Making MapReduce Scheduling Effective in Erasure-Coded Storage Clusters

Runhui Li and Patrick P. C. Lee
The Chinese University of Hong Kong
{rhli,pclee}@cse.cuhk.edu.hk
(Invited Paper)

***Abstract*—With the explosive growth of data, enterprises increasingly adopt erasure coding on storage clusters to save storage space. On the other hand, erasure coding incurs higher performance overhead, especially during recovery. This motivates us to study the feasibility of alleviating performance overhead of erasure coding, while maintaining its storage efficiency advantage. In this paper, we study the performance issue of MapReduce when it runs on erasure-coded storage. We first review our previously proposed degraded-first scheduling, which avoids network bandwidth competition among degraded map tasks in failure mode, and hence improves the MapReduce performance over the default locality-first scheduling in MapReduce. We then show that the basic degraded-first scheduling may not work effectively when there are multiple running MapReduce jobs, and hence we propose heuristics to enhance the degraded-first scheduling design. Simulations demonstrate the performance gain of our enhanced degraded-first scheduling in a multi-job scenario. Our work makes a case that a new design of MapReduce scheduling is critical when we move to erasure-coded storage.**

***Index Terms*-Erasure coding, MapReduce, storage systems.**

I. Introduction

Enterprises have increasingly adopted *erasure coding* in storage clusters to address both availability and scalability of data storage. For example, Google's GFS [14], Facebook's HDFS [22, 25, 27] and Microsoft's Azure [16] have deployed erasure coding to save storage space. Erasure coding is typically configured by two parameters n and k (where $k < n$). An (n, k) erasure code works by dividing data into fixed-size *blocks*. It encodes every group of k (uncoded) data blocks to form additional $n - k$ (coded) parity blocks, such that any k out of n data and parity blocks suffice to reconstruct the original k data blocks. Classical examples of erasure coding constructions are Reed-Solomon codes [26] and Cauchy Reed-Solomon codes [5]. In general, erasure coding achieves higher fault tolerance than replication (i.e., making identical copies for each data block), while incurring much less storage overhead [28]. Although disk prices significantly drop nowadays, the overwhelming scale of data makes erasure coding necessary to keep the operational cost low. For example, Azure's researchers [16] claim that erasure coding can reduce the operational cost of Azure storage by over 50%, compared to traditional triple replication. Facebook [22] also uses erasure coding to reduce over 53PB of storage.

A. Challenges

Erasure coding generally has higher performance overhead than replication. The performance issues of erasure coding are further complicated by the following challenges:

Resource-constrained architectures: Network bandwidth is a scarce resource in practical storage clusters [7, 9]. Network congestion is common due to the imbalance use of bottleneck links [7], and network links are often over-subscribed [2, 15]. This poses performance overhead to the management of erasure-coded data.

Recovery is expensive: Node failures are common in storage clusters [14], and recovery is necessary. We consider two types of recovery of node failures: (i) *crash recovery* for permanently lost data (e.g., due to disk crashes) and (ii) *degraded reads* to temporarily unavailable data (e.g., due to power outages or reboots). In both cases, recovery is performed when users access lost or unavailable data due to node failures, by retrieving enough data and parity blocks from other surviving (non-failed) nodes to reconstruct failed data. Thus, it triggers a significant amount of traffic. Field measurements [24] show that recovery generates nearly 200TB of traffic per day in production clusters.

Performance degradation of analytics applications: Analytics paradigms, such as MapReduce [9] and Dryad [17], are designed with replication-based storage in mind. For example, in replication, if failures happen, MapReduce schedules tasks to run on other nodes with the replicas so as to preserve locality; in erasure coding, MapReduce tasks need to retrieve data from other surviving nodes to recover failed data. Thus, erasure coding can increase the overall analytics runtime. However, the actual performance implication of erasure coding on MapReduce remains largely unexplored in the literature.

