Supplementary Material for FCS-18403

Yanwen Xie
Wuhan National Laboratory for Optoelectronics
Huazhong University of Science and Technology
Wuhan, China

I. Introduction

We provide the following contents in this supplementary material (compared with the previous version [1] presented in ICPP):

- We rewrote the design and the implement of SICE
- We re-evaluated the simulation of SICE with settings for larger clusters.
- We presented the evaluation results of "Influence on MapReduce Applications" more clearly.
- We added a new test: recovering a single-node failure.
- We described a tradeoff in the design of ASICE, the implement of ASICE in Hadoop and the detail about the simulation environment of ASICE.

II. SICE

A. Design

Fig. 1 shows the procedure of SICE. It first constructs stripes non-sequentially. Then it performs encoding according to local computing and writes single-replica parity blocks. Finally it discards spare replicas to save the storage cost.

1) Constructing stripes: CrossrackTraffic represents the cross-rack traffic for reading data blocks during encoding. *ColocatedBlocks* represents the co-located blocks after encoding. No *CrossrackTraffic* when encoding a stripe means every block in the stripe has a replica located in the same rack. No *ColocatedBlocks* in a stripe means blocks in the stripe has replicas located in the different racks. Therefore, for every stripe, SICE attempts to search for the blocks which have a replica in the same rack and a replica in the different racks. In other words, the blocks should match the block pattern shown in Fig. 2.

In order to search for the perfect block pattern in the dataset, SICE gets the whole layout of the dataset and classifies blocks to racks which own the replicas of blocks. Then SICE chooses a rack as *WorkRack* where the stripe is supposed to be encoded. How to choose *WorkRack* will be discussed in the next subsection.

Afterwards, SICE iterates the blocks in the *WorkRack*, tests and finds the blocks meeting the pattern. Every chosen block has a replica at *PreferRack* where the replica is supposed to survive for the block after encoding. SICE maintains a *PreferRack* blacklist for a stripe. The blacklist is empty at first. SICE chooses the new blocks which have replicas at the racks outside the blacklist into the stripe. The *PreferRack*s of the chosen blocks will then be put into the blacklist so that no co-located blocks will exist after encoding.

Fig. 3 shows an example with 6 racks and EC(3, 5). SICE chooses *Rack1* (with block 0, 2, 3, 4 ...) as the *WorkRack*, and then it iterates the blocks in turn 0, 2, 3, 4 ... Block 0 and 2 are chosen into the stripe, and their *PreferRack*s are *Rack1* and *Rack2*. The *PreferRack* blacklist right now is $\{Rack1, Rack2\}$. Block 3 has no replicas at the racks outside the blacklist, so SICE abandons it. Afterwards, Block 4 has a replica in *Rack3*, so it is chosen into the stripe.

When there are no blocks matching the pattern, SICE looses the limit of *CrossrackTraffic* because reliability is much more important. At that point, SICE will select blocks from the racks outside the blacklist to fill the incomplete stripe even though the selected blocks have no replicas in *WorkRack*. Even at the last gasp when there are no blocks from the racks outside the blacklist, SICE avoids *ColocatedBlocks* by accepting incomplete stripes. However, these situations are not common. The results of the following evaluation show that they only result in nearly zero incomplete stripes and acceptable cross-rack traffic.

After finding all blocks for the stripe, SICE will then decide the distribution of parity blocks. It checks the blacklist and finds random racks outside the blacklist for parity blocks. For the example of Fig. 3, the blacklist is $\{Rack1, Rack2, Rack3\}$. SICE chooses two random racks in $\{Rack0, Rack4, Rack5\}$, and the final decision is *Rack4* for the first parity block and *Rack5* for the second.

2) Iterating WorkRack: SICE selects the *WorkRack* in a round-robin way among the racks with enough blocks. It helps to make sure all racks are assigned to as approximately equivalent encoding tasks as possible. 'Enough blocks' means a customized factor times the number of data blocks per stripe. The factor has only a moderate impact because it only affects the end of the period of constructing striping. Therefore, the factor is casually set to 1.5 in this paper.

