Parallelizing Degraded Read for Erasure Coded Cloud Storage Systems using Collective Communications

Peng Li, Xingtong Jin, Rebecca J. Stones, Gang Wang, Zhongwei Li, Xiaoguang Liu, Mingming Ren✉
Nankai-Baidu Joint Lab, College of Computer and Control Engineering
Nankai University, Tianjin, China
Email: {lipeng, jinxingtong, rebecca.stones82, wgzwp, lizhongwei, liuxg, renmingming}@nbjl.nankai.edu.cn

***Abstract*—For lower storage costs, storage systems are increasingly transitioning to the use of erasure codes instead of replication. However, the increase in the amount of data to be read and transferred during recovery for an erasure-coded system results in the problem of high degraded read latency.**

We design a new parallel degraded read method, Collective Reconstruction Read, which aims to overcome the problem of high degraded read latency of erasure coding by utilizing parallel reconstruction. By introducing collective communication operations (e.g. all-to-one reduction and all-to-all reduction) into distributed storage systems, data reading, transferring and decoding are preformed by all of the involved data nodes in parallel rather than the client itself. Therefore, the time complexity of the degraded read operation is reduced from linear time to logarithmic time.

We implement Collective Reconstruction Read in HDFS-RAID and evaluate it as the block size and stripe size vary. We find that these algorithms can reduce degraded read latency significantly, thereby improving system availability. Specifically, experimental results indicate an approximate 55% **to** 81% **round off drop in degraded read latency.**

1. Introduction

Recently peta-level storage systems are becoming more and more common. The failures they have to face with are not as simple as in the past [1]. To avoid data loss when failure occurs while maintaining high reliability and availability, storage systems use a replication strategy to distribute several copies of data across multiple machines and racks. For example, the Google File System (GFS) [2] and the Hadoop Distributed File System (HDFS) [3] typically maintain three copies for each data block. This method results in a high reliability and availability in practical application at the expense of significant storage overhead (i.e. 300%), which becomes costly for large scale systems. Thus, cloud storage systems [4, 5] use erasure codes to store data instead, which can provide high reliability with significantly lower storage costs.

An erasure coding scheme is defined by parameters (k,m) where k original data blocks are encoded into m coding blocks, which can tolerate any m failures. Erasure coding reduces storage overhead significantly compared with replication, but it suffers a high reconstruction cost when node or disk failures happen. If a data block is 3-way replicated and one replica among them is corrupted, the request for the unavailable block can be accomplished simply by fetching the data from one of the other two available replicas. However, in an erasure-coded system, the request for an corrupted data block will launch a block reconstruction process which fetches k blocks (belonging to the same stripe as the corrupted block) from multiple remote nodes. The network links of clients become the major bottleneck, which takes considerable data transfer time and impacts system throughout significantly, especially when a large amount of data need to be recovered, such as all the data of a corrupt disk or node. This problem is called *degraded read*.

Recently much research has been carried out studying how to reduce the high degraded read latency and shorten the long reconstruction time. For example, Huang *et al.* presented LRC [6], which divides the coding blocks into global and local types to reduce the recovery cost. RASHMI *et al.* presented Hitchhiker [7], which "rides" on the top of RS codes and reduces both network traffic and disk I/O by around 25% to 45% during reconstruction of missing or unavailable data without additional storage. Mitra presents PPR [8], which focuses on scheduling small partial reconstruction operations more efficiently to reduce bandwidth pressure. A few researchers focus on proactive disk failure warning process such as RaidShield [9] which quantifies the vulnerability of RAID group and replaces the in-danger disks in advance to reduce recovery cost when failures truly occur.

In this paper, we present Collective Reconstruction Read (CRR) for erasure-coded systems, which uses parallel operations in the process of recovering lost data. CRR reduces the degraded read latency by making full use of the resources (i.e. bandwidth and computational power) of involved server nodes. We implement a prototype based on HDFS-RAID [4]. Current distributed systems [4, 5] generally use a Serial Reconstruction Read algorithm (SRR) by default, in which the surviving data/coding blocks are read and transferred one by one and then use those blocks to compute the lost data block. By contrast, in CRR, all of the involved surviving nodes read disks in parallel, and then perform network transfer and decoding in parallel by using collective

2324-9013/16 $31.00 © 2016 IEEE
DOI 10.1109/TrustCom.2016.202

communication algorithms, reducing its reconstruction time.

Compared with some popular optimization method [6, 10] which reduces the required data for recovery at the expense of storage overhead, CRR incurs no extra storage costs to decrease degraded read latency without reducing the total amount of transferred data. Moreover, CRR has a wide range of applications and can be applied flexibly to erasure-coded systems and proactive fault-tolerant systems as well.

