XORInc: Optimizing Data Repair and Update for Erasure-Coded Systems with XOR-Based In-Network Computation

Fang Wang*, Yingjie Tang*, Yanwen Xie*, Xuehai Tang†
*Wuhan National Laboratory for Optoelectronics, School of Computer Science and Technology
Huazhong University of Science and Technology, Wuhan, China
*Key Laboratory of Information Storage System, Engineering Research Center of Data Storage Systems and Technology
Ministry of Education of China, Wuhan, China
†Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
wangfang@hust.edu.cn, *tangyingjie@hust.edu.cn*, *ywxie@hust.edu.cn*, *tangxuehai@iie.ac.cn*

***Abstract*—*Erasure coding* is widely used in the distributed storage systems due to its significant storage efficiency compared with *replication* at the same fault tolerance level. However, erasure coding introduces high cross-rack traffic since (1) repairing a single failed *data block* needs to read other available blocks from multiple nodes and (2) updating a data block triggers parity updates for all *parity blocks*. In order to alleviate the impact of these traffic on the performance of erasure coding, many works concentrate on designing new transmission schemes to increase bandwidth utilization among multiple storage nodes but they don't actually reduce network traffic.**

With the emergence of programmable network devices, the concept of in-network computation has been proposed. The key idea is to offload compute operations onto intermediate network devices. Inspired by this idea, we propose *XORInc*, a framework that utilizes programmable network devices to XOR data flows from multiple storage nodes so that XORInc can effectively reduce network traffic (especially the cross-rack traffic) and eliminate network bottleneck. Under XORInc, we design two new transmission schemes, *NetRepair* and *NetUpdate*, to optimize the repair and update operations, respectively. We implement XORInc based on HDFS-RAID and SDN to simulate an in-network computation framework. Experiments on a local testbed show that NetRepair reduces the repair time to almost the same as the normal read time and reduces the network traffic by up to 41%, meanwhile, NetUpdate reduces the update time and traffic by up to 74% and 30%, respectively.

***Index Terms*—erasure coding, in-network computation, repair, update**

I. Introduction

Modern distributed storage systems that manage large-scale data are constantly facing the risk of data loss, which can be caused by frequent hardware and software failures [1–4]. A common way to improve data reliability is data redundancy [5–8]. Replication, a traditional redundancy scheme, is a way that locates copies of each data block in different storage nodes, which can ensure high system reliability [1, 9]. When a data block is unavailable, the repair operation only needs to read the same size of data from another available copy. However, since data volume has been expanded to an incredible scale [1, 7, 9, 10], the dramatic storage cost of replication (normally three replicas) has motivated an increasing number of distributed storage systems to use erasure coding, due to its significantly lower storage overhead. For example, the classical Reed-Solomon (RS) codes [11] are deployed in Google ColossusFS [5], HDFS-RAID (also known as HDFS in Facebook) [6] and QFS [8]; the recent Local Reconstruction Codes (LRC) are deployed in Azure [7]. Generally, erasure coding takes a fixed number of data blocks as an input and then encodes them into a fixed number of parity blocks. The data blocks and the related parity blocks form a *stripe* and a sufficient number of available blocks can repair all lost blocks in the same stripe.

However, compared with replication, erasure coding introduces additional overhead during the repair and update operations. For repair operations (e.g. degraded read and reconstruction job), repairing an unavailable block requires reading multiple data and parity blocks within the same stripe, which results in a significant increase in network traffic. Moreover, these data flows from multiple storage nodes are aggregated at the downlink of a client or reconstruction node, causing a network bottleneck and increasing the repair time. Alongside this, the increasing traffic also has a bad effect on the performance of other applications in the system [3]. Besides repair operations, update operations also become more common in many real-world storage systems [12–15]. Whereas updating a data block requires updating all other parity blocks in remote nodes, as a result, introducing multiple network traffic and increasing the update time. On the other hand, the inconsistent time of data and parity blocks is determined by the completion time of the update operation. Once the system encounters node failures during the update operation, it will have a great influence on system reliability and even cause permanent data loss.

As a result, optimizing the repair and update operations of erasure coding have received significant attention [3, 7, 14, 16–29]. A general solution is to design new erasure codes that can reduce the amount of repair and update traffic [7, 16–21]. These codes trade off among system reliability, network

overhead, storage overhead, and computational complexity. Another choice is to design a fast transmission path that fully utilizes the network bandwidth among storage nodes, so as to alleviate the pressure of the bottleneck link [24–27]. For example, the state-of-the-art repair technique called *repair pipelining* [27] pipelines the repair of a block in small-size units across storage nodes. Compared with the conventional star-structured transmission path, repair pipelining presents a chain-structured repair scheme that each node sends data to the next in order. In this way, the bandwidth of each node is fully utilized and the repair time of a failed block is close to the read time of a normal block. Similarly, *T-Update* [24], a tree-structured update scheme with the top-down transmission is presented to improve update efficiency. However, these optimizations eliminate the network bottleneck merely by spreading traffic to other links, rather than reduce the network traffic in the system. On the other hand, in modern data centers, for better system reliability, blocks on the same stripe are usually distributed to multiple racks. Some studies [2, 7, 10, 16] even adopt *flat placement*, in which blocks are distributed across distinct racks to maximize the tolerance against rack failures. Therefore, the network traffic distributed to other links is often cross-rack traffic. Unfortunately, the cross-rack bandwidth is often oversubscribed and much more limited than the inner-rack bandwidth [30, 31]. So, how to improve the performance of erasure coding and practically reduce the network traffic at the same time becomes our major concern.

At present, the optimization of erasure coding basically can't avoid such end-to-end network transmission (from multiple nodes to one node or vice versa). So, their reduction of network traffic is very limited. The recent researches [32–34], which use programmable network devices to optimize other distributed applications, enlighten us to take the advantage of in-network computation to optimize erasure coding. In-network computation is driven by software-defined networking (SDN) [35] and the rapid development of programmable network devices (e.g. programmable switches [36, 37] and network accelerators [38, 39]). SDN emphasizes the separation of control plane and data plane of network devices. The network devices, which are required to be programmable to support user-defined computing functions, are managed by the centralized controller. In the repair and update operations, if the network device performs computation on transmitted data, it only needs to forward the intermediate results without transmitting a large amount of raw data, leading to less network traffic. Limited by the hardware resources of current switches, there are two different ways to implement in-network computation. (1) Only programmable switches are used to perform computing tasks with low computational complexity and low storage overhead [32]. (2) Combining programmable switches and network accelerators to handle more complex computing tasks [33, 34], where programmable switches are responsible for parsing application-specific header and customizing forwarding strategies, and network accelerators perform computations with a low-power multi-core processor.

In this paper, we present XORInc, an in-network computation framework that uses network devices to XOR the relevant data flows from multiple nodes and generate new packets to forward. In XORInc, the computing function that programmable network devices need to support is just XOR, which has a very small computational overhead. In addition, XORInc has fewer storage requirements for network devices. It only needs to temporarily buffer a small number of packets and the storage space can be reclaimed when the computation is completed. Therefore, only the programmable switch is needed to support XORInc. But considering overload in some scenarios, we can also adopt another implementation that combines the programmable switch and the network accelerator. In this way, the pressure of computation and storage is transferred to the network accelerators. In summary, we make the following contributions.

