K **do**
7 $\grave{}' = N_i(\grave{})$;
8 **if** $tabu[\grave{}'] == 0$ **then**
9 **if** $w(\grave{}') < w(\grave{}^{*})$ **then**
10 Update $w^{*}, \grave{}^{*}$;
11 Update $tabu[]$;
12 $\grave{} = \grave{}'$;
13 break;
14 **if** $tabu[\grave{}'] > 0$ *and* $w(\grave{}') < w(\grave{}^{*})$ **then**
15 Update $w^{*}, \grave{}^{*}, tabu[]$;
16 $\grave{} = \grave{}'$;
17 breaks
18 return $\grave{}^{*}$;
19 **EndFunction**

Tabu search based path selection: To construct an update path, we should define a metric to identify the condition of links in the path. The two commonly used metric is the available bandwidth [4] [28] and network distance [24] (the number of hops between two racks). However, the network distance cannot always reflect the condition of each link due to the link-sharing among other applications. Thus, we associate a *weight*, the inverse of the link bandwidth, for each link between two racks, such that higher weight implies longer transmission time. The intuition here is that network conditions are reasonably stable on short timescales and usually do not change drastically during a short horizon [39], [40]. The weight can be obtained by periodic measurements on link utilization [40]. Since selecting a path with min-max link weight is time-consuming when the searching set is large, we propose a path search algorithm based on Tabu Search, called *TSPS*, which can escape the local optimum trap with acceptable complexity. Besides, the running time of *TSPS* is stable. The key elements of Tabu Search demonstrated as follows.

Evaluation function: The maximum link weight $w(\grave{})$ of the solution $\grave{}$.
Move operator: To generate a set of neighbors for the current solution (path) $\grave{}$, we use *swap* as the move operator. A $swap(i,j); i \neq 0; j \neq 0$ move means that exchange the i-th rack with the j-th rack in the current solution $\grave{}$. As such, the neighborhood $N(\grave{})$ of a solution $\grave{}$ includes all possible solutions that can be obtained by applying the *swap* operator to $\grave{}$, the size of which is $\frac{R(R-1)}{2}$ (Line 4).
Tabu table: To escape a local optimum trap, the table maintains a list of solution points that must be avoided, and updated based on tabu tenure. The tabu tenure defines the duration of the recorded solution.
Neighbor selection strategy: If the $\grave{}'$ is the optimal solution in the neighbors and not in the tabu table, this algorithm designates the solution obtained as the new current solution (Line 8-13). However, such a move, leading to a solution with a better result compared with the recorded best solution, is allowed even if it exists in the tabu table (Line 14-17). And, the new best solution is recorded if it improves on the previous best (Line 15).
Aspiration criterion: A move leading to promising solutions is allowed even if it is in the tabu table (Line 14).
Stop criterion: The tabu search stops if a specified number of iterations has elapsed in total (Line 3).

Slice size selection: We first relax the slice size and slice number to continuous value ϵ and l/ϵ. Thus, given the selected update path, finding the optimal slice size is a concave problem. Based on equation (9), we have

$$\frac{\partial T_{trans}(\grave{};\epsilon)}{\partial \epsilon} = \frac{l}{b_{min}\epsilon^2} - \frac{1}{b_{min}} + \sum_{r=1}^{R}\frac{1}{b_r}. \tag{12}$$

Then, the optimal slice size

$$\epsilon = \sqrt{\frac{l}{\sum_{r=1}^{R}\frac{b_{min}}{b_r} - 1}}. \tag{13}$$

Based on constraint (10b), the slice size must be a multiple of *word*. Thus, the practical slice size is obtained based on the equation (14).

$$c = \begin{cases} \left\lceil \frac{\epsilon^*}{w} \right\rceil w, & \epsilon \bmod w > \frac{w}{2} \\ \left\lfloor \frac{\epsilon^*}{w} \right\rfloor w, & \epsilon \bmod w < \frac{w}{2} \end{cases} \tag{14}$$

Simplification for slice size: To further accelerate the update process, a simple way to do slice size selection is further proposed for practical DSSs. When slice size is larger than c_{low}, the network transmission time decreases as the slice size decreases. However, for every update path, the optimal update time cannot be less than T^{0}_{max}. Then,

$$\frac{T_{trans}(\grave{};\gamma)}{T_{trans}(\grave{})} \leq 1 + \frac{R-1}{\gamma} \tag{15}$$