B. Our Work

In this paper, we study one performance challenge of deploying erasure coding, namely: *how to make MapReduce perform efficiently on erasure-coded storage clusters*. Our previous work [19] argues that traditional locality-first scheduling (LF) in MapReduce can *hurt* the performance of MapReduce when it operates on erasure-coded storage in failure mode. Thus, we propose *degraded-first scheduling* (DF) [19], which improves MapReduce performance on erasure-coded storage by carefully scheduling different types of map tasks, so that degraded map tasks (i.e., tasks that process lost or unavailable data blocks) will not compete for network bandwidth.

On the other hand, our basic degraded-first scheduling design only focuses on task-level scheduling of a single MapReduce job. We show that when there are multiple running MapReduce jobs, the basic design may generate lingering degraded tasks at the end of a MapReduce job, and such lingering degraded tasks can compete for network bandwidth with other degraded tasks of running MapReduce jobs. Thus, we propose heuristics to enhance our degraded-first scheduling design for the multi-job setting.

We conduct discrete-event simulations based on CSIM20 [8], and demonstrate the performance gains of our enhanced DF in a multi-job scenario. Specifically, we show that the enhanced DF reduces the MapReduce runtime of LF by 22.9-28.8%, and also suppresses the outliers that have significantly long runtime as found in LF. Furthermore, the enhanced DF reduces the MapReduce runtime of the basic DF by 4.5-11.5%.

The rest of the paper proceeds as follows. Section II presents the background details and reviews the basic design of degraded-first scheduling. Section III discusses the problem of our basic design in multi-job scheduling and proposes heuristics to address the problem. Section IV presents evaluation results based on discrete-event simulations. Section V reviews related work, and finally Section VI concludes the paper and presents future work.

II. Background

In this section, we provide background details of an erasure-coded storage cluster. We also review the design of degraded-first scheduling [19].

A. Basics

We consider a storage cluster that stores files as a collection of fixed-size *blocks* over multiple *nodes* (i.e., server machines). Nodes are typically organized in *racks*, such that nodes within the same rack are connected via the same top-of-rack switch, while different racks are connected via a network core composed of core/aggregation switches [2]. Cross-rack links are often over-subscribed [2, 15]. In this work, we assume that cross-rack links are the performance bottleneck.

With erasure coding, an (n, k) code encodes every group of k blocks into $n - k$ parity blocks (see Section I). To provide fault tolerance against node and rack failures, the n data and parity blocks are often distributed to the nodes in different racks [16, 22]. Also, the erasure-coded data is typically constructed *asynchronously* in the background [13], in which all blocks are first replicated when being written to a storage cluster, and the cluster later transforms the replicas into erasure-coded data.

MapReduce [9] processes data in units of *jobs*. Each job is composed of multiple *tasks* of two types: a *map* task processes an input block and generates intermediate outputs, and a *reduce* task collects and processes the outputs from different map tasks through a *shuffle* operation and produces final outputs. MapReduce uses a *master* node to coordinate multiple *slave* nodes for task executions. For brevity, we refer to the slave nodes simply as "nodes". MapReduce allocates resources by assigning each node a fixed number of *slots*, such that a node can run a map or reduce task only when a free slot is available. Since moving data over the network is expensive, MapReduce emphasizes *locality* and attempts to run a map task on a node that stores the input block to avoid cross-rack transfers (note that reduce tasks cannot exploit locality as they need to collect outputs from multiple map tasks). We can categorize map tasks into two types: *local tasks* process blocks stored in the same node or the same rack (assuming that intra-rack bandwidth is sufficient), and *remote tasks* download and process blocks from other racks. By default, MapReduce scheduling gives a higher priority to launch local tasks, followed by remote tasks. We call this *locality-first scheduling* (LF).

In erasure-coded storage, MapReduce also emphasizes locality of map tasks as in replication-based storage. One key difference is that when MapReduce operates in failure mode, it performs a new type of map tasks called *degraded tasks*, which perform degraded reads by retrieving data from other surviving nodes to reconstruct the failed block for processing. In Facebook's Hadoop [12], its LF implementation gives degraded tasks the lowest priority and schedules them after local and remote tasks.

