Repair Pipelining for Erasure-Coded Storage

Runhui Li, Xiaolu Li, Patrick P. C. Lee, Qun Huang
The Chinese University of Hong Kong
lrhdiy@gmail.com, {lixl,pclee}@cse.cuhk.edu.hk, huangqundl@gmail.com

Abstract

We propose *repair pipelining*, a technique that speeds up the repair performance in general erasure-coded storage. By pipelining the repair of failed data in small-size units across storage nodes, repair pipelining reduces the repair time to approximately the same as the normal read time to the same amount of data in homogeneous environments. We further extend repair pipelining for heterogeneous environments. We implement a repair pipelining prototype called ECPipe and integrate it as a middleware system into two open-source distributed storage systems HDFS and QFS. Experiments on a local testbed and Amazon EC2 show that repair pipelining significantly improves the performance of both degraded reads and full-node recovery over existing repair techniques.

1 Introduction

Distributed storage systems rely on data redundancy to provide fault tolerance, so as to maintain availability and durability. *Replication*, which is traditionally used by production systems [4, 11], provides the simplest form of redundancy by keeping identical copies of data in different storage nodes. However, the raw storage cost of replication is overwhelming, especially with the massive scale of data we face today. *Erasure coding* provides a low-cost redundancy alternative that incurs significantly lower storage overhead than replication at the same fault tolerance level [39]. In a nutshell, erasure coding transforms fixed-size units, called *blocks*, of original data into a set of coded blocks, such that any subset of a sufficient number of available coded blocks can reconstruct all original data. Today's distributed storage systems adopt erasure coding to protect data against failures in clustered [10, 15, 29] or geo-distributed environments [21, 33], and reportedly save PBs of storage [15, 21].

Although achieving storage efficiency, erasure coding has a drawback of incurring high repair penalty. Specifically, the repair of a single failed coded block (either lost or unavailable) needs to read multiple available coded blocks for reconstruction; in other words, it reads more available data than the actual amount of failed data. This is in contrast to replication, whose repair can be simply done by reading another replica that is of the same size as the failed block. The excessive data not only increases the read time to failed data as opposed to normal reads, but also consumes bandwidth resources that could otherwise be made available for other foreground jobs [29]. Thus, erasure coding in practice is mainly used for storing less frequently read (i.e., warm/cold) data that needs long-term persistence [2, 15, 21], while frequently read (i.e., hot) data remains replicated for efficient access. To mitigate the repair penalty of erasure coding, prior studies either propose new erasure codes that reduce the amount of repair traffic (e.g., [8, 15, 17, 25, 28, 30, 34]), or design fast repair approaches for existing erasure codes (e.g., lazy repair [3, 37] or partial-parallel-repair (PPR) [20]). While the repair time is effectively reduced, it remains higher than the normal read time in general. In view of this, we pose the following question: *Can we further reduce the repair time of erasure coding to almost the same as the normal read time?* This creates opportunity for applying erasure coding to hot data for high storage efficiency, while preserving read performance.

We present a new technique called *repair pipelining* to speed up the repair performance in general erasure-coded storage. Its main idea is to pipeline the repair of a coded block in small-size units across storage nodes (analogous to wormhole routing [22]), so as to distribute repair traffic and fully utilize bandwidth resources across storage nodes. Contrary to the conventional wisdom that the repair of erasure coding is a slow operation, repair pipelining can reduce the repair time of a failed coded block to approximately the same as the read time of a normal coded block, regardless of coding parameters, in homogeneous environments (i.e., link bandwidths are identical). It is also general to support various practical erasure codes that are adopted by today's production systems, including classical Reed-Solomon codes [32] and recent Local Reconstruction Codes [15]. To summarize, we make the following contributions.

- We design repair pipelining to address two types of repair operations: degraded reads and full-node recovery. We show that repair pipelining achieves $O(1)$ repair time in homogeneous environments.
- We extend repair pipelining to address heterogeneous environments (i.e., link bandwidths are different). We present two variants of repair pipelining. The first one allows parallel reads of reconstructed data when the bandwidth between the storage system and the node that issues repair is limited, while the second one finds

an optimal repair path across storage nodes such that the repair time is minimized.

- We implement a repair pipelining prototype called ECPipe, which runs as a middleware layer atop an existing storage system and performs repair operations on behalf of the storage system. As a proof of concept, we integrate ECPipe into two widely adopted open-source distributed storage systems HDFS [36] and QFS [24]. Both integrations only make minor changes (with no more than 200 lines of code) to the code base of each storage system.
- We evaluate repair pipelining on a local cluster and two geo-distributed Amazon EC2 clusters (one in North America and one in Asia). We compare it with two existing repair approaches: conventional repair that is used by classical Reed-Solomon codes [32] and achieves $O(k)$ repair time, and the recently proposed PPR [20] that achieves $O(\log k)$ repair time by parallelizing partial repair operations in a hierarchical manner (§2.2). Our experiments show that in many cases, repair pipelining reduces the single-block repair time by around 90% and 80% compared to conventional repair and PPR, respectively. It also improves repair performance in HDFS and QFS deployments.

2 Background and Motivation

2.1 Basics

We consider a distributed storage system (e.g., GFS [11], HDFS [36], and Azure [4]) that manages large-scale datasets and stores files as fixed-size *blocks*, which form the basic read/write units. The block size is often large, ranging from 64 MiB [11] to 256 MiB [30], to mitigate I/O overhead. Erasure coding is applied to a collection of blocks. Specifically, an erasure code is typically configured with two integer parameters (n,k), where $k < n$. An (n,k) code divides blocks into groups of k. For every k (uncoded) blocks, it encodes them to form n coded blocks, such that any k out of n coded blocks can be decoded to the original k uncoded blocks. The set of n coded blocks is called a *stripe*. A large-scale storage system stores data of multiple stripes, all of which are independently encoded. The n coded blocks of each stripe are distributed across n distinct nodes to tolerate any $n-k$ node failures. Most practical erasure codes are *systematic*, such that k of n coded blocks are identical to the original uncoded blocks and hence can be directly accessed without decoding. Nevertheless, our design treats both uncoded and coded blocks the same, so we simply refer to them as "blocks".

Many erasure code constructions have been proposed in the literature (see survey [26] and §7). Among all erasure codes, Reed-Solomon (RS) codes [32] are the most popular erasure codes that are widely deployed in production [10, 24, 29]. Note that RS codes achieve the minimum storage redundancy among any (n,k) codes, and are said to be *maximum distance separable (MDS)*. Some erasure codes used in production, such as locally repairable codes [15,34], introduce slightly higher redundancy than RS codes for better repair performance.

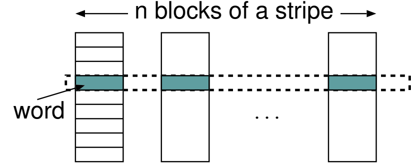


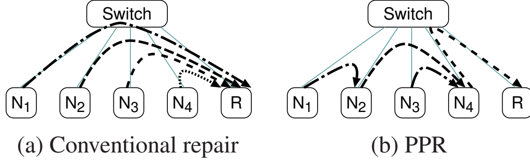
Figure 1: In erasure coding, blocks are partitioned into words, such that words at the same offset of each block of a stripe are encoded together.