Thus, the network transmission time exceeds no more than $(R-1)/\gamma$ of the optimal network transmission time. To

save storage cost, the number of parity chunks in a stripe is less than 10 for most practical DSSs (e.g., by default, RS(6,3) for HDFS). Due to *Big Chunk* and *Small overhead*, the delta-data chunk can be divided into over thousands of slices without enrolling significant traffic overhead. Moreover, the gain from an additional parallel transmission decreases as the number of slices increase ($\frac{@T_{trans}}{@g}$ / $\frac{1}{g^2}$). Thus, there must be an *RoI* as illustrated in slice size analysis in §6, where the update time of *CPU* only has little deviation compared with the optimal update time. Although *RoI* is different for different update paths given by *TSPS*, we can adopt the *RoI* obtained under the worst case that all link bandwidths in the found update path equal to the bandwidth of the worst link as the default value, which bounds the worst performance of *CPU*. As shown in equation (12), although the *RoI* is obtained under the worst case, the performance of *CPU* can be still promised because the number of slices is greater than the number of parity racks. Therefore. instead of finding the optimal slice size, we can choose a slice size in the *RoI* as the default size with little degradation on the performance of *CPU*. The *RoI* can be obtained based on the deviation value we set, the number of parity racks, and chunk size before deploying *CPU* on practical DSSs, which is one time running.

4.6 Discussion

Reliability: *CPU* keeps reliability for following reasons. Before an update process, the path selection algorithm can essentially exclude the links with high loss possibility and failure links, because the algorithm tries to find a path with maximum minimum link. During an update process, although the selected path exists packet loss, the lost packets can also be successfully transmitted due to the retransmission scheme of TCP. In the worst case that a link failure happens during an update process, *CPU* can solve it through the combination of *ACK* messages from the parity nodes and *update timer timeout (UTT)*. Receiving an *ACK* message from a parity node means the parity node has finished the parity chunk update. After an *UTT*, if there is no *ACK* from parity nodes, *CPU* inserts the selected update path into the tabu table and reruns *TSPS* to get a new path to do update. Moreover, if the data node receives several *ACKs* but not all after an *UTT*, *CPU* just re-transmits the missing parity chunks to the corresponding parity nodes.

Computation Overhead: The computation overhead introduced by *pipelining* mainly comes from the disk I/O and pipelining keeping. For star update and tree update, the read-write operation is done in chunk level, and communication session between two storage servers is terminated once a delta-data chunk is received. However, as stated in §5, *CPU* reduces the disk I/O to one read-write for an update on each parity node by buffering the received slices and writing them into local disks only when receiving all slices of a delta-data chunk. Therefore, the disk I/O overhead in *CPU* is equivalent to that in star update and tree update. Moreover, each storage server in an update path only keeps the TCP connection alive before the next storage server receiving all the slices of a delta-data chunk. Even though *CPU* keeps the communication session a little longer due to the traffic overhead, it is negligible when the slice size locates in the *RoI*.

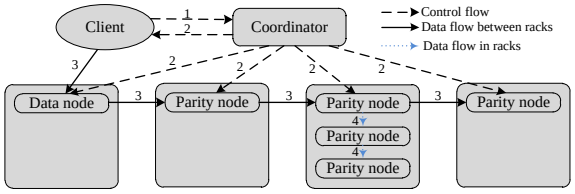


Fig. 4. CPU architecture.

Resource Competition: Pipelining update will consume the bandwidth resources between parity racks, which may slow down the data transmission of other applications running there. However, the goal of *CPU* is trying to fully utilize the available network resource to speed up the update process. Besides, the path selection algorithm will filter the busy links occupied by other applications, which extremely mitigates the influence on ongoing traffic.