The remainder of the paper is structured as follows. we motivate the CRR method in Section 2 and present a detail description of CRR, a theoretical analysis and the implementation in Section 3 and evaluate it in Section 4. We summarizes related work in Section 5 and conclude the paper in Section 6.

2. MOTIVATION

In this section, we first illustrate the typical implementation for degraded read—Sequential Reconstruction Read (SRR)—in current distributed storage systems and give the reason why SRR will bring about large degraded read latency. This will motivate our proposed method.

2.1. SRR Method

One kind of erasure coding, Reed Solomon (RS) coding [11] is deployed in HDFS-RAID [4] which uses the SRR method to recover a lost block. Unless otherwise specified, we assume RS(k, m) is used to encode the blocks. During the recovery, it simply builds k connections used for reading the k surviving data/coding blocks needed to reconstruct the corrupted block in the same stripe. Then it reads these surviving data blocks via these connections and decodes the lost block, and then sends the results back to the client. In practice, though it uses multiple threads to accelerate reading of the surviving blocks, it often becomes a network bottleneck because of the huge data to read which reduces the transfer speed. Compared to fault-free read, degraded read will result in more than k times the amount of latency because it performs the reading operations of k surviving data blocks from remote nodes in serial, as well as decoding operations.

The SRR process in a RS(6, 3)-coded system is illustrated in Figure 1; it is time consuming. For example, it requires about $6t$ time to read any corrupted data block encoded with RS(6, 3), even ignoring the decoding time, where t is the time spent in reading and transferring a single block. By contrast, t will be enough if it is directly readable.

2.2. CRR Method

We propose a solution to reduce the degraded read latency in a concurrent way, which is illustrated in Figure 2. When a lost data block is required by a client, we carry out parallel reconstruction as follows: first, all the nodes with the surviving blocks required for recovery read the block

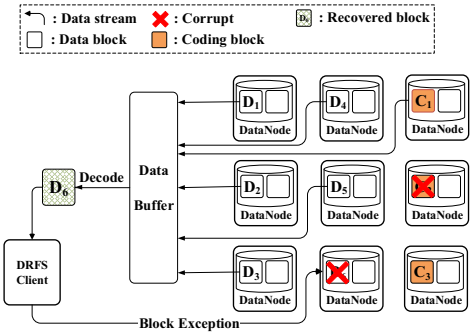

Figure 1. Illustrating the SRR process using RS(6, 3) in HDFS-RAID.

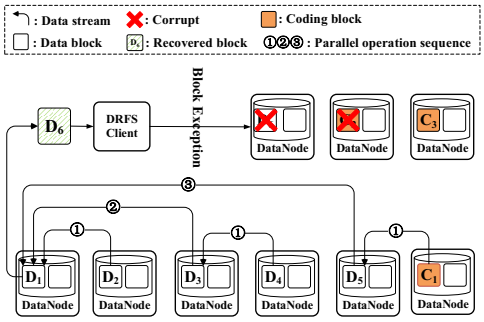

Figure 2. Illustrating the CRR process using RS(6, 3) in HDFS-RAID.

from disk and preprocess the data. The preprocessing step is detailed in Section 2.3; second, we divide the involved nodes into pairs and within each pair, one node sends its block to the other node where the two are combined into one (by XOR-sum); third, we repeat the second step until only one block left, namely the lost data block. Figure 2 indicates the parallel procedure can achieve a shorter read latency ($3t$ compared to $6t$) when the client requests an unavailable data block, although the total amount data to be transferred is not decreased. This procedure is based on the classic algorithm for the *All-to-One Reduction* which is one of the common *Collective Communication* operations. So we name our method *Collective Reconstruction Read* (CRR).

2.3. CRR Preprocessing

With RS(k, m), k data blocks labeled $D_1, \ldots, D_k$ are encoded into m coding blocks labeled $C_1, \ldots, C_m$ over a finite field by the matrix-vector multiplication shown in (2.3). The $k+m$ blocks constitute a stripe and the matrix is constructed from a Vandermonde matrix.

$$\begin{bmatrix} 1 & 1 & \dots & 1 \\ 1 & {a_1}^1 & \dots & {a_1}^{k-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & {a_{m-1}}^1 & \dots & {a_{m-1}}^{k-1} \end{bmatrix} \begin{bmatrix} D_1 \\ D_2 \\ \vdots \\ D_k \end{bmatrix} = \begin{bmatrix} C_1 \\ C_2 \\ \vdots \\ C_m \end{bmatrix} \quad (1)$$

It tolerates any m failures. Any f-erasure ($f \leq m$) can be recovered using the following linear equations:

$$\text{For } 1 \leq i \leq f: \quad L_i = \sum_{j=1}^{k} e_{i,j} A_j \quad (2)$$

where L_i is the lost block, $e_{i,j}$ is the coefficient (determined by some linear operations or matrix inversion [11, 12]) and A_j is one of available block in the stripe. It indicates that all the nodes which have A_j are required to read the block and multiply it by $e_{i,j}$ first. We call this step CRR *preprocessing*. Afterwards, the nodes can perform XOR operation among $e_{i,j}A_j$ concurrently, as described in Section 2.2.