It's well known that uniform distribution of data helps to balance the load. We have considered selecting the rack with most blocks as *WorkRack* to balance data in different racks because replicas in *WorkRack* are mostly discarded after encoding. However, it exposes the imbalance of encoding

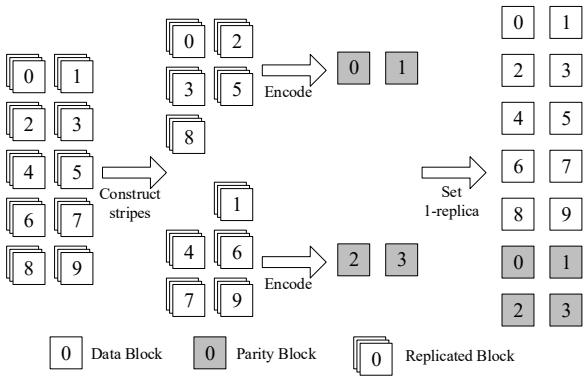


Figure 1: **Design of SICE.** It constructs stripes non-sequentially, performs encoding, writes single-replica parity blocks, and sets all data blocks to single-replica.

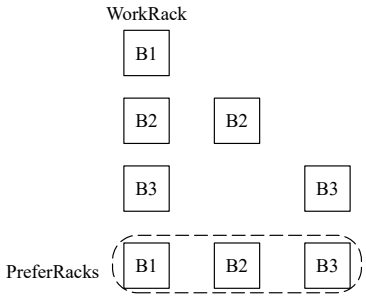


Figure 2: **A perfect block pattern.** Every column represents a rack. Encoding at *WorkRack* and choosing replicas according to *PreferRack*s benefits the transition period best.

tasks in our early tests. The racks with more blocks will have more tasks while some racks are free. In contrast, SICE works well by a simple round-robin way. It does not cause severe imbalance in the distribution of blocks. That is because it is more likely to select blocks from the racks with more blocks to the stripes and discard replicas in the racks, and lots of stripes will improve the balance of data distribution.

When there are no racks with enough blocks, SICE selects the rack with most blocks as *WorkRack*. It happens only at the end of the period of constructing striping, so it has only a moderate impact on the cross-rack traffic.

3) Encoding: After constructing stripes, SICE encodes the stripes at *WorkRack*s to generate parity blocks. It exploits local computing just like LOCAL to reduce cross-rack traffic. However, because of non-sequential striping, SICE can save more traffic than LOCAL.

Many classes of erasure codes [2]–[4] are proposed to improve repair performance. SICE treats the erasure code

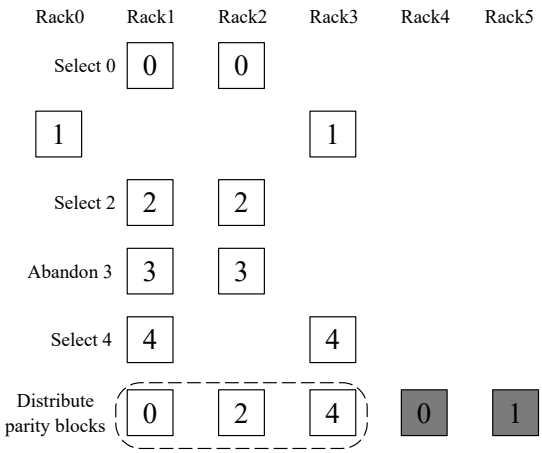


Figure 3: **An example for constructing a stripe.** SICE iterates blocks in *Rack1*, selects data block 0, 2, 4 in turn, and distributes parity blocks.

as a black box, so it can work well under all of them.

4) Writing single-replica parity blocks: After encoding, SICE writes single-replica parity blocks, which reduces resource consumption greatly. When node failure happens, it may cause:

- Some replicas of data blocks are missing. As data blocks are multi-replica, they can tolerate failure.
- Some replicas of parity blocks are missing after their encoding tasks are completed. As parity blocks are single-replica, they are lost. They can be recovered by re-running the encoding tasks or decoding after transition. As a stripe may have several parity blocks and only a part of them are lost, it causes more consumption or more complexity to re-run the encoding tasks. SICE just ignores the failure in the transition period and recovers the lost with erasure codes after conversion.
- Parity blocks are incomplete because their writing processes are interrupted by the failure. As parity blocks are written to single nodes, the clients can't re-construct the writing pipelines and they will throw exceptions which cause the failure of the encoding tasks. However, the system will recognize all failed tasks and re-run these tasks. New tasks will write parity blocks to new nodes and everything will go on well.

Therefore, it will not cause any problems of reliability to write single-replica parity blocks.