In failure mode, we find that LF launches degraded tasks after all local tasks are launched. Thus, the degraded tasks are launched together, and they compete for network bandwidth to perform degraded reads and retrieve data from other surviving nodes, which typically resides in other racks. This significantly increases the overall MapReduce runtime.

B. Degraded-First Scheduling

Degraded-first scheduling (DF) [19] is a task-level scheduling algorithm that improves MapReduce performance on erasure-coded storage in failure mode. The idea is that in failure mode, we evenly spread the launch of degraded tasks among the whole map phase, so as to (i) use the idle network bandwidth while local tasks are running and (ii) avoid bandwidth competitions for degraded reads among degraded tasks.

We compare LF and DF via an example. Figure 1(a) shows a two-rack cluster with five nodes (denoted by Nodes 1 to 5). Suppose that the cluster has 12 data blocks, and we use (4,2) erasure coding. This leading to six stripes: the i-th stripe ($0 \leq i \leq 5$) has two data blocks $B_{i,0}$ and $B_{i,1}$ and two parity blocks $P_{i,0}$ and $P_{i,1}$. Each node is allocated two slots to run at most two map tasks simultaneously. Let the processing time of a map task be 10s, and the cross-rack transfer time of a block when the network is idle be also 10s. Now, consider the case where a MapReduce job processes the stored data while Node 1 is failed, so the failed blocks $B_{0,0}$, $B_{1,0}$, $B_{2,0}$, and $B_{3,0}$ will be processed by degraded tasks. Figure 1(b) shows the map task activities for default scheduling, which first assigns local tasks and finally degraded tasks. Suppose that Nodes 2, 3, 4, and 5 are scheduled to run degraded tasks

Algorithm 1 Basic Degraded-First Scheduling

```
1: while a heartbeat comes from node $s$ do
2:     for job $j$ in job list do
3:         if $j$ has a degraded task and $\lambda \geq \lambda_d$ then
4:             assign the degraded task to $s$
5:         end if
6:         assign other map slots as in LF
7:     end for
8: end while
```

for $B_{0,0}$, $B_{1,0}$, $B_{2,0}$, and $B_{3,0}$, respectively, and that each node downloads the first parity block $P_{0,0}$, $P_{1,0}$, $P_{2,0}$, $P_{3,0}$, respectively, from another node to reconstruct the failed block. Node 4 downloads $P_{2,0}$ from Node 5, and Node 5 downloads $P_{3,0}$ from Node 3. Note that nodes 4 and 5 do not compete for the same download link. However, Nodes 2 and 3, located in the same rack, need to compete for the download link of the rack in order to download $P_{0,0}$ and $P_{1,0}$ from another rack, respectively. This doubles the download time, from 10s to 20s. The entire map phase lasts for 40s. On the other hand, DF tries to launch some of the degraded tasks earlier. Figure 1(c) shows the map task activities for DF, which moves two degraded tasks for processing $B_{0,0}$ and $B_{2,0}$ to the beginning of the map phase. This eliminates bandwidth competition, and reduces the map-phase duration to 30s (25% saving).

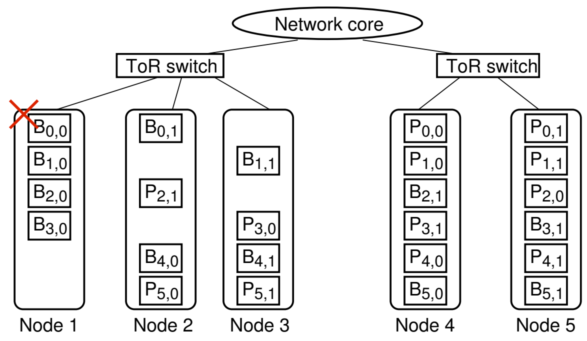
Algorithm 1 shows the high-level pseudo-code of the basic DF algorithm (the detailed pseudo-code is in [19]). The idea is to launch a degraded task only if the fraction of degraded tasks that have been launched (denoted by λ_d) is no more than the fraction of all map tasks that have been launched (denoted by λ). This controls the pace of launching degraded tasks, and hence keeps the degraded tasks separated in the whole map phase and prevents network competition among them. Note that DF has exactly the same behavior as LF when there is no node failure.