3. Design of CRR

In this section, we present the system design of CRR, which executes decoding operations in a concurrent way. This new method improves a system in two ways. First, it reduces the read latency from $O(k)$ to $O(\log k)$. Second, unlike serial decoding that concentrates the total workload in one place, parallel decoding distributes the decoding cost over several nodes which better utilizes the network and computing resources in the system.

3.1. CRR Architecture

CRR is implemented based on HDFS-RAID [4], which consists of modules such as NameNode, RaidNode, DataNode and Distributed Raid File System (DRFS) client. NameNode is responsible for the management of files and blocks' meta data information of the system and provides control of reading and writing for DRFS clients. DataNodes are in charge of the blocks' physical storage and executes block operations received from NameNode. RaidNode module is used to encode the blocks of cold files and recover corrupted blocks. We add a parallel degraded read module (PDRM) to the DRFS client and a parallel reconstruction module (PReM) to the DataNode. PDRM sends reconstruction commands to multiple involved PReMs when it encounters a corrupted block needed to be recovered and PReMs launch parallel reconstruction. Another task of PReMs is to send the results back to the DRFS client once the reconstruction phase finishes.

A DRFS client requiring a corrupted block causes k DataNodes (DN_0 to DN_{k-1}) be involved in the reconstruction operation. The DRFS client will select a *root* among the involved DataNodes, which is responsible for accommodating the reduction result and sending it back to the client. We assume that DN_0 serves as the root.

As described in Section 2.2, the reconstruction process works as *all-to-one reduction*, a common *collective communication* operation, which can be implemented in three ways. The first way is a linear time centralized solution, in the same way as the default SRR. The other two algorithms, the *recursive halving* algorithm and the *message splitting* algorithm, are both more efficient parallel methods and have logarithm time complexity.

3.2. Algorithm Descriptions

CRR can be implemented based on two different algorithms, which lead to different performance. The *recursive halving* algorithm is the classic solution to all-to-one reduction, and the *message splitting* algorithm adopts all-to-all reduction to utilize network and computing resources more efficiently when reducing large data blocks.

Recursive Reconstruction Read (CRR-R). Algorithm 1 shows the principle behind recursive reconstruction read. First, all involved DataNodes search their local block storage, read the blocks needed for reconstruction and preproccess them. Second, each DataNode whose indices are indivisible by 2 sends preproccessed data to its even-numbered predecessor which is responsible for XORing them and storing the result for the following steps. Third, DataNodes whose indices are indivisible by 4 send their intermediate results to those whose number can be divisible by 4. By repeating this procedure until only the *root* DataNode remains, the lost block can be reconstructed in $\lceil \log_2 k \rceil$ steps. To further reduce latency, we divide each data block into multiple small units (default size of one unit is 1MB) and all of the read, transfer and calculation operations involving them will work in a pipeline.

Splitting Reconstruction Read (CRR-S). From Algorithm 1, we see that concurrency decreases as the algorithm proceeds. Although we use a pipeline strategy, the utilization of network and computing resources is only improved slightly. Considering the relatively large block size used in current distributed systems [4, 5], we adopt a *message splitting* algorithm for all-to-one reduction to improve concurrency of degraded read. The key idea is to split the large data blocks into small fragments. And then all of the involved DataNodes launch an all-to-all reduction on the fragments. Finally, the root DataNode collects (gathers) the partial results from all the DataNodes into the final reduction result and sends it to the DRFS client.

Algorithm 2 shows the pseduo-code of the splitting reconstruction read algorithm; its key part is the all-to-all reduction. First, each of k DataNodes locates its block, reads it from disk to the data buffer and preprocesses the buffer. We treat each block as k fragments, each with a size of bs/k, where bs denotes the block size. In this way, we have k^2 fragments. We can imagine them as a matrix, that is, each block forms a column and the fragments with the same in-block offset forms a row. Then, an all-to-all reduction on this matrix is performed and we can imagine it as k parallel all-to-one reductions on k rows respectively. In each step, all

Algorithm 1 Recursive Reconstruction Read

Input: the total number k of DataNodes involved, and the current DataNode's identifier id.