5) Discarding spare replicas: After all parity blocks are written successfully, SICE discards spare replicas to save the storage cost. For every data block, it checks *PreferRack* of the block, keeps the replica at *PreferRack* and discards the other replicas. Because blocks in the same stripe have different *PreferRack*s, there will be no co-located blocks at

last.

Sometimes when there are no replicas for a block at its *PreferRack* because of failure, SICE will check whether there is any replica at the rack outside the blacklist or not. If so, SICE keeps the replica and discards the other replicas. If not, SICE will choose a random rack outside the blacklist, copy the block to the rack and discard the other replicas. All blocks from the same stripe finally have replicas in different racks.

6) Metadata management: SICE employs non-sequential striping, so extra metadata should be maintained to tell the ownerships between stripes and blocks. First, SICE maintains an integer field for every block to indicate the ID of the stripe the block belongs to. Thus when a block is missing, SICE can find out the corresponding stripe according to the stripe ID to recover the missing block. Second, SICE maintains a list of stripe structures for all stripes to hold the IDs of the blocks belonging to the stripes.

Because the extra metadata are required only when any failure happens and their size is not modest especially the size of the list of stripe structures, they are often stored outside the original metadata structures. For example, when encoding a folder "/root/foo", SICE will store all extra metadata in the file "/raidfs/root/foo/.meta" rather than add extra fields to the block metadata structure or node structure. Thus no extra metadata burden is added when responding to I/O requests normally. More detail about the metadata management will be presented in the implement of SIZE.

7) When cold data turn hot: This paper is focusing on the scenario when hot data turn cold. The SICE method has not considered the scenario when cold data turn hot, so a single conversion method converting erasure-coded data to replicated data can be added to complement SICE. When cold data turn hot, the conversion method first enlarges the number of replicas of data blocks and distributes new replicas according to the specific replication distribution method. After that, parity blocks are discarded and metadata about stripes are removed.

B. Implement

To accelerate the conversion, it is important to perform encoding parallel. Research works [5]–[7] are based on HDFS-RAID, which runs a MapReduce job to encode all stripes parallel. Thus we implement our method in Hadoop v2.7.1 based on MapReduce and HDFS-RAID. In HDFS-RAID, RaidNode is the server program responsible for managing erasure codes. Most functions of SICE are implemented in RaidNode.

1) Constructing stripes: SICE starts as RaidNode gets all block locations of the dataset and constructs stripes according to the design of SICE. Information of all stripes will then be written to HDFS and become the input of the following MapReduce job. The stripe information format is shown in Table I. Every stripe has a *StripeIndex*, a *WorkRack* and a number of blocks. Every block has a *FileName*, a *BlockIndex* and a *PreferRack*. RaidNode can locate the unique block with *FileName* and *BlockIndex* and get its *BlockId*.

Table I: An example of constructed stripes.

StripeIndex	WorkRack	FileName	BlockIndex	PreferRack
0	Rack0	0.data	0	Rack0
		0.data	3	Rack1
		0.data	4	Rack2
1	...	...	.	...

2) Running MapReduce: The encoding MapReduce job will start soon after its input is ready. As Fig. 1 shows, the job will split all stripes into multiple groups according to their *WorkRack* and start one map task per group. Each map task is responsible for encoding one or several stripes. *StripesPerMap* represents the maximum number of stripes per map task. Larger *StripePerMap* helps to reduce the number of map tasks, which helps to reduce the cost of initialization and finalization of tasks. Thus the conversion time is reduced. However, it makes the conversion unsteady because feature of small tasks benefits the flexibility of scheduling and enhances the balance of the cluster. There is a trade-off between them. The best *StripesPerMap* is suggested to be the smallest number which does not speed up the conversion greatly. *StripesPerMap* can be calculated by the size of dataset, the cluster scale and a customized factor $f(f > 0)$:

$$StripesPerMap = \frac{the\ number\ of\ stripes}{RackCount * f} \quad (1)$$

In Hadoop, tasks are classified to local tasks and remote tasks according to whether the required data are located in the computing nodes or not. The task scheduler of Hadoop schedules local tasks first to exploit local computing. LOCAL from EAR modifies the task scheduler to force tasks to be executed strictly in *WorkRack*. SICE keeps the original task scheduler and makes use of the free resource. It may cause some *CrossrackTraffic* when the cluster is not heavily busy. There is a trade-off between them and it is easy to modify SICE to work in the LOCAL way.