Practical erasure codes, including RS codes and locally repairable codes, satisfy *linearity*. Specifically, for each stripe of an (n,k) code, let $\{B_1, B_2, \cdots, B_k\}$ denote any k blocks of a stripe. Any block in the same stripe, say B^*, can be computed from a linear combination of the k blocks as $B^* = \sum_{i=1}^{k} a_i B_i$, where a_i's ($1 \leq i \leq k$) are decoding coefficients specified by a given erasure code. All additions and multiplications are based on Galois Field arithmetic over w-bit units called *words*; in particular, an addition is equivalent to bitwise XOR. Note that the additions of $a_i B_i$'s are associative. Some constraints may be applied; for example, RS codes require $n \leq 2^w + 1$ [27]. Each block is partitioned into multiple w-bit words, such that the words at the same offset of each block of a stripe are encoded together, as shown in Figure 1.

2.2 Repair

In this paper, repair in erasure-coded storage can refer to one of the following: (i) *full-node recovery* for restoring lost blocks (e.g., due to disk crashes, sector errors, etc.), or (ii) *degraded reads* to temporarily unavailable blocks (e.g., due to power outages, network disconnection, system maintenance, etc.) or lost blocks that are yet recovered. Each failed block (either lost or unavailable) is reconstructed on a destination termed *requestor*, which can be a new node that replaces a failed node, or a client that issues degraded reads. Note that there may be one or multiple requestors when multiple failed blocks are reconstructed.

Erasure coding triggers more repair traffic than the size of failed data to be reconstructed. For example, for (n,k) RS codes, repairing a failed block reads k available blocks of the same stripe from other nodes (i.e., k times the block size). Some repair-friendly erasure codes (e.g., [8, 15, 17, 25, 28, 30, 34]) are designed to reduce repair traffic, but the size of repair traffic per block remains larger than the size of a block. In distributed storage sys-



(a) Conventional repair (b) PPR

Figure 2: Examples of conventional repair and PPR.

tems, network bandwidth is often the most dominant factor in repair performance as extensively shown by previous work [8, 20, 37] (see further justifications in §2.3). Thus, the amplification of repair traffic implies the congestion at the downlink of the requestor, thereby increasing the overall repair time.

To understand the repair penalty of erasure coding, we use RS codes as an example and call this repair approach *conventional repair*. Suppose that a requestor R wants to repair a failed block B^*. It can be done by reading k available blocks from any k working nodes, called *helpers*. Without loss of generality, let R contact k helper nodes $N_1, N_2, \cdots, N_k$, which store available blocks B_1, $B_2, \cdots, B_k$, respectively. To make our discussion clear, we divide the repair process into *timeslots*, such that only one block can be transmitted across a network link in each timeslot. Figure 2(a) shows the conventional repair for $k = 4$. Since R needs to retrieve the k blocks B_1, B_2, $\cdots$, B_k, all k transmissions must traverse the downlink of R. Overall, the repair takes four timeslots.

The drawback of conventional repair is that the bandwidth usage distribution is highly skewed: the downlink of the requestor is highly congested, while the links among helpers are not fully utilized. PPR [20] builds on the linearity and addition associativity of erasure coding by decomposing a repair operation into multiple partial operations that are distributed across all helpers. This distributes bandwidth usage across the links of helpers. Figure 2(b) shows how PPR repairs B^* for $k = 4$. In the first timeslot, N_2 and N_4 receive blocks a_1B_1 and a_3B_3 from N_1 and N_3, respectively. Since the transmissions use different links, they can be done simultaneously in a single timeslot. In the second timeslot, N_2 combines the received a_1B_1 and its locally stored block B_2 to obtain $a_1B_1 + a_2B_2$ and sends it to N_4. In the third timeslot, N_4 combines all received blocks and its own block B_4 to obtain $a_1B_1 + a_2B_2 + a_3B_3 + a_4B_4$, and sends it to R. This hierarchical approach reduces the overall repair time to only three timeslots. In general, PPR needs $\lceil \log_2(k+1) \rceil$ timeslots to repair a failed block.

2.3 Motivation

Although PPR reduces repair time, the bandwidth usage distribution remains not fully balanced; for example, the downlink of N_4 in Figure 2(b) still carries more repair traffic than other links. Thus, the repair time is still bottlenecked by the link with the most repair traffic. This motivates us to design a new repair scheme that can more efficiently utilize bandwidth resources, with the primary goal of minimizing repair time.

Minimizing repair time is critical to both availability and durability. In terms of availability, field studies show that transient failures (i.e., no data loss) account for over 90% of failure events [10]. Thus, most repairs are expected to be degraded reads rather than full-node recovery. Since degraded reads are issued when clients request unavailable data, achieving fast degraded reads not only improves availability but is also critical for meeting customer service-level agreements [15]. In terms of durability, minimizing repair time also minimizes the window of vulnerability before unrecoverable data loss occurs.

Our work targets distributed storage environments in which network bandwidth is the bottleneck. Although modern data centers scale to 10Gb/s or higher speeds, they are shared by a mix of application workloads. Thus, the network bandwidth available for repair tasks is often throttled [15, 37]. Also, the cross-rack links of modern data centers are oversubscribed [5], yet blocks are striped across racks to tolerate rack failures [10, 15, 30, 34]. Repair of failed blocks inevitably reads available blocks from other racks, and its performance becomes constrained by the limited cross-rack bandwidth.

3 Repair Pipelining

We present the design of repair pipelining for both degraded reads and full-node recovery.

3.1 Goals and Assumptions

Repair pipelining also exploits the linearity and addition associativity of erasure codes as in PPR [20], yet it parallelizes the repair across helpers in an inherently different way. It focuses on (i) eliminating bottlenecked links (i.e., no link transmits more traffic than others) and (ii) effectively utilizing bandwidth resources during repair (i.e., links should not be idle for most times), so as to ultimately achieve $O(1)$ repair time in homogeneous environments where all links have the same bandwidth. In addition, we show that repair pipelining can be extended for practical distributed environments with heterogeneous links (§4), which are not addressed by PPR.

Repair pipelining is designed for speeding up the repair of a single failed block per stripe, which accounts for the most repair scenarios in practice [15, 29] (e.g., over 98% of cases [29]). If a stripe has multiple failed blocks, we trigger a multi-failure repair, in which we resort to conventional repair (§2.2) by reading a sufficient number of available blocks. Optimizing single-block repair is also the main design goal of repair-friendly erasure codes [8, 15, 17, 25, 28, 30, 34]. In this paper, we study the single-block repair for one stripe and multiple stripes. The former occurs when a requestor issues a de-

graded read to an unavailable block, which are the majority (§2.3); the latter occurs when all lost data of a single failed node is recovered at one or multiple requestors in full-node recovery.

As in PPR, we do not design new repair-friendly erasure codes that minimize repair traffic; instead, each repair of a single failed block still reads k blocks, yet it spreads the repair traffic across all helpers to fully utilize bandwidth resources and reduce the overall repair time.