Output: The DRFS client receives the recovered data block.

alloc two buffers buf and tmp
read the block from the disk to buf and **preprocess** buf
$mask := 0$
$d := \lceil \log_2 k \rceil$
for $i := 0$ to $d-1$ **do**
    /* DataNodes whose lower i bits are 0 participate in the i-th step */
    if (id AND $mask$) = 0 **then**
        if (id AND 2^i) $\neq 0$ **then**
            $dest := id$ XOR 2^i
            send buf to dest
        else
            $src := id$ XOR 2^i
            receive tmp from src
            $buf := buf$ XOR tmp
        end if
    end if
    $mask := mask$ XOR 2^i
end for
if $id = 0$ **then**
    send buf to the DRFS client
end if

Algorithm 2 Splitting-based Reconstruction Read

Input: the total number k of DataNodes involved, and the current DataNode's identifier id.

Output: The DRFS client receives the recovered data block.

alloc two arrays of buffers $buf[k]$ and $tmp[k]$
read the block (k fragments) into $buf[1..k]$ and **preprocess** $buf[1..k]$
$r\_idx := 0$
$d := \lceil \log_2 k \rceil$
for $i := d-1$ to 0 **do**
    $partner := id$ XOR 2^i
    $j := id$ AND 2^i
    $l := (id$ XOR $2^i)$ AND 2^i
    $s\_idx = r\_idx + l$
    $r\_idx = r\_idx + j$
    send $buf[s\_idx..s\_idx+2^i-1]$ to $partner$
    receive $tmp[0..2^i-1]$ from $partner$
    for $j := 0$ to 2^i-1 **do**
        $buf[r\_idx+j] := buf[r\_idx+j]$ XOR $tmp[j]$;
    end for
end for
gather($buf[id]$)
if $id = 0$ **then**
    send buf to the DRFS client
end if

of the DataNodes communicate in pairs so that the network and computing resources are better utilized. Once the all-to-all reduction is complete, DN_i will hold the reduction result of the i-th row. Then we can get the recovered block through a gather operation on these partial results.

Algorithm 3 shows the gather algorithm. It uses the same *recursive halving* algorithm as Algorithm 1 except that the receiver simply concatenates the two buffers rather than combining them into one in each step. Intuitively, the splitting-based algorithm is superior to the recursive halving algorithm because of the more efficient use of network links, especially when the data size is large. A detailed performance analysis is presented in the next subsection.

3.3. Theoretical Analysis

In this section, we first give a theoretical analysis to illustrate the performance in practical applications, and then we analyze the complexity of the two parallel algorithms presented above to show the improvement vs. SRR.

Normal Read vs. Degraded Read. Although CRR is superior to SRR in terms of latency, they consume the same amount network and computing resources. If there are many degraded read requests that need to be processed simultaneously which saturate the entire system or a part of the system, the advantage of CRR over SRR will be diminished. However, when considering a multi-clients situation, which is very common in real environment, we can show that the ratio of degraded read requests to normal read requests is small. As such, we can parallelize the rare degraded read to achieve good average latency (including both normal and degraded read). The ratio (denoted by $R_{n:d}$) of the number of normal reads to the number of degraded reads is approximated by the ratio of the number normal blocks to the number of corrupted blocks:

$$R_{n:d} = \frac{1-q}{q}$$

where q is the disk failure rate. In a real-world environment, the requests to unavailable data always account for a small proportion [13] (typically 1 in 300) compared to the available data. Based on this, the system will still achieve a significant reduction of degraded read latency by using the parallel algorithms while maintaining a low overall average read latency.

Algorithm Complexity Consider a cluster composed of multiple nodes with a block size of b bytes and using $RS(k,m)$ to encode k data blocks into m coding blocks and $n = k+m$. The time of a degraded read operation mainly consists of four parts: reading k blocks from disk, preprocessing the data, computing XOR results and sending the recovered block back to the client. Thus the total time of a degraded read operation launched by a client is given by

$$T_{total} = T_{read} + T_{prep} + T_{recover} + T_{write}, \qquad (3)$$

Algorithm 3 Recursive Gather

Input: *buf* stores the data to be gathered, the total number k of DataNodes involved, and the current DataNode's identifier *id*.

Output: The root DataNode holds the recovered block.

```
1: alloc a buffer tmp
2: mask := 0
3: d := ⌈log2 k⌉
4: for i := 0 to d − 1 do
5:     /* DataNodes whose lower i bits are 0 participate in the i-th step */
6:     if (id AND mask) = 0 then
7:         if (id AND 2^i) ≠ 0 then
8:             dest := id XOR 2^i
9:             send buf to dest
10:        else
11:            src := id XOR 2^i
12:            receive tmp from src
13:            append tmp to buf
14:        end if
15:    end if
16:    mask := mask XOR 2^i
17: end for
```

where T_{read} is the time spent reading the block data from the disk, T_{prep} is the time spent preprocessing data, $T_{recover}$ is the time spent in reconstructing a lost block, and T_{write} indicates the time spent in sending back the recovered data.