The parity blocks will be output into the racks selected by SICE. When opening a HDFS file for writing, SICE sets the parameter *replicas* to 1 and *preferHost* to a random host in *PreferRack*. We name the processing parity file with a timestamp and rename it without the timestamp after success. When some encoding map tasks fail, the re-run tasks will check whether the particular files exist or not so that all encoded stripes will not be re-encoded. It also helps to make SICE compatible with the speculative mechanism of Hadoop. The speculative mechanism is that two same

tasks run concurrently and only the result of the fast task is accepted.

3) Updating metadata: After the MapReduce job is completed successfully, RaidNode will update related metadata. According to Table I, only *StripeIndex* is required to be added to the metadata of block so that when a block is lost, the system can get the whole information of the block's stripe. The stripe structure contains only *StripeIndex* and every block's *BlockId*.

Because the extra metadata is required only when failure happens, it is maintained by RaidNode and stored as a normal file in HDFS, called the metadata file, outside the metadata server of HDFS. Thus the original metadata structures haven't been changed and SICE puts no more storage occupation on the metadata server and no more overhead when processing normal I/O requests. For example, encoding the file "/root/foo.data" will generate a metadata file "/raidfs/root/foo.data.meta", and encoding the folder "/root/foo" will generate a metadata file "/raidfs/root/foo/.meta". The metadata file structure is similar to the input of a MapReduce encoder in Table I. The fields *WorkRack* and *PreferRack* are not required in the metadata file structure because they are required only when encoding. RaidNode also maintains a metadata cache so that the stripe information can be retrieved fast. The cache includes a hash map mapping *BlockId* to *StripeIndex* and an array of stripe structures which contain *BlockId*s of the blocks in the stripes. When the cache is missed, the map and the array can be constructed from the metadata file. Optionally, just like HDFS-RAID, the extra metadata can also be stored in an outside database.

4) Adjusting data blocks to single-replica: After updating the metadata, RaidNode will set the replica number of all data blocks to 1. Here, we implement our BlockDistributionPolicy for HDFS, named as SICEPolicy to make sure the replicas in *PreferRack*s survive. SICEPolicy serves blocks in RS with the default BlockDistributionPolicy and serves blocks in ECS in a new way.

SICE employs a push way to tell the metadata server all *PreferRack*s, which helps to reduce the burden of the metadata server. When constructing the stripes, RaidNode has gathered locations of all blocks and calculated the *PreferRack*s. It just pushes the results of all *PreferRack*s to the metadata server. Thus the metadata server needn't recalculate which replicas should be discarded.

C. Evaluation

1) Testbed: As Fig. 4 shows, we construct a Hadoop testbed in a small cluster with 9 nodes: one Master responsible for HDFS NameNode and YARN ResourceManager, and eight Slaves for HDFS DataNodes and YARN NodeManagers. All nodes are connected with 1000Mbps Ethernet, and each node has an Intel Xeon E5-2620 CPU, 12 GB memory (Hyundai, 1066 MHz). a disk (Seagate ST600MM0006)

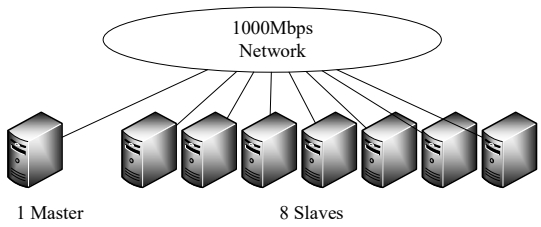


Figure 4: **Testbed.** It contains one Master and eight Slaves and the nodes are connected with 1000Mbps Ethernet.

with about 150Mbytes/sec sequential I/O throughput. The operating system is CentOS 7 with Linux 3.10. The Hadoop version is 2.7.1. Due to the limitation of cluster scale, we configure one node per rack, and deploy two-way replication and RS(3,5) (Reed-Solomon code with 3 data blocks and 2 parity blocks).

2) New simulation: In order to present the problems and show the performance of SICE in large scale, we developed a simulator and made the following simulation in the Master node of the testbed.