3.2 Degraded Reads

We first study how repair pipelining reconstructs a single block of a stripe at a requestor in a degraded read. We start with a naïve approach. Specifically, we arrange k helpers and the requestor as a linear path, i.e., $N_1 \rightarrow N_2 \rightarrow \cdots \rightarrow N_k \rightarrow R$. At a high level, to repair a lost block B^*, N_1 sends $a_1 B_1$ to N_2. Then N_2 combines $a_1 B_1$ with its own block B_2 and sends $a_1 B_1 + a_2 B_2$ to N_3. The process repeats, and finally, N_k sends R the combined result, which is B^*. The whole repair incurs k transmissions that span across k different links. Thus, there is no bottlenecked link. However, this naïve approach underutilizes bandwidth resources, since there is only one block-level transmission in each timeslot. The whole repair still takes k timeslots, same as the conventional repair (§2.2).

Thus, repair pipelining decomposes the repair of a block into the repair of a set of s small fixed-size units called *slices* $S_1, S_2, \cdots, S_s$. It pipelines the repair of each slice through the linear path, and each slice-level transmission over a link only takes $\frac{1}{s}$ timeslots. Figure 3 shows how repair pipelining works for $k = 4$ and $s = 6$.

A slice can have an arbitrarily small size, provided that Galois Field arithmetic can be performed (§2.1). For RS codes, the minimum size of a slice is a w-bit word; if $w = 8$, a word denotes a byte. On the other hand, practical distributed storage systems store data in large-size blocks, typically 64 MiB or even larger (§2.1). Since a coding unit (i.e., word) has a much smaller size than a read/write unit (i.e., block), we can parallelize a block-level repair operation into more fine-grained slice-level repair sub-operations. Having small-size slices can improve parallelism, but also increases the overhead of issuing many requests for transmitting slices over the network. We study the impact of the slice size in §6.

We analyze the time complexity of repair pipelining. Here, we neglect the overheads due to computation and disk I/O, which we assume cost less time than network transmission; in fact, they can also be executed in parallel with network transmission in actual implementation (§5). Each slice-level transmission over a link takes $\frac{1}{s}$ timeslots. The repair of each slice takes $\frac{k}{s}$ timeslots to traverse the linear path, and N_1 starts to transmit the last slice after $\frac{s-1}{s}$ timeslots. Thus, the whole repair time,

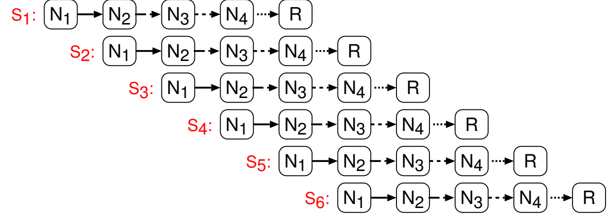


Figure 3: Repair pipelining with $k = 4$ and $s = 6$.

which is given by the total number of timeslots to transmit all slices through the linear path, is $\frac{s-1+k}{s} = 1 + \frac{k-1}{s}$ timeslots. In practice, k is of moderate size to avoid large coding overhead [27] (e.g., $k = 12$ in Azure [15] and $k = 10$ in Facebook [29]), while s can be much larger (e.g., $s =$ 2,048 for 32 KiB slices in a 64 MiB block). Thus, we have $1 + \frac{k-1}{s} \rightarrow 1$ as s is sufficiently large.

Repair pipelining connects multiple helpers as a chain, so its repair performance is degraded by the presence of poorly performed links/helpers (i.e., stragglers). We emphasize that any repair scheme of erasure coding faces the similar problem, as it needs available data from multiple helpers for data reconstruction; for example, the conventional repair of (n, k) MDS codes needs available data from k helpers. We address the straggler problem by taking into account heterogeneity and bypassing stragglers via helper selection (§4.2). Also, if any helper fails during an ongoing repair, the progress of repair pipelining will be stalled. In this case, we restart the whole repair process with a new set of available helpers and trigger a multi-failure repair (§3.1), since the repaired stripe now has multiple failed blocks; however, multi-failure repairs are rare in practice [15, 29].

3.3 Full-Node Recovery

We now study how repair pipelining addresses multi-stripe repair (one failed block per stripe) when recovering a full-node failure. As the stripes are independently encoded, we can parallelize the multiple single-stripe repair operations. However, since each repair involves a number of helpers, if one helper is chosen in many repair operations of different stripes, it will become overloaded and slow down the overall repair performance. In practice, each stripe is stored on a different set of storage nodes spanning across the network. Our goal is to distribute the load of a multi-stripe repair across all available helpers as evenly as possible.

We adopt a simple greedy scheduling approach for the selection of helpers. For each node in the storage system, repair pipelining keeps track of a timestamp indicating when the node was last selected as a helper for a single-stripe repair. To repair a failed block of a stripe, we select k out of $n-1$ available helpers in the stripe that have the smallest timestamps; in other words, the k se-

lected helpers are the least recently selected in previous requests. Choosing the k out of the $n-1$ helpers can be done in $O(n)$ time using the quick select algorithm [13] (based on repeated partitioning of quick sort). We use a centralized coordinator to manage the selection process (§5). Our greedy scheduling emphasizes simplicity in deployment. We can also adopt a more sophisticated approach by weighting node preferences in real time [20].

Unlike the degraded read scenario, the multiple reconstructed blocks can be stored on multiple requestors. Under this condition, the gain of repair pipelining over conventional repair decreases, as the latter can also parallelize the repair across multiple requestors. Nevertheless, our evaluation indicates that repair pipelining still provides repair performance improvements (§6).

Note that the number of requestors that can be selected and the choices of requestors may also depend on various deployment factors [20]. In this work, we assume that the requestors are selected offline in advance.

4 Heterogeneity

In practice, the links of a distributed storage system have different bandwidths [9, 18]. We now extend the design of repair pipelining in §3 in two aspects: (i) a requestor can read slices from multiple helpers in parallel, and (ii) we solve a weighted path selection problem to find an optimal path of k helpers that maximizes repair performance. Each extension addresses a different heterogeneous setting.

4.1 Parallel Reads

In the original design of repair pipelining, a requestor always reads slices from one helper. This may lead to last-mile congestion. For example, a client (requestor) sits at the network edge and accesses a cloud storage system that is far from the client. We propose a *cyclic version* of repair pipelining that allows a requestor to read slices from multiple helpers.

We now describe the cyclic version. Our discussion assumes that all links are homogeneous and it takes one timeslot to transmit a block size of data in a link. The cyclic version again divides a failed block into s fixed-size slices S_1, S_2, $\cdots$, S_s, and repairs each slice through some linear path to eliminate any bottlenecked link. However, it now maps the k helpers N_1, N_2, $\cdots$, N_k into different *cyclic paths* that can be cycled from N_k through N_1. Specifically, it partitions the s slices into $\lceil \frac{s}{k-1} \rceil$ groups, each of which has $k-1$ slices (the last group has fewer than $k-1$ slices if s is not divisible by $k-1$). The repair of each group of slices is then performed in two phases. Without loss of generality, we only consider how to repair the first group S_1, S_2, $\cdots$, S_{k-1}. In the first phase, repairing each slice S_i $(1 \leq i \leq k-1)$ traverses through the cyclic path

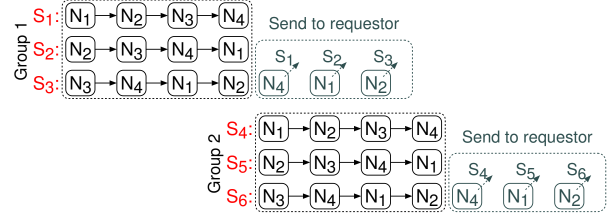


Figure 4: Cyclic version of repair pipelining with $k=4$ and $s=6$.

$N_i \to N_{i+1} \to \cdots N_k \to N_1 \to \cdots N_{i-1}$. We repair all slices through different cyclic paths simultaneously, and each slice-level transmission takes $\frac{1}{s}$ timeslots. The first phase can be done in $\frac{k-1}{s}$ timeslots. In the second phase, the last helper of each cyclic path delivers the repaired slice to the requestor. The second phase is also done in $\frac{k-1}{s}$ timeslots. Figure 4 shows the cyclic version for $k=4$ and $s=6$.

Note that we can start repairing the slices of the next group simultaneously while we deliver the repaired slices for the current group. Specifically, while $k-1$ helpers simultaneously transmit slices for the repair in the next group, there is one idle helper that can transmit the repaired slice for the current group to the requestor. They can be done together in $\frac{k-1}{s}$ timeslots.

We analyze the time complexity of the cyclic version under the homogeneous link assumption. We only consider the case where s is divisible by $k-1$, while the same result can be derived otherwise. Repairing each group of slices takes $\frac{2(k-1)}{s}$ timeslots, and the repair of the last group starts after $(\frac{s}{k-1}-1)\frac{k-1}{s}$ timeslots. The whole repair time is $(\frac{s}{k-1}-1)\frac{k-1}{s}+\frac{2(k-1)}{s}=1+\frac{k-1}{s}\to 1$, as s is sufficiently large.

Note that the cyclic version now allows a requestor to read slices from $k-1$ helpers. If the repair bottleneck lies on the network transfer from the helpers to the requestor, our evaluation shows that the cyclic version significantly outperforms the original design of repair pipelining (§6).

4.2 Weighted Path Selection

We now study a more general heterogeneous setting in which link bandwidths can have arbitrary values. To motivate, we consider geo-distributed data centers that span multiple geographic regions [1, 10]. They typically stripe redundancy across regions to protect against large-scale correlated failures. However, intra- and inter-region bandwidths are highly different. Table 1 shows one of our `iperf` [16] measurement tests for the intra- and inter-region bandwidths on Amazon EC2 across four regions respectively in North America and Asia. We observe that intra-region bandwidths are in general more abundant than inter-region bandwidths, and inter-region bandwidths have a high degree of variance.

Table 1: A test of intra- and inter-region bandwidth measurements (in Mb/s) on Amazon EC2 in North America and Asia. Each number is the measured bandwidth from the row region to the column region.

(a) North America

Bandwidth	California	Canada	Ohio	Oregon
California	501.3	57.2	44.1	299.9
Canada	55.3	732.0	63.3	48.0
Ohio	46.3	65.7	332.5	95.6
Oregon	297.8	50.2	93.6	250.1

(b) Asia

Bandwidth	Mumbai	Seoul	Singapore	Tokyo
Mumbai	624.8	62.3	39.5	37.7
Seoul	63.8	265.7	86.1	183.2
Singapore	41.5	88.1	493.0	49.1
Tokyo	39.7	181.0	46.9	489.1

In the following, we extend repair pipelining to solve a *weighted path selection* problem. We focus on extending the design for the single-block repair (of a single stripe) for degraded reads (§3.2). We later discuss how our extended design is applied to full-node recovery (§3.3).

4.2.1 Formulation

Recall that for a single-block repair, repair pipelining transmits a number of slices along a path of k helpers, say $N_1 \to N_2 \to \cdots \to N_k \to R$. Suppose that the link bandwidths are different. If the number of slices is sufficiently large, then the slices are transmitted in parallel through the path (Figure 3), and the performance of repair pipelining will be bottlenecked by the link with the minimum available bandwidth along the path. To minimize the single-block repair time, we should find a path that maximizes the minimum link bandwidth.

To repair a failed block, we need to find k out of $n-1$ available helpers of the same stripe as the failed block, and also find the sequence of link transmissions so that the path along the k selected helpers and the requestor minimizes the single-block repair time. Specifically, there are a total of n nodes, including the $n-1$ available helpers and the requestor. We associate a *weight* with each (directed) link from one node to another node, such that a higher weight implies a longer transmission time along the link. For example, the weight can be represented by the inverse of the link bandwidth obtained by periodic measurements on link utilizations [5]. Then our objective is to find a path of $k+1$ nodes (i.e., k selected helpers and the requestor) that minimizes the maximum link weight of the path. Here, we focus on link weights, and the same idea is applicable if we associate weights with nodes. Any straggler is assumed to be associated with a large weight, so it will be excluded from the selected path.

To solve the above problem, a naïve approach is to perform a brute-force search on all possible candidate paths. However, there are a total of $\frac{(n-1)!}{(n-1-k)!}$ permutations, and the brute-force search becomes computationally expensive even for moderate sizes of n and k. Since the link weights vary over time, the path selection should be done quickly on-the-fly based on the measured link weights.

Algorithm 1 Weighted Path Selection

Input: link weights
Output: optimal path P^*

```
1: procedure MAIN
2:    P = R
3:    P* = null
4:    w* = ∞
5:    𝒩 = set of n − 1 available helpers
6:    EXTENDPATH
7:    return P*
8: end procedure
9: function EXTENDPATH
10:   if P.length < k + 1 then
11:      for each node N ∈ 𝒩 not in P do
12:         if weight(N, head node of path P) < w* then
13:            P = N → P
14:            EXTENDPATH
15:            remove N from P
16:         end if
17:      end for
18:   else
19:      P* = P
20:      w* = maximum link weight of P
21:   end if
22: end function
```

4.2.2 Algorithm

We present a fast yet optimal algorithm that quickly identifies an optimal path. The algorithm builds on brute-force search to ensure that all candidate paths are covered, but eliminates the search of infeasible paths. Our insight is that if a link L has a weight larger than the maximum weight of an optimal path candidate that is currently found, then we no longer need to search for the paths containing link L, since the maximum weight of any path containing L must be larger than the maximum weight of the optimal path candidate.

Algorithm 1 shows the pseudo-code of the weighted path selection algorithm. Let P be the path that we currently consider, P^* be the optimal path candidate that we have found, w^* be the maximum link weight of P^*, and $\mathcal{N}$ be the set of $n-1$ available helpers. We first initialize a path P with only the requestor R (Line 2), such that R will be the tail node of P. We also initialize P^*, w^*, and $\mathcal{N}$ (Lines 3-5). We call the recursive function EXTENDPATH (Line 6) and finally return the optimal path P^* (Line 7).