To make it clear in the following section, we use T_{SRR}, T_{CRR-R} and T_{CRR-S} to represent the total time cost used in SRR, CRR-R and CRR-S respectively. We also define the following notation: t_d is the disk seek time, t_{rw} is the time required for reading or writing one byte from the disk, t_p is the time required for preprocessing one byte, t_w is the time required for transferring one byte, t_s is the time required for preparing messages and calculating route and so on, t_x is the time of XORing two bytes, and t_m is the time required to copy one byte in memory.

For SRR, $T_{recover}$ consists of two parts: (a) transferring k blocks to clients (denoted by T_{net}); (b) XORing k blocks to reconstruct a lost block (denoted by T_{xor}). Thus, we can describe the time cost as follows:

$$\begin{aligned} T_{read} &= t_d + t_{rw}b \\ T_{prep} &= t_p b \\ T_{recover} &= T_{net} + T_{xor} \\ &= k(t_s + t_w b) + (k-1)t_x b \\ T_{write} &= t_m b. \end{aligned}$$

Thus, combining these formulas and (3), the SRR latency can be described by

$$\begin{aligned} T_{SRR} &= (t_d + t_{rw}b) + t_p b + (k(t_s + t_w b) + (k-1)t_x b) + t_m b \\ &= t_d + t_{rw}b + t_p b + kt_s + kt_w b + (k-1)t_x b + t_m b. \end{aligned}$$

In SRR, t_d, t_{rw}, t_p and t_m are all smaller than t_w, meanwhile, k is greater than 1. So we can conclude that the bottleneck of degraded read latency is the number of DataNodes, which brings about a factor of k block transfer overhead.

CRR-R is different from SRR mainly in three aspects: (a) it needs $\lceil \log_2 k \rceil$ steps to perform the transfer of k blocks; (b) multiple DataNodes are involved in the XOR operations in the same time; (c) the reduction results are sent to client over the network rather than the memory copy in SRR. To recover a corrupted block, all DataNodes will read the required blocks from their disks first. Then, they will perform a *recursive halving* algorithm which will take $\lceil \log_2 k \rceil$ steps and each step will require one block transfer and one XOR operation. Finally, the recovered data will be sent back to the client via the network. Thus, the latency can be described as follows:

$$\begin{aligned} T_{read} &= t_d + t_{rw}b \\ T_{recover} &= \lceil \log_2 k \rceil (t_s + t_w b + t_x b) \\ T_{write} &= t_s + t_w b. \end{aligned}$$

Thus, combining these equations with (3), the CRR-R latency can be described by

$$\begin{aligned} T_{CRR-R} &= (t_d + t_{rw}b) + t_p b + \lceil \log_2 k \rceil (t_s + t_w b + t_x b) \\ &\quad + (t_s + t_w b) \\ &= t_d + t_{rw}b + t_p b + (\lceil \log_2 k \rceil + 1)t_s \\ &\quad + (\lceil \log_2 k \rceil + 1)t_w b + \lceil \log_2 k \rceil t_x b. \end{aligned}$$

Compared with CRR-R, CRR-S spends the same time reading blocks from the disk, preprocessing them and sending result back to client; the main difference is how to reconstruct lost blocks. In CRR-S, the involved DataNodes perform an all-to-all reduction and a gather operation in order to reconstruct a block. We use T_{reduc} and T_{gather} to indicate their time respectively. The reduction process requires $\lceil \log_2 k \rceil$ steps and the i-th step transfers $2^{\lceil \log_2 k \rceil - i - 1} b/k$ bytes and performs a XOR operation. So T_{reduc} can be described by

$$\begin{aligned} T_{reduc} &= \sum_{i=1}^{\lceil \log_2 k \rceil} (t_s + 2^{i-1} t_w b/k + 2^{i-1} t_x b/k) \\ &= t_s \lceil \log_2 k \rceil + t_w b(k-1)/k + t_x b(k-1)/k. \end{aligned}$$

The gather operation always requires $\lceil \log_2 k \rceil$ steps and the i-th step simply needs to transfer $2^{i-1} b/k$ bytes, so the time overhead of gather operation is $t_s \lceil \log_2 k \rceil + t_w b(k-1)/k$. With the T_{reduc} and T_{gather}, the recover time is described as follows:

$$\begin{aligned} T_{recover} &= T_{reduc} + T_{gather} \\ &= 2\lceil \log_2 k \rceil t_s + (2(k-1)/k) t_w b + ((k-1)/k) t_x b. \end{aligned}$$

So the total time cost of CRR-S is

$$\begin{aligned} T_{CRR-S} &= (t_s + t_{rw}b) + t_p b + T_{recover} + (t_s + t_w b) \\ &= t_s + t_{rw}b + t_p b + (2\lceil \log_2 k \rceil + 1)t_s \\ &\quad + (2(k-1)/k + 1)t_w b + ((k-1)/k)t_x b. \end{aligned}$$

It can be inferred from the theoretical analysis that T_{CRR-R} can reduce the degraded read latency from $O(k)$

to $O(\log k)$. However, according to our experiments, this method can only get a 57.4% reduction compared to SRR latency. This can occur in two reasons: (a) the SRR implementation is optimized by multi-threads and pipeline blocks reading and transferring which will reduce the time-span; (b) k is comparatively small (e.g. k might be equal to 5).