Our work [19] studies DF via mathematical analysis, simulations, and testbed experiments. Our preliminary testbed experiments show that DF improves MapReduce runtime by 27% in failure mode when compared to LF. We refer readers to [19] for our detailed results.

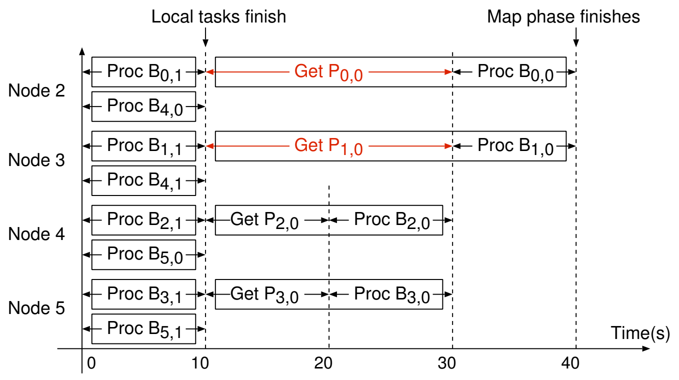
III. Integration with Job-Level Scheduling

Our previously proposed DF [19] operates at the task level, and assumes the default first-in-first-out scheduling (FIFO) at the job level. In FIFO, MapReduce sorts the jobs by their submission times, and schedules tasks from one job (i.e., the earliest submitted job) at a time. However, in general, MapReduce may need to schedule tasks from multiple jobs simultaneously. In this section, we explore how to integrate DF with general job-level scheduling.

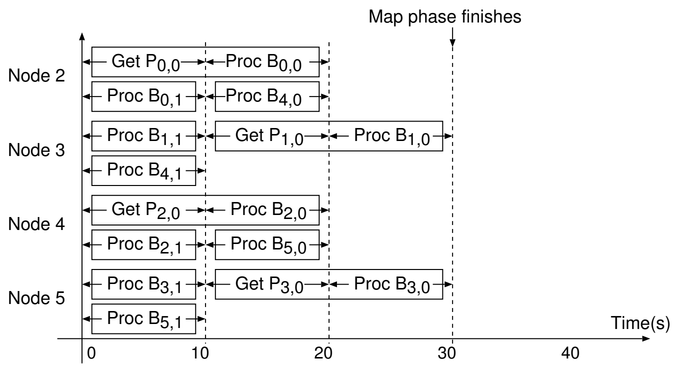
The default FIFO suffers from the starvation problem since short jobs cannot be scheduled until all the tasks of the long jobs ahead have been scheduled. In this work, we consider a special job-level scheduling scheme called *fair sharing* (FS) [30], whose objective is to fairly allocate computational resources among multiple jobs. Specifically, FS attempts to



(a) Two-rack cluster with 12 data blocks and 12 parity blocks, using (4,2) erasure coding



(b) Locality-first scheduling (LF)



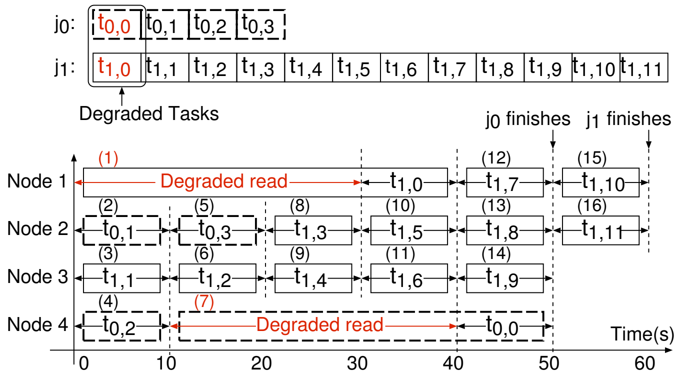
(c) Degraded-first scheduling (DF)

Fig. 1. Map task activities of LF and DF when Node 1 is failed. We assume that each node has two map slots.