5 Implementation

System architecture: We implemented a prototype of *CPU* to realize pipelining update. Figure 4 shows the architecture of *CPU*, which compromises of a client, a coordinator, and multiple storage nodes located in distinct racks. The coordinator manages the metadata information of every stored chunk, such as the chunk ID, the stripe that the chunk belongs to, the data node where the chunk is stored, and the ID of related parity node. It also collects the bandwidth between racks, and schedules the update procedure. When an update is triggered by the client, it first sends a request to the coordinator (step 1). Then, the coordinator uses the updated chunk ID to identify the locations of (n k) parity chunks within the stripe, and feeds back the control message to the client and the leading storage node selected randomly of each related rack for scheduling the update data flows (step2). Once the leading storage node receives a slice from other parity rack, it first forwards the received slice to the next parity rack in the main pipelining path if it is not the end of the pipelining (step3). Then, the slice is pipelined to the parity nodes in the same rack if there are more than one parity node (step4). Due to the abundant bandwidth in a rack, the pipelining path in a rack is also randomly selected.

Implementation details: The *CPU* prototype is implemented in C on Linux. We achieve the erasure coding operations with the Jerasure Library v1.2 [51]. Instead of establishing a TCP connection for each slice, every two adjacent racks in the update path keep the connection alive before receiving all slices of a data-delta chunk to eliminate the overhead from connection establishment. To speed up the performance, we parallel computation and disk I/O with slice transmission via multiple threads. To reduce the computation loads on storage servers caused by disk I/O, a storage server executes coding operations in slice level but disk I/O in chunk level. Specifically, once receiving a slice, *CPU* computes and overwrites the slice with the corresponding buffered parity chunk reading from local file system immediately, but overwrites the buffered parity chunk into local file system only when all the slices for the parity chunk are received. Since there may be a new update (identified by the chunk ID in slices) arrives before the current updates on the parity chunk complete, *CPU* uses a flag to indicate whether all the on-going updates on a

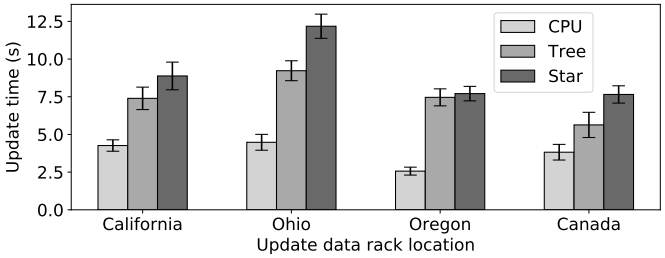


Fig. 5. Overall performance of *CPU*.

TABLE 2
Measured bandwidth among regions (Mb/s)

Bandwidth	California	Ohio	Oregon	Canada
California	998.2	183.0	259.2	47.15
Ohio	91.55	999.0	72.1	153.0
Oregon	201.0	69.3	997.8	78.9
Canada	69.5	162.5	71.9	998.4

parity chunk are finished or not, and does write operations based on the flag. We also implement star update and tree update in the same environments to evaluate *CPU*.

6 Evaluation

6.1 Experiment Setup

To demonstrate the performance of *CPU*, we choose tree update and star update as baselines. The two baselines are also implemented into rack-aware update architecture to compare them equivalently. We design two test scenarios (homogeneous environment and heterogeneous environment) and deploy them on Amazon EC2. All the instances are t2.micro type with 2.5GHz 1vCPU, 1GB RAM, 1Gb/s bandwidth and run 64-bit Ubuntu 16.04 LTS.

Heterogeneous environment: To illustrate how *CPU* performs in a heterogeneous environment and the impact of update path selection on *CPU*, we evaluate *CPU* in the heterogeneous network environment by creating a set of Amazon instances across different regions, namely California, Ohio, Oregon, and Canada. During running our experiments, we periodically obtain the inter-regions and cross-region bandwidth across four regions using *iperf*. One of the measured results is presented in Table 2, in which each number is the measured bandwidth (in Mb/s) from the row region to the column region. It shows that cross-region bandwidth is much more scarce than inner-region bandwidth. We deploy RS(12,9) and create three different instances to store three chunks of each stripe in each region. Additional two instances are created as the coordinator and client. Since the data transmission between different regions is slow, we set chunk size as 32MB to save the experiment time.