Now we compare T_{CRR-R} with T_{CRR-S} to show the reason why splitting-based reduction read makes a (slight) further reduction on degraded read latency. Their difference is given by

$$\begin{aligned} T_{diff} &= T_{CRR-R} - T_{CRR-S} \\ &= (\lceil \log_2 k \rceil - 2(k-1)/k) t_w b \\ &\quad + (\lceil \log_2 k \rceil - (k-1)/k) t_x b - \lceil \log_2 k \rceil t_s. \end{aligned}$$

Given the fact that k is generally greater than 4, $\lceil \log_2 k \rceil - 2(k-1)/k$ and $\lceil \log_2 k \rceil - (k-1)/k$ are both greater than 0. Thus only one item, $-\lceil \log_2 k \rceil t_s$, is less than 0. In our circumstance, the block size is so large that the preparation time for connection, $\lceil \log_2 k \rceil t_s$, can be ignored safely with no impact compared with the network overhead of block transfer. Thus, T_{diff} is larger than 0 and the splitting-based algorithm can achieve a better performance in reducing degraded read latency.

3.4. Prototype Implementation

We implement CRR as an extension of HDFS-RAID. The new algorithms are implemented in the RaidNode and DataNode modules without modifying other modules such as NameNode and JobTracker.

If a client requests an unavailable block, it will receive a remote Block Missing Exception or Checksum Error Exception. Then the underlying Distributed Raid File System (DRFS) will trigger a stripe read operation, which builds input streams for the blocks required to recover the unavailable block. After block reading, it launches decoding operations and returns the decoded results back to the client. As the operation used to recover the block is performed in serial, the read latency will be a significant problem for the client.

The parallel decoding algorithm is implemented within DataNodes and is triggered by DRFS in the case of a degraded read operation. For the convenience of experiments, we add an item to the Raid configuration file to determine which algorithm will be used.

4. Experiments

We build a small cluster to evaluate the performance of CRR-R and CRR-S vs. SRR from two aspects: (i) to demonstrate the validity of the theoretical analysis in Section 3.3 using a single client request, (ii) to apply them to multiple clients to evaluate the performance by simulating more realistic environments.

Experimental Setup: CRR is deployed in a cluster consisting of 14 nodes, a NameNode and thirteen DataNodes, which are linked by 1Gbps network. Each node is equipped with 4 Intel Xeon CPU cores, six 7.2K RPM disks each of 320GB capacity and 2GB of memory, running CentOS 6.5 and the modified HDFS-RAID [4]. We design a fault injector to randomly delete blocks in the system to simulate the block loss scenario and add configurations to arbitrarily switch the reconstruction methods from SRR to CRR. Each result we report is the average of 30 trials. We measure the degraded read latency between the time when a client encounters a missing block and the time when the client finishes reconstructing it and receives the data.

TABLE 1. EXPERIMENTAL PARAMETER SETTINGS

	variable	fixed parameter
Single Client	RS(k,m)	Block Size
	Block Size	RS(k,m)
Multiple Clients	No. of Clients	RS(k,m) and Block Size
	Block Size	RS(k,m) and No. of Clients

Experiment parameters are listed in Table 1, evaluating several RS coding parameters where $(k,m) \in \{(4,2),(6,2),(8,2),(10,2),(12,2)\}$ and the block size ranges from 32MB to 192MB with varying numbers of clients.

4.1. Degraded Read Performance

To evaluate the performance of CRR method without interference, we conducted a serial of experiments by comparing the degraded latency of CRR with SRR in single client requests mode.

Different RS Codes: We run the experiment with the block size of 64MB and RS coding parameters varying. After randomly deleting blocks, we launch a client to read the lost blocks and measure the average degraded read latency in SRR, CRR-R and CRR-S.

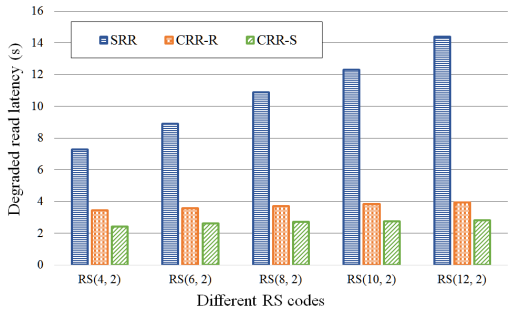

Figure 3. Degraded read latency in different RS codes.

