Fast Proactive Repair in Erasure-Coded Storage: Analysis, Design, and Implementation (Supplementary File)

Xiaolu Li, Keyun Cheng, Zhirong Shen, and Patrick P. C. Lee

The following materials are supplementary to our main file. In this digital supplementary file, we analyze two repair methods for Azure-LRC (Section 1). We present the time complexity analysis for FastPR (Section2). We present the large-scale simulation results for RS codes (Section 3). We finally include more experiment results for RS codes on Amazon EC2.

1 Analysis for the Reactive Repair of Azure-LRC

We compare two reactive repair methods for Azure-LRC, namely *independent repair* and *mixed repair*. We divide the reconstruction process into multiple rounds. For independent repair, we only repair chunks from either the local parity group or the global parity group in each repair round. For mixed repair, we can repair chunks from both groups. We prove that independent repair is faster than mixed repair in ideal cases, where in each round, we can reconstruct the maximum number of chunks. Table 1 shows the symbols used in this proof.

TABLE 1
Notation for Section 1.

Symbol	Description
U	Total number of chunks in the STF node.
M	Total number of nodes in a storage cluster.
U_l	Number of chunks in the local parity group.
$t_{r,l}$	Time to repair a chunk in the local parity group.
k_l	Number of chunks required to repair a chunk in the local parity group.
U_g	Number of chunks in the global parity group.
$t_{r,g}$	Time to repair a chunk in the global parity group.
k_g	Number of chunks required to repair a chunk in the global parity group.

- *Xiaolu Li is with the School of Computer Science and Technology, Huazhong University of Science and Technology (E-mail: lixl666@hust.edu.cn). This work was partially done when Xiaolu Li worked as a Postdoctoral Fellow at The Chinese University of Hong Kong.*
- *Keyun Cheng, and Patrick P. C. Lee are with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong (E-mails: {kycheng, pclee}@cse.cuhk.edu.hk).*
- *Zhirong Shen is with the School of Informatics, Xiamen University, China. (E-mail: shenzr@xmu.edu.cn).*

Independent repair: In each round, we can repair $G_l = \frac{M-1}{k_l}$ chunks from the local parity group in parallel. The number of rounds to repair all chunks from the local parity group is $\frac{U_l \cdot k_l}{M-1}$. Thus, the time to repair all the chunks in the local parity group is $\frac{U_l \cdot k_l \cdot t_{r,l}}{M-1}$. Similarly, the time to repair all the chunks in the global parity group is $\frac{U_g \cdot k_g \cdot t_{r,g}}{M-1}$. Thus, the total repair time for the independent repair $T_{R,idd}$ is

$$T_{R,idd} = \frac{U_l \cdot k_l \cdot t_{r,l}}{M-1} + \frac{U_g \cdot k_g \cdot t_{r,g}}{M-1} \tag{1}$$

Mixed repair: We divide the repair process into three parts for mixed repair: (i) U_l' out of U_l chunks from the local parity group are repaired independently in each round; (ii) U_g' out of U_g chunks from the global parity group are repaired independently in each round; and (iii) for remaining chunks, we repair both chunks in each round. The repair times for (i) and (ii) are $\frac{U_l' \cdot k_l \cdot t_{r,l}}{M-1}$ and $\frac{U_g' \cdot k_g \cdot t_{r,g}}{M-1}$, respectively. For (iii), the total number of rounds is at least $\frac{(U_l - U_l') \cdot k_l + (U_g - U_g') \cdot k_g}{M-1}$. Suppose $t_{r,g} \geq t_{r,l}$, which means that the repair time of each round is bottlenecked by $t_{r,g}$. Thus, the total repair time $T_{R,mix}$ is

$$\begin{aligned} T_{R,mix} \quad &\geq \quad \frac{U_l' \cdot k_l \cdot t_{r,l}}{M-1} + \frac{U_g' \cdot k_g \cdot t_{r,g}}{M-1} \\ &+ \quad \frac{(U_l - U_l') \cdot k_l + (U_g - U_g') \cdot k_g}{M-1} \cdot t_{r,g} \end{aligned} \tag{2}$$

Considering that $t_{r,g} \geq t_{r,l}$, we can readily show that $T_{R,mix} \geq T_{R,idd}$.

2 Time Complexity Analysis

We first consider the MATCH function in Algorithm 1. MATCH aims to find the maximum matching on a candidate reconstruction set, whose size is at most $\frac{M-1}{k}$. Thus, the resulting bipartite graph in MATCH has at most $\frac{M-1}{k} \cdot k = M-1$ chunk vertices (together with $M-1$ node vertices). Each chunk vertex connects to $n-1$ node vertices, so there are at most $(M-1)(n-1)$ edges in the bipartite graph. Thus, MATCH finds a maximum matching in a bipartite graph in $O(M^2 n)$ time.