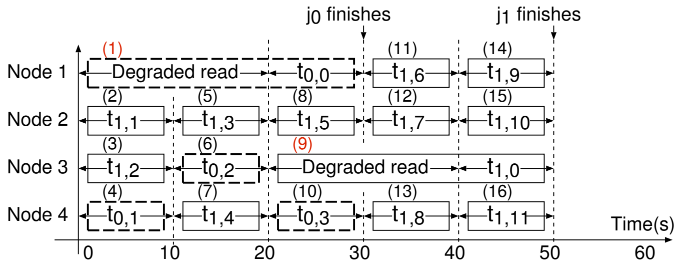
evenly distribute all available map slots to all jobs. MapReduce sorts the jobs by how many map slots they currently hold. The job that occupies the fewest map slots has the highest priority to have its tasks assigned to available map slots. Here, we consider how DF behaves when FS is used. Note that the same work [30] also proposes a more robust job-level scheduling scheme called *delay scheduling* to achieve both locality and fair sharing, and we pose the study of DF on delay scheduling as future work.

A. Lingering Degraded Task Problem

We argue that the basic DF may suffer from the *lingering degraded task problem* when we schedule tasks for multiple jobs simultaneously. We consider the case of FS, and explain



(a) Basic DF



(b) With lingering degraded task elimination

Fig. 2. Comparison of map task activities with and without lingering degraded task elimination. The numbers in the brackets show the order of the blocks being assigned a map slot.

the problem and propose our solution using an example shown in Figure 2.

Consider a cluster with four surviving nodes (denoted by Nodes 1 to 4). The cluster is about to process two map-only jobs: a short job j_0 with four map tasks and a long job j_1 with 12 map tasks. Let $t_{a,b}$ denote the b-th map task of j_a, where $a = 0$ or 1, and $b \geq 0$. Suppose that job j_0 needs to process a degraded task $t_{0,0}$ and job j_1 also needs to process a degraded task $t_{1,0}$, and both degraded tasks share the same network bandwidth. We assume that each node is allocated one map slot. It takes 10s to process each normal or degraded map task for both jobs, and an additional 20s for a degraded task to issue a degraded read to an unavailable block when the network is idle.

We now apply the basic DF (Algorithm 1) for scheduling. When assigning the 1st degraded task to a node, DF can choose either $t_{0,0}$ or $t_{1,0}$. Suppose that $t_{1,0}$ is selected. Figure 2(a) shows the corresponding map task activities. When DF assigns the 7th task, it should choose a task of job j_0 to ensure fair sharing of both j_0 and j_1 across all nodes. However, j_0 has only one remaining task $t_{0,0}$, which is a degraded task. Thus, DF assigns $t_{0,0}$. We see that both degraded tasks $t_{0,0}$ and $t_{1,0}$ issue degraded reads simultaneously and compete for the same network bandwidth (as we have assumed). Thus, both degraded tasks have their degraded read time extended from 20s to 30s. While deferring the launch of the *lingering* degraded task $t_{0,0}$ may eliminate the competition with $t_{1,0}$, it also defers the completion of job j_0 and impairs the fairness of scheduling.

Our observation is that a lingering degraded task appears when a job has launched all normal (i.e., local or remote) tasks. This problem is likely to occur when a job satisfies the following two conditions: (i) the job has a small number of unassigned map tasks and (ii) the job has at least one unassigned degraded task. Our intuition is that we should give a higher priority to assign a degraded task of this type of jobs to avoid having any lingering degraded task.

