Reliability-Aware Deduplication Storage: Assuring Chunk Reliability and Chunk Loss Severity

Youngjin Nam*†
*School of Computer and Information Technology
Daