

Archpipe: Fast and Flexible Pipelined Erasure-coded Archival Scheme for Heterogeneous Networks

Bin Xu*, Jianzhong Huang*, Xiao Qin†, Qiang Cao*, Yuanyuan Dong+, and Weikang Kong+

*Huazhong University of Sci.& Tech, Wuhan National Laboratory for Optoelectronics,
{binxu,hjzh,caoqiang}@hust.edu.cn

†Auburn University, xqin@auburn.edu

+Alibaba Group, {yuanyuan.dyy,weikang.kwk}@alibaba-inc.com

Abstract—Erasure-coded archival converts the redundancy mechanism of low access-frequency data from replication to erasure coding for balancing access performance and storage efficiency. A variety of pipelined schemes are designed to speed up the archival operation, however they neglect such three factors as heterogeneous network, under-utilization of replica resources and tight coupling with underlying platforms which restrict or even negate the performance gains.

In this paper, we propose *Archpipe*, a fast and flexible pipelined erasure-coded archival scheme. It exhibits three distinct features: 1) *heterogeneous network awareness*, for a single-pipelined construction, sufficient-bandwidth links are given high scheduling priority to avoid network congestion, while considering locality to reducing network transmissions; 2) *parallel encoding*, the unused replica resources are exploited to adaptively construct multiple pipelines for each stripe based on the single-pipelined algorithm, thereby enabling parity blocks to be encoded in parallel; 3) *loose coupling*, it does not rely on specific block placement policies and stripe construction algorithms. Experimental results indicate that, *Archpipe* can be seamlessly integrated with common distributed storage systems, and it improves the erasure-coded archival performance by $3.6 \sim 4.7\times$ and $1.3 \sim 2.6\times$ in on-disk and in-memory scenarios, respectively.

Index Terms—erasure coding, replication, heterogeneous network, pipelined archival

I. INTRODUCTION

International Data Company predicted that the amount of data produced by humans in 2025 will reach $175ZB$ [19]. One of the main reasons of such data explosion comes from data redundancy for guaranteeing reliability. As an attractive option reducing storage requirement at a given fault tolerance, Erasure coding (EC) has been widely used in various kinds of practical systems, such as distributed storage systems [16][2][29] and in-memory key-value systems [22] [33] [34].

Compared with replication, EC exhibits higher storage space efficiency and lower monetary cost, but suffers from extra encoding/decoding overhead, heavy partial-stripe update penalty and high recovery cost. For reaping advantages of replication and EC, the hybrid redundancy mechanism is adopted ingeniously in a tiered cluster-storage architecture for guaranteeing high access performance at low storage cost [4] [8] [35]. Commonly, the replication-tier fast stores newly-written and hot data, and then cold data with decreased access frequency

are dumped into the EC-tier asynchronously. The downward transition process of redundancy mechanism from replication to EC is called *erasure-coded archival*.

Traditional centralized erasure-coded archival (see Section II-C) performs three consecutive operations as local I/O, network transmission, and encoding computation. Each operation may become a single-point performance bottleneck in runtime, dynamically limiting the archival throughput. Furthermore, the redundancy transition causes heavy and unbalanced I/O, network, and computation overheads, heavily affecting the performance stability of foreground applications. Therefore, '*how to efficiently accomplish erasure-coded archival*' is a research-worthy issue.

Many researches have been devoted to address the above-mentioned issue. Some non-sequential stripe construction algorithms [28] [31] and encoding-oriented replica placement policies [14][32] were proposed to reduce the inter-rack traffic caused by data-block-retrieval and parity-block-relocation in the rack-layout cluster. Recently, pipelining has been efficiently applied to erasure-coded archival [7][9][17]. Different fine-grained tasks are evenly distributed to multiple nodes which are organized as a pipeline and executed in parallel, the overlapping of multiple tasks reduces the overall archival time. Such pipelined archival schemes avoid the single-point bottleneck and improve the parallelism, thus achieving better performance than the centralized archival.

However, existing pipelined archival schemes still encounter the following three dilemmas: (i) existing pipelined schemes are all based on an ideal homogeneous network, they ignore heterogeneous networks widely existed in real-world clusters with different-type devices, dynamic network bandwidths, etc. Experiment results in Section V-B1 demonstrate that the pipelined schemes upon heterogeneous networks lose the expected performance gains; (ii) a single pipeline for generating parity blocks sequentially only needs one replica of its involving blocks in a stripe. This means the other replicas of these blocks are unused; (iii) existing archival schemes strictly rely on the unique characteristics of their platforms such as coding schemes and data placement, limiting their portability.

Motivated by the above considerations, we propose a fast and flexible pipelined erasure-coded archival scheme – *Archpipe* – in heterogeneous network for distributed storage

*Qiang Cao is the corresponding author.

systems. First, we model the overhead of pipelined archival process. In-depth analysis reveals that the bottleneck bandwidth and the number of pipelining stages are two main factors affecting the archival performance. This guides us to perceive network heterogeneity to bypass bandwidth-limited links, and to exploit data locality by choosing nodes that reside more replicas to construct the archival pipeline. Second, we develop a time-efficient algorithm based on space-for-time to optimize the single-pipelined construction, thereby avoiding the NP complexity of brute-force searching solution. Third, we extrapolate the constraint of overall performance gains to make a good trade-off between the number of pipelines and the bottleneck bandwidth. Afterwards, *Archpipe* exploits the under-utilization replica resources to adaptively construct multiple pipelines for each stripe. Parity blocks are encoded in parallel instead of repeating r times sequentially, thereby improving the overall archival throughput. *Archpipe* takes data blocks in a stripe as input for pipeline constructions. In this way, it is totally independent with the underlying platforms with different replica placement policies and stripe construction algorithms.

We implement a plug-in of *Archpipe*, seamlessly integrated with common distributed systems without modification in the original codebase. The target systems only need to provide metadata query instructions and permissions. We deploy *Archpipe* on the popular on-disk and in-memory distributed systems (i.e., HDFS [2] and Alluxio [1]) in a real-world cluster with simulated heterogeneous network environments. We conduct extensive experiments under the representative configurations. The results demonstrate that compared with existing state-of-the-art archival schemes, *Archpipe* can effectively improve the erasure-coded archival performance by up to 4.7 times in heterogeneous network.

The contributions of this paper are summarized as follows:

- We propose *Archpipe*, a fast and flexible pipelined erasure-coded archival scheme that jointly considers network heterogeneity and locality to construct an optimal pipeline. We also present a time-efficient algorithm to construct the optimal pipelines.
- Based on the single-pipelined construction and constraint of overall performance gains, we exploit under-utilized replica resources to construct multiple pipelines, thus generating parity blocks within each stripe in parallel.
- We implement a flexible plug-in of *Archpipe*, which can be seamlessly integrated with common distributed systems. Through extensive experiments, we verify the effectiveness and generality of *Archpipe*.

II. BACKGROUND AND MOTIVATION

We first introduce the network heterogeneity and hybrid redundancy, then review the erasure-coded archival and related schemes, finally draw three insights to motivate our design.

A. Heterogeneous Network

In practice, storage clusters are upgraded and expanded in routine maintenance. The clusters could deploy different

types of servers, network devices and various versions of operating systems. The configuration differences in hardware and software lead to heterogeneous networks of distinct bandwidths between nodes in the cluster. Meanwhile, foreground applications with imbalance and dynamic load make available network bandwidth for background archival tasks fluctuated over time [20] [26]. This motivates the distributed systems to employ adaptive network scheduling to overcome network bottleneck and ensure stable performance.