The function EXTENDPATH recursively extends P by

one node in $\mathcal{N}$ and appends the node to the head of P if the link weight from the node to the current head node of P is less than w^*; otherwise, the path containing the link cannot minimize the maximum link weight as argued above. Specifically, the algorithm appends $N \in \mathcal{N}$ to P if the current path length is less than $k+1$ and the weight from N to the head node of P is less than w^* (Lines 10-13). It calls EXTENDPATH again to consider candidate paths that now include $N \rightarrow P$ (Line 14). It then removes N from P (Line 15), and tries other nodes in $\mathcal{N}$. If the length of P is now $k+1$, it implies that all of its links have weight less than w^*, so we update P as the new optimal path P^* and w^* as the maximum link weight of P^* (Lines 19-20).

Algorithm 1 significantly reduces the search time. We evaluate the search time for (14,10) codes using Monte-Carlo simulations over 1,000 runs on a machine with 3.7 GHz Intel Xeon E5-1620 v2 CPU and 16 GiB memory. The brute-force search takes 27s on average, while Algorithm 1 reduces the search time to only 0.9ms.

4.2.3 Discussion

Algorithm 1 also addresses full-node recovery (§3.3). Specifically, we apply Algorithm 1 to each stripe. If we apply greedy scheduling on helper selection, we simply substitute $\mathcal{N}$ with the set of k selected helpers. Note that the brute-force search for the optimal path on the k selected helpers remains expensive, since it still needs to consider $k!$ permutations on the sequence of link transmissions along the path. Thus, Algorithm 1 still significantly saves the search time in this case.

We can also apply Algorithm 1 to the cyclic version in §4.1. Instead of searching for an optimal path, we now search for an optimal cycle of k helpers that minimizes the maximum link weight.

5 Implementation

We implemented a prototype called ECPipe to realize repair pipelining. ECPipe runs as a middleware atop an existing storage system and performs repair operations on behalf of the storage system. Moving the repair logic to ECPipe greatly reduces changes to the code base of the storage system to realize new repair techniques, while we focus on optimizing ECPipe to maximize the repair performance gain. We have integrated ECPipe with two widely deployed distributed storage systems HDFS [36] and QFS [24]. HDFS is written in Java, while QFS is written in C++. Our ECPipe prototype is mostly written in C++, and the part for HDFS integration is in Java. Our ECPipe prototype has around 3,000 lines of code.

5.1 Erasure Coding in HDFS and QFS

HDFS: Erasure coding in HDFS is done by the HDFS-RAID module [12]. HDFS-RAID deploys a *RaidNode* atop HDFS for erasure coding management. HDFS initially stores data as fixed-size blocks (64 MiB by default) with replication; later, the RaidNode encodes replicated blocks into coded blocks via MapReduce [7]. The RaidNode also checks for any lost or corrupted coded block (by verifying block checksums). If so, it repairs the failed blocks, either by itself in local mode or via a MapReduce job in distributed mode. Both modes will issue reads to k available blocks of the same stripe in parallel from HDFS, reconstruct the failed block, and write back to HDFS. HDFS-RAID also provides a RAID file system client to access coded blocks. For a degraded read to a failed block, the RAID file system reads k available blocks of the same stripe in parallel and reconstructs the failed block.

QFS: Different from HDFS, which stores data with both replication and erasure coding, QFS stores all data in erasure-coded format. QFS supports (9,6) RS codes [32]. The QFS client writes data into six 1 MiB buffers. When the buffers fill up, it encodes the six 1 MiB buffers into three 1 MiB parity buffers. It then appends the nine 1MB buffers to nine data and parity blocks (the default block size is 64 MiB) that are stored in nine storage nodes. To repair any failed block, a storage node retrieves six available blocks from other storage nodes for reconstruction.

5.2 ECPipe Design

Figure 5 shows the ECPipe architecture. It uses a *coordinator* to manage the repair operation between a requestor and multiple helpers. ECPipe runs on top of a storage system. To repair a failed block, the storage system creates a requestor object, which sends a repair request with the failed block ID to the coordinator (step 1). The coordinator uses the failed block ID to identify the locations of k available blocks of the same stripe. It notifies all helpers with the block locations (step 2). The helpers retrieve the blocks, perform repair pipelining in slices, and deliver the repaired slices to the requestor (step 3).

We integrate ECPipe with a storage system in three aspects. First, we implement the requestor as a class (in C++ and Java) that can be instantiated by the storage system to reconstruct failed blocks. For HDFS, the requestor is created in either the RaidNode or the RAID file system client; for QFS, it is created by the storage node that starts a repair operation. Second, we implement each helper as a daemon that is co-located with each storage node to directly read the locally stored blocks. Our insight is that both HDFS and QFS store each block in the underlying native file system as a plain file, and use the block ID to form the file name. Thus, each helper can directly read the stored blocks through the native file system. This eliminates the need of helpers to fetch data through the distributed storage system routine. It not only reduces the burden of metadata management of the

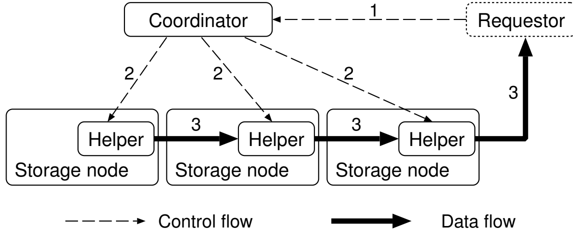


Figure 5: ECPipe architecture.

distributed storage system, but also improves repair performance (§6.3). Finally, the coordinator needs to access both block locations and the mappings of each block to its stripe. For HDFS, we retrieve the information from the RaidNode; for QFS, we retrieve the information from a storage node when it starts a repair operation.

To simplify our implementation, ECPipe uses Redis [31] to pipeline slices across helpers. Each helper maintains an in-memory key-value store based on Redis, and uses the client interface of Redis to transmit slices among helpers. In addition, each helper performs disk I/O, network transfer, and computation via multiple threads for performance speedup. Adding ECPipe into HDFS and QFS only requires changes of around 110 and 180 lines of code, respectively.

To provide fair comparisons (§6), we also implement conventional repair (§2.2) and PPR [20] under the same ECPipe framework, by only changing the transmission flow of data during repair.

6 Evaluation

We conducted experiments on a local cluster and two geo-distributed clusters deployed on Amazon EC2. We show that repair pipelining outperforms both conventional repair and PPR [20], for both degraded reads and full-node recovery.

6.1 ECPipe Performance on a Local Cluster

6.1.1 Methodology

We first evaluate ECPipe when it runs as a standalone system. We conducted experiments on a local cluster of 19 machines, each of which has a quad-core 3.1 GHz Intel Core i5-2400 CPU, 8 GiB RAM, and a Seagate ST31000524AS 1 TiB SATA hard disk. We host the coordinator on one machine, and 18 helpers on the remaining machines. All machines are connected via a 1 Gb/s Ethernet switch. The 1 Gb/s bandwidth can be viewed as modeling the cross-rack bandwidth available for repair tasks in a production cluster [34], in which the blocks of a stripe are stored in distinct racks and there will be cross-rack transfers during repair.

Initially, we store coded blocks in the local file system of each machine, and load block locations and stripe information into the coordinator. We simulate a "failed" machine by erasing blocks there, and repair the failed block of each stripe on a requestor. To fairly evaluate the impact of network transfers on repair, we host the requestor on a machine that does not store any available block of the repaired stripe, so as to ensure that the available blocks are always transmitted over the network. By default, we configure 64 MiB block size, 32 KiB slice size (for repair pipelining only), and (14,10) RS codes; note that (14,10) RS codes are also used by Facebook [30, 34]. We vary one of the settings at a time and evaluate its impact.