- We propose XORInc, a specific implementation of in-network computation that reduces network traffic by offloading the XOR operations from the end to the programmable switches.
- We implement a prototype of XORInc with HDFS-RAID [6] and SDN. Applying XORInc to the repair and update operations of erasure coding, we implement a new repair scheme NetRepair and a new update scheme NetUpdate.
- We design different scheduling strategies to select appropriate network devices to perform computing tasks for NetRepair and NetUpdate respectively, so as to avoid overloading some network devices and affecting their basic packet forwarding functions.
- We perform an evaluation by simulating a programmable switch. The results show that in the repair operation, NetRepair can achieve the optimal repair time of a lost data block, which is close to the read time of a normal block. Compared with the conventional repair scheme and repair pipelining, it reduces cross-rack traffic by up to 41%. In addition, in the update operation, NetUpdate also outperforms the conventional star-structured update schemes. It reduces the update time and cross-rack traffic by up to 74% and 30%, respectively.

II. Background and Motivation

A. Principle of Erasure Coding

Erasure coding typically has two configurable integer parameters n and k (where $k < n$). In an erasure-coded distributed storage system that manages large-scale datasets, a source file is divided into multiple fixed-size blocks, which are called *data blocks*. We consider erasure codes that are *systematic code* which means raw data remains after encoding. So, every k data blocks (denoted by $d_1, \cdots, d_k$) are encoded into n blocks (denoted by $d_1, \cdots, d_k, p_1, \cdots, p_r$, where $r = n - k$) which consist of k original data blocks and r parity blocks. These n blocks are grouped as a *stripe* and distributed to n different storage nodes to maximize system reliability. For erasure codes that are *maximum distance separable* (MDS), such as prevalent RS codes, they can tolerate up to r node failures since any k

blocks of the same stripe can be decoded to original k data blocks. Other common erasure codes, such as LRC codes, sacrifice some reliability in order to improve single fault repair performance. Compared with RS codes, LRC codes require more storage redundancy to achieve *r-fault* tolerance.

An erasure-coded storage system includes a plurality of independent stripes. For a stripe, any data block (denoted by d^*) in it can be represented by a linear combination of any other k blocks (denoted by $b_1, \cdots, b_k$) as (1), where α_i's $(1 \leq i \leq k)$ are the coefficients depending on the decoding matrix. According to (1), it can be known that repairing any lost data block requires acquiring data of other k available blocks for decoding. In addition, a parity block (denote by p_i) could be computed from a linear combination of k data blocks as (2), where $\beta_{i,j}$'s are the coefficient depending on encoding matrix. For the update operation, Equation (2) shows that any modification of the data blocks will result in an update of all the parity blocks. It is also worth noting that all additions and multiplications of the erasure coding are based on the Galois Field, where an addition is equivalent to XOR, and a multiplication is an operation with relatively large computational overhead. In the following description, we simply refer to addition as XOR.

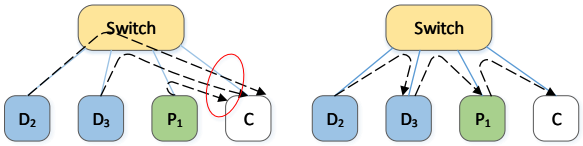
$$d^* = \sum_{i=1}^{k} \alpha_i b_i \tag{1}$$

$$p_i = \sum_{j=1}^{k} \beta_{i,j} d_j, 1 \leq i \leq r \tag{2}$$

B. Repair Operation

In erasure-coded storage systems, repair operations occur in a degraded read or a reconstruction job. When a client reads an unavailable data block, it will receive an exception reply and trigger a degraded read, where the client temporarily decodes the data block by retrieving k available blocks within the same stripe from other nodes. Obviously, the read to the unavailable block is degraded due to multiple network transmission and decoding operations. On the other hand, a reconstruction job is mainly launched to repair a disk failure. The job executes parallel repair of several lost blocks and permanently store them on other live nodes. Considering that the reconstruction of a single block is similar to a degraded read, we will only discuss the degraded read in this paper.

In order to explain the degraded read in a concise manner, we choose RS $(5, 3)$ code as an example. Assume that the data blocks d_1, d_2, d_3 and parity blocks p_1, p_2 form a stripe and are stored on the data nodes D_1, D_2, D_3 and the parity nodes P_1, P_2, respectively. Fig. 1(a) shows the conventional repair, where a client *C* triggers a degraded read to repair the lost data block d_1 by fetching three other available blocks d_2, d_3, p_1 stored in nodes D_2, D_3, P_1. Instead of disk I/O and decoding operations, network traffic is considered to be a major factor affecting the repair performance of erasure coding because bandwidth resources are always scarce in distributed systems [26–28]. Thus, during the repair operation, traffic from these



(a) Conventional repair. (b) Repair pipelining.

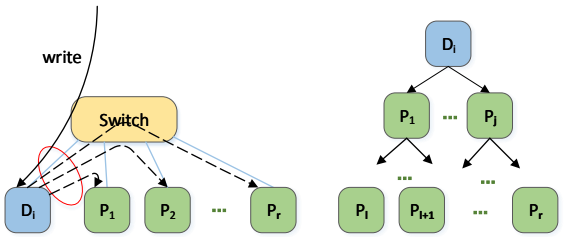
Fig. 1: Examples of conventional repair and repair pipelining with RS $(5, 3)$ code.

three nodes congest the downlink of the client, forming the network bottleneck and increasing the repair time of degraded read.

The drawback of the conventional repair is that all data is transmitted to one node for decoding, resulting in a single link overload. Equation (1) shows that the repair operation of a single block can be decomposed into several XOR operations, so the idea is that if these XOR operations are spread over multiple nodes instead of a single node. In this way, the network traffic is dispersed, so that the network bottleneck is eliminated. Both PPR [26] and repair pipelining [27] use this idea to optimize the transmission path. The performance comparison between PPR and repair pipelining has been described in the literature (see survey [27]), so we only discuss the repair pipelining, which has better performance. With RS $(5, 3)$ code, Equation (1) can be simplified to $d_1 = a_1 d_2 + a_2 d_3 + a_3 p_1$. Fig. 1(b) shows how the repair pipelining repairs d_1 in the client C. First, the data node D_2 reads data from the local disk to compute $a_1 d_2$ and send it to the next data node D_3. Then the node combines the received $a_1 d_2$ and its locally stored block d_3 to obtain $a_1 d_2 + a_2 d_3$ and continues to send it forward. Because the data of $a_1 d_2$ and $a_2 d_3$ are XORed on the node D_3, the amount of data sent forward is only one block-size. When the parity node P_1 receives the computation result of $a_1 D_2 + a_2 D_3$, the lost data block d_1 is obtained by $a_1 d_2 + a_2 d_3 + a_3 p_1$. Finally, it is sent to the client C and the network traffic traversing the downlink of the client is only the data of block d_1. In order to improve efficiency, repair pipelining decomposes a block into a set of fixed-size units to perform the operations mentioned above so that the entire transmission process is fully pipelined. Compared with the conventional repair, repair pipelining reduces the network traffic of the bottleneck link from the original three block size to one block size by making full use of the network bandwidth across multiple nodes. Ultimately the performance of degraded read is close to the normal read.

C. Update Operation

Erasure coding guarantees high system reliability by storing parity blocks, so when the data block is modified, the parity blocks in the same stripe also need to be updated to maintain consistency. According to (2) and addition associativity of erasure coding, when a data block d_j is updated to d'_j, the corresponding parity blocks p_i are updated into p'_i as follows:



(a) Delta-based update scheme. (b) Update tree of T-Update.

Fig. 2: Delta-based update scheme and T-Update with generalized RS(n, k) code.

$$p'_i = p_i + \beta_{i,j}(d'_j - d_j), 1 \leq i \leq r \tag{3}$$

The difference between the original data block d_j and the updated block d'_j is defined as *delta*. We use the symbol δ to present the delta and it can be obtained by $\delta = d'_j - d_j$. As is shown in (3), while updating all parity blocks, only the delta δ along with the original parity block p_i is required to complete the update. This update scheme is called *delta-based update* scheme. In a delta-based update scheme (see Fig. 2(a)), when the data node D_1 receives a write request, it first reads the modified data of block d_1 is to obtain the δ. Then the node updates data block d_1 into a new data block d'_1 and sends the δ to all parity nodes $P_i(1 \leq i \leq r)$. Finally, each parity node completes the update with (3). In the update operation, the uplink of the data node D_1 becomes the network bottleneck since the node is responsible for transmitting the delta to all parity nodes. Moreover, as the number of parity blocks increases, the load on the uplink becomes heavier.