Homogeneous environment: For different available bandwidth distributions among the storage nodes, the performance of the three update schemes may vary and be hard to evaluate. To be fair, we further compare *CPU* with the two baselines in the homogeneous network environment under different parameter settings. The instances of the scenario are created in the same region (e.g., Ohio-2b in North America). The bandwidths among instances in private networks

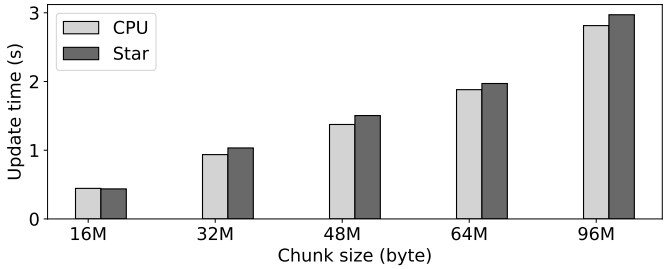


Fig. 6. Gain from pipelining.

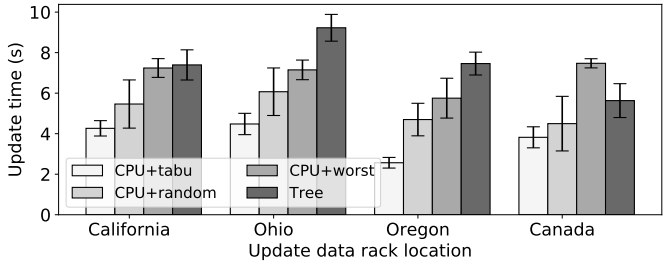


Fig. 7. The impact of path selection.

are stable. Thus, we transfer the update data using private network address of instances, which make the traffic go through the private network instead of the public Internet.

Evaluation metrics: Update time denotes the duration from the start of the update to the end. The less update time, the faster the update is. All the update times of experiments are average results over 10 runs.

6.2 Overall Performance

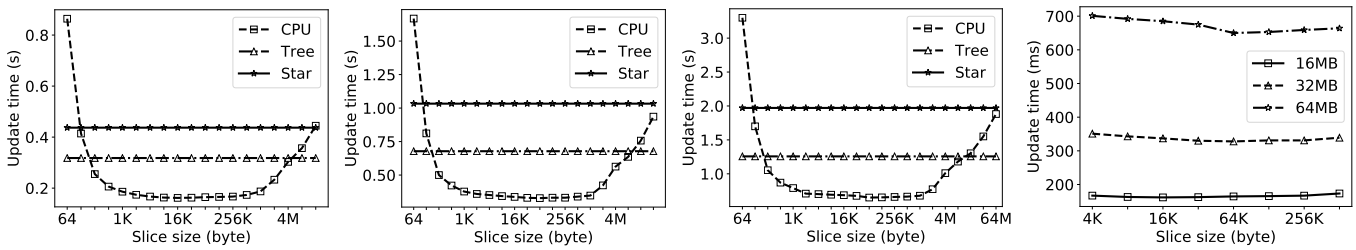
Figure 5 shows the update time and the standard deviations of *CPU* compared with the two baselines in the heterogeneous scenario. Since the bandwidth distribution between data racks and dependent parity racks for updates triggered in distinct regions is different, the time for updates initiated in different regions varies. However, *CPU* achieves time-saving over star update and tree update in all regions where the updates are started. *CPU* reduces update time by 50.1%-66.7% and 32.2%-65.6% compared with star update and tree update, respectively.

6.3 Performance Gain Decomposition

This part demonstrates where the performance gain of *CPU* come from, including pipelining, parallelism, and path selection. The setting of the experiments in this part is three parity chunks distributed in three distinct parity racks.

Pipelining: We set the slice size to the chunk size for *CPU*, and compare the performance of *CPU* with star update in the homogeneous scenario to show the gain from pipelining. Figure 6 shows the update time of *CPU* and star update scheme with the variation of chunk size. Although there is no parallelism in transmission, *CPU* still performs better than star update, especially for large chunk size (e.g., larger than 32M). Compared with star update, pipelining update can release the burden on data racks, which will lag the update process when the resource of data racks is up to the limit.