B. Hybrid Redundancy

N -way replication distributes N replicas to N nodes, offering $N - 1$ fault tolerances and high access parallelism, at the expense of $N - 1$ redundancy overhead. In contrast, EC with high space efficiency at the same fault tolerance reserves only one replica of raw data blocks, thus providing low access parallelism. Reed-Solomon (RS) as a popular erasure code is employed below. A $RS(k + r, k)$ stripe organizes k data blocks and generated r parity blocks, to tolerate up to r blocks faults. Hybrid Redundancy storage architecture combining replication with EC is a common solution to reduce the redundancy overhead while guaranteeing system performance. Hot data with high access-frequency are stored in replication-based storage tier with high performance. Cold data with low access-frequency are preserved in EC-based storage tier with high storage efficiency. With hotness varying, data in such two storage tiers can be downward from replication to EC or upward inversely. The upward transition from EC to replication can be achieved simply by adding replicas. However, the downward process (i.e., erasure-coded archival) is not straightforward and encounters some pitfalls.

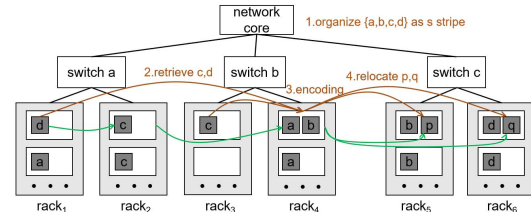


Fig. 1: centralized and pipelined archival in a heterogeneous network.

C. Erasure-coded Archival

Fig.1 shows an example of centralized and pipelined archival process in a heterogeneous network. The process of traditional erasure-coded archival is centralized and slated to undertake four steps. 1) Data blocks are logically orchestrated as stripes governed by a given stripe construction algorithm. 2) After an encoding node containing most data blocks is elected based on locality, it retrieves the rest k data blocks from other nodes. 3) The encoding node computes r parity blocks via erasure codes. 4) Parity-relocation is triggered to ensure node-level fault tolerance. Only one replica is kept for each data block in a stripe, the other replicas are deleted to claim storage capacity. In the centralized archival, the encoding node needs

to utilize both up- and down-stream bandwidth for data-block-retrieval and parity-block-relocation respectively. Without sufficient bandwidth available, the encoding node could become a bottleneck, thus severely degrading the archival performance. Therefore, pipelined archival is proposed to dispatch partial encoding tasks to multiple nodes due to the linearity of erasure codes. It provides high parallelism and great load balancing, but also manufactures a raft of new challenges in heterogeneous networks.

D. Archival-related Schemes

The large amount of local I/O, network transmission, and encoding overheads incurred slows down the erasure-coded archival process. A diversity of archival-related schemes, summarized in Table. I, propel archival performance from various angles. They can be divided into two camps according to the organization of nodes: centralized and pipelined schemes.

The centralized erasure-coded archival schemes include traditional scheme (TRAD)[8], EAR [14], DSC [28], Sice [31] and TEA [32]. TRAD chooses a encoding node based on locality and randomly schedules nodes for data-block-retrieval. EAR, DSC, Sice and TEA all consider common rack-layout environments and attempt to avoid inter-rack traffic caused by retrieving data blocks and relocating parity blocks to improve archival performance. More prosaically, EAR is an encoding-oriented placement based on the rack-aware one in HDFS and deployed in conjunction with the sequential stripe construction (SSC) algorithm which organizes data blocks according to the time written. DSC and Sice are suitable to most replica placement policies (e.g., RR short for random replica placement), but use non-sequential stripe construction (non-SSC) algorithms with the constraint that all the data blocks in a stripe have a unique replica in the same rack and their remaining replicas are kept in separate racks. TEA, advocating for an interleaved-type placement called ERP, exploits spatial locality to construct stripes to speed up the process of archival and maximize the utilization of memory space.

Representative pipelined erasure-coded archival schemes are RapidRAID [17], aHDFS [7] and DP/3X [9]. RapidRAID employs a new erasure coding customized for pipelined encoding, which generates all the parity blocks through a single pipeline without any need for the pipeline to encode multiple times. The cost of RapidRAID entails that each read request requires decoding calculations. aHDFS - an erasure-coded data archival system customized for Hadoop clusters - employs an array of idle nodes to build pipelines. Meanwhile, aHDFS splits data blocks and applies the Map/Reduce framework to enhance encoding parallelisms. DP/3X uses a chained-type placement, taking multiple stripes as a group to build multiple pipelines for parallel encoding. As such, DP/3X exhibits an improved overall archival throughput. Compared with centralized archival schemes, pipelined archival one can alleviate single-point bottlenecks and improve the parallelism of tasks, thereby improving the overall performance.

E. Motivation

We draw three insights to motivate our work through comparing existing archival-related schemes listed in Table. I. First, none of them take heterogeneous networks into account. Only some centralized schemes consider the rack-layout network. Unfortunately, experimental results show that pipelined archival schemes can not gain expected performance advantages in a heterogeneous network. Second, existing pipelined archival schemes merely construct one single pipeline for each stripe with multiple replicas of its involving data blocks. Obviously, when one single pipeline needs a replica of each data block, hence its other replicas are unused. Rather than relying on a single pipeline repeatedly running r times, the remaining nodes storing unused replicas can be employed to construct multiple pipelines to encode all the parity blocks in parallel. Third, existing archival schemes are customized to match replica placement policies and stripe construction algorithms in specific systems. The tight coupling limits their widespread applications.

III. DESIGN

A. Overview

Motivated by the above considerations, we design *Archpipe*, a fast and flexible pipelined erasure-coded archival scheme based on heterogeneous networks. For ease of understanding, we use an erasure-coded stripe as an archival unit to explain. Multiple stripes can be encoded in parallel and independently.

We model various overheads in the erasure-coded archival and identify the bottleneck bandwidth as the dominant factor affecting the performance. Therefore, *Archpipe* perceives the bandwidth discrepancy among links in heterogeneous networks and elects links with relatively high bandwidths. Meanwhile, *Archpipe* also considers locality to reduce the number of pipelining stages. Due to the high time-complexity of brute-force searching, we propose a space-for-time algorithm to determine optimal single-pipelined construction.

Considering the under-utilization of replica resources, *Archpipe* constructs multiple pipelines for each stripe, generating parity blocks in parallel instead of repeating executing the pipeline sequentially. We extrapolate the constraint to ensure performance improvement, and based on the optimal single-pipelined construction algorithm, we propose a multi-pipelined construction algorithm which can adaptively determine the number of pipelines and avoid contention among pipelines.

In addition, *Archpipe* takes stripe information as input and focuses on the encoding process. Consequently, it is independent of the specific underlying systems and can be complementary with other archival-dedicated schemes.

B. Optimal Single Pipeline

1) *Model*: Pipelining, a practical technology of temporal-parallel processing, is used to expedite erasure-coded archival[7][9][17]. In the pipelined archival schemes, for each data block in a stripe, a node storing its replicas is elected as the encoding node to control the pipelining. Data blocks are divided into several equal-sized slices. The slices with the

TABLE I: Comparisons among archival-related schemes.

archival-related scheme	stripe construction	placement	encoding process	locality	network
TRAD [8]	SSC	any	centralized	✓	×
EAR [14]	SSC	encoding-oriented	centralized	×	rack-layout
DSC [28]/Sice [31]	non-SSC	any	centralized	×	rack-layout
TEA [32]	non-SSC	encoding-oriented	centralized	✓	rack-layout
RapidRAID [17]/aHDFS [7]	any	any	single-pipelined	×	×
DP/3X [9]	SSC	chained	single-pipelined	×	×
Our <i>Archpipe</i>	any	any	multi-pipelined	✓	heterogeneous

same offset from all the data blocks are organized as a group. Different groups can be encoded in parallel via the constructed pipeline. Each node in a pipeline reads and encodes local slices, adds the slices to received partial-encoded slices from a preceding node, and sends current outcome of encoded slices to a succeeding node (more details in SectionIV-B). Due to the linearity of RS encoding, parity blocks can be obtained through combining the resulting fragments of each group. As illustrated in Fig.2, 4 nodes are organized as a pipeline to encode 2 parity blocks. Each data block is divided into 4 slices, so $4 \times 2 = 8$ slice groups need to be encoded.

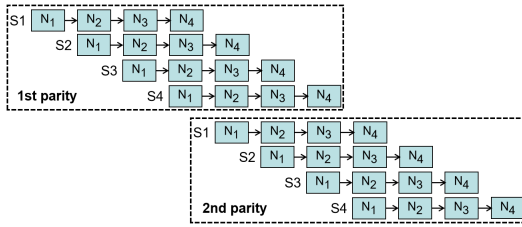


Fig. 2: An example of single-pipelined archival, with $k=4$, $r=2$ and $N_{slice}=4$. Label ' N_i ' represents the i^{th} node keeping a data block. Label ' S_j ' represents a group containing the j^{th} slices from each data block.

We model the execution time of single-pipelined archival (T_{sp_arch}) in Equ.1, which includes the latencies of local reading, CPU encoding and network transmission. S_{sli} and S_{blk} denote the size of a slice and data block, respectively; BW_{read} , BW_{enc} and BW_{net} denote the bandwidth of local reading, CPU encoding and network transmission, respectively; N_{pipe} and N_{slice} denote the number of pipelining stages and slices. Only the time of reading and encoding of the first slice are added, and the time of subsequent slices are overlapped by network transmissions. Because in common clusters, network is 1Gb or 10Gb Ethernet, which is evidently less than aggregated disk and CPU encoding bandwidth in a node. Furthermore, we usually configure limited network bandwidth for background tasks, e.g., 30MB/s for reconstruction in Pangu [26]. We know that a total of $N_{slice} \times r$ slice groups need to be executed via a pipeline, and the execution time of a pipelining stage is $\frac{S_{sli}}{BW_{net}}$. The execution time of I instructions in a pipeline is equal to that of the first instruction plus $(I-1)$ multiplying that of each stage. The time of network transmission is $\frac{S_{sli}}{BW_{net}} * N_{pipe} + \frac{S_{sli}}{BW_{net}} \times (N_{slice} \times r - 1)$. We further simplify and get Equ.1. We can reduce N_{pipe} to speed up the archival process by exploiting locality to elect encoding nodes containing most data blocks in an archival pipeline.

$$T_{sp_arch} = \frac{S_{sli}}{BW_{read}} + \frac{S_{sli}}{BW_{enc}} + \frac{S_{blk}}{BW_{net}} \times \left(r + \frac{N_{pipe} - 1}{N_{slice}} \right),$$

$$BW_{net} = \min\{BW_{net}^i\}, \text{ with } i = (1, 2, \dots, N_{pipe}). \quad (1)$$

In a heterogeneous network, the link bandwidths are diverse. The smallest bandwidth of all the stages becomes the bottleneck limiting the archival performance. It is necessary to consider BW_{net} and N_{pipe} simultaneously to determine optimal archival pipeline. There are k data blocks in a stripe; each data block has c replicas, which correspond to c locations. For a permutation, each location of data blocks has c possibilities, so there is a total of $k! \cdot c^k$ cases. We obtain from Eq. 1 the pipeline's execution time of all the cases, in which the least one is determined as an optimal archival pipeline. Obviously, this is an NP-Hard problem and the brute-force solution-seeking method has unacceptably high time-complexity.

2) A Time-efficient Construction Algorithm: We optimize the searching algorithm from the following two perspectives, (1) we first consider network-awareness to enlarge BW_{net} and then exploit locality to reduce N_{pipe} , because N_{slice} is commonly much larger than N_{pipe} , or $\frac{N_{pipe} - 1}{N_{slice}} \rightarrow 0$, enabling BW_{net} more affecting archival time than N_{pipe} ; (2) we adopt a well-known dynamic programming (DP) idea based on space-for-time, which connects links to determine qualified paths as archival pipelines rather than selecting and arranging nodes.

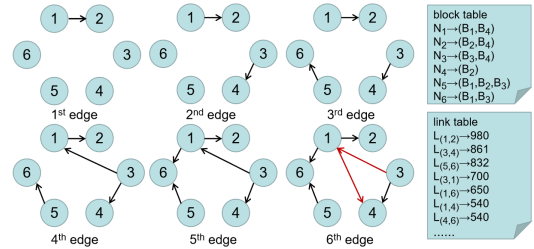


Fig. 3: A example showing the process of constructing an optimal single pipeline using the time-efficient algorithm.

As illustrated in Fig. 3, we formalize the optimal single-pipelined construction problem as finding a qualified path with the max-minimal weight in the graph, where vertexes correspond to nodes containing data blocks; directed edges represent network links used to transfer data from head node to tail node, and the weight of an edge signifies its link bandwidth. Some edges are allowed to form several paths. The main process of our algorithm is as follows. All links are sorted in a descending order of bandwidth, and added to the graph in turn.

New paths are formed through connecting the added edge to existing paths, and checked whether nodes in the path contain all the data blocks needed for encoding. Until some qualified paths are found in a round, the algorithm stops because the path with shortest length will be selected as an optimal archival pipeline. We use $E_{m,n}$ to represent an edge from m to n . The example in Fig. 3 demonstrates that when adding the 5th edge $E_{1,6}$, a new path $3 \rightarrow 1 \rightarrow 6$ is built; however, the path does not embrace all the data blocks. After continuously augmenting the 6th edge $E_{1,4}$, a new path $3 \rightarrow 1 \rightarrow 4$ is constructed to include all the data blocks. At this point, the algorithm obtains the optimal pipeline and quits. Our DP algorithm pays space to record the constructed paths, which are used to guide the constructions of new paths and avoid most invalid searches in advance, thus significantly improving searching efficiency.

3) *Algorithm Details*: The pseudo code of algorithm for optimal single-pipelined construction is shown in Algorithm.1:

In function *single_pipeline_cons*, three sets $\mathbb{EP}$, $\mathbb{NP}$ and $\mathbb{MP}$ are initialized to store existing, newly and matching constructed paths (line 2). Location information of blocks in $\mathbb{B}$ are queried, followed by constructing a node set ($\mathbb{N}$) by nodes containing replicas of all the data blocks (line 3). The algorithm retrieves a link set $\mathbb{L}$ embracing all the links that can be used in the archival pipeline. Note that each node, acting as a special link with a length of zero and infinite bandwidth, joins $\mathbb{L}$. Then, the quick-sort algorithm is employed to rank all the links in a descending order of bandwidth (line 4-5). Afterwards, links are added to the graph in turn until the optimal single pipeline is obtained. In function *construct_newpaths*, an added link L itself acts as a new path. All the paths containing the same head/tail node as that of L are found from $\mathbb{EP}$, and then they are connected to establish new paths inserted into $\mathbb{NP}$ (line 7). Function *check_path()* checks whether the nodes of each path in $\mathbb{NP}$ contain all the data blocks required for encoding. If it is, then the path will be included into $\mathbb{MP}$; the one with shortest length in this round is returned as an optimal pipeline. Otherwise, new paths with a length of less than k are placed in $\mathbb{EP}$ (line 8-17).

In the worst case, the algorithm traverses all the paths whose length is less than or equal to k , so it is bound to find a pipeline that meets the restraints as an optimal solution. In addition, the algorithm gives a high priority to the bandwidth bottleneck while granting a moderate priority to locality. As a result, the algorithm is unable to achieve theoretical optimal solutions in some cases. Fortunately, the number of slices is usually much larger than that of pipelining stages, so the result is identical to the optimal solution in most cases. The experimental results show that under the configurations of ‘RS(10, 8) and 18 nodes’, our DP method takes 130ms whereas the brute-force method takes 34608ms.

C. Adaptive Multi-pipelined Archival

1) *Model*: Each data block in a stripe has N replicas distributed on multiple nodes. Rather than relying on a single pipeline repeatedly running r times, the pipeline unrelated

Algorithm 1 optimal single-pipelined construction

Input: $\mathbb{B}$ (blocks in a stripe)
Output: P (optimal pipeline for a stripe)

```

1: function SINGLE_PIPELINE_CONS( $\mathbb{B}$ )
2:   Initialize sets  $\mathbb{EP}, \mathbb{NP}$  and  $\mathbb{MP}$ 
3:   Query locations of  $\mathbb{B}$  and construct  $\mathbb{N}$ 
4:   Construct link set  $\mathbb{L}$  using  $\mathbb{N}$ 
5:   QUICK_SORT( $\mathbb{L}$ )
6:   for all  $L$  with  $L \in \mathbb{L}$  do
7:      $\mathbb{NP} = \text{CONSTRUCT\_NEWPATHS}(L, \mathbb{EP})$ 
8:     for all  $P$  with  $P \in \mathbb{NP}$  do
9:       if CHECK_PATH( $P, \mathbb{B}$ ) then
10:         $\mathbb{MP} = \mathbb{MP} \cup \{P\}$ 
11:       end if
12:     end for
13:   if  $\mathbb{MP} \neq \emptyset$  then
14:     return  $P$  with smallest length in  $\mathbb{MP}$ 
15:   else
16:      $\mathbb{EP} = \mathbb{EP} \cup \{P \mid P \in \mathbb{NP} \& P.length < k\}$ 
17:   end if
18: end for
19: end function

```

nodes storing replicas can be employed to construct multiple pipelines for parallel encoding of parity blocks. As shown in Fig.4, nodes $\{N_1, N_2, N_3, N_4\}$ containing a replica of 4 data blocks forms a pipeline to encode the first parity block, while nodes $\{N_5, N_6, N_7, N_8\}$ storing another replica of 4 data blocks forms another pipeline to encode the second parity block. The two pipelines are executed in parallel. Apparently, the whole archival time can be reduced by exploiting full replicas.

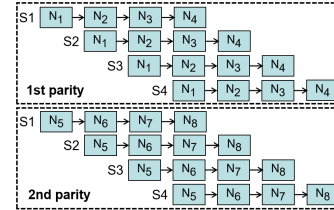


Fig. 4: An example of multi-pipelined archival, with $k=4$, $r=2$ and $N_{slice}=4$.

We extend Equ.1 to obtain the model of multi-pipelined archival time (T_{mp_arch}) to guide the multi-pipelined construction. In Equ.2, p (with $p \leq r$) captures the number of pipelines. $BW_{net}^{P_i}$ represents the bandwidth bottleneck in the i th pipeline. The lowest bandwidth among multiple pipelines (i.e. BW_{net}^{multi}) and p mainly dominate the overall archival time. A large p results in high encoding parallelisms improving the overall archival performance; however, increasing p leads to severe competition among links. Thus, the large value of p causes the small bandwidth BW_{net}^{multi} , which degrades the archival performance. To foster parallel encoding, we are motivated to propose an adaptive multi-pipelined scheduling technique to strike a good trade-off between p and BW_{net}^{multi} . We first derive the constraint of performance gains, then decide the number of pipelines, and finally schedule the multi-pipelined construction based on the above optimal single-pipelined algorithm. In addition, we introduce a link-shared model which can help us make full use of bandwidth re-

sources. Assume that n users share a link with a bandwidth BW , the bandwidth available to a user is averaged at BW/n . With the link-shared model in place, the links being utilized by one pipeline can be employed by the other pipelines.

$$T_{mp_arch} = \frac{S_{sli}}{BW_{read}} + \frac{S_{sli}}{BW_{enc}} + \frac{S_{blk}}{BW_{net}^{multi}} \times \left(\frac{r}{p} + \frac{N_{pipe}-1}{N_{slice}} \right),$$

$$BW_{net}^{multi} = \min\{BW_{net}^{P_i}\}, \text{ with } i = (1, 2, \dots, p), p \leq r. \quad (2)$$

2) *Constraint on Performance Gains:* Our goal is to devise a multi-pipelined archival that beats the performance of a single-pipelined one. Therefore, we put forth a formal analysis to derive the constraint on performance gains. Equ.3 reveals the final result is that $BW_{net}^{multi} \geq BW_{net}^{sgl}/p$, where BW_{net}^{sgl} is the bottleneck bandwidth of the optimal single pipeline, p denotes the number of constructed pipelines, and BW_{net}^{multi} is the minimal bandwidth of all the pipelines. Because the multi-pipelined construction is reliant on the link-shared model, the bandwidth of a link is the ratio of its original bandwidth to its used times. With the constraint in place, the following algorithm excludes links without enough bandwidths in advance and judiciously decides whether to continue constructing more pipelines under current link usage, thereby guaranteeing archival performance gains.

$$\frac{S_{blk}}{BW_{net}^{sgl}} \times \left(r + \frac{N_{pipe}}{N_{slice}} \right) \geq \frac{S_{blk}}{BW_{net}^{multi}} \times \left(\frac{r}{p} + \frac{N_{pipe}}{N_{slice}} \right),$$

$$BW_{net}^{multi} \geq \frac{BW_{net}^{sgl}}{p}, \quad (N_{slice} \rightarrow \infty, \frac{N_{pipe}}{N_{slice}} \rightarrow 0). \quad (3)$$

3) *Decision of the Number of Pipelines:* The number of pipelines should be decided elaborately based on replica resources. For N -way replication, we first determine an ideal number of pipelines (p') based on r . With $p' \in \{1, 2, \dots, N\}$, it is inclined to choose the maximum p' , which is the divisor of r . For example, for RS(12,8) in F4 [21] and RS(9,6) in QFS [16], we set p' to 2 and 3, respectively. In this way, our scheme can cover most erasure coding parameters. With p' in place, we assemble a single optimal pipeline and compute the constraint. Afterwards, the links without enough bandwidth are purged. We continue constructing pipelines using the remaining links. If a newly created pipeline is identical to an existing one, meaning that this pipeline should be run one more time. After finishing p' times constructions, p pipelines are finalized and each pipeline runs r_i times ($i = \{1, 2, \dots, p\}$). Such an adaptive process is slated to compose optimal multiple pipelines for parallel archival.

4) *Algorithm Detail:* The pseudo code of algorithm for multi-pipelined construction is shown in Algorithm.2:

In function *multi_pipelines_cons*, we maintain an auxiliary set $\mathbb{U}$ which records the number of pipelines using the shared links and is initialized to empty. The shared-bandwidth of a link is equal to the ratio of its original bandwidth and used times plus one. In addition, we initialize two sets $\mathbb{P}$ and $\mathbb{R}$, which store the constructed pipelines and run-times; two variables CB and $step$ are used to record constraint bandwidth

Algorithm 2 multi-pipelined construction

Input: $\mathbb{B}$ (blocks in a stripe)
Output: $\mathbb{P}$ (pipelines for a stripe), $\mathbb{R}$ (pipelines' runtimes)

```

1: function MULTI_PIPELINES_CONS( $\mathbb{B}$ )
2:   Initialize used times set  $\mathbb{U}$ ,  $\{U|U=0, U \in \mathbb{U}\}$ 
3:    $BW_L^{sh} = BW_L^{og}/(U_L + 1)$ ,  $L \in \mathbb{L}$ 
4:   Initialize sets  $\mathbb{P}$  and  $\mathbb{R}$ .
5:   Initialize constraint bandwidth  $CB = 0$  and  $step = 1$ 
6:   Decide the initial number of pipelines  $p'$ .
7:   while  $step++ \leq p'$  do
8:      $P = \text{SINGLE\_PIPELINE\_CONS}(\mathbb{B})$ .
9:     if  $P \notin \mathbb{P}$  then
10:        $\mathbb{P} = \mathbb{P} \cup \{P\}$  and  $R_P = 1$ ,  $\mathbb{R} = \mathbb{R} \cup \{R_P\}$ 
11:     else
12:        $R_P++$ ,  $R_P \in \mathbb{R}$ 
13:     end if
14:      $CB = BW_{min}^{sgl}/step$ .
15:      $U_L++$ ,  $L$  is used in  $P$ 
16:     Update  $BW_L^{share}$  using updated  $\mathbb{U}$  and  $BW_L^{og}$ .
17:     Filter out links with  $\{L|BW_L^{share} < CB, L \in \mathbb{L}\}$ .
18:   end while
19:   return  $\mathbb{P}$  and  $\mathbb{R}$ .
20: end function

```

and the number of iterations, respectively (line 2-5). The initial number p' of pipelines is determined according to r (line 6); then multiple pipelines are repeatedly assembled in the following three steps. First, the function *single_pipeline_cons* shown in Algorithm.1 is invoked to construct an optimal pipeline P . Second, if P does not exist in $\mathbb{P}$, then it is added to $\mathbb{P}$ with its run-times being set to 1 and inserted to $\mathbb{R}$. Otherwise, the pipeline's corresponding run-time is increased by 1 (line 8-12). Afterwards, we updates CB , $\mathbb{U}$, and shared-bandwidth of links according to the utilized links in P . The links with limited bandwidth are pruned from $\mathbb{L}$ (line 13-16). When iterating p' rounds, the resulting pipelines $\mathbb{P}$ and run times $\mathbb{R}$ are applied to subsequent erasure-coded archival.

In the worst case, Algorithm.2 constructs p pipelines that are the same as the optimal single pipeline. It means the multi-pipelined archival degenerates into a single-pipelined archival. Our experiments reveal that the algorithm can achieve higher time efficiency, compared with the brute-force solution. Under the same configurations of the single-pipelined algorithm, the time of consumption of our algorithm is 260ms, while the brute-force method takes 70619ms.

D. Extension

1) *Relocation:* Supporting r node-level fault tolerance, data and parity blocks in a stripe ought to be scattered among $k+r$ different nodes. When the encoding procedure of pipelined archival is accomplished, multiple parity blocks are generated in the same node and; hence, it is essential to relocate a portion of parity blocks. Keeping the archival efficiency in mind, we strive to elect large-bandwidth links to transfer parity blocks. The latency of relocation in a multi-pipelined archival is usually smaller than that in a single-pipelined one. In particular, r parity blocks in a single-pipelined archival are located in the same node, so $r-1$ relocations are mandatory. In a multi-pipelined archival, r parity blocks may be scattered

among various nodes because the tail nodes of pipelines are different commonly, thus cutting down relocations.

2) *Adaptation to network changes*: In practice, a cluster's expansion or its components' upgrades in software and hardware occur routinely, which change heterogeneous network configurations. For adapting to the changes, each time executing upgrade and expansion, each link bandwidth is measured. Besides, dynamic workloads also cause available network-bandwidth fluctuations. *Archpipe* can recalibrate link bandwidths in event-triggered or periodical manner.

IV. IMPLEMENTATION

We implement a distributed archival plug-in of *Archpipe* with about 4600 LoCs. *Archpipe* not only conveniently integrates with various distributed systems, but also can freely complement the other archival-related schemes. Target systems only need to support permissions and instructions for querying the metadata of blocks (i.e., "hdfs fsck" and "alluxio fs stat" in HDFS and Alluxio). We make use of Jerasure v.1.2 [3] to realize the encoding functionality.

A. Architecture

Fig.5 shows the architecture of our plug-in. More specifically, it encompasses a global coordinator and multiple workers. The coordinator periodically queries locations of blocks and stripes from metadata server (i.e., MDS) of target systems and probes link bandwidths in the cluster using iperf[23]. Then, the coordinator employs our algorithm to construct archival pipelines, generates archival tasks and assigns them to workers in parallel. Upon receiving tasks, the workers establish network connections and start local tasks. Finally, each worker acknowledges its accomplishment to the coordinator.

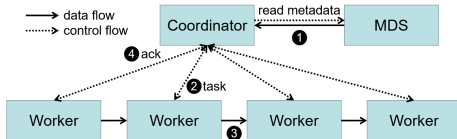


Fig. 5: The architecture of our plug-in (*Archpipe*).

B. Multi-threading

Fig.6 shows the multi-threading work process of a node in a pipeline. There are four threads. The first continually receives encoded slices from a predecessor worker while keeping the slices to a receiving buffer. The second proceeds to read local slices and stores them in a reading buffer. The third computes the partial encoded data using received and local slices with same offset, and stores them in a sending buffer. The last continually dispatches slices to a successor worker. Such a multi-threading work is pipelined by overlapping local I/Os, encoding and network transfers time, improving the overall archival throughput.

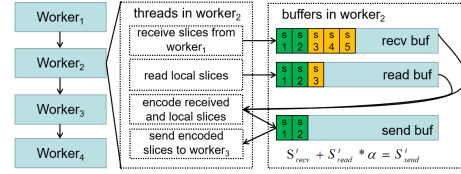


Fig. 6: The task threads in worker.

V. EXPERIMENTS AND RESULTS

We carry out extensive application-level trace-driven experiments on a real-world computing cluster. We are slated to answer the following performance evaluation questions.

- Can *Archpipe* improve overall archival performance combining with different replica placement policies and stripe construction algorithms? (§V-B1)
- How *Archpipe* is sensitive to different configuration parameters? (§V-B2, §V-B3, and §V-B4)
- Is our algorithm for multi-pipelined constructions more efficient than the brute-force one? (§V-B5)
- Is *Archpipe* more beneficial than existing pipelined schemes? (§V-B6)
- Can *Archpipe* adapt to network changes? (§V-B7)

A. Experimental Environment

System setup: Our testbed cluster consists of 19 servers, each of which is equipped with a Intel Xeon 4 core E5-2609 2.4GHz CPU, 8GB DDR3 memory and 1TB Western-Digital SATA HDD. They are connected by a 1Gb/s Ethernet switch. 18 servers are storage nodes and the remaining one is reserved as MDS. To mimic the heterogeneous network, we generate multiple independent link bandwidth records based on the Pareto distribution [24] and use the traffic control tool (i.e., TC) to adjust the bandwidths between servers [5]. The distribution has a default shape parameter $\alpha = 0.1$. The larger α , the more obvious the long tail feature. The bandwidth is allocated to front-end users and restricted to back-end tasks. Therefore, we make the link bandwidths obey the uniform distribution $U[100Mb/s, 1000Mb/s]$, according to the measurement of real world networks [12].

Comparison schemes: We adopt HDFS and Alluxio as on-disk and in-memory distributed systems respectively. We combine different types of replica placement policies (RR, encoding-oriented placements including EAR, ERP) and stripe construction algorithms (SSC, non-SSC including Sice/DSC, TEA) to obtain four categories (RR+SSC, RR+DSC, EAR+SSC, ERP+TEA). In the context of these categories, we compare *Archpipe* (i.e., optimal multi-pipelined archival) with optimal and random single-pipelined archival, random centralized archival (i.e., *opt.s.pipe*, *r.s.pipe* and *r.ctl*).

Default configurations: The block size is set as 64MB and 4KB for on-disk and in-memory scenarios, respectively. There are 6000 data blocks, each of which has three replicas placed via RR. We opt for $RS(8, 6)$ to organize a stripe via SSC, meaning that 1000 stripes generated. A data block is divided into 1024 slices; the rationale is articulated in §V-B2.

The shape of Pareto distribution is 0.1. We perform each test for multiple runs to acquire average values.

B. Experimental Results

1) *Performance Improvement and Flexibility*: Fig.7 shows the archival performance in term of archival time with *Archpipe* and three candidate schemes in on-disk and in-memory scenarios, under various replica placement policies and stripe construction algorithms being adopted.

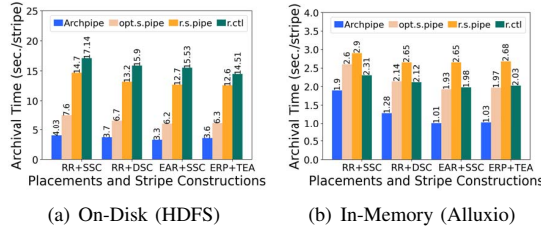


Fig. 7: Comparisons of archival time with different replica placements and stripe constructions under the default configurations.

Fig.7 unveils that *Archpipe* has a performance edge over the other three schemes in both on-disk and in-memory scenarios, regardless of replica placement policies and stripe construction algorithms. The impressive results validate the effectiveness and flexibility of *Archpipe*. Such performance improvement of *Archpipe* benefits from the following three factors. (1) Compared with centralized schemes (i.e., rctl), *Archpipe* constructs pipelines with distributing load to each involved node, effectively avoiding the single-point bottleneck. Meanwhile, local I/Os, network transmissions, and encoding computations are executed in parallel to further curtail the archival time. (2) Unlike the schemes favoring random scheduling (e.g., rctl and r.s.pipe), *Archpipe* perceives the diversity of link bandwidths to avert bandwidth-limited links while incorporating locality to reduce the number of pipelining stages in heterogeneous networks. (3) Compared with single-pipelined schemes such as opt.s.pipe, *Archpipe* constructs multiple pipelines for each stripe, enabling parity blocks encoded in parallel instead of r times repeatedly performed by a single pipeline. In particular, *Archpipe* improves the erasure-coded archival performance of the r.s.pipe and rctl archival schemes by $3.6 \sim 4.7\times$ and $1.3 \sim 2.6\times$ in on-disk and in-memory scenarios, respectively. Note that opt.s.pipe is left out of the comparisons because it is *Archpipe* powered by a single pipeline.

It should be noteworthy that in in-memory scenarios, smaller-sized blocks lead to a narrow gap of network transmission latency. Thus the performance improvement of *Archpipe* is smaller than that in on-disk scenarios. In addition, the transmission of slices requires more network requests, causing lower archival performance of opt.s.pipe than that of rctl in in-memory scenarios.

2) *Sensitivity of the Number of Slices*: We set N_{slice} to various values to examine the sensitivity of *Archpipe* to the number of slices (N_{slice}). Fig.8 reveals the archival time of all the schemes with respect to N_{slice} under the default

configurations, with the maximum N_{slice} being set to 2^{22} and 2^{12} for on-disk and in-memory scenarios, respectively.

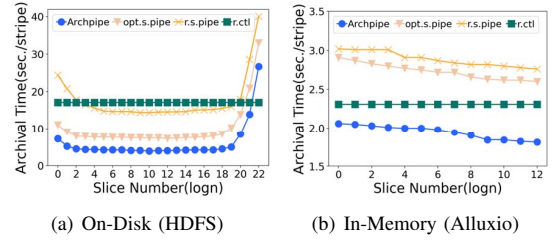


Fig. 8: Comparisons of archival time with different N_{slice} under the default configurations.

As shown in Fig.8, *Archpipe* exhibits the best archival performance among all the candidate schemes regardless of the value of N_{slice} . In on-disk scenarios, increasing N_{slice} makes the performance trend of all the pipelined archival schemes a bathtub curve. The reason is threefold. (1) When N_{slice} is small, the low parallelism leads to relatively poor archival performance. (2) As N_{slice} goes up, the performance is gradually boosted while remaining stable. (3) When N_{slice} continues to increase, a large number of requests are needed to deliver slices, thus extending the archival time. More specifically, the latency sharply increases when N_{slice} reaches a certain value (e.g., 2^{20} in this experiment). In in-memory scenarios, the maximum N_{slice} is set to 2^{12} due to the small size of data blocks. The trend is still at the early stage of the bathtub curve. In this way, the performance continues to improve as N_{slice} is shooting up. It is, therefore reasonable and prudent to set N_{slice} to 1024 for the default configurations.

3) *Sensitivity of Erasure Coding Parameter*: Intuitively, erasure coding parameters k and r are closely associated with different steps during the archival. Hence, it is mandatory to examine the sensitivity of *Archpipe* to erasure coding parameters. Fig.9 unravels the archival time of all the schemes with respect to different RS parameters.

Fig.9 demonstrates that under any erasure coding parameters, *Archpipe* delivers superb archival performance thanks to network-awareness, locality-awareness and parallel encoding of parity blocks. increasing k or r deteriorates the archival performance of all the schemes. The reason is twofold. (1) A larger k entails a larger number of pipelining stages, thus prolonging the latency of network transmissions accordingly. (2) Increasing r implies that more parity blocks ought to be encoded and relocated. Meanwhile, compared against the other schemes, *Archpipe* is less sensitive to k or r . This trend is expected because of two factors. (a) A large k value leverages more node resources to mitigate the performance degradation. (b) After encoding, r parity blocks may be residing in multiple nodes instead of a single node. Therefore, few parity blocks should be relocated for node-level fault tolerance.

4) *Sensitivity of Block Size*: Large-sized data blocks require long network transmission latency, so block size (S_{blk}) may take an effect on archival operations. Fig.10 shows the archival time of all the tested schemes in which S_{blk} is set

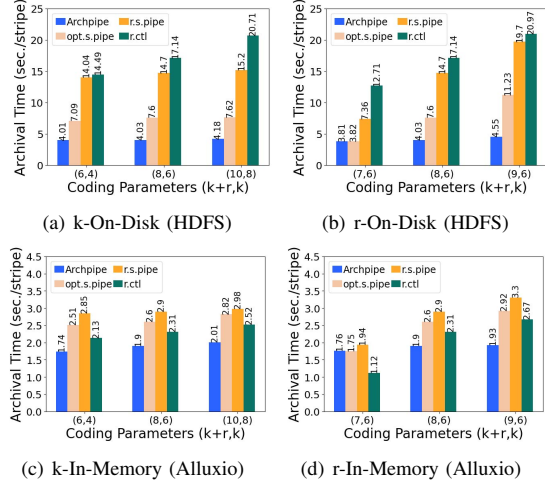


Fig. 9: Comparisons of archival time with different erasure coding schemes under the default configurations.

to $\{2^0, 2^1, \dots, 2^8 MB\}$ and $\{2^0, 2^1, \dots, 2^{10} KB\}$ in on-disk and in-memory scenarios, respectively.

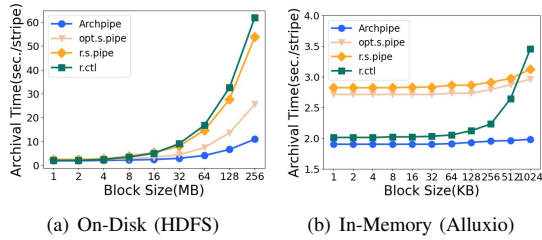


Fig. 10: Comparisons of archival time with different S_{blk} under the default configurations.

Fig.10 unveils that for any S_{blk} value, *Archpipe* is a clear winner in terms of archival performance among its peers. As S_{blk} increases, the archival time obeys an upward trend because large data blocks push up time spent in local I/Os, network transmissions and encoding computations. When it comes to in-memory scenarios, the prolonged archival times of the pipelined schemes are smaller than that of the centralized one. The observation is expected because an increase in S_{blk} may enlarge the pressure of the encoding node - a single-point bottleneck in the centralized schemes. The pipelined schemes, in contrast, mitigate the bottleneck in the centralized schemes by dividing data blocks among nodes into several slices to foster parallel reads, encoding, and transfers.

5) *Algorithm Complexity*: In order to verify the efficiency of our DP algorithm, we adopt DP against the force-brute and random algorithms to construct multiple pipelines. We refer to these algorithms as *DP.Archpipe*, *FB.Archpipe*, and *RND*.

TABLE II: Algorithm runtime with different k (unit:ms.).

Algo.	k	2	4	6	8
<i>DP.Archpipe</i>		0.064	0.842	15.6	259.7
<i>FB.Archpipe</i>		0.006	0.457	111.2	70619
<i>RND</i>		0.005	0.006	0.006	0.008

Table II summarizes the runtimes of the *DP.Archpipe*, *FB.Archpipe*, and *RND* under various k values. We observe from the table that *RND* stably maintains a minimum runtime while suffering from poor archival performance. The runtime measures of other algorithms rise when k goes up. The reason of this trend is that a large k means a large number of data blocks and nodes participating in the archival process shooting up the number of pipelining stages. The second observation drawn from the table is that the runtime of *DP.Archpipe* is less sensitive to k than *FB.Archpipe* thanks to *DP.Archpipe*'s low complexity. When we have $k = 2$ and $k = 4$, the runtime of *FB.Archpipe* is smaller than that of *DP.Archpipe* because there are a small number of possible situations to search under a small k . In case of $k = 6$ and $k = 8$, *FB.Archpipe* sharply surges because its time complexity is $k! \times c^k$. On the flip side, *DP.Archpipe* adopts a space-for-time strategy to purge a large number of invalid searches in advance.

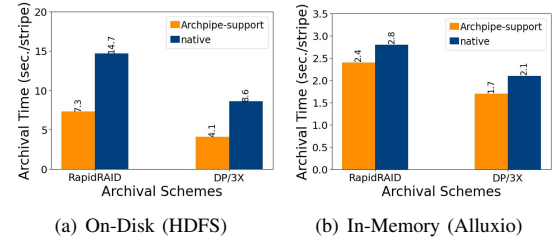


Fig. 11: Comparisons of archival time of existing pipelined schemes w/o *Archpipe* under the default configurations.

6) *Archpipe vs. Existing Pipelined Schemes*: The existing pipelined archival schemes including RapidRAID and DP/3X, are primed to deliver good archival performances in a homogeneous network. However, they overlook the performance issues in a heterogeneous network. On the basis of the native schemes, we implement *Archpipe* to orchestrate node scheduling to construct pipelines. Fig.11 reveals the archival time measures of RapidRAID and DP/3X in different scenarios, with/without *Archpipe*'s support. We draw two observations. (1) *Archpipe* is conducive to significantly improving the archival performance over the existing schemes. The performance optimization is mainly contributed by the network-awareness and locality awareness, which are explained in §V-B1. (2) The archival performance boosts stemmed from *Archpipe* in on-disk scenarios are more pronounced than those in in-memory scenarios because of the small-sized data blocks resulting in low transmission latency.

7) *Network Condition*: We first set the shape parameter of Pareto distribution α to $\{0.01, 0.1, 1\}$ to examine the sensitivity of *Archpipe* to degree of network heterogeneity. The larger α , the smaller the heterogeneity, which means that more link bandwidths approach 100Mb/s. As shown in Fig.12(a) and Fig.12(b), as α increases, performance degradations of all the schemes occur, because of more small bandwidth links; tiny bandwidth variability makes algorithms lose optimization potential, the performance gap between opt.s.pipe and r.s.pipe is reduced; however, *Archpipe* still obtains the lowest archival

time, benefiting from multi-pipelined parallelism.

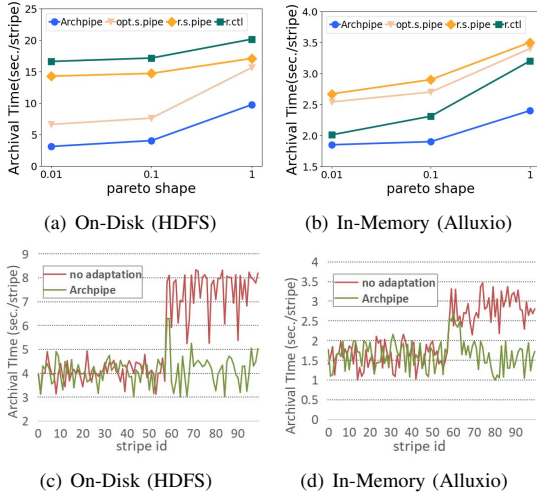


Fig. 12: Comparisons of archival time of with different shapes of network distribution under the default configurations and adaptation to network changes.

In addition, for examining adaptation to network changes, during the archival process, we replace the network layout to simulate dynamic networks caused by node failures, struggles and device upgrades, etc. Fig.12(c) and Fig.12(d) reveal that network changes lead to inaccurate pipeline scheduling and performance degradation if no adaptation. Our *Archpipe* triggers the task of network measurement, when actual archival time falls short of expectations. As a result, *Archpipe* can recover stable and efficient archival performance again quickly.

VI. RELATED WORK

Let us elaborate the innovation and excellence of our *Archpipe* by qualitatively comparing it with related works.

Hybrid redundancy and efficient redundancy transition can exploit the diversity of hardware, data and workload to balance multiple aspects of system performance. Hybrid redundancy mainly includes 1) mixture of different erasure codes and 2) mixture of replication and erasure codes. The representatives of multi-erasure-code-enabled hybrid redundancy schemes are HACFS, LHH, and EC-Fusion, which adopt different erasure codes for hot/cold data aiming to reduce space and recovery overhead. These schemes exploit local parity blocks to accelerate their transitions [30][27][18]. HeART [10] selects RS parameters based on the life cycle of disks; PACEMAKER [11] reduces transition I/Os in HeART by proactively arranging data layouts and initiations. Ring [22] configures RS parameters for different transactions while employing SRS code for fast switch. The second hybrid-approach type, which is championed in our proposed solution, embraces the mixture of replication and erasure codes. The archival-related schemes in Table.I have been developed to optimize erasure-coded archival performance. Distinctively, *Archpipe* is orthogonal to the above-mentioned archival schemes and decoupled from the underlying platforms.

Pipelining has been applied into erasure-coded reconstruction, updating and archival to overcome single-point performance bottlenecks while fostering parallelisms. ECPipe first introduces pipelining to recovery erasure-coded blocks [13]. RPR and PPCT build repairing pipelines for rack-layout clusters [15] [6]. TA-Update constructs an update tree according to the network layout, where tasks are distributed to nodes and processed in a pipelined manner [25]. Although encoding nodes are organized as pipelines to achieve archival performance improvement over the centralized archival ones, the existing schemes are developed and customized for clusters in an ideal homogeneous network and possess under-utilization of replica resources [17] [7] [9]. On the contrary, to our knowledge, *Archpipe* is the first multi-pipelined archival scheme, considering the bandwidth bottleneck and locality simultaneously, thereby catering for heterogeneous networks.