In order to eliminate the network bottleneck and improve update efficiency, T-Update [24] proposes a tree-structured update scheme, which constructs an update tree with a data node as the root and *r* parity nodes as children. Fig. 2(b) illustrates how the scheme completes the update. In the update tree, the root node (a data node storing updated data block d_i) is responsible for computing the delta, updating data block d_i to d'_i, and forwarding the δ to its direct children nodes (a part of parity nodes). This updating and forwarding process repeat until the δ reaches leaf nodes from top to bottom. In the T-Update, the parity node is not only responsible for receiving delta to complete the update but also needs to perform some data transmission tasks. On the contrary, the data node only needs to transmit delta to a small part of parity nodes, thereby reducing the network traffic of the uplink and improving the update performance.

D. Motivation

No matter whether it is a repair operation or an update operation, there is a common problem that the bandwidth usage distribution is highly skewed: a large amount of network traffic is aggregated on the downlink of the client and the uplink of the data node. Repair pipelining and T-Update respectively propose a chain-structured transmission path and a tree-structured transmission path to eliminate network bottlenecks and improve the performance of erasure coding. These optimized transmission paths essentially only move the network traffic from the bottleneck to other links, but the amount of traffic in the system is not actually reduced. What's worse, the network traffic produced by erasure coding is almost cross-rack traffic, and the cross-rack bandwidth shared by various types of workloads is typically limited [3, 30, 31]. Moreover, with the rapid development of storage devices (e.g., NVMe SSD and Optane SSD reduce the average latency to less than 100 microseconds and less than 10 microseconds respectively [40]), network latency becomes a major performance bottleneck for distributed applications. Thus, it is necessary to effectively reduce cross-rack traffic, but it has not received much attention in previous works.

In addition, in the delta-based update scheme, the uplink of the data node is a network bottleneck, which needs to transmit *r* delta. In T-Update, each non-leaf node only needs to send $m(1 \leq m \leq r)$ delta to its children nodes. In other words, although T-Update eliminates the network bottleneck of the uplink, the update time is still bottlenecked by the link with the most update traffic. This motivates us to design a new update scheme, enabling the update traffic on each link is only one delta.

Therefore, we apply in-network computation to erasure coding by offloading XOR operations to programmable network devices. In this way, not only the performance of repair and update operations is improved but also the network traffic in the system is reduced.

III. System Design

In this section, we first describe how the function of in-network computation is integrated into erasure-coded storage system. Then we discuss how to utilize in-network computation to eliminate network bottleneck in the repair and update operations of erasure coding.

A. In-network Computation in HDFS-RAID

In our design, the entire system mainly consists of two components (see Fig. 3): HDFS-RAID and SDN Network. HDFS-RAID is a distributed storage system that implements an erasure coding storage scheme on top of the Hadoop distributed file system (HDFS). On the storage nodes, there are three daemons running: NameNode managing metadata information, DataNode storing the data and the RaidNode managing all functions of erasure coding. Storage nodes communicate with clients through the network. Based on the idea of SDN, a network is divided into two parts: the network core and the SDN controller. The network core is usually a layered network from Top of Rack (TOR) switches at the bottom to core switches at the top. All switches are managed by a centralized SDN controller, which communicates with the switches to obtain network and device information.

In HDFS-RAID, files are encoded and stored in units of blocks. The DataNode storing the data block and the parity block are respectively referred to as the *data node* and *parity*

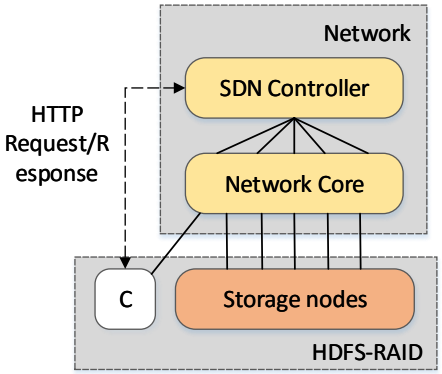


Fig. 3: XORInc architecture.

node, and a DataNode can be both a data node and parity node. All information about erasure coding is managed by RaidNode. In the degraded read and the update operation, the client accesses RaidNode to obtain stripe information.

In addition to maintaining the communication between nodes, the network core also supports XOR operations during the period of repair and update, which makes traffic merge in the network and reduces the amount of data forwarded. But two problems remain to be solved when offloading computation to the network. First, which link is most optimal for effective data transmission between multiple nodes; Second, which switch along the path is suitable for performing XOR operations.

We extend the function of the SDN controller to solve the above problems. Benefiting from the global visibility on the control plane, controllers can easily obtain the network topology, and accurately make a judgment as to which path is the shortest and which switch is the best candidate. Therefore, when a client initiates a degraded read or an update operation, it first obtains this information by sending an HTTP request to the controller. The controller maintains a state for each switch, which indicates the number of repair tasks that the switch is currently responsible for. When the controller receives the request from the client, it takes the path length and load of switches into account to select the transmission path and the switch for computation. At the same time, it updates the state, organizes the results in a tree structure and returns them to the client. In the following description, we refer to this tree structure as a *connection tree*. After receiving the connection tree, the client sends a request to the corresponding node. When the degraded read or update operation is completed, the client sends a signal to the controller to update the state again for the next decision.

B. NetRepair

Repair pipelining makes full use of the bandwidth resources across storage nodes to eliminate the network bottleneck of conventional repair scheme. It allows network traffic to be evenly distributed across each link, and ultimately in a homogeneous network environment where all links have the same bandwidth, the repair overhead of a lost block can be close to

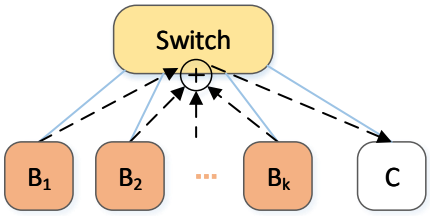


Fig. 4: NetRepair.

the read time of a normal block. However, the repair pipelining only distributes the traffic from the bottleneck link to other links that means it does not reduce the amount of network traffic generated by the degraded read. These network traffic still occupies the bandwidth resources, especially the scarce cross-rack bandwidth, which will affect other applications running in the system. Thus, we propose NetRepair, which decomposes the decoding operations of the erasure coding like the repair pipelining, but it chooses to perform the XOR operations at the network instead of the idle node. NetRepair is designed to achieve two goals: (1) to reduce the repair time of a lost block to near the read time of a normal block as well as repair pipelining (2) to reduce cross-rack traffic in the system compared with repair pipelining. In this paper, we use the RS code as an example to illustrate the effectiveness of NetRepair, but in fact, our scheme can be applied to any erasure code that satisfies the linearity and addition associativity.