As we see in Figure 3, as the coding parameters vary, the cost of degraded read under SRR shows an liner growth as the number of blocks required in reconstruction grows, while for CRR, we see a slight increase consistent with the logarithmic growth predicted by theory in Section 3.3. Specially, CRR-R improves the degraded read latency by a factor of 55% to 73%, and using the CRR-S method during recovery improves the performance by 68% to 81%. Theory predicts an improvement of 74.6% for CRR-R and 74.6%

for CRR-S, consistent with the experiment. We conclude that CRR reduces degraded read latency regardless of different coding parameters.

Different Block Size: Figure 4 shows the evaluation of the degraded read latency as the block size varies from 32MB to 192MB while maintaining the same erasure code RS(6,2). We can infer that as block size increases, the degraded read

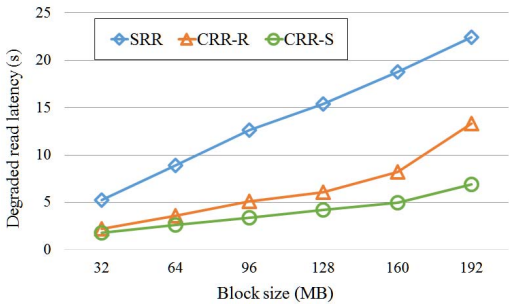

Figure 4. Degraded read latency with RS(6,2) as the block size varies.

latency also increases and that of CRR-R and CRR-S grows slower than SRR. Specially, CRR-R improves the performance by a factor of 57% to 62%, and using the CRR-S method during recovery improves the performance by a factor of 67% to 75% compared with SRR. As expected, CRR improves the performance as the block size changes.

Theoretically, a large block size would mean that network transfers dominate over recovery costs. Therefore, as one might expect that, with certain limits, the larger the block size, the better the degraded read latency. However, Figure 4 does not show this trend. This is mainly because in our experiments, the smallest data block size is 32MB, which is large enough (second level) to overwhelm other factors (millisecond level) such as preparation time for transfer, computing time, and so on.

In general, a request for an unavailable block results in reading multiple blocks for reconstruction, which dominates the recovery costs. Using a parallel instead of serial algorithm can reduce this component, thereby improving the degraded read performance.

4.2. Multiple Clients

In a realistic setting, the requests for data blocks in a storage system are often initiated from multiple clients simultaneously, rather than just one and normal read and degraded read both occur. It should be noted that according to our experiment, the block size is large enough to saturate the network link even with a small number of clients in our small cluster.

Different Block Sizes: We evaluate the cost of degraded read latency with the block size varying from 32MB to 192MB while the number of clients is fixed at 3 and the erasure code is RS(6,2). We set the ratio of normal read to degraded read as 30 : 1. We compare the average read

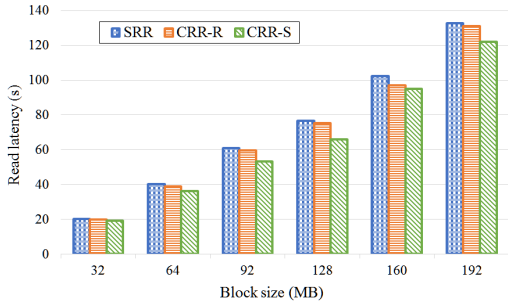

Figure 5. Read latency with multiple clients (3).

latency (including normal read and degraded read) of SRR, CRR-R and CRR-S under different block sizes.

Figure 5 plots the read latency as the block size varies. We make the following observations. First, as the block size increases, the average read latency increases just as in Figure 4. Second, compared to the SRR method, CRR-R and CRR-S all reduce the read latency. Specifically, CRR-R improves the performance by a factor of 2.9% to 9.7%, and CRR-S improves the performance by a factor of 6.7% to 19%. The improvement is not as significant as that in single client mode, which is because that only 1 request in 31 is for a corrupted block in the multi-client environment, that is, most of the requests are launched for available data blocks, where the advantage of CRR over SRR is less significant.

5. Related Work

Petabytes of storage is becoming common with the fast growing data requirements of modern systems today. Erasure codes offer an attractive means of providing lower storage overhead than data replication. As a result, many distributed storage systems [4–6, 14] are moving to the use of erasure codes. However, erasure codes have the problems of a high degraded read latency and longer reconstruction time.

Some researchers have aimed to reduce the amount of data that need to download during the reconstruction of common failure scenario (i.e. single failure) at the cost of additional storage overhead in the recent past [6, 10]. The optimization of encoding or decoding has also been carried out [15–17], which requires no extra storage and less network transfer compared to classical erasure codes. In contrast, CRR improves degraded read latency by parallelizing the recovery among all involved nodes and thus reducing degraded latency greatly and CRR can be employed in conjunction with almost all kinds of erasure-coded systems. PPR [8], which shares a similar idea of parallelization but focuses on scheduling the reconstruction operations more efficiently by using a distributed protocol while CRR aims to design multiple reduction-based methods to reduce degraded read latency.