We add a new function called CHOOSEDEGRADEDTASK to decide which job should have one of its degraded tasks assigned to a node in order to remove any lingering degraded task. Suppose that a node is considered to be suitable for launching a degraded task. Then among all scheduled jobs, CHOOSEDEGRADEDTASK chooses the one that has the least number of unassigned map tasks and has at least one degraded task. It finally returns a degraded task of the selected job.

For example, when CHOOSEDEGRADEDTASK is added, the revised degraded-first scheduling algorithm first assigns the degraded task $t_{0,0}$ of j_0, since j_0 has the least number of unassigned map tasks. Figure 2(b) shows the resulting map task activities. We see that both the runtimes of j_0 and j_1 are reduced by 20s (or 40%) and 10s (or 16.7%), respectively.

B. Enhanced Algorithm

Algorithm 2 shows the high-level pseudo-code of the enhanced version of DF, which includes CHOOSEDEGRADED-TASK to address the lingering degraded task problem. In addition, we add a heuristic function ASSIGNTORACK [19] to ensure that multiple degraded tasks are not assigned to the same rack at nearly the same time, thereby avoiding competition of network bandwidth through the same top-of-rack switch. Specifically, we keep track of the duration since the last degraded task is assigned to each rack r (denoted by T_r), and the average duration $E[T_r]$ across all racks. We avoid assigning a degraded task to a node in rack r if r satisfies both of the following conditions: (i) if $T_r < E[T_r]$, and (ii) if T_r is less than some threshold. For condition (ii), we choose the threshold as the expected time for a degraded read [19]. Satisfying both conditions implies that rack r has just recently launched a degraded task (for condition (i)) and that the degraded task is still performing a degraded read (for condition (ii)). If we launch another degraded task to rack r, it will unnecessarily compete for network bandwidth.

IV. EVALUATION

In this section, we evaluate via simulations different MapReduce scheduling schemes for erasure-coded storage clusters. We consider the scenario where MapReduce schedules multiple jobs and operates in failure mode. Our goal is to show that the enhanced DF (Algorithm 2) outperforms both LF (the default MapReduce scheduling) and the basic DF (Algorithm 1). To help our discussion, we denote the enhanced DF and the basic DF by EDF and BDF, respectively.

A. Setup

We conduct our evaluation on a MapReduce simulator developed in our previous work [19]. The simulator is written

Algorithm 2 Enhanced Degraded-First Scheduling

```
 1: function AssignToRack(rack $r$)
 2:     if $T_r < \min(E[T_r], \text{threshold})$ then
 3:         return false
 4:     end if
 5:     return true
 6: end function

 7: function ChooseDegradedTask()
 8:     sort jobs by the increasing number of unassigned map tasks
 9:     for job $j$ in job list do
10:         if $j$ has a degraded task then
11:             return the degraded task
12:         end if
13:     end for
14: end function

15: procedure Main Algorithm
16:     while a heartbeat comes from node $s$ do
17:         if $\lambda \geq \lambda_d$ and AssignToRack(rackID($s$)) == true then
18:             assign ChooseDegradedTask() to $s$
19:         end if
20:         for job $j$ in job list do
21:             assign other map slots as in LF
22:         end for
23:     end while
24: end procedure
```

in C++ based on CSIM20 [8], and performs discrete event simulations. The simulator deploys *processes* to simulate map and reduce tasks. We extend the simulator to consider job-level scheduling with FS, and simulate the cluster and workload settings similar to [30]. Specifically, we consider a cluster of 100 nodes evenly grouped into 10 racks (i.e., 10 nodes per rack). The nodes are connected via top-of-rack switches and a network core, all of which have bandwidth 1Gb/s. We generate a workload of 100 map-only jobs, detailed in Table I. We classify the jobs into three types, namely long jobs, medium jobs, and short jobs, according to the number of map tasks per job. We randomly permute the 100 jobs, and submit jobs such that the job inter-arrival times follow an exponential distribution with mean 25s. Also, the expected map-task processing time of each job follows a normal distribution with mean 20s and standard deviation 5s. We allocate each node four map slots. Also, we encode the data blocks with (20,15) erasure coding. In each run of simulation, we randomly pick a node as the failed node to simulate a single-node failure, which is the most commonly found in production clusters [14, 16, 24].