We consider two versions of repair pipelining: the basic version in §3 and the cyclic version in §4.1. We compare them with conventional repair (§2) and PPR [20].

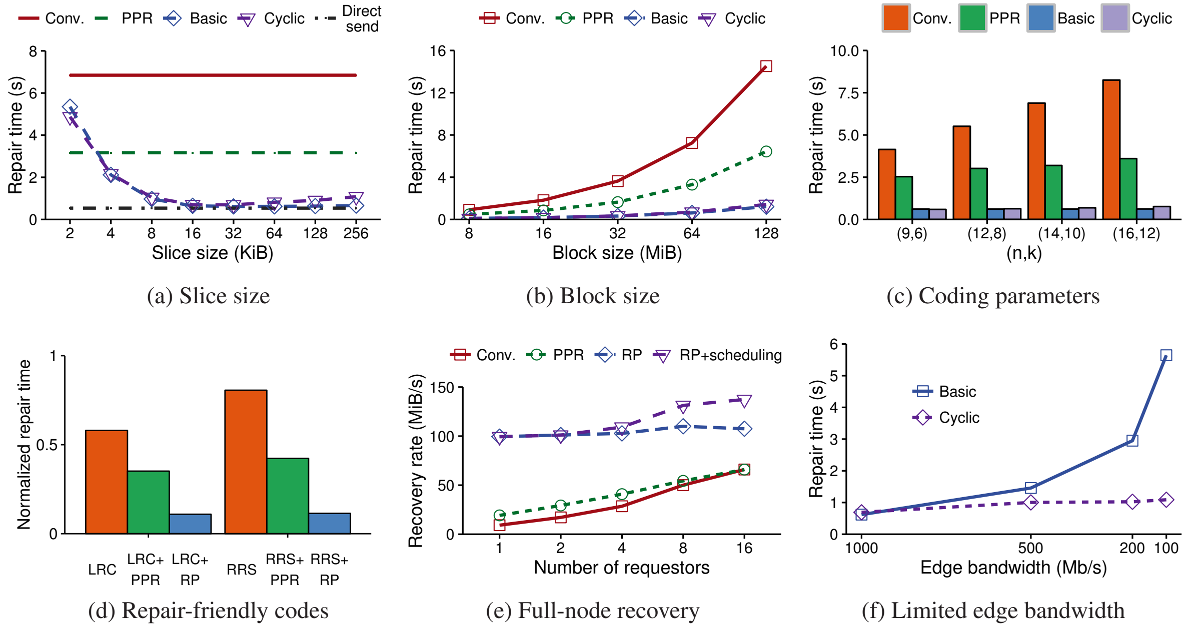
We evaluate both degraded reads and full-node recovery. For degraded reads (Figures 6(a)-(d) and 6(f)), we measure the *single-block repair time*, defined as the latency from issuing a degraded read request to a failed block until the block is reconstructed. For full-node recovery (Figure 6(e)), we measure the *recovery rate*, defined as the amount of recovered data by the total repair time. All results are averaged over 10 runs. The standard deviations are small and hence omitted from the plots.

6.1.2 Results

Slice size: Figure 6(a) shows the single-block repair time versus the slice size in repair pipelining. It also plots the transmission time of directly sending a single block over a 1 Gb/s link (labeled as "Direct send"). Both basic and cyclic versions of repair pipelining have high repair times when the slice size is small, even though more slices are pipelined during a repair (i.e., s is large). The reason is that the overhead of issuing transmission requests for many slices becomes significant. Nevertheless, the repair times of both versions decrease as the slice size increases up to 32 KiB (where $s = 2{,}048$) since fewer transmission requests are issued, and then increase since there are fewer slices in a block being pipelined. When the slice size is 32 KiB, the basic version reduces the single-block repair time by 90.9% and 80.4% compared to conventional repair and PPR, respectively. The basic version achieves 10.7% less single-block repair time than the cyclic version. The reason is that a helper in the cyclic version sends data to both the requestor as well as its next-hop helper, so the two transfers interfere with one another and (slightly) increase the repair time.

Also, the direct send time of transferring a 64 MiB block is 0.57s, which is almost network-bound in our 1 Gb/s network. The single-block repair time of the basic version is only 7.0% more than the direct send time, showing the feasibility of achieving $O(1)$ repair time.

Block size: Figure 6(b) shows the single-block repair time versus the block size. The repair time reduction of



(a) Slice size (b) Block size (c) Coding parameters

(d) Repair-friendly codes (e) Full-node recovery (f) Limited edge bandwidth

Figure 6: ECPipe performance on a local cluster.

repair pipelining over conventional repair and PPR increases with the block size as it can partition a block into more slices for better network usage. The basic version of repair pipelining reduces the single-block repair time by up to 91.4% and 80.9% compared to conventional repair and PPR, respectively. It is also faster than the cyclic version by up to 16.7%.

Coding parameters: Figure 6(c) shows the single-block repair time versus (n, k). The single-block repair times of both conventional repair and PPR increase with k, while that of repair pipelining is almost unchanged. As k increases from 6 to 12, the repair time reduction of the basic version increases from 85.1% to 92.1% compared to conventional repair, and from 75.7% to 83.3% compared to PPR.

Repair-friendly codes: We demonstrate how repair pipelining is compatible with practical erasure codes. We consider two state-of-the-art repair-friendly codes: LRC [15] and Rotated RS codes [17]. LRC has higher storage redundancy than RS codes by associating local parity blocks with a subset of data blocks, so as to improve single-block repair performance. On the other hand, rotated RS codes arrange the layout of parity blocks to improve the performance of a degraded read to a series of data blocks. We configure LRC with $k = 12$ data blocks, and Rotated RS codes with $(n, k) = (16,12)$. LRC needs to read only six blocks (five data blocks plus one local parity block) for repairing a failed data block, while Rotated RS codes on average read nine blocks for repairing a failed data block. Here, we focus on the basic version of repair pipelining.

Figure 6(d) shows the normalized single-block repair time with respect to the conventional repair of (16,12) RS codes. Although repair pipelining does not reduce the amount of repair traffic as in LRC and Rotated RS codes, its normalized repair time (around 0.1) is much smaller than those of LRC and Rotated RS codes by effectively utilizing the bandwidth resources of all helpers. We observe the same improvement in PPR, but its repair time reduction is less than that of repair pipelining.

Full-node recovery: We now evaluate full-node recovery with multiple requestors and our greedy scheduling in helper selection (§3.3). We randomly write multiple stripes of blocks across all 18 helpers in the local cluster. We erase 64 blocks from 64 stripes (one block per stripe) in one helper to mimic a single node failure, and recover all the erased blocks simultaneously. We distribute the reconstructed blocks evenly across a number of requestors (i.e., 1, 2, 4, 8, and 16).

We consider two cases of helper selection based on the basic version of repair pipelining: (i) we index the helpers from 1 to 18, and always select the available blocks from the k helpers that have the smallest indexes in a stripe for repair (labeled as "RP"); and (ii) we use the greedy approach to select k helpers that are least recently accessed for repair (labeled as "RP+scheduling"). We also evaluate conventional repair and PPR, both of which select helpers as in RP without greedy scheduling.