First, in a simple network topology with only one switch as shown in Fig. 4, we analyze how NetRepair repairs a single block in a degraded read. In order not to lose generality, we consider deploying the RS(n, k) code in the system, and now we need to repair the lost data block b^* by retrieving the k available blocks (denoted by $b_1, b_2, \cdots, b_k$) within the same stripe. These blocks are distributed among k storage nodes (denoted by $B_1, B_2, \cdots, B_k$) which are located in different racks to maximize system reliability. To simplify our description, we assume that there are no switches other than the link intersection. In the following, we all follow this assumption unless otherwise stated. According to (1), the decoding operation can be decomposed into addition and multiplication. The k nodes first complete the multiplication operation locally to obtain $\alpha_i b_i (1 \leq i \leq k)$ and sends it to the switch. The switch completes $a_1 b_1 + a_2 b_2 + \cdots + a_k b_k$ and send the result b^* to the client. Because the switch combines data from k different links before forwarding, the traffic on each link has only one block size. Therefore, there is no bottleneck link. In order to improve transmission efficiency, NetRepair, as repair pipelining does, decomposes the repair of one block into the repair of a set of small fixed-size units to pipeline the entire transmission process, and ultimately the overall repair performance depends on the slowest link. Thus, in a homogeneous network environment, NetRpair can achieve the repair overhead of a lost block closely to the read time of a normal block.

We further analyze the cross-rack traffic generated by Ne-

tRepair and repair pipelining when repairing a single data block. To simplify discussion, we assume that the switch is denoted as *S* and the control information in the repair process is ignored. In the repair pipelining, the entire transmission process can be denoted as $B_1 \rightarrow S \rightarrow B_2 \rightarrow S \rightarrow \cdots \rightarrow S \rightarrow B_k \rightarrow S \rightarrow C$. Except for the node B_1 and the client, the remaining $k-1$ nodes not only need to receive data but also send data. Therefore, the transmission process will generate $(k-1)*2+2=2k$ block size cross-rack traffic. However, for NetRpair, the transmission process can be simplified to $\{B_1, B_2, \cdots, B_k\} \rightarrow S \rightarrow C$, where $\{B_1, B_2, \cdots, B_k\} \rightarrow S$ means that *k* nodes transmit data to the switch concurrently through different links. In the whole process, the data transmitted on each link is one block size, so the $k+1$ block size cross-rack traffic is finally generated. Compared with repair pipelining, NetRepair reduces cross-rack traffic by half. On the one hand, in practical distributed storage systems that manage large-scale datasets typically divide files into large-size blocks, such as 128 MB or even larger. On the other hand, the choice of parameter *k* will be a trade-off between system reliability and storage overhead. Common configurations are $k=12$ in Azure and $k=10$ in Facebook. In summary, NetRepair can bring significant traffic reduction when repairing a lost data block.

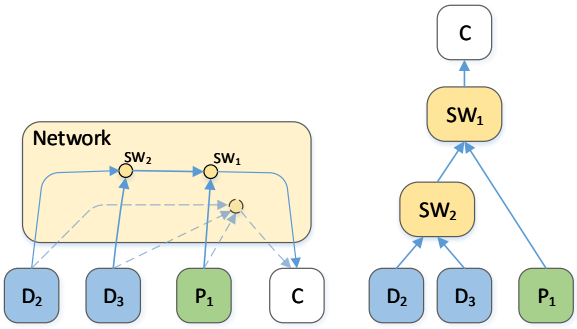
We next consider a more complex network topology. The datacenter network is usually a hierarchical topology, from the lowest TOR switch to the uppermost core switch, the number of switches decreases layer by layer. To simplify the discussion, we abstract the network topology: switches in the same layer are logically considered to be in a horizontal direction, and switches in different layers are considered to be in a vertical direction. Extend the network with only one switch: (1) Regardless of repair pipelining or NetRepair, if a switch (denoted as S_2) is added in the horizontal direction, the transmission process increases the link of $S \rightarrow S_2$, so the traffic will increase by a block size. (2) If a switch (denoted as S_3) is added in the vertical direction, assuming that the added switch is located between B_2 and S, then the transmission path of the repair pipelining will change from $S \rightarrow B_2 \rightarrow S$ to $S \rightarrow S_3 \rightarrow B_2 \rightarrow S_3 \rightarrow S$, and the transmission path of NetRepair will change from $B_2 \rightarrow S$ to $B_2 \rightarrow S_3 \rightarrow S$. That is, for every switch added in the vertical direction, the traffic of repair pipelining will increase by two block sizes, and the traffic of NetRepair will increase by one block size. To generalize our traffic model, when there are *h* switches in the horizontal direction and *v* switches in the vertical direction, the traffic generated by repair pipelining and NetRepair is $2k+h-1+2v=2(k+v)+h-1$ and $k+1+h-1+v=k+v+h$ block size, respectively. It can be seen that when the network topology is complex, NetRepair still generates less network traffic than repair pipelining. In addition, in extreme cases, a node is far away from other nodes, causing the value of *h* to be much larger than *k*. At this time, the network traffic mainly comes from the horizontal forwarding between switches, but the performance of NetRepair is no worse than Repair pipelining in the worst case. However, this situation is relatively rare. Because we can deploy blocks on some more centralized racks, or try to select *k* blocks that are close to each other to repair a lost data. Therefore, in most scenarios, NetRepair can achieve the desired goal.

In the design of NetRepair, the most critical part is using switches to perform XORs over packets from multiple nodes. In order to achieve this function, we need to answer the following questions:

What conditions should be satisfied when a switch is chosen to perform XORs? (1) At least two data flows from different nodes are aggregated on the switch; (2) The switch has sufficient storage and computational resources to perform XORs. In other words, a switch should not be responsible for too many computing tasks.

Is there a switch that satisfy the conditions? This question can be transformed into another equivalent question: Will data flows from different nodes necessarily aggregate on a switch? Today's datacenter networks have a clear advantage in achieving flow aggregation. First, the number of aggregation switches and core switches in the network are far less than the number of nodes. A large amount of traffic is aggregate on these switches, so it is common that data from different nodes flows through the same switch. Second, in order to ensure the high availability of the network, the datacenter networks offer multiple paths between storage nodes, which makes it easy to change the data transmission path by configuring routing policy. Finally, even if data from different nodes are transmitted to the same target node through a completely different path, the data will eventually converge on the TOR switch, which directly connects the target node. As a result, there must be an appropriate switch to perform the XOR operations.

How to find the switch that satisfies the conditions? This work is done by the SDN controller. When the client triggers a degraded read, it first accesses RaidNode to retrieve the other *k* available blocks on the stripe and obtain the IP addresses of the nodes where these *k* blocks are located. Then the client sends its IP address and the *k* nodes to the SDN controller via an HTTP request. Because SDN controller maintains the global network topology information, it is easy to select the appropriate path for data transmission between nodes. In order to avoid the excessive consumption of switch storage and computational resources, we implement a simple scheduling strategy in SDN controller to find relatively idle switches to perform computing tasks. First, for the *k* nodes participating in the decoding, the SDN controller finds the *x* shortest path from each node to the client based on the number of network hops. It is common that there are multiple shortest paths between nodes in the datacenter network. But there are also some nodes that cannot find the *k* shortest paths. We use $\mathcal{P}_i$ to represent the set of shortest paths from the *i*-th ($1 \leq i \leq k$) node to the client. There are *k* such sets. Then we select a path from each set, and the selected *k* shortest paths form a transmission network of the degraded read. Since there are *k* sets and up to *x* shortest paths in each set, at most x^k transmission networks can be formed. Fig. 5(a) shows a transmission network with the



(a) Examples of transmission network. (b) Connection tree.

Fig. 5: Example of the scheduling strategy in NetRepair.

RS$(5, 3)$ code, where the intersection of the paths represents a switch where data flows aggregated. Each transmission network can be represented by the connection tree shown in Fig. 5(b). When the controller obtains a transmission network, it is also necessary to judge whether these switches at the intersection are *busy*. We define a switch is *busy* when the number of degraded reads dealt by the switch reaches a predefined threshold, otherwise, it is *free*. If all switches at the intersection are free, data from k different nodes can be transmitted according to the transmission network and XOR operations can be performed on these switches. But once a switch is busy, we need to reselect the transmission network until it meets the criterion. If the SDN controller fails to find an appropriate transmission network, NetRepair will abandon computing in the network and put compute operations back on storage nodes. At this time, NetRepair will use the chain-structured transmission scheme in repair pipelining to repair data.