VII. CONCLUSION

Compared with the cutting-edge schemes, *Archpipe* brags three distinctive and salient features: (1) adapting to heterogeneous networks, (2) making the full use of replica resources, and (3) decoupling from underlying platforms. First, *Archpipe* perceives network heterogeneity to bypass bandwidth-limited links and considers locality to reduce the number of pipelining stages simultaneously. Second, *Archpipe* exploits the resources of nodes keeping other replicas of data blocks to adaptively construct multiple pipelines, which allow parity blocks to be encoded in parallel. Third, *Archpipe* envisions encoding as an issue of archival performance rather than replica placement policies and stripe construction algorithms, thus boosting deployment flexibility. We implement and open source *Archpipe* as a plug-in. The systematic evaluations unveil that compared with the existing schemes, *Archpipe* significantly shapes up the erasure-coded archival performance by $3.6 \sim 4.7\times$ and $1.3 \sim 2.6\times$ in on-disk and in-memory scenarios, respectively.

ACKNOWLEDGMENT

This work was supported in part by National key research and development program of China under Grant 2018Y-FA0701800, NSFC No.62172175, Creative Research Group Project of NSFC No.61821003, and by Alibaba Group through Alibaba Innovative Research (AIR) Program. Xiao Qin's work is supported by the U.S. National Science Foundation under Grants IIS-1618669, OAC-1642133, CCF-0845257.

REFERENCES

- [1] "Alluxio," <https://www.alluxio.io/>.
- [2] "HDFS," <http://hadoop.apache.org/>.
- [3] "Jerasure," <https://jerasure.org/>.
- [4] M. Abd-El-Malek, W. V. C. II *et al.*, "Ursa minor: Versatile cluster-based storage," in *Proc. of FAST*, 2005.
- [5] W. Almesberger, "Linux network traffic control implementation overview," 1999.
- [6] Y. Bai, Z. Xu *et al.*, "Fast recovery techniques for erasure-coded clusters in non-uniform traffic network," in *48th International Conference on Parallel Processing*, 2019, pp. 61:1–61:10.
- [7] Y. Chen, Y. Zhou *et al.*, "ahdfs: An erasure-coded data archival system for hadoop clusters," *IEEE Trans. Parallel Distributed Syst.*, vol. 28, no. 11, pp. 3060–3073, 2017.

- [8] B. Fan, W. Tantisiriroj *et al.*, “Diskreduce: Replication as a prelude to erasure coding in data-intensive scalable computing,” *SC11*, 2011.
- [9] J. Huang, Y. Wang *et al.*, “Exploiting pipelined encoding process to boost erasure-coded data archival,” *IEEE Trans. Parallel Distributed Syst.*, vol. 26, no. 11, pp. 2984–2996, 2015.
- [10] S. Kadekodi *et al.*, “Cluster storage systems gotta have heart: improving storage efficiency by exploiting disk-reliability heterogeneity,” in *17th USENIX FAST*, 2019, pp. 345–358.
- [11] S. Kadekodi, F. Maturana *et al.*, “PACEMAKER: Avoiding heart attacks in storage clusters with disk-adaptive redundancy,” in *14th USENIX SOSP*, 2020, pp. 369–385.
- [12] S. Lee, P. Sharma *et al.*, “Measuring bandwidth between planetlab nodes,” vol. 3431. Springer, 2005, pp. 292–305.
- [13] R. Li *et al.*, “Repair pipelining for erasure-coded storage,” in *USENIX Annual Technical Conference*, 2017, pp. 567–579.
- [14] R. Li, Y. Hu *et al.*, “Enabling efficient and reliable transition from replication to erasure coding for clustered file systems,” in *International Conference on Dependable Systems and Networks*, 2015, pp. 148–159.
- [15] T. Liu *et al.*, “A rack-aware pipeline repair scheme for erasure-coded distributed storage systems,” in *49th International Conference on Parallel Processing*, 2020, pp. 37:1–37:11.
- [16] M. Ovsianikov, S. Rus *et al.*, “A the quantcast file system,” *Proc. VLDB Endow.*, vol. 6, no. 11, pp. 1092–1101, 2013.
- [17] L. Pamies-Juarez, A. Datta *et al.*, “Rapidraid: Pipelined erasure codes for fast data archival in distributed storage systems,” in *Proceedings of the IEEE INFOCOM*, 2013, pp. 1294–1302.
- [18] H. Qiu, C. Wu *et al.*, “Ec-fusion: An efficient hybrid erasure coding framework to improve both application and recovery performance in cloud storage systems,” in *International Parallel and Distributed Processing Symposium*, 2020, pp. 191–201.
- [19] D. Reinsel, J. Gantz *et al.*, “Data age 2025: the digitization of the world from edge to core,” *IDC White Paper*, pp. 1–29, 2018.
- [20] C. Reiss, A. Tumanov *et al.*, “Heterogeneity and dynamicity of clouds at scale: Google trace analysis,” in *SoCC '12*, 2012.
- [21] M. Subramanian, W. Lloyd *et al.*, “f4: Facebook’s warm BLOB storage system,” in *11th USENIX Symposium on Operating Systems Design and Implementation*, 2014, pp. 383–398.
- [22] K. Taranov *et al.*, “Fast and strongly-consistent per-item resilience in key-value stores,” in *Proc. of EuroSys Conference*, 2018, pp. 39:1–39:14.
- [23] A. Tirumala, “Iperf: The tcp/udp bandwidth measurement tool,” 1999.
- [24] A. Varet and N. Larrieu, “How to generate realistic network traffic?” in *IEEE Annual Computer Software and Applications Conference*. IEEE Computer Society, 2014, pp. 299–304.
- [25] Y. Wang, X. Pei *et al.*, “TA-update: An adaptive update scheme with tree-structured transmission in erasure-coded storage systems,” *IEEE Trans. Parallel Distributed Syst.*, vol. 29, no. 8, pp. 1893–1906, 2018.
- [26] Z. Wang, G. Zhang *et al.*, “Dayu: Fast and low-interference data recovery in very-large storage systems,” in *Annual Technical Conference*, 2019.
- [27] Z. Wang *et al.*, “An adaptive erasure-coded storage scheme with an efficient code-switching algorithm,” in *49th International Conference on Parallel Processing*, 2020, pp. 35:1–35:11.
- [28] S. Wei, Y. Li *et al.*, “DSC: dynamic stripe construction for asynchronous encoding in clustered file system,” in *INFOCOM*, 2017, pp. 1–9.
- [29] S. A. Weil, S. A. Brandt *et al.*, “Ceph: A scalable, high-performance distributed file system,” in *7th SOSP*, 2006, pp. 307–320.
- [30] M. Xia, M. Saxena *et al.*, “A tale of two erasure codes in HDFS,” in *Proceedings of the 13th USENIX FAST*, 2015, pp. 213–226.
- [31] Y. Xie *et al.*, “Non-sequential striping for distributed storage systems with different redundancy schemes,” in *46th International Conference on Parallel Processing*, 2017, pp. 231–240.
- [32] B. Xu, J. Huang *et al.*, “TEA: A traffic-efficient erasure-coded archival scheme for in-memory stores,” in *48th International Conference on Parallel Processing*, 2019, pp. 24:1–24:10.
- [33] M. M. T. Yiu *et al.*, “Erasure coding for small objects in in-memory KV storage,” in *10th SYSTOR Conference*, 2017, pp. 14:1–14:12.
- [34] H. Zhang *et al.*, “Efficient and available in-memory kv-store with hybrid erasure coding and replication,” in *14th FAST*, 2016, pp. 167–180.
- [35] P. Zhou *et al.*, “PaRS: A popularity-aware redundancy scheme for in-memory stores,” *IEEE Trans. Computers*, vol. 68, pp. 556–569, 2019.