Parallelism-slice size: *RoI* is obtained in the homogeneous scenario because *CPU* profiles the *RoI* under the worst



(a) 16MB chunk (b) 32MB chunk (c) 64MB chunk (d) Rang of interesting

Fig. 8. The update time comparison between *CPU*, Tree update, and Star update with the variation of slice size.

case. Figure 8(a-c) shows the parity chunks update time versus the slice size for *CPU* with chunk size 16MB, 32MB, and 64MB, respectively. When slice size is larger than 512KB, the average update time of *CPU* for all chunk sizes decreases significantly as the slice size decreases. Before the slice size exceeds 512KB, the performance of *CPU* remains almost the same as the slice size decreases. Until the slice size reaches 4KB, the average update of *CPU* increases gradually as the slice size decreases. The phenomenon results from the two reason:1) the incremental gain *CPU* from an additional parallel transmission becomes smaller as the number of slice increases; 2) the additional traffic introduced by slicing becomes significant when slice size is less than a certain value (e.g., 4KB), but remains neglectable when slice size is larger than that value. The overlapped RoI (4KB 512KB) for the different chunks (16MB, 32MB, and 64MB) is showed in Figure 8(d), in which *CPU* achieves almost the same performance. Although the RoI for the DSSs with different settings (chunk size, number of parity racks) varies, we can obtain the RoIs for different settings by one time running before deploying *CPU* on piratical DSSs. Thus, we can select a slice size from this range as the default slice size with little degradation on the performance of *CPU*. We choose 64KB, in the following experiments, as the default slice size, with which *CPU* can reduce the average update time by 67% and 48.2% for 64MB chunk size compared with star update and tree update schemes, respectively.

Path selection: To illustrate how a good update path improves the performance of *CPU*, the experiment is done in the heterogeneous scenario. Figure 7 shows the average update time and standard deviations of pipelining update with different path selection algorithms. We consider selecting update path randomly (labeled as *CPU+random*), and selecting update path with *tabu* algorithm (labeled as *CPU+tabu*, and used in previous experiments). We also show the worst case of pipelining update (labeled as *CPU+worst*), and tree-update for comparison. The results of star update are not shown, whose update time is worse than that of tree-update and very large. *CPU+tabu* can reduce the update time by 15.0%-45.4% compared with *PU+random*. Meanwhile, the update time of *CPU+random* suffers from a larger standard deviation. Besides, for some worst cases, the performance of pipelining update is even worse than tree update (e.g., update process started in Canada). Thus, selecting an optimal or near-optimal update path can improve the performance of *CPU* significantly. Note that path selection with the proposed algorithm can be done less than 3.5ms (§6-E), which is neglectable compared with the update time in our evaluation.

6.4 Different Parameters

To further demonstrate the efficiency of *CPU*, we further conduct some experiments with different parameter settings, like the number of parity racks, chunk size, and multiple update requests.

Number of parity racks: The experiment is done with 64MB chunk size. Figure 9 shows the average update time versus the number of parity racks. The average update time of star update and tree-update both increases with the increase of parity racks since more updated parity racks involve larger transmitted update traffic. However, since slicing can parallel the transmission of update data, the average update time of *CPU* remains almost unchanged as the number of parity changes, which approximately equals to the time of transmitting a delta-data chunk (64MB here). As the number of parity racks increases from 1 to 4, *CPU* reduces the update time by 6.0%-75.7% compared with star update, and 6.0%-66.2% compared with tree update.

Size of Chunk: The experiment is done with three parity racks. Figure 10 shows the average update time versus the chunk size. The average update time for all three update schemes increases as chunk size increase since larger chunk size means more update data to be transmitted. However, *CPU* performs better than star update and tree update under all chunk size because it can partition a chunk into slices for better bandwidth utilization. Compared with star update and tree update, *CPU* reduces the average update time by no less than 62.3% and 48.2%, respectively.

Multiple updates: There are two cases of multiple updates: multiple updates within a stripe, and multiple updates in different stripes. The two types of experiments are conducted under 32MB chunk size and three parity racks. Figure 11 shows the update time versus the number of data nodes to be updated within a stripe concurrently. With more data nodes participating in the update, the average update time increases for all three update schemes, since all the updated data nodes correspond to the same three parity racks. Thus, more updated data nodes within a stripe not only bring more update traffic but also increase the resource competition on the parity racks, including network bandwidth and computation resource. Figure 12 shows the average update time versus the number of stripes to be updated concurrently. With more stripes participating in the update, the average update time increases for all three update schemes, since more stripes bring more storage nodes to be updated and more update traffic, increasing

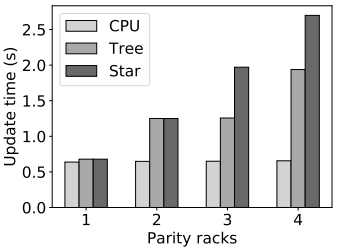


Fig. 9. Number of parity racks.