We next consider the FIND function in Algorithm 1. Forming an initial reconstruction set (Lines 10-17) calls

MATCH $|\mathcal{C}|$ times. Optimizing the reconstruction set (Lines 18-38) expands $\mathcal{R}$ no more than $\frac{M-1}{k}$ times (Line 34), and each time calls MATCH $|\mathcal{C}|^2|\mathcal{R}| \leq |\mathcal{C}^2\frac{M-1}{k}|$ times (Lines 20-31). Thus, the time complexity of FIND is $O(|\mathcal{C}|^2M^4n)$.

Overall, Algorithm 1 calls FIND at most $|\mathcal{C}|$ time, so its time complexity $O(|\mathcal{C}|^3M^4n)$.

Finally, we analyze the time complexity of Algorithm 2. Sorting d reconstruction sets (Line 1) takes $O(d\log d)$ time. For each repair round, we scan d reconstruction sets to find the largest x (Line 9) in $O(d)$ time, and find the subset $\mathcal{R}'_x$ in $\mathcal{R}_x$ (Line 10) in $O(|\mathcal{R}_x|)$ time. Since the number of repair rounds is at most d, the complexity of Lines 3-15 is $O(d(d+|\mathcal{R}_x|))$. Also, as $d \leq |\mathcal{C}|$ and $|\mathcal{R}_x| \leq \frac{M-1}{k}$, the time complexity of Algorithm 2 is $O(|\mathcal{C}|(|\mathcal{C}|+\frac{M}{k}))$.

Discussion: Note that Algorithm 1, even with polynomial complexity, incurs high running time for large $|\mathcal{C}|$ and M. We suggest two options to mitigate the overhead. The first option is to partition the chunks of the STF node into *chunk groups* and find the reconstruction sets for each chunk group (which now becomes $\mathcal{C}$). Another option is that we can run Algorithm 1 for each possible STF node in advance and store the results when they are required [3].

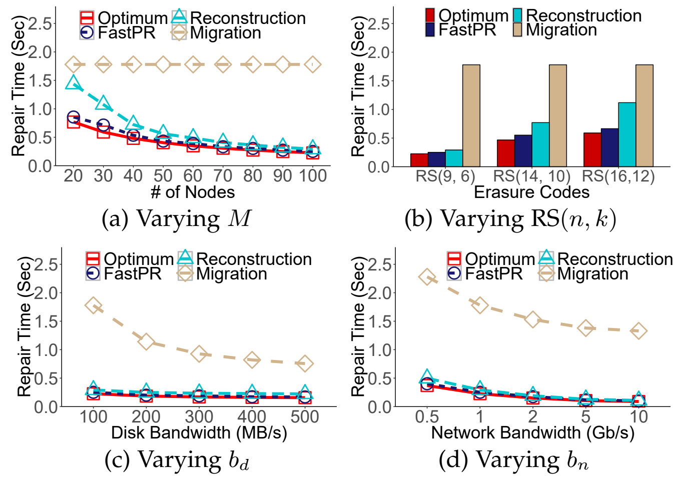
3 SIMULATION RESULTS

We conduct simulation on FastPR to evaluate its performance in a large-scale storage cluster. We focus on RS codes. We design a single-machine simulator for FastPR by modifying our prototype. In the simulator, we remove all the actual operations of disk I/Os and network transmission from the prototype, and simulate the operations by computing their execution times based on the input network and disk bandwidths. Note that the main algorithms, including finding reconstruction sets and repair scheduling, are still preserved.

We compare FastPR with three approaches: (i) *migration-only*, in which we directly migrate all the chunks of the STF node to other healthy nodes; (ii) *reconstruction-only*, in which we find the reconstruction sets based on Algorithm 1, but we repair each of them in a repair round by reconstruction only without calling Algorithm 2 (note that it corresponds to the conventional reactive repair); and (iii) *optimum*, which we derive from the mathematical analysis based on our modeled t_m and t_r.

We assume the following default configurations. We configure a storage cluster of $M = 100$ nodes with the disk bandwidth $b_d = 100$ MB/s and network bandwidth $b_n = 1$ Gb/s. We encode the chunks by RS$(9,6)$ adopted by QFS [5], while we also consider RS$(14,10)$ (adopted by Facebook [4]) and RS$(16,12)$ (coding parameters used by Azure [2]). We fix the chunk size as 64 MB, and randomly distribute 1,000 stripes of chunks across the storage cluster. For hot-standby repair, we fix the number of hot-standby nodes $h = 3$. We vary one configuration parameter in our simulation experiments and evaluate its impact. We measure the repair time per chunk, averaged over 30 runs.