We measure the MapReduce runtime of each of the 100 jobs in failure mode, and repeat the measurements for 30 runs. We obtain the runtime of each job from the average of the 30 runs. We also normalize the runtime of each job over its runtime in normal mode (i.e., the cluster has no node failure while MapReduce operates).

B. Results

Figure 3 shows the normalized runtimes of LF, BDF, and EDF for short, medium and long jobs, averaged over all jobs of the same type (the endpoints of the error bars are the minimum and maximum normalized runtimes among all jobs of the same type). Note that the normalized runtimes of short jobs are generally higher than those of medium and long jobs, because the overhead of a degraded task of a small job becomes more dominant. Overall, we observe that EDF outperforms LF and BDF in all the three job types.

TABLE I
DETAILS OF MULTI-JOB WORKLOAD.

Job type	# of map tasks per job	Number of jobs
Long	4,800	4
Medium	750	8
Short	150	88

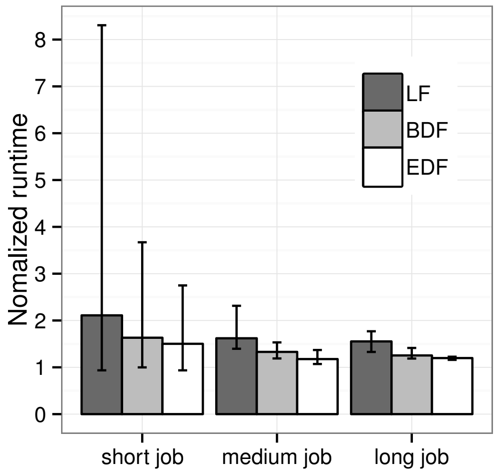


Fig. 3. Normalized runtimes of LF, BDF, and EDF (the endpoints of the vertical bars are the minimum and maximum runtimes among all jobs of the same type).

Compared to LF, EDF reduces the normalized runtime by 28.8% (2.10 to 1.50), 27.4% (1.62 to 1.18) and 22.9% (1.55 to 1.20) for short, medium, and long jobs, respectively. In some cases, we see that some short jobs under LF have significantly long runtimes (e.g., up to 8.31× of normal mode). The reason is that LF schedules the degraded tasks to the end of a job. Such degraded tasks compete with the network bandwidth with the degraded tasks of the running medium or long jobs. This significantly increases the degraded read time, thereby increasing the overall job runtime as well. On the other hand, EDF reduces the maximum runtime of short jobs to 2.75× of normal mode.

Compared to BDF, EDF reduces the normalized runtime by 8.0% (1.63 to 1.50), 11.5% (1.32 to 1.17), 4.5% (1.25 to 1.19) for short, medium, and long jobs, respectively. We also note that EDF reduces the maximum runtime of BDF by 25.1%.

V. Related Work

The application of erasure coding in storage clusters has been extensively studied in the literature. Some studies propose new erasure code designs to improve recovery performance, for example, by minimizing bandwidth [10] or I/O [11, 16, 18, 23, 25, 27]. Some studies implement and evaluate erasure coding on existing storage clusters such as Azure

[16] and HDFS [11, 13, 20, 21, 25, 27, 32], or design new erasure-coding-based storage clusters [1, 6, 29]. In particular, the studies [25, 27, 32] evaluate MapReduce performance on erasure-coded storage, but do not customize the MapReduce design specifically for erasure-coded storage. On the other hand, there are many proposals on enhancing MapReduce performance by taking into account heterogeneous hardware [4, 7, 31] and access skewness [3], yet they target replication-based storage only. Our work proposes an erasure-coding-aware MapReduce scheduling algorithm to complement the prior studies.