It is worth noting that the values of x and k are usually relatively small. The default value of x is 2, and k is usually around 10 in production environments, so the value of x^k is not large and in most cases, the controller does not traverse all x^k transmission networks. Therefore, the main overhead of our scheduling strategy comes from finding the shortest path (the time complexity of finding multiple shortest paths is equal to find one shortest path), which is one of the core functions of the controller. The controller optimizes this function by updating the topology information in real time and caching the shortest path between switches. Therefore, the scheduling strategy neither takes up too many additional resources of the controller nor has any impact on the degraded read. In addition, we can define more suitable algorithms for different scenarios to select the transmission path rather than simply based on the shortest path. On the other hand, the performance of switches also needs to be evaluated more accurately. These will be our future work.

C. NetUpdate

In addition to the repair operation, the update operation of erasure coding can also be optimized by means of in-network computation. Fig. 2(a) shows the performance bottleneck in the update operation, which is mainly caused by a star-structured transmission scheme. T-Update proposes a tree-structured transmission scheme to eliminate the bottleneck. Its main idea is to let the parity nodes also participate in the transmission of the *delta* to share the pressure on the network transmission, which originally belongs to the data node. Although T-Update improves the performance of the update operation, the total network traffic does not decrease because the traffic is only dispersed from the bottleneck link to other links. Moreover, the performance of the update operation still depends on the link with the most update traffic. Therefore, we propose NetUpdate, which reduces the latency of update operations by calculating the delta on switches. At the same time, it utilizes the multicast function of switches to make only one delta is transmitted on each link, which not only eliminates the network bottleneck but also reduces the total network traffic.

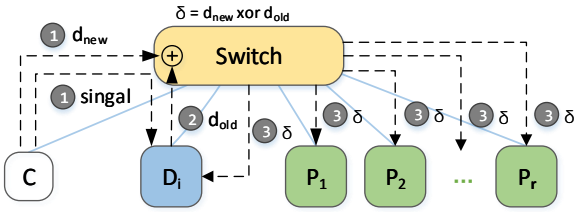


Fig. 6: NetUpdate.

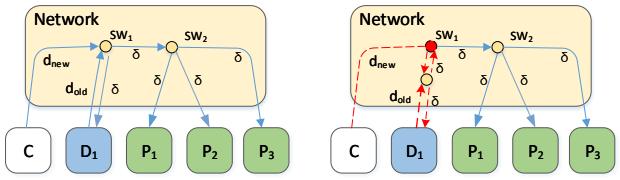
The entire process of NetUpdate is shown in Fig. 6. We assume that the client needs to update a file on the data node D_i, the data written to the file is denoted as d_{new}, and the original data before updating is denoted as d_{old}. Moreover, the size of data d_{new} is equal to d_{old}. The client first sends the data d_{new} to the switch, and at the same time sends a signal to the data node D_i. The signal is a very small packet that carries various information such as the file path and offset, which is used by the D_i to locate where the file is modified. After receiving the signal, D_i will send d_{old} to the switch, which calculates the delta by $\delta = d_{new} \oplus d_{old}$. The deltas are then multicast to data node D_i and all parity nodes. Finally, D_i completes the update by $d_{new} = d_{old} \oplus \delta$ and parity nodes complete the update according to (3). Compared with conventional delta-based update scheme and T-Update, NetUpdate can achieve better performance in two aspects:

Update time: We compare the three update schemes with the example in Fig. 7(a). Fig. 8 shows the data transmission paths of the three update schemes. In the conventional delta-based update scheme, the data d_{new} sent by the client reaches the data node D_1 via the switch SW_1. After D_1 receives the data, it calculates the delta and completes the update. Then D_1 sends a delta to the parity nodes P_1, P_2, P_3, respectively. These three deltas pass through switches SW_1, SW_2 and then are forwarded to different links by switch SW_2. Finally, parity nodes P_1, P_2, P_3 receive the delta and complete the update. In T-Update, we assume that a tree-structured transmission

path such as $D_1 \rightarrow P_1 \rightarrow \{P_2, P_3\}$ (a tree with degree 2) is constructed. Data node D_1 only needs to transmit a delta to parity node P_1, which receives the delta and then continues to transmit it to P_2 and P_3, thus sharing the transmission pressure of data node D_1. It should be noted that although T-Update eliminates the network bottleneck in the delta-based update scheme, there still is a link ($P_1 \rightarrow SW_2$) that needs to transmit multiple deltas. This link will become a new bottleneck, but the transmission pressure on this link will be less than the previous bottleneck link. In NetUpdate, d_{new} and the signal are simultaneously sent by the client, while the signal is much smaller than d_{new}, leading to the signal arriving at data node D_1 earlier than d_{new} reaching SW_1. Upon receiving the signal, data node D_1 immediately send the d_{old} to SW_1. For the sake of illustration, we assume that D_1 does not start sending the data d_{old} until SW_1 receives the d_{new}. When SW_1 receives the data and calculates the delta, the delta is forwarded in parallel to data node D_1 and switch SW_2. SW_2 then continues to forward the delta to all parity nodes. During the period of update, NetUpdate can reduce update time in two ways. First, NetUpdate move the XOR operations from the data node to the switch, making the delta closer to parity nodes. Compared with transmitting the delta from a data node, starting the transmission from the network will go through fewer links, so that the data encounter less network congestion and queuing. Second, NetUpdate completely eliminates bottleneck link so that only one delta is needed to be transmitted on each link. In the conventional delta-based update scheme, the bottleneck link is the uplink of the data node, because it needs to transmit *r* deltas. In T-Update, the bottleneck link is the link that needs to transmit the most deltas.

Network traffic: Fig. 8 also reflects the traffic generated by the three update schemes in an update operation. The delta-based update scheme generates eleven delta-sized network traffic. Moreover, for each additional bottleneck link ($D_1 \rightarrow SW_1$ and $SW_1 \rightarrow SW_2$ in Fig. 8 are bottleneck links), the traffic will increase by *r* delta size ($r = 3$ in Fig. 8). Although T-Update reduces the traffic of the original bottleneck link, it still generates nine delta-sized network traffic after completing an update. This is because the reduced traffic is actually transferred to another link ($P_1 \rightarrow SW_2$), which becomes a new bottleneck link. Therefore, T-Update does not have much advantage in reducing network traffic, and even in some topologies (such as Fig. 6) have no advantage. But in NetUpdate, not only the bottleneck link is eliminated but also no traffic is added on other links. So, NetUpdate really reduces network traffic. In Fig. 8, NetUpdate generates seven delta-sized network traffic in an update, which reduces the network traffic by 36% and 22% compared with the delta-based update scheme and T-Update.

In the update operation, we also need to consider how to choose the switches that perform the XOR operation. Compared with repair operation, update operation brings less computational load to switches. Because an update operation requires only one switch to participate in the computation, and the switch only needs to XOR the data from the client



(a) Before scheduling. (b) After scheduling.

Fig. 7: Example of the scheduling strategy in NetUpdate.