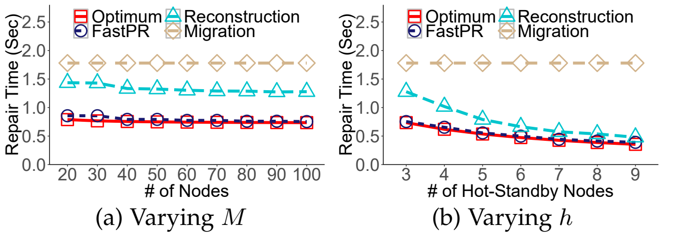
Experiment 1 (Scattered repair): Figure 1 shows the simulation results of the repair time per chunk in scattered repair, in which we vary M, RS(n,k), b_d, and b_n. Migration-only is the worst among all approaches, since its performance



(a) Varying M (b) Varying RS(n,k)

(c) Varying b_d (d) Varying b_n

Fig. 1. Experiment 1: Simulation results of repair time per chunk in scattered repair.



(a) Varying M (b) Varying h

Fig. 2. Experiment 2: Simulation results of repair time per chunk in hot-standby repair.

is bottlenecked by the STF node. Reconstruction-only has similar performance to FastPR since it can exploit the available bandwidth resources of the storage cluster as FastPR. However, it incurs higher repair time per chunk than FastPR when M is small (Figure 1(a)) or (n,k) is large (Figure 1(b)), since the available bandwidth resources of the storage cluster are more limited for smaller M and the repair traffic increases for larger k. Overall, FastPR reduces the repair times of both migration-only and reconstruction-only, for example, by 62.7% and 40.6% for RS$(16,12)$ (Figure 1(b)).

In practice, FastPR deviates from the optimum since the number of chunks of the STF node that can be repaired in parallel depends on the chunk distribution. Nevertheless, our simulation results show that the repair time of FastPR is only 11.4% more than the optimum on average.

Experiment 2 (Hot-standby repair): Figure 2 shows the simulation results of the repair time per chunk in hot-standby repair, in which we vary M and h. The repair performance is mainly bottlenecked by the hot-standby nodes, and the repair time has limited variance across different values of M (Figure 2(a)). When $h = 3$, FastPR reduces the repair times of migration-only and reconstruction-only by 57.7% and 41.0%, respectively. FastPR maintains high performance in hot-standby repair, and its repair time is only 5.4% more than the optimum on average.

Experiment 3 (Impact of the number of stripes): Figure 3 shows the repair time per chunk versus the number of stripes repaired in total. Here, we only focus on FastPR and the optimum to compare their differences. Increasing the number of stripes provides more flexibility of FastPR to

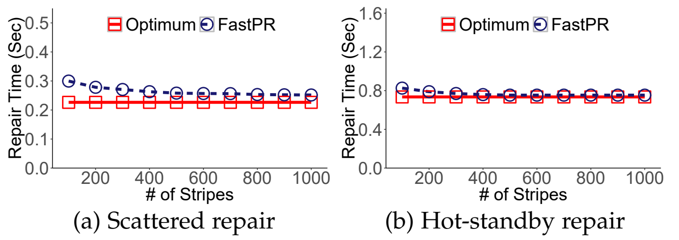


Fig. 3. Experiment 3: Simulation results of repair time per chunk versus the number of stripes.

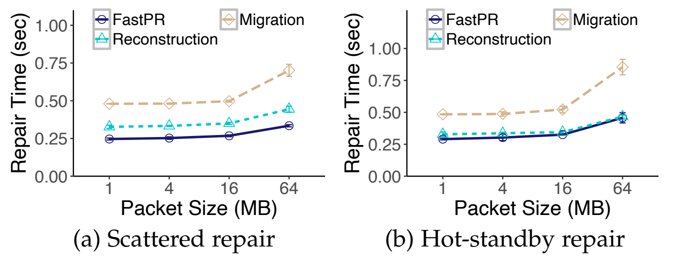


Fig. 4. Experiment 4: Impact of the packet size.

identify the reconstruction sets that maximize parallelism. We observe that when the number of stripes is at least 400, the differences between FastPR and the optimum are very small (within 15%). This implies that we can limit our selection of reconstruction sets (i.e., Algorithm 1) to a smaller group of chunks to mitigate the running time overhead, while achieving the near-optimal repair performance.

4 Experiments for RS Codes

We now include more experiment results for RS codes in the same Amazon EC2 clusters we configure in Section 6. Here, we study the impact of packet size and network bandwidth. We also conduct microbenchmarks to study the performance of Algorithm 1.