Figure 6(e) shows the recovery rates. As the number of requestors increases, the recovery rates of all schemes increase. Conventional repair sees the largest gain by distributing the repair load across more requestors. Its

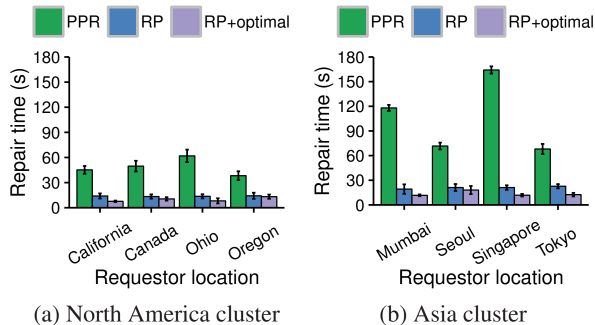
performance is also close to that of PPR as the number of requestors increases. However, repair pipelining still outperforms conventional repair by making bandwidth utilization more balanced. Furthermore, our greedy scheduling achieves a higher gain when there are more requestors by better distributing the repair load across all helpers. For example, when there are 16 requestors, the recovery rate of repair pipelining without greedy scheduling is 1.63× that of conventional repair, and our greedy scheduling further improves the recovery rate of repair pipelining by 27.9%.

Limited edge bandwidth: In previous tests, the basic version of repair pipelining always outperforms the cyclic version. We now show the benefits of the cyclic version when a requestor sits at the network edge and the edge bandwidth from the storage system to the requestor is limited (§4.1). We use the Linux command `tc` [38] to limit the edge bandwidth from each helper to the requestor. Figure 6(f) shows the single-block repair time versus the edge bandwidth. As the edge bandwidth decreases, the repair time of the basic version increases significantly, while that of the cyclic version only increases mildly by allowing the requestors to read repaired data from multiple helpers in parallel. For example, the cyclic version has 80.1% less repair time than the basic version when the edge bandwidth is 100 Mb/s.

6.2 ECPipe Performance on Amazon EC2

Methodology: We evaluate ECPipe on two independent Amazon EC2 clusters, one in North America and one in Asia. Each cluster is deployed in four regions as shown in Table 1. We deploy four EC2 instances per region per cluster to host helpers (i.e., 16 helpers in total), and one EC2 instance in Ohio and Singapore to host the coordinator for the North America and Asia clusters, respectively. Note that the overhead of accessing the coordinator has negligible impact on the overall repair performance. We focus on evaluating the degraded reads (in terms of single-block repair time) issued by a requestor. We host the requestor on an EC2 instance in each region and study how the performance varies across regions. All EC2 instances are of type `t2.micro`.

We configure 64 MiB block size and 32 KiB slice size for repair pipelining. We use (16,12) RS codes and distribute the 16 blocks of each stripe across the 16 EC2 instances in four regions; this also provides fault tolerance against any single-region failure. We consider two versions of repair pipelining: the basic version in §3 (labeled as "RP"), which finds a random path across k randomly selected helpers, and the optimal version in §4.2 (labeled as "RP+optimal"), which finds an optimal path via Algorithm 1. Note that the network bandwidth fluctuates over time, although intra-region bandwidth remains higher than inter-region bandwidth, as shown in Table 1.



(a) North America cluster (b) Asia cluster

Figure 7: ECPipe performance on Amazon EC2.

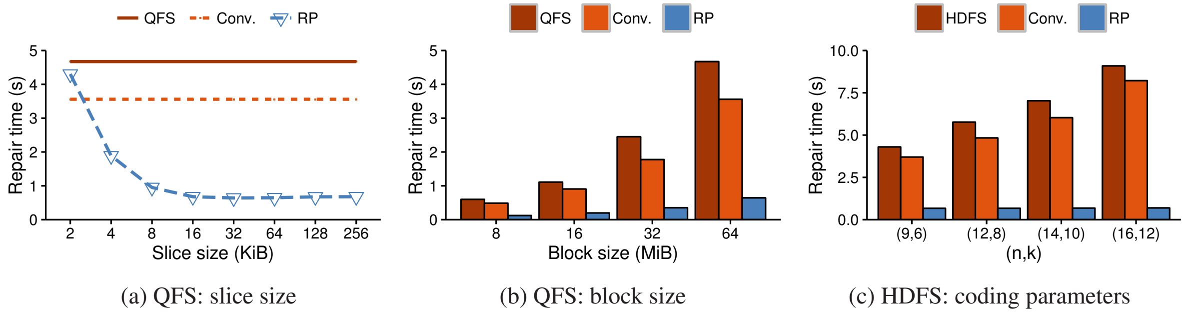
Thus, the optimal version probes the network bandwidth via `iperf` before each run of experiments. We average our results over 10 runs, and also include the standard deviations as the results have higher variances than in our local cluster.

Results: Figure 7 shows the single-block repair times and the standard deviations of PPR and the two versions of repair pipelining in both clusters; we do not show the results of conventional repair, whose repair time goes beyond 200s. Repair pipelining (without weighted path selection) achieves repair time saving over PPR in all cases when the requestor is in different regions. The repair time reduction is 62.7-78.0% for North America and 66.6-87.1% for Asia. Our weighted path selection further reduces the repair time by 7.3-45.4% for North America and 14.5-45.0% for Asia, compared to repair pipelining without weighted path selection. Note that our weighted path selection can be done in around 1ms (§4.2), which is negligible compared to the repair time in our evaluation.

6.3 Performance on HDFS and QFS

Methodology: We evaluate the integration of ECPipe into HDFS and QFS, both of which are deployed on our local cluster (§6.1). We co-locate a helper daemon with each storage node (18 nodes in total). By default, we set the slice size of repair pipelining as 32 KiB and block size as 64 MiB. For QFS, we use its default (9,6) RS codes and vary the slice size and block size. For HDFS, we vary (n, k). We consider three repair schemes: (i) the original repair implementations of HDFS and QFS, both of which are based on conventional repair, (ii) the conventional repair under ECPipe, and (iii) the basic version of repair pipelining in §3 under ECPipe. We evaluate degraded reads (in terms of single-block repair time) issued by a requestor that is attached with either an HDFS or QFS client. We report averaged results over 10 runs as in §6.1 (the standard deviations are small and omitted).

Results: Figure 8 shows the evaluation results. First, repair pipelining under ECPipe significantly improves the repair performance of the original repair implementa-



(a) QFS: slice size (b) QFS: block size (c) HDFS: coding parameters

Figure 8: Performance on HDFS and QFS.

tions of HDFS and QFS. It reduces the single-block repair time by up to 86.3% when the slice size is 32 KiB and the block size is 64 MiB (Figures 8(a) and 8(b)), and by 84.4–92.4% for different coding parameters (Figure 8(c)). The results are consistent with those in §6.1.