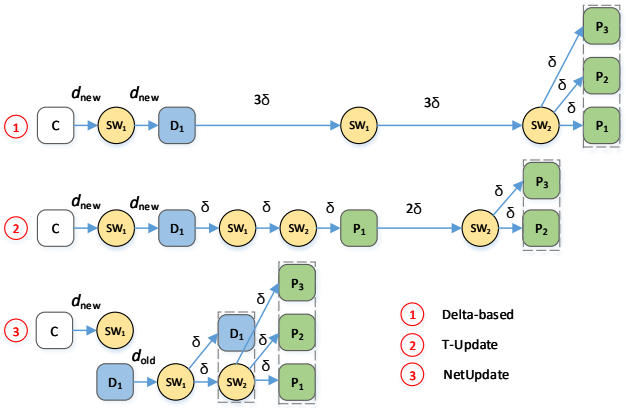


Fig. 8: Data transmission paths of three update schemes under the topology shown in Fig. 7(a).

and the data node. We design another strategy for the update operation to select the switch and prevent the switch from being overloaded. This strategy is still performed by the SDN controller. First, the controller finds the shortest paths from the client to the updated data node and all parity nodes. These $r+1$ paths form a transmission network. Fig. 7(a) is a transmission network with three parity nodes. The switch SW_1 is selected to perform the XOR operation. In order to prevent SW_1 from undertaking too many computing tasks, we set a threshold for switches as in the repair operation. The threshold indicates the maximum number of update tasks a switch can undertake. A switch that exceeds the threshold is considered busy and can no longer take on new tasks. Unlike repair request, the update request is small but the frequency is high. If each time a client generates an update request, it accesses the controller once to generate the transmission network and update the state of switches, then the overhead of accessing controller cannot be ignored. To solve this problem, we buffer the previous update information on the client and set a validity period for the buffer. During the validity period, the client updates the same data block without accessing the controller. Moreover, since a single update request is small, which imposes little computational load on the switch, so it is not necessary to update the state of switches every operation. The state can be updated once in a validity period. When we find that a switch is busy, we need to select another free switch to perform the computing task. The scheduling strategy we adopt is to find a new switch on the link between the updated data node and

the switch that is busy. For example, in Fig. 7(a), when SW_1 is busy, we choose a new switch on the path $SW_1 \rightarrow D_1$. The reason for this choice is that choosing any switch on this path to perform XOR operations guarantees minimal network traffic. In addition, we give priority to switches closer to SW_1, because computing at the network center can reduce more update latency than computing at the edge. If all switches on $SW_1 \rightarrow D_1$ are not suitable, we will choose the data node to calculate the delta. But unlike the delta-based update scheme, the data node in NetUpdate only need to send a delta to the corresponding switch, and then the switch forwards the delta to each link in a broadcast manner. This means that NetUpdate always guarantees that there are no bottleneck links.

In summary, the advantages of NetUpdate are reflected in two aspects. First, when we move the compute operations from a data node to a switch, the delta is generated directly in the network. Sending deltas from the network to all parity nodes has less latency than sending from a data node. Second, NetUpdate broadcasts deltas so that each link in the network only needs to transmit one delta and no new links need to be added. This not only completely eliminates the network bottleneck but also reduces the network traffic. Importantly, these advantages of NetUpdate do not depend on network topology. So, considering a more general network topology, NetUpdate is still valid.

IV. Evaluation

We conduct simulation-based experiments to extensively evaluate the XORInc framework. Our evaluation reveals the advantages of XORInc in improving repair/update performance and reducing network traffic.

A. Simulation Setup

In our experiments, we simulate a virtual network environment in which the SDN controller chooses floodlight-0.90 and the switch is simulated using Open vSwitch 2.0.2. In order to ensure the unification of the experimental environment, the server is also generated by simulation. We generate 18 docker containers on a physical machine to build a distributed storage system. The physical server has two quad-core 2.4GHz Intel Xeon E5620 CPUs, 12GB DDR3 RAM, and two 1TB SATA hard disks. The operation system running on the server is Ubuntu 14.04, and the version of docker is 1.7.1. Each container acts as a virtual server. All of these virtual servers, locating in different racks, are connected via a 1Gb/s virtual switch. To fairly evaluate the impact of network transfers, we host the client on a single server that does not store any data, so as to ensure that all data are always transmitted over the network. In addition, there is also a single server performing as the NameNode and RaidNode, with the rest acting as the DataNode.

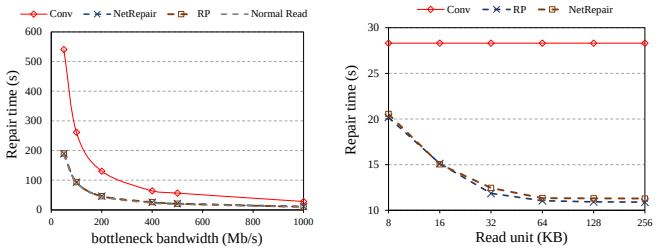
We use *RandomTextWriter* on HDFS-RAID to generate files of different sizes according to the coding parameters. The file is divided as data blocks with a fixed size. With the *Jerasure* open source coding library, the *k* data blocks in one stripe are encoded into *n* blocks, which are distributed to *n* different racks. We configure 128MB block size and RS $(5, 3)$ code. In NetRepair and repair pipelining, we pipeline the repair of a large data in small-size units. The default size of a unit is 64KB and we call it "read size". In update operations, the write request size is denoted as "write size", which defaults to 48KB. We vary one of the settings at a time and evaluate its impact.

We use Open vSwitch to simulate a virtual switch, which is only responsible for data forwarding between nodes and communication with the SDN controller. We also create a docker container to act as a computing node, which is directly connected to the switch to provide computing power for the switch. All data that need to be computed in the network is forwarded by the switch to the computing node as it flows through the switch. When the computing node completes the calculation, the results are returned to the switch for further forwarding. In order to make our simulation close enough to real in-network computation, we need the link from the switch to the computing node as fast as possible (this means that there is a high bandwidth link from the switch to the computing node), so that we can ignore the transmission overhead of this link. Eventually, it achieves the same effect as when the computations are performed locally on the switch. Therefore, in our experiment, the bandwidth of the link from the switch to the computing node is set to 10Gb/s, which is much larger than the 1Gb/s bandwidth of other links. With such a simulation framework, we conduct some tests. In these tests, we will further explain the impact of this simulation on the test results.

B. Results of Repair Operation

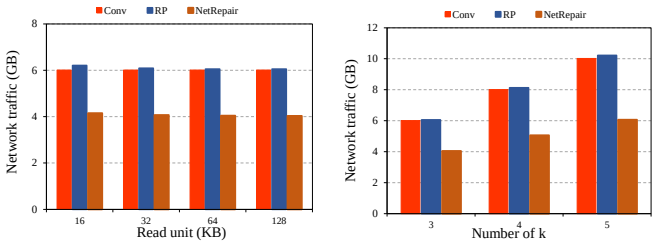
In order to evaluate the performance of NetRepair, we implement the conventional repair scheme (denoted as "Conv") and repair pipelining (denoted as "RP") as comparison schemes and illustrate the advantages of NetRepair from two aspects of repair time and network traffic.

Bottleneck bandwidth: In the conventional repair scheme, the downlink of the client becomes a bottleneck link due to the aggregation of data from multiple nodes. Fig. 9(a) shows the repair time of 1GB data versus the bottleneck bandwidth. It also plots the transmission time of directly reading 1GB data (denoted as "Normal read"). It can be seen that the transmission time is inversely proportional to the bottleneck bandwidth. This is because the transmission process is completely pipelined, and the time to read 1GB data entirely depends on the bottleneck bandwidth. In the conventional repair scheme where RS $(5, 3)$ code is adopted, 3 GB data needs to be transmitted on the bottleneck link, resulting in the repair time being about 3 times that of the "Normal read". Both NetRepair and repair pipelining eliminate the bottleneck link and their performance is close to the "Normal read". In NetRepair, the bandwidth between the switch and the computing node is always 10Gb/s, and the amount of data transmitted on this link is 3GB. Compared with other links that transmit 1GB of data with 1GB/s bandwidth, this link will not become the slowest part of the pipeline. In other words, even if this link is added to the pipeline, it will not affect the overall



(a) Bottleneck bandwidth. (b) Repair time versus read unit.