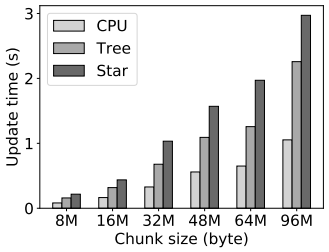


Fig. 10. Chunk size.

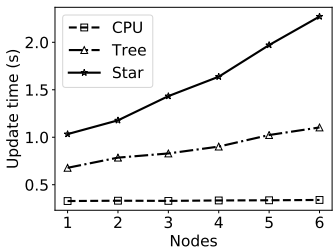


Fig. 11. Updated nodes.

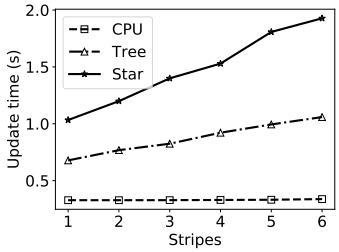


Fig. 12. Updated stripes.

the resource competition, especially the network bandwidth competition. Nevertheless, *CPU* outperforms the two baselines and keeps almost same performance as the number of updates increases, which benefits from the efficient utilization of bandwidths among racks and good update traffic balance among racks (only one data-delta chunk transferred between two racks).

TABLE 3
Performance of TSPS with different locations.

Location	OPT(s)	*TSPS* of *CPU*			
		Best(s)	AVG(s)	Gap	PCT of best
1	14.55	14.55	15.08	3.6%	34%
2	14.55	14.55	15.09	3.7%	36%
3	15.17	15.17	15.86	4.5%	22%
4	15.17	15.17	15.94	5.1%	32%
5	14.99	14.99	15.27	1.9%	65%
6	14.55	14.55	15.38	5.7%	38%
7	14.55	14.55	15.54	6.8%	22%
8	14.71	14.71	15.69	6.7%	36%
9	14.55	14.55	15.31	5.2%	21%
10	14.55	14.55	15.22	4.6%	26%
11	14.55	14.55	15.22	4.6%	32%

6.5 Efficiency of TSPS

Since the number of parity chunks of a stripe in most practical DSSs is less than 10, we evaluate the time for *TSPS* to find an update path consisting of 11 storage nodes on the Amazon EC2 t2.micro instance. We initiate 11 t2.micro instances in 11 distinct regions on Amazon EC2 and obtain the bandwidth distribution among these instances by *iperf*. Table 3 shows the performance of *TSPS* over 1000 runs with the variation of location where an update is triggered. The numbers in the table denote the maximum link weight (the time to send 1Gb data) of the found path. Although the brute-force search can find the optimal update path, it takes over 350ms. However, 1) *TSPS* reduces the search time to less than 3.5ms under 100 iterations only with less than 7% deviation from the optimal solution for all locations, and 2) over 20% solutions found by *TSPS* hit the optimal solution.

7 Related Work

Erasure codes: There are many kinds of erasure codes applied in storage systems, e.g., RS codes [42], fountain codes (also known as rateless erasure codes [43], [44], [45]), regenerating codes (also called network coding [48], [49] and etc.. To improve reliability and reduce storage overhead, most today's storage systems adopt erasure codes to store the data, including RS codes [33], [34], [35] and fountain codes [46]. However, the most commonly used erasure codes for storage are RS codes. Thus, in this paper, we mainly target at RS code based storage systems. However, coming with the benefit of coding, it suffers from the high overhead in repair bandwidth. Thus, the regenerating codes are proposed to reduce the repair bandwidth significantly [11], [47]. With these erasure codes, erasure-coded storage systems have been practical and popular.

Update in erasure codes: Researches of the update on erasure coding have attracted many attentions. Based on these works, we mainly classify the update methods into two categories: reducing update traffic, and refining update transmission structure.