We observe that moving the repair logic to ECPipe improves repair performance. Specifically, conventional repair under ECPipe reduces the single-block repair time by up to 16.2% and 23.8% in HDFS and QFS, respectively, compared to the original conventional repair implementation. The reason of the performance gain is that the helpers of ECPipe can directly access the stored blocks via the native file system, instead of fetching the blocks through the distributed storage system routine. Nevertheless, we emphasize that the repair performance gain mainly comes from repair pipelining, rather than the implementation of ECPipe. Although moving repair to ECPipe reduces repair time, the reduction is minor compared to the reduction achieved by repair pipelining.

7 Related Work

Many new erasure codes have been proposed to mitigate repair overhead, especially for single-node repair. To name a few, regenerating codes [8] minimize repair traffic by allowing storage nodes to send encoded data for repair. Rotated RS codes [17] reduce repair traffic and disk I/O of a degraded read to a sequence of data blocks. Hitchhiker [30] extends RS codes [32] to piggyback parity information of one stripe into another stripe, and is shown to reduce both bandwidth and I/O for repair by up to 45%. PM-RBT codes [28] are special regenerating codes that simultaneously minimize bandwidth, I/O, and storage redundancy. Butterfly codes [25] are systematic regenerating codes that provide double-fault tolerance. Locally repairable codes [15, 34] add local parity blocks to mitigate repair I/O with extra storage.

Instead of constructing new erasure codes, we design new repair strategies for general practical erasure codes (including repair-friendly codes). Some prior studies are also along this direction. Lazy repair [3, 37] defers immediate repair action until a tolerable limit is reached. To speed up full-node recovery, the repair of multiple stripes can be parallelized across available nodes, as also adopted by replicated storage [6, 23] and de-clustered RAID arrays [14]. Degraded-first scheduling [19] targets MapReduce on erasure-coded storage by scheduling map tasks to fully utilize bandwidth in degraded reads. CAR [35] focuses on RS codes in data centers, and computes partial repaired results in each rack to mitigate cross-rack repair traffic. The most closely related work to ours is PPR [20], which reduces repair time from $O(k)$ to $O(\log k)$. Repair pipelining further reduces it to $O(1)$. We also show how repair pipelining addresses heterogeneous environments with different link bandwidths.

8 Conclusions

Repair pipelining is a general technique to reduce the repair time to almost the same as the normal read time in erasure-coded storage. It pipelines the repair of a failed block across storage nodes in units of slices, so as to evenly distribute repair traffic and fully utilize bandwidth resources across storage nodes. Our contributions include: (i) the design of repair pipelining for both degraded reads and full-node recovery, (ii) the extensions of repair pipelining with parallel reads and weighted path selection for heterogeneous environments, (iii) a repair prototype ECPipe and its integrations into HDFS and QFS, and (iv) experiments that show the repair speedup through repair pipelining on a local cluster and Amazon EC2. The source code of our ECPipe prototype is available at: **http://adslab.cse.cuhk.edu.hk/software/ecpipe**.

Acknowledgments: We thank our shepherd, Ryan Huang, and the anonymous reviewers for their valuable comments. We thank Allen Poon for contributing to the early implementation. This work was supported in part by the Research Grants Council of Hong Kong (GRF 14216316 and CRF C4047-14E), VC Discretionary Fund of CUHK (VCF2014007), and Cisco University Research Program Fund (CG#593756) from Silicon Valley Community Foundation.

References

[1] M. K. Aguilera. Geo-distributed Storage in Data Centers. In *Slides presented at OPODIS*, 2013.

[2] F. André, A.-M. Kermarrec, E. L. Merrer, N. L. Souarnec, G. Straub, and A. van Kempen. Archiving Cold Data in Warehouses with Clustered Network Coding. In *Proc. of ACM EuroSys*, Apr 2014.

[3] R. Bhagwan, K. Tati, Y. Cheng, S. Savage, and G. Voelker. Total Recall: System Support for Automated Availability Management. In *Proc. of NSDI*, 2004.

[4] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, et al. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proc. of ACM SOSP*, Oct 2011.

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

[6] B. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. F. Kaashoek, J. Kubiatowicz, and R. Morris. Efficient Replica Maintenance for Distributed Storage Systems. In *Proc. of NSDI*, 2006.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. of USENIX OSDI*, Dec 2004.

[8] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran. Network Coding for Distributed Storage Systems. *IEEE Trans. on Info. Theory*, 56(9):4539–4551, Sep 2010.

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

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

[11] S. Ghemawat, H. Gobioff, and S. Leung. The Google File System. In *Proc. of ACM SOSP*, Dec 2003.

[12] HDFS-RAID. `http://wiki.apache.org/hadoop/HDFS-RAID`.

[13] C. A. R. Hoare. Algorithm 65: find. *Communications of the ACM*, 4(7):321–322, 1961.

[14] M. Holland and G. A. Gibson. Parity Declustering for Continuous Operation in Redundant Disk Arrays. In *Proc. of ASPLOS*, 1992.

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

[16] Iperf. `https://iperf.fr/`.

[17] O. Khan, R. 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. of USENIX FAST*, Feb 2012.

[18] J. Li, S. Yang, X. Wang, and B. Li. Tree-structured Data Regeneration in Distributed and Storage Systems with Regenerating Codes. In *Proc. of IEEE INFOCOM*, 2010.

[19] R. Li, P. P. C. Lee, and Y. Hu. Degraded-First Scheduling for MapReduce in Erasure-Coded Storage Clusters. In *Proc. of IEEE/IFIP DSN*, 2014.

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

[21] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and S. Kumar. f4: Facebook's Warm BLOB Storage System. In *Proc. of USENIX OSDI*, 2014.

[22] L. M. Ni and P. K. McKinley. A Survey of Wormhole Routing Techniques in Direct Networks. *IEEE Computer*, 26(2):62–76, Feb 1993.

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

[24] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly. The Quantcast File System. In *Proc. of VLDB Endowment*, 2013.

[25] L. Pamies-Juarez, F. Blagojević, R. Mateescu, C. Gyuot, E. E. Gad, and Z. Bandic. Opening the Chrysalis: On the Real Repair Performance of MSR Codes. In *Proc. of USENIX FAST*, Feb 2016.

[26] J. S. Plank. Erasure Codes for Storage Systems: A Brief Primer. *;login: the Usenix magazine*, 38(6):44–50, Dec 2013.

[27] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, and Z. Wilcox-O'Hearn. A Performance Evaluation and Examination of Open-source Erasure Coding Libraries for Storage. In *Proc. of USENIX FAST*, 2009.

[28] K. V. 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 *Proc. of USENIX FAST*, 2015.

[29] K. V. 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 *Proc. of USENIX HotStorage*, 2013.

[30] 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. of ACM SIGCOMM*, 2014.

[31] Redis. `http://redis.io/`.

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

[33] J. K. Resch and J. S. Plank. AONT-RS: Blending Security and Performance in Dispersed Storage Systems. In *Proc. of USENIX FAST*, 2011.

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

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

[36] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proc. of IEEE MSST*, May 2010.

[37] M. Silberstein, L. Ganesh, Y. Wang, L. Alvizi, and M. Dahlin. Lazy Means Smart: Reducing Repair Bandwidth Costs in Erasure-coded Distributed Storage. In *Proc. of ACM SYSTOR*, 2014.

[38] `tc`. `https://linux.die.net/man/8/tc`.

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