Fig. 9: Repair time of 1GB data with the variation of bottleneck bandwidth and read size.



(a) Network traffic versus read unit. (b) Coding parameters.

Fig. 10: Network traffic generated by repairing 1GB data with the variation of read size and coding parameters.

performance. So, the results we measured in the simulation environment are reasonable.

Read size: Fig. 9(b) shows the repair time of 1GB data versus the read size. The smaller the read size is, the more read requests the client sends out. NetRepair and repair pipelining begin to show significant performance degradation when the read size is small. This is because frequent interactions between devices introduce additional communication overhead. As the read size increases, the repair time of the two schemes is gradually reduced until they are stable. It is also worth noting that the performance of "Conv" is independent of the read size. This is because in the "Conv", data is transmitted directly from one end to the other, and no other nodes participate in the transmission. Therefore, it is not necessary to divide a data block into small-size units, and the packet switching is sufficient to ensure that the data transmission is completely pipelined. Compared with repair pipelining, NetRepair not only achieves the optimal repair time but also greatly reduces the network traffic. Fig. 10(a) shows the network traffic generated by repairing 1GB of data versus the read size. The test results are obtained by accessing the switch using a REST API. It can be seen that the network traffic generated by the repair pipelining is basically the same as the conventional repair scheme. with 64KB read size, they repair 1GB of data and generate 6.05GB and 6.01GB network traffic, respectively. As the read size decreases, the traffic generated by the repair pipelining will increase slightly. This is mainly due to the extra network transmission overhead caused by too many read requests. NetRepair only generates 4.04GB of network traffic because it moves compute operations from nodes to the network. Compared with the other two repair solutions, the network traffic is reduced by 33%.

Coding parameters: Fig. 10(b) shows the network traffic generated by repairing 1GB of data versus the coding parameters. Since the number of available blocks that need to be read to repair a data block depends on the value of k, we only change the value of k while keeping the value of r unchanged in the test. The results show that as the value of k increases, the network traffic generated by the three schemes will increase. Repair pipelining and the conventional scheme are always consistent, but the increase of NetRepair is only half of the other two schemes. As k increases from 3 to 5, the network traffic reduction of NetRepair increases from 33% to 41%, which exactly shows NetRepair is conducive to reduce traffic. More specifically, The larger the k value is, the greater advantage NetRepair has.

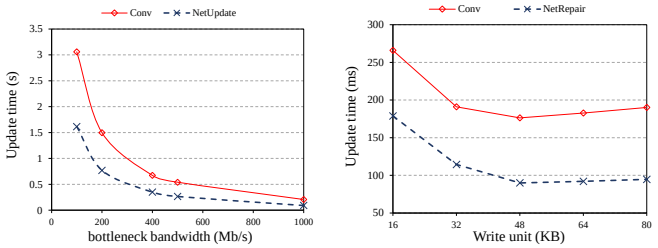
C. Results of Update Operation

In order to evaluate the performance of NetUpdate, we implement the conventional delta-based update scheme (denoted as "Conv") as comparison schemes and illustrate the advantages of NetUpdate from the two aspects of update time and network traffic. All test results are obtained by updating 10MB data.

Bottleneck bandwidth: In the conventional delta-based update scheme, the uplink of the data node becomes a bottleneck link because it needs to bear the burden of delta forwarding for all parity nodes. Fig. 11(a) shows the update time versus the bottleneck bandwidth. A total of the 10MB updated data volume is divided into many 64KB write units so that the entire update process is completely pipelined. The update time is inversely proportional to the bottleneck bandwidth. The update time of NetUpdate is always half of that of "Conv". This is because NetUpdate eliminates the bottleneck so that only one delta is transmitted on each link, while two deltas on the bottleneck link of "Conv".

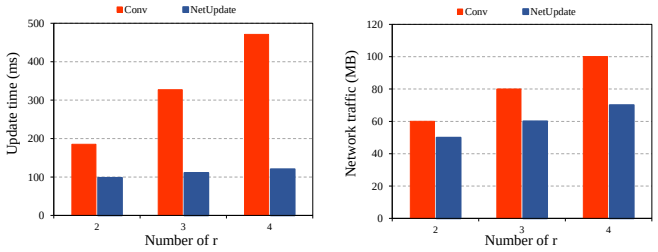
Write size: Fig. 11(b) shows the update time versus the write size. The update time continues to decrease until the write size grows to 48KB, and then remains stable as the size increases. Since the total amount of updated data volume is 10MB, the smaller write unit it is, the more request number it generates, which increases the transmission overhead.

Coding parameters: Fig. 12(a) shows the update time versus the coding parameters. As r increases from 2 to 4, the update time of conventional update scheme increases significantly, while that of NetUpdate is almost unchanged. When the value of r is 4, NetUpdate reduces the update time by 74%. This is because in the conventional update scheme, with the increase of parity node number, more deltas need to be transmitted via the uplink of the data node. But in NetUpdate, no matter how many parity nodes there are, only one delta needs to be transmitted on each link. Fig. 12(b) shows the network traffic versus the coding parameters. A larger value of r increases the amount of network traffic generated by these two schemes since more parity nodes need to be updated.



(a) Bottleneck bandwidth. (b) Update time versus write unit.

Fig. 11: Update time with the variation of bottleneck bandwidth and write size.



(a) Update time versus r. (b) Network traffic versus r.

Fig. 12: Update time and traffic with the variation of coding parameters

However, the increased network traffic of NetUpdate is always half of the conventional update scheme. When r increases from 2 to 4, the network traffic reduction of NetUpdate increases from 17% to 30%.

V. Related Work

Repair of erasure coding: There are many works that focus on optimizing the repair performance of erasure coding of which designing a new code is the most common approach. LRC codes [7, 18] add local parity blocks to optimize single-block repair. Rotated RS codes [17] require fewer data reads and disk I/O to accelerate degraded read. Hitchhiker [16] proposes a new stripe construction method to reduce both bandwidth and disk I/O in repair operations. Regenerating codes [19, 20] repair the lost data blocks by reading a small amount of encoded data, thereby reducing network transmission overhead. HACFS [22] uses two different erasure codes to maintain a low storage overhead and improve the repair performance. Another optimization direction is to design new repair paths that are orthogonal to the design of new erasure codes. PPR [26] decomposes a repair operation into multiple partial operations to relieve transmission pressure on the bottleneck link. The most closely related work to NetRepair is repair pipelining [27], which proposes a chain-structured repair path to pipeline the repair and reduce the repair time to almost the same as the normal read time. But NetRepair not only achieves the optimal repair time but also significantly reduces the network traffic introduced by repair operations.

Update of erasure coding: Existing parity update solutions mostly adopt delta-based updates to reduce disk I/O and network traffic. Parity logging [29] saves the disk read overhead of parity blocks by storing deltas in a log and delaying the updates until the log is full. CodFS [14] extends the parity logging by storing deltas in reserved log space next to the parity chunks to reduce disk seeks in repair operations. T-Update [24] designs a tree-structured update scheme to distribute update traffic and utilize bandwidth across storage nodes. Neither does T-Update achieve an optimal update performance nor does T-Update reduce the amount of update traffic. It's worth saying that NetUpdate can achieve these two goals with in-network computation and multicast.