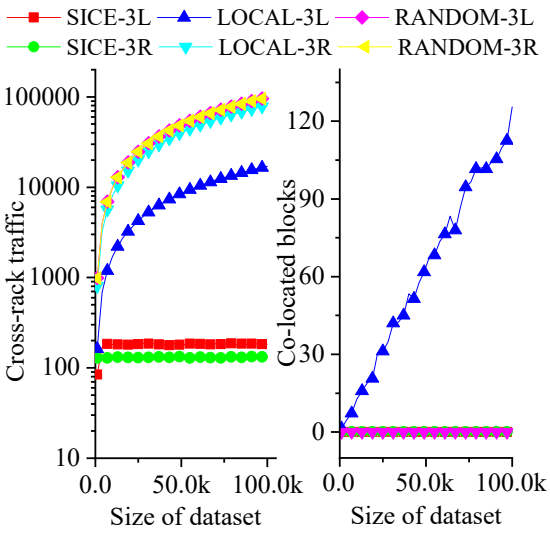
First, the simulator generates a dataset with *BlockCount* data blocks in three-way replication, and distributes them in *RackCount* racks. The dataset contains many files, and every file contains $10 \sim 20$ data blocks. Here we only simulate two distribution methods, 3L (three-way replication in nL) and 3R (three-way replication in nR). That's because Sinbad and EAR-R are close to nL and CDRM is between nL and nR.

Afterwards, every *StripeSize* data blocks are grouped into a stripe, and the simulator will choose a rack where to encode the stripe. We stop here and calculate *CrossrackTraffic* and *ColocatedBlocks*. The simulator will test 3 solutions, RANDOM from HDFS-RAID, LOCAL from EAR, and SICE. Therefore, RANDOM-3R, RANDOM-3L and LOCAL-3R, LOCAL-3L, SICE-3L, SICE-3R are compared. However, as *ColoatedBlocks* here only counts the data blocks, RANDOM and LOCAL perform the same, so we only list 4 comparison items when presenting the figures of *ColocatedBlocks*.

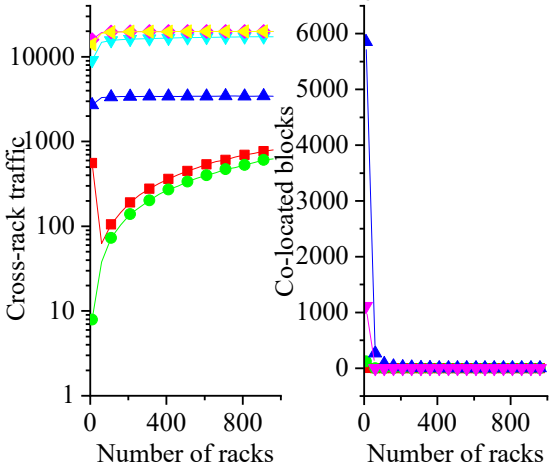
The simulation results are the average values of five-time running. We set *BlockCount* default to 20000, *RackCount* default to 200, and *StripeSize* default to 10. Fig. 5(a), Fig. 5(b) and Fig. 5(c) present how the results change as *BlockCount*, *RackCount* or *StripeSize* changes. Fig. 6 shows the time spent on selecting the blocks for constructing stripes.

Co-located blocks after encoding: As Fig. 5(a), Fig. 5(b) and Fig. 5(c) show, SICE generates no co-located blocks under both 3L and 3R. For SICE, the number of incomplete stripes is so small (only 0 to 2) that it is acceptable.

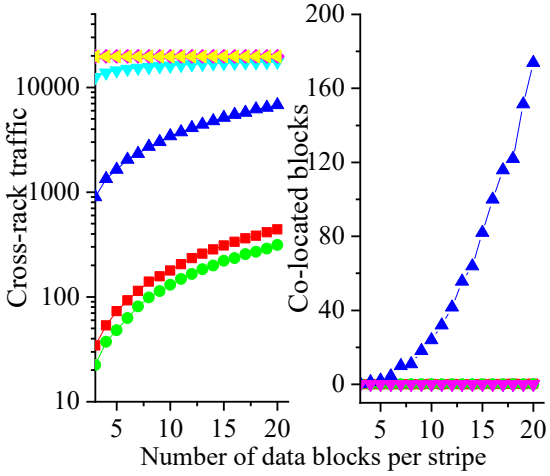
RANDOM-3R reaches the same magnitude as SICE while RANDOM-3L generates some co-located blocks. *Colocat-*



(a) *BlockCount* changes



(b) *RackCount* changes



(c) *StripeSize* changes

Figure 5: **Simulation results**. It presents the *CrossrackTraffic* (in blocks) and the *ColocatedBlocks* (in blocks) in different situations.