There is some research [9, 18, 19] focusing on proactive fault tolerance methods which are able to migrate data on

the soon-to-fail disks to other healthy ones in advance. Therefore, when the failure happens, the data on the failed drives can be accessed directly from the replaced disks. For example, RAIDShield [9] uses a fixed threshold-based predictor to determine the soon-to-fail disks and replace them with good ones while IDO [20] proactively migrates data of hot zones which bear the most requests to surrogate RAID to improve RAID reconstruction performance. By combining these methods with CRR, we can improve their performance by parallelizing read operations to the unavailable blocks missed by the predictor when failures occur.

6. Conclusions

Large scale storage systems are increasingly transitioning to erasure coding instead of replication due to its higher reliability and lower storage cost. However, this brings about two major problems: high degraded read latency and long reconstruction time when failure occurs. In this paper, we explore the performance of using parallel disk read, collective communication and computation, instead of the serial ones in erasure coding storage systems. We present CRR, designed for improving the performance of requesting for unavailable blocks from clients, reducing the degraded read latency significantly. In the future, we plan to deploy the idea of CRR to other kinds of erasure coding systems so as to speed up the read latency of corrupted blocks and even the reconstruction of node or disk failures as well.

References

[1] L. N. Bairavasundaram, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, G. R. Goodson, and B. Schroeder, "An analysis of data corruption in the storage stack," *ACM Transactions on Storage (TOS)*, vol. 4(3), 2008.

[2] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 29–43, Oct. 2003.

[3] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proc. MSST*, 2010, pp. 1–10.

[4] "HDFS RAID," http://wiki.apache.org/hadoop/HDFS-RAID, [Online; accessed 14-September-2015].

[5] "Colossus, successor to Google file system," http://static.googleusercontent.com/media/research.google.com/en/us/university/relations/facultysummit2010/storage_architecture_and_challenges.pdf, [Online; accessed 14-September-2015].

[6] 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 *USENIX Annual Technical Conference*, 2012, pp. 15–26.

[7] K. V. 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." in *Proc. ACM SIGCOMM*, 2014.

[8] S. Mitra, R. Panta, M.-R. Ra, and S. Bagchi, "Partial-parallel-repair (PPR): a distributed technique for repairing erasure coded storage," in *Proc. ACM EuroSys*, 2016, p. 30.

[9] A. Ma, R. Traylor, F. Douglis, M. Chamness, G. Lu, D. Sawyer, S. Chandra, and W. Hsu, "RAIDShield: characterizing, monitoring, and proactively protecting against disk failures," *ACM Transactions on Storage (TOS)*, vol. 11, no. 4, p. 17, 2015.

[10] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. Dimakis, R. Vadali, S. Chen, and D. Borthakur, "Xoring elephants: Novel erasure codes for big data," in *Proc. VLDB Endowment*, 2013.

[11] I. Reed and G. Solomon, "Polynomial codes over certain finite fields," *J. SIAM*, vol. 2, pp. 300–304, 1960.

[12] J. S. Plank, "A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems," *Softw., Pract. Exper.*, vol. 27, no. 9, pp. 995–1012, 1997.

[13] B. Schroeder and G. A. Gibson, "Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you?" in *Proc. FAST*, vol. 7, 2007, pp. 1–16.

[14] "Openstack Swift," http://docs.openstack.org/developer/swift/overview_erasure_code.html, [Online; accessed 5-December-2015].

[15] L. Xiang, Y. Xu, J. Lui, and Q. Chang, "Optimal recovery of single disk failure in RDP code storage systems," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, no. 1, 2010, pp. 119–130.

[16] Z. Wang, A. Dimakis, and J. Bruck, "Rebuilding for array codes in distributed storage systems," in *Proc. ACTEMT*, 2010.

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

[18] A. Qin, D. Hu, J. Liu, W. Yang, and D. Tan, "Fatman: Cost-saving and reliable archival storage based on volunteer resources," in *Proc. VLDB Endowment*, vol. 7, no. 13, 2014, pp. 1748–1753.

[19] X. Ji, Y. Ma, R. Ma, P. Li, J. Ma, G. Wang, X. Liu, and Z. Li, "A proactive fault tolerance scheme for large scale storage systems," in *Algorithms and Architectures for Parallel Processing*. Springer, 2015, pp. 337–350.

[20] S. Wu, H. Jiang, and B. Mao, "Proactive data migration for improved storage availability in large-scale data centers," *IEEE Transactions on Computers*, vol. 64, no. 9, pp. 2637–2651, 2015.