To reduce update traffic, existing works mainly focus on proposing new parity update scheme, designing new update-friendly codes, and grouping update. A class of parity updates [52], [53], [54], called the delta-based approaches, eliminate redundant network traffic by only transferring a parity delta which is of the same size as the modified data range, on which most update schemes are built. Designing a new class of codes or refining existed erasure coding scheme to improve the update-efficiency is another encouraging method [20], [21], [22], [23]. Mehrabi etc. [20] propose a class of locally repairable codes (LRCs) with small update complexity, which can achieve the lower bound on the update complexity associated with one chunk update. Ankit etc. [21] establish the existence of the codes which require only logarithmic updates when data changes. TIP-code [22], based on XOR code, uses three independent parities to offer optimal update complexity.

Since there will be multiple updates within a stripe, the update traffic can be reduced by grouping, also called logged-based update, which selects a data node to accumulate and merge the delta-data chunks within a stripe, and send the merged delta-data chunk to related parity nodes after a given time [50] [52]. Although grouping update can reduce update traffic, it will result in parity chunks inconsistency, which may degrade the repair performance because the repair operations must be executed after the old parity chunks are updated with the merged delta-data chunk [24]. Thus, we focus on optimizing delta-based in-place update, which means all the data chunks and the related parity chunks are updated at the same time, to support the data access and data repair well.

Besides, scheduling the computation and transmission flow of the update data carefully could also improve the update efficiency of erasure coding significantly. Traditionally, the update process is performed on a star structure [16],

which means that the data node with an update computes the update data and transmits it to all the related parity nodes, respectively. The data node would be burdened with the data computation and transmission under intensive updates with star update. To mitigate the load on data nodes and exploit the idle links between parity nodes, T-Update [24] organizes update data to be transmitted along a tree, which reduces the update time by 27% compared with star update. However, T-update still cannot fully utilize the bandwidth between parity nodes. Pipelining technique is what we adopt to further improve the bandwidth utilization for update in erasure codes, which also is applied in other situations, e.g., repair in erasure codes [28], write in replication-based HDFS [55]. Beside the different applications of pipelinig, we also have some unique designs, which are slice size adaptation and path selection algorithm. However, both replication-based HDFS and repair pipelinig select fixed slice size and there is no path selection algorithm in [55]. Besides, the path selection algorithm in repair pipelining is based on brute force search, which eliminates the search of infeasible paths. Although this algorithm can find the optimal pipelining path, its performance highly depends on the bandwidth distribution, the worst case of which is unacceptable equaling the performance of brute force search.

8 Conclusion

In this paper, we propose a cross-rack-aware update mechanism for erasure-coded storage to improve the update efficiency of erasure coding, called *CPU*. It pipelines update data in smaller units along a path consisting of data rack (source) and related parity racks to alleviate the burden on the data rack and utilize the bandwidths between parity racks fully. The optimization problem based on the proposed pipelining update model is NP-hard. Based on *Big chunk* and *Small overhead*, a heuristic algorithm is proposed to solve it by determining the best update path first through solving a max-min problem and then deciding the slice size. *RoI* is further proposed to simply the slice selection. To evaluate the performance, we have implemented a *CPU* prototype and done a series of experiments on Amazon EC2. The results have demonstrated that *CPU* can improve the update efficiency of erasure coding significantly compared with star update and tree update schemes, especially for more parity racks, which can reduce the update time to almost the time of transmitting one delta-data chunk.

Acknowledgments

This work was done when Haiqiao Wu visited the Department of Computer Science and Engineering, University of California, Merced, with a scholarship from the China Scholarship Council. The corresponding author is Peng Gong.

References

[1] J. Tang, X. Tang, and J. Yuan. Traffic-Optimized Data Placement for Social Media. *IEEE Transactions on Multimedia*, 20(4): 1008-1023, 2018.

[2] Z. Shen, W. Du, X. Zhao and J. Zou. DMM: fast map matching for cellular data. In *Proc. of MobiCom*, 2020.

[3] J. George, C. Chen, R. Stoleru, and G. G. Xie. Hadoop MapReduce for Mobile Clouds. *IEEE Transactions on Cloud Computing*, 7(1):224-236, 2019.