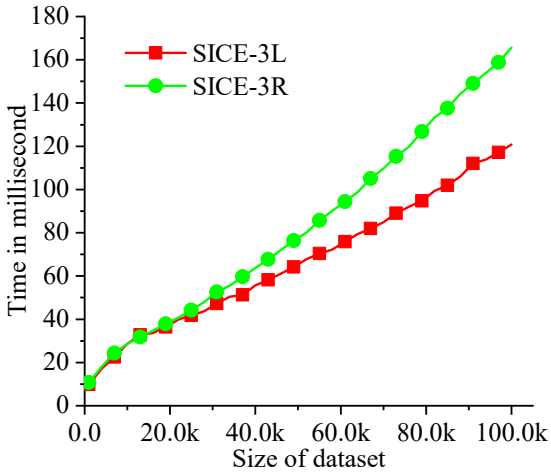


Figure 6: **Time for constructing stripes**. As data scale up, the time increases linearly.

edBlocks of RANDOM-3L increases linearly as the dataset scales up. It decreases quickly as the *RackCount* scales up because the remote racks can be chosen in a larger scope. It increases as *StripeSize* increases because more blocks are counted in the same stripes.

Cross-rack traffic in reading data blocks: As Fig. 5(a) shows, RANDOM generates large cross-rack traffic in reading data blocks under 3L and 3R. Compare to RANDOM, LOCAL-3R reduces *CrossrackTraffic* by about 25% and LOCAL-3L reduces that by one order of magnitude. However, SICE further reduces it by one more order of magnitude than LOCAL-3L. Moreover, *CrossrackTraffic* of SICE keeps almost the same as the dataset scales up. It shows highly scalable, so it's more applicable to large storage systems.

Fig. 5(b) shows that as *RackCount* scales up, *CrossrackTraffic*s of RANDOM-3R, RANDOM-3L and LOCAL-3R, LOCAL-3L keep almost the same. That's because *RackCount* is often much larger than the number of replicas, so the possibility for hitting the replica does not change a lot as *RackCount* increases. In contrast, *CrossrackTraffic* of SICE increases nearly linearly. More racks, less blocks every rack owns, so it's slightly harder to find suitable blocks for stripes. However, *CrossrackTraffic* of SICE is still lower by one order of magnitude at least than that of RANDOM and LOCAL. There is a point of inflection for SICE-3L because the cluster is small. Tt is really hard to find blocks with replicas in different rack and SICE makes concessions on cross-rack consumption for the promise that no co-located blocks exist.

Fig. 5(c) shows that as the *StripeSize* increases, *CrossrackTraffic* of RANDOM-3R, RANDOM-3L and LOCAL-3R increases in very small speeds. That's because they use a random method or a random distribution, and *StripeSize* is usually much smaller than *RackCount*. In contrast, LOCAL-

3L, SICE-3R and SICE-3L all schedule encoding tasks according to the local computing, so more blocks per stripe, more difficult it is to read all blocks locally. Thus their cross-rack traffic increases a little more quickly as *StripeSize* increases.

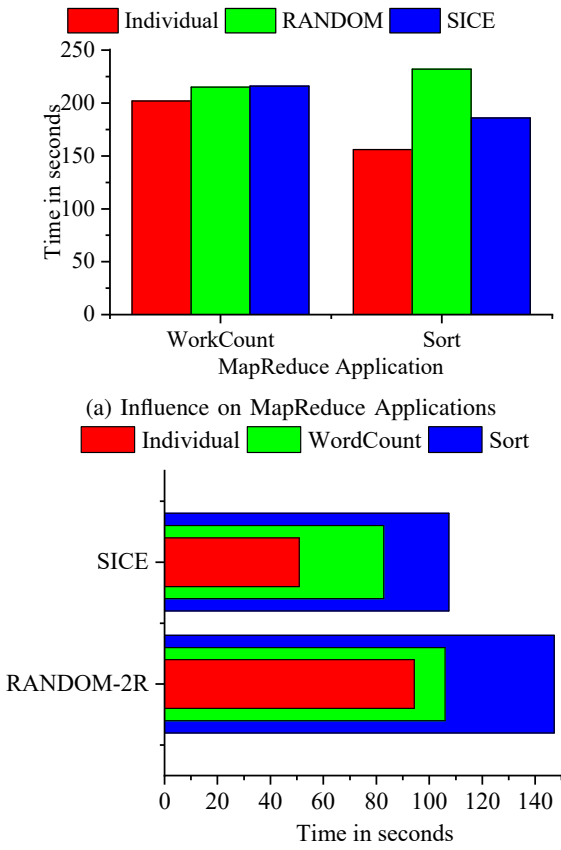
All these figures show that SICE-3R generates about two-thirds *CrossrackTraffic* of SICE-3L. That's because 3R distributes data more uniformly than 3L. It benefits SICE more to construct stripes in a round-robin way.