Experiment 4 (Impact of the packet size): We first study the impact of multi-threading by evaluating the repair time per chunk versus the packet size, varied from 1 MB to 64 MB; note that for the packet size 64 MB, we do not enable multi-threading as it is equal to the default chunk size. Figure 4 shows that the repair time reduces for smaller packet sizes, as multi-threading can parallelize different steps of a repair operation. For example, when the packet size reduces from 64 MB to 4 MB, the repair time of FastPR reduces by 31.4% (the reduction is negligible when the packet size further reduces to 1 MB). For all packet sizes, FastPR reduces the repair times of migration-only and reconstruction-only by 37.7-52.3% and 1.9-24.7%, respectively.

Experiment 5 (Impact of network bandwidth): We study how the network bandwidth (i.e., b_n) affects the repair time. We use the Wonder Shaper tool [1] to control the network adapter bandwidth. Here, we vary b_n as 0.5 Gb/s, 1 Gb/s, and 5 Gb/s (the default one without bandwidth limiting). Figure 5 shows that the repair time of reconstruction-only significantly increases when the network bandwidth is limited, as it incurs a high amount of repair traffic. Overall, FastPR reduces the repair times of migration-only and reconstruction-only by 27.7% and 62.5% when $b_n = 0.5$ Gb/s, and 27.1% and 61.5% when $b_n = 1$ Gb/s, respectively.

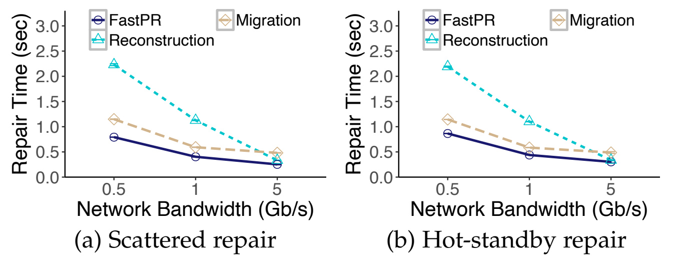


Fig. 5. Experiment 5: Impact of network bandwidth.

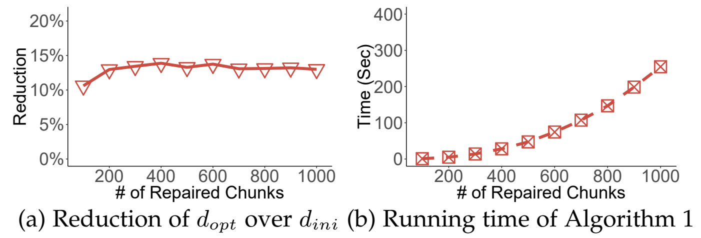


Fig. 6. Experiment 6: Microbenchmarks.

Experiment 6 (Microbenchmarks): A key component of FastPR is to find the reconstruction sets (Algorithm 1). We study Algorithm 1 in two aspects, based on the evaluation setting for RS codes on Amazon EC2 as described in Section 6.1 of the main file.

First, we analyze the effectiveness of optimizing the selection of the initial reconstruction set in Algorithm 1 (i.e., Lines 18-38). We compare the numbers of reconstruction sets returned by Algorithm 1 with and without Lines 18-38, denoted by d_{opt} and d_{ini}, respectively. Intuitively, if $d_{opt} < d_{ini}$, the optimization step reduces the number of reconstruction sets returned and hence includes more chunks in each reconstruction set on average to exploit a higher degree of parallelism. Figure 6(a) shows the reduction of d_{opt} compared to d_{ini} versus the number of repaired chunks (i.e., $|\mathcal{C}|$), averaged over 30 runs. Overall, d_{opt} is 13% less than d_{ini}, while the reduction becomes fairly stable when the number of repaired chunks is at least 200.

Second, we measure the running time of Algorithm 1 in one Amazon EC2 instance of type `m5.large` in the US East (North Virginia) region. Figure 6(b) shows the running time of Algorithm 1 versus the number of repaired chunks, averaged over 30 runs. The running time increases from 0.84 s for 100 repaired chunks to 254.63 s for 1,000 repaired chunks. Nevertheless, we can run Algorithm 1 on the repaired chunks of the STF node by groups and pre-compute Algorithm 1 for each STF node offline to mitigate its overhead.

References

[1] The Wonder Shaper 1.4. https://github.com/magnific0/wondershaper.
[2] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, S. Yekhanin, et al. Erasure Coding in Windows Azure Storage. In *Proc. of USENIX ATC*, 2012.
[3] O. Khan, R. C. Burns, J. S. 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*, 2012.
[4] S. Muralidhar, W. Lloyd, S. Roy, et al. F4: Facebook's Warm Blob Storage System. In *Proc. of USENIX OSDI*, 2014.
[5] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly. The Quantcast File System. *Proc. of the VLDB Endowment*, 6(11):1092–1101, 2013.