VI. Conclusions and Future Work

This paper studies the problem of how to improve the performance of MapReduce when it operates on erasure-coded storage in failure mode. We enhance our previously proposed degraded-first scheduling algorithm when it is integrated with job-level scheduling. In particular, we propose a heuristic to eliminate lingering degraded tasks when MapReduce uses fair sharing for job-level scheduling. We show via discrete event simulations that our enhanced degraded-first scheduling outperforms the default locality-first scheduling and the basic degraded-first scheduling in a large-scale storage cluster.

Our future work includes the following: (i) examine the integration of degraded-first scheduling with more sophisticated job-level scheduling designs, such as delay scheduling [30], (ii) examine the use of degraded-first scheduling on new erasure coding schemes (e.g., regenerating codes [10] and locally repairable codes [11, 16, 27]), and (iii) extend our prior HDFS-based implementation [19] and conduct more extensive testbed experiments in general scenarios.

Acknowledgments

This work was supported in part by grants AoE/E-02/08 and ECS CUHK419212 from the University Grants Committee of Hong Kong.

References

[1] M. Abd-El-Malek, W. Courtright II, C. Cranor, G. Ganger, J. Hendricks, A. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. Sambasivan, et al. Ursa Minor: Versatile Cluster-based Storage. In *Proc. of USENIX FAST*, Dec 2005.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Proc. of ACM SIGCOMM*, 2008.

[3] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: Coping with Skewed Content Popularity in MapReduce Clusters. In *Proc. of ACM EuroSys*, Apr 2011.

[4] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *Proc. of USENIX OSDI*, page 14, 2010.

[5] J. Bloemer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman. An XOR-Based Erasure-Resilient Coding Scheme. Technical Report TR-95-048, International Computer Science Institute, UC Berkeley, Aug 1995.

[6] J. C. W. Chan, Q. Ding, P. P. C. Lee, and H. H. W. Chan. Parity Logging with Reserved Space: Towards Efficient Updates and Recovery in Erasure-coded Clustered Storage. In *Prof. of USENIX FAST*, Feb 2014.

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

[8] CSIM. http://www.mesquite.com/products/csim20.htm.

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

[10] 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.

[11] K. S. Esmaili, L. Pamies-Juarez, and A. Datta. CORE: Cross-Object Redundancy for Efficient Data Repair in Storage Systems. In *Proc. of IEEE BigData*, 2013.

[12] Facebook. Facebook's Realtime Distributed FS based on Apache Hadoop 0.20-append. https://github.com/facebookarchive/hadoop-20.

[13] B. Fan, W. Tantisiriroj, L. Xiao, and G. Gibson. DiskReduce: RAID for Data-Intensive Scalable Computing. In *Proc. of Annual Workshop on Petascale Data Storage (PDSW)*, Nov 2009.

[14] 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.

[15] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *Proc. of ACM SIGCOMM*, Aug 2009.

[16] 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.

[17] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *Proc. of ACM EuroSys*, Jun 2007.

[18] 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.

[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] R. Li, J. Lin, and P. P. C. Lee. CORE: Augmenting Regenerating-Coding-Based Recovery for Single and Concurrent Failures in Distributed Storage Systems. In *Proc. of IEEE MSST*, May 2013.

[21] R. Li, J. Lin, and P. P. C. Lee. Enabling Concurrent Failure Recovery for Regenerating-Coding-Based Storage Systems: From Theory to Practice. *IEEE Trans.on Computers*, 2014.

[22] 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.

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

[24] 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 FacebookWarehouse Cluster. In *Proc. of USENIX HotStorage*, 2013.

[25] 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.

[26] 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.

[27] 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*, 2013.

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

[29] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable Performance of the Panasas Parallel File System. In *Proc. of USENIX FAST*, 2008.

[30] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *Proc. of ACM EuroSys*, pages 265–278, 2010.

[31] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *Proc. of USENIX OSDI*, 2008.

[32] Z. Zhang, A. Deshpande, X. Ma, E. Thereska, and D. Narayanan. Does Erasure Coding Have a Role to Play in my Data Center? Technical Report MSR-TR-2010-52, Microsoft Research, May 2010.