Time for constructing stripes: As Fig. 6 shows, both SICE-3R and SICE-3L requires a short time for selecting blocks for stripes. The time increases approximately linearly as the dataset scales up, so it proves highly scalable. For a dataset of about 2.44TB (20000 blocks * 128 MB in HDFS), SICE requires about 40ms to work out the encoding solution. It's really fast compared to the encoding time. SICE-3R requires a little more time than SICE-3L because blocks distributed in 3L tend to be more similar to the perfect block pattern than those distributed in 3R.

Summary: For different replication distribution methods, SICE achieves its goals for getting the system rid of co-located blocks and reducing the network consumption. In contrast, for RANDOM and LOCAL, 3R helps to reduce the risky blocks, but it generates large cross-rack traffic. Compared to RANDOM, LOCAL helps to reduce the cross-rack traffic by one order of magnitude, but SICE reduces it by one more order of magnitude.

3) Influence on MapReduce Applications: In this evaluation, we run an encoding job and a MapReduce application concurrently to simulate how encoding affects other services. In general, MapReduce applications run in foreground while encoders run in background. Therefore, we set the encoder in a low priority and the MapReduce application in a high priority. For comparison, we also run the encoder and the MapReduce application individually. The encoders for comparison are SICE and RANDOM. The MapReduce applications for comparison are *WordCount* and *Sort*. MapReduce applications can be divided into two categories, CPU-intensive and I/O-intensive. *WordCount* statistics the frequencies of every word and it is CPU-intensive. *Sort* sorts all records and it is I/O-intensive. The input sizes of *WordCount* and *Sort* are fixed to 10G, and the size of dataset to be encoded is fixed to 10G under 2R.

Due to the influence of encoder, the performance of MapReduce application degrades. As Fig. 7(a) shows, both SICE and RANDOM slightly affect *WordCount*, leading to about 10% degradation of *WordCount*. *Sort* spends 48.7% more time Because of the influence of RANDOM. *Sort* spends 19.2% more time Because of the influence of SICE. It shows that the I/O-intensive application is affected more heavily than the CPU-intensive application because the encoder competes for the I/O resource. Compared to RANDOM, SICE cuts down the influence by 60% because SICE is more resource-saving than RANDOM.



(a) Influence on MapReduce Applications

(b) Influence on Encoders

Figure 7: **Influence on MapReduce applications.** Different kinds of MapReduce applications are affecting and affected by the encoders differently.

Due to the influence of MapReduce application, the performance of encoder degrades. As Fig. 7(b) shows, SICE needs 60%∼120% more time and RANDOM needs 10%∼50% more time. SICE is influenced more heavily by MapReduce applications because SICE is so resource-saving that it tends to be CPU-intensive while RANDOM is still I/O-intensive. The reduction of computing resource influence more heavily on CPU-intensive applications than on I/O-intensive applications. However, as SICE still consumes less resources than RANDOM, it runs about 20% faster.

4) Failure recovery: In this evaluation, a node failure is simulated by shutting down a node randomly after encoding. To handle the failure, a MapReduce job is started and each map task of the job is responsible for recovering one or more missing blocks. The number of missing blocks per map task was set to *StripesPerMap* according to Equation (1). In the procedure of failure recovery, the major difference between sequential striping and non-sequential striping lies in retrieving the metadata. In general, each map task can retrieve the required metadata respectively. However, to intuitively show the total overhead of retrieving metadata retrieve,

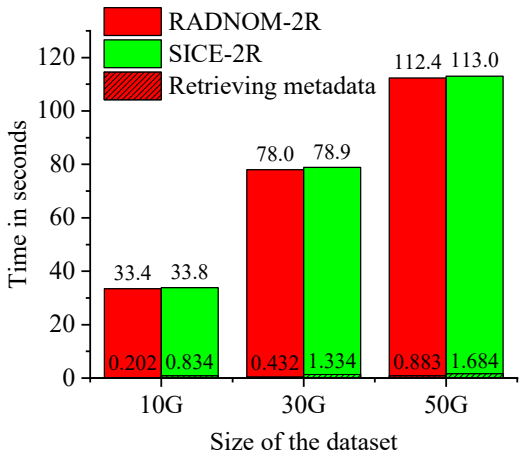


Figure 8: **Failure recovery.** It shows that for recovering a failure, the time spent in retrieving metadata is so short compared to the recovery time.

the MapReduce job will retrieve all the metadata before starting any map tasks. Only SICE-2R and RANDOM-2R are evaluated for comparison because the recovery performs on single-replica data.