In-network computation: In prior studies, in-network computation is widely used to enhance the performance of key-value stores. NetCache [32] leverages programmable switches to cache hot data to effectively balance the load of the storage system. IncBricks [33] combines programmable switches and network accelerators to build an in-network caching fabric, in which the network accelerators provide larger storage space to improve the cache hit ratio. NetRS [34] enables in-network replica selection for key-value stores to reduce response latency. Compared with these applications of in-network computation, XORInc mainly utilizes in-network computation to improve the performance of erasure coding. Moreover, XORInc brings less storage and computational pressure to the programmable switches, and it only needs the switches to perform XOR operations on the data flowing through it.

VI. Conclusion

We propose XORInc, an in-network computation framework that supports XOR operations. With this framework, we design new repair and update schemes to optimize the performance of erasure coding. The new repair scheme NetRepair can make the performance of the degraded read the same as the normal read, while greatly reduces network traffic brought by the degraded read. In addition, the update scheme NetUpdate can also achieve an optimal update performance and generate as little network traffic as possible. To the best of our knowledge, we believe XORInc is the first work that uses in-network computation to optimize erasure coding, and we plan to extend this work on a real programmable switch in the future.

VII. Acknowledgment

We thank the anonymous reviewers for their valuable feedback. This work was supported in part by National Key R&D Program of China No. 2018YFB10033005, NSFC No. 61832020, No. 61772216, National Defense Preliminary Research Project (31511010202), Hubei Province Technical Innovation Special Project (2017AAA129), Wuhan Application Basic Research Project (2017010201010103), Fundamental Research Funds for the Central Universities, and CERNET Innovation Project NGII20170120. (*Corresponding author: Xuehai Tang*)

REFERENCES

[1] S. Ghemawat, H. Gobioff, and S.-T. Leung, *The Google file system*. ACM, 2003, vol. 37, no. 5.

[2] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan, "Availability in globally distributed storage systems." in *OSDI*, vol. 10, 2010, pp. 1–7.

[3] K. Rashmi, N. B. 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 *HotStorage*, 2013.

[4] B. Schroeder, R. Lagisetty, and A. Merchant, "Flash reliability in production: The expected and the unexpected." in *FAST*, 2016, pp. 67–80.

[5] A. Fikes, "Colossus, successor to google file system," 2015.

[6] "Facebooks erasure coded hadoop distributed file system (hdfs-raid)," https://github.com/facebook/hadoop-20, 2007.

[7] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, S. Yekhanin *et al.*, "Erasure coding in windows azure storage." in *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC12)*, 2012, pp. 15–26.

[8] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly, "The quantcast file system," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1092–1101, 2013.

[9] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, 2010, pp. 1–10.

[10] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang *et al.*, "f4: Facebooks warm blob storage system," in *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, 2014, pp. 383–398.

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

[12] D. Narayanan, A. Donnelly, and A. Rowstron, "Write off-loading: Practical power management for enterprise storage," *ACM Transactions on Storage (TOS)*, vol. 4, no. 3, p. 10, 2008.

[13] I. F. Adams, M. W. Storer, and E. L. Miller, "Analysis of workload behavior in scientific and historical long-term data repositories," *ACM Transactions on Storage (TOS)*, vol. 8, no. 2, p. 6, 2012.

[14] J. C. Chan, Q. Ding, P. P. Lee, and H. H. Chan, "Parity logging with reserved space: towards efficient updates and recovery in erasure-coded clustered storage." in *FAST*, 2014, pp. 163–176.

[15] Y. L. Chen, S. Mu, J. Li, C. Huang, J. Li, A. Ogus, and D. Phillips, "Giza: Erasure coding objects across global data centers," in *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC17)*, 2017.

[16] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A hitchhiker's guide to fast and efficient data reconstruction in erasure-coded data centers," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 331–342, 2015.

[17] O. Khan, R. C. Burns, J. S. Plank, W. Pierce, and C. Huang, "Rethinking erasure codes for cloud file systems: minimizing i/o for recovery and degraded reads." in *FAST*, 2012, p. 20.

[18] 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 *Proceedings of the VLDB Endowment*, vol. 6, no. 5, 2013, pp. 325–336.

[19] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran, "Network coding for distributed storage systems," *IEEE transactions on information theory*, vol. 56, no. 9, pp. 4539–4551, 2010.

[20] K. Rashmi, P. Nakkiran, J. Wang, N. B. Shah, and K. Ramchandran, "Having your cake and eating it too: Jointly optimal erasure codes for i/o, storage, and network-bandwidth." in *FAST*, 2015, pp. 81–94.

[21] A. S. Rawat, S. Vishwanath, A. Bhowmick, and E. Soljanin, "Update efficient codes for distributed storage," in *Information Theory Proceedings (ISIT), 2011 IEEE International Symposium on*, 2011, pp. 1457–1461.

[22] M. Xia, M. Saxena, M. Blaum, and D. Pease, "A tale of two erasure codes in hdfs." in *FAST*, 2015, pp. 213–226.

[23] M. Manasse, C. Thekkath, and A. Silverberg, "A reed-solomon code for disk storage, and efficient recovery computations for erasure-coded disk storage," *Proc. Inf*, pp. 1–11, 2009.

[24] X. Pei, Y. Wang, X. Ma, and F. Xu, "T-update: A tree-structured update scheme with top-down transmission in erasure-coded systems," in *INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications, IEEE*, 2016, pp. 1–9.

[25] J. Huang, X. Liang, X. Qin, Q. Cao, and C. Xie, "Push: A pipelined reconstruction i/of or erasure-coded storage clusters," *IEEE Transactions on Parallel & Distributed Systems*, no. 2, pp. 516–526, 2015.

[26] S. Mitra, R. Panta, M.-R. Ra, and S. Bagchi, "Partial-parallel-repair (ppr): a distributed technique for repairing erasure coded storage," in *Proceedings of the Eleventh European Conference on Computer Systems*, 2016, p. 30.

[27] R. Li, X. Li, P. P. Lee, and Q. Huang, "Repair pipelining for erasure-coded storage," in *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC17)*, 2017, pp. 567–579.

[28] M. Silberstein, L. Ganesh, Y. Wang, L. Alvisi, and M. Dahlin, "Lazy means smart: Reducing repair bandwidth costs in erasure-coded distributed storage," in *Proceedings of International Conference on Systems and*

Storage, 2014, pp. 1–7.

[29] D. Stodolsky, G. Gibson, and M. Holland, "Parity logging overcoming the small write problem in redundant disk arrays," in *ACM SIGARCH Computer Architecture News*, vol. 21, no. 2, 1993, pp. 64–75.

[30] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. Vijaykumar, "Shufflewatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters." in *USENIX Annual Technical Conference*, 2014, pp. 1–12.

[31] M. Chowdhury, S. Kandula, and I. Stoica, "Leveraging endpoint flexibility in data-intensive clusters," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, 2013, pp. 231–242.

[32] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "Netcache: Balancing key-value stores with fast in-network caching," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 121–136.

[33] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya, "Incbricks: Toward in-network computation with an in-network cache," *ACM SIGOPS Operating Systems Review*, vol. 51, no. 2, pp. 795–809, 2017.

[34] Y. Su, D. Feng, Y. Hua, Z. Shi, and T. Zhu, "Netrs: Cutting response latency in distributed key-value stores with in-network replica selection," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, 2018, pp. 143–153.

[35] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[36] "Barefoot tofino: P4-programmable ethernet switch asics," https://barefootnetworks.com/products/brief-tofino/, 2017.

[37] "Intel ethernet switch fm6000 series, white paper," 2013.

[38] "Octeon multi-core mips64 processor family," https://www.cavium.com/octeon-mips64.html, 2017.

[39] "Netronome nfe-3240 family appliance adapters," https://www.netronome.com/products/nfe/, 2017.

[40] "Ceph optimizations for nvme," https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2018/20180808_SOFT-202-1_Liu.pdf, 2018.