[4] J. Li, S. Yang, X. Wang, and B. Li. Tree-structured data regeneration in distributed storage systems with regenerating codes. In *Proc. of IEEE INFOCOM*, 2010.

[5] E. Pinherio, W. Weber, and L. Barroso. Failure trends in a large disk drive population. In *Proc. of USENIX FAST*, 2007.

[6] B. Schroeder, G. Gibson. Disk failures in the real world: What does an MTTF of 1000000 hours mean to you? In *Proc. of USENIX FAST*, 2007.

[7] R. Potharaju, N. Jain. An empirical analysis of intra- and inter-datacenter network failures for geo-distributed services. In *Proc. of ACM SIGMETRICS*, 2014.

[8] M. Abebe, K. Daudjee, B. Glasbergen, and Y. Tian. EC-Store: Bridging the Gap Between Storage and Latency in Distributed Erasure Coded Systems. In *Proc. of ICDCS*, 2018.

[9] A. Mazumdar, V. Chandar, and G. W. Wornell. Update-efficiency and local repairability limits for capacity approaching codes. *IEEE Journal on Selected Areas in Communications*, 32(5):976-988, 2014.

[10] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. Xoring elephants: Novel erasure codes for big data. *Proceedings of the VLDB Endowment*, 6(5):325-336, 2013.

[11] D. S. Papailiopoulos, J. Luo, A. G. Dimakis, C. Huang, and J. Li. Simple Regenerating Codes: Network Coding for Cloud Storage. In *Proc. of IEEE INFOCOM*, 2012.

[12] J. Li, X. Wang, and B. Li. Cooperative Pipelined Regeneration in Distributed Storage Systems. In *Proc. of IEEE INFOCOM*, 2013.

[13] J. Li and B. Li. Demand-aware Erasure Coding for Distributed Storage Systems. *IEEE Transactions on Cloud Computing*, 2018.

[14] J. Darrous, S. Ibrahim, and C. Perez. Is it Time to Revisit Erasure Coding in Data-Intensive Clusters? In *Proc. of MASCOTS*, 2019.

[15] T. Ernvall, S. El Rouayheb, C. Hollanti, and H. Vincent Poor. Capacity and Security of Heterogeneous Distributed Storage Systems. *IEEE Journal on Selected Areas in Communications*, 31(12): 2701-2709, 2013.

[16] F. Zhang, Jianzhong Huang, and C. Xie. Two efficient partial-updating schemes for erasure-coded storage clusters. In *Proc. of IEEE NAS*, 2012.

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

[18] I. Adams, M. Storer, and E. Miller. Analysis of workload behavior in scientific and historical long-term data repositories. *ACM Transactions on Storage*, 8(2), 2012.

[19] D. Narayannan, A. Donnelly, and A. Rowstron. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage*, 4(3), 2008.

[20] M. Mehrabi, M. Shahabinejad, M. Ardakani, and M. Khabbazian. Improving the Update Complexity of Locally Repairable Codes. *IEEE Transactions on Communications*, 66(9):3711-3720, 2018.

[21] A. Singh Rawat, S. Vishwanath, A. Bhowmick, and E. Soljanin. Update Efficient Codes for Distributed Storage. In *Proc. of IEEE ISIT*, 2011.

[22] Y. Zhang, C. Wu, J. Li, and M. Guo. TIP-code: A Three Independent Parity Code to Tolerate Triple Disk Failures with Optimal Update Complexity. In *Proc. of IEEE DSN*, 2015.

[23] A. Mazumdar, V. Chandar, G. W. Wornell. Update-Efficiency and Local Repairability Limits for Capacity Approaching Codes. *IEEE Journal on Selected Areas in Communications*, 32(5):975-988, 2014.

[24] X. Pei, Y. Wang, X. Ma, F. Xu. T-Update: A Tree-Structured Update Scheme with Top-Down Transmission in Erasure-Coded Systems. In *Proc. of IEEE INFOCOM*, 2016.

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

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

[27] A. Vulimiri, C. Curino, P. B. Godfrey, T. Jungblut, J. Padhye, and G. Varghese. Global analytics in the face of bandwidth and regulatory constraints. In *Proc. of USENIX NSDI*, 2015.

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

[29] K.V. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A "Hitchhiker's" Guide to Fast and Efficient Data