Fig. 8 shows the evaluation result. It shows that non-sequential striping requires more time in retrieving metadata than sequential striping. However, the time spent in retrieving metadata is so short compared to the recovery time. Therefore, the overhead caused by non-sequential striping in failure recovery is modest and acceptable.

III. ASICE

A. A tradeoff in design

As ASICE only constructs the stripes that benefits the storage occupation, the searching space of ASICE for blocks is smaller than that of SICE. When there are no enough blocks with replicas in the same rack, keeping them in multi-replica will increase the popularity mismatching, or encoding them will increase the cross-rack network traffic. There is a tradeoff between them. ASICE sets a threshold for the number of multi-replica blocks which should be encoded, and only when the threshold is exceeded, ASICE constructs new stripes. With the threshold, ASICE trades off the popularity mismatching and the cross-rack network consumption. Moreover, the simulation results latter show that the reducing searching space only causes a small amount of popularity mismatching and cross-rack traffic.

B. Implement in Hadoop

Compared with SICE, the implement of ASICE requires HDFS to maintain access frequencies for every block. On every checking period, ASICE first gets all block locations of the dataset and the information of existing stripes to separate blocks in stripes and blocks not in stripes. For blocks in stripes, ASICE calculates $NR_{i,t}$ for each block, adjusts the number of replicas to $NR_{i,t}$ if needed by discarding spare replicas or complementing required replicas, and removes stripes which are hot. For blocks not in stripes, ASICE tries to search for more suitable stripes to reduce the storage occupation. The checking interval is supposed to be determined by the change rate of popularity and for HDFS, coarse-grained interval such as several days is recommended.

C. Simulation environment

We extend the simulation evaluation of SICE with popularity function. The dataset is generated to be totally hot and turns cold gradually over time. In detail, blocks of the dataset have a different popularity degradation rates. Obeying the Pareto Principle (also known as the 80/20 rule), about 80% of blocks turn cold for 1 replica, about 16% of blocks turn warm for 2 replicas and about 4% of blocks stay hot for 3 replicas. In the simulation, the generated dataset contains 20000 blocks, the cluster has 200 racks, the number of data blocks per stripe is set to 10, the total simulation time is 1 year and the checking interval is 1 day. The simulator runs under two replication distribution methods 3R and 3L for hot data, but their results show almost the same, so we only present the simulation results of 3R here.

REFERENCES

[1] Y. Xie, D. Feng, and F. Wang, "Non-sequential striping for distributed storage systems with different redundancy schemes," in *Proceedings of the 46th International Conference on Parallel Processing*, ICPP'17, pp. 231–240, Aug 2017.

[2] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, "Erasure coding in windows azure storage," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, (Berkeley, CA, USA), pp. 2–2, USENIX Association, 2012.

[3] M. N. Krishnan, N. Prakash, V. Lalitha, B. Sasidharan, P. V. Kumar, S. Narayanamurthy, R. Kumar, and S. Nandi, "Evaluation of codes with inherent double replication for hadoop," in *Proceedings of the 6th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'14, (Berkeley, CA, USA), pp. 12–12, USENIX Association, 2014.

[4] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A "hitchhiker's" guide to fast and efficient data reconstruction in erasure-coded data centers," *SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 331–342, Aug. 2014.

[5] B. Fan, W. Tantisiriroj, L. Xiao, and G. Gibson, "Diskreduce: Raid for data-intensive scalable computing," in *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, PDSW '09, (New York, NY, USA), pp. 6–10, ACM, 2009.

[6] R. Li, Y. Hu, and P. P. C. Lee, "Enabling efficient and reliable transition from replication to erasure coding for clustered file systems," in *Proceedings of the 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '15, (Washington, DC, USA), pp. 148–159, IEEE Computer Society, 2015.

[7] M. Xia, M. Saxena, M. Blaum, and D. A. Pease, "A tale of two erasure codes in hdfs," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, (Berkeley, CA, USA), pp. 213–226, USENIX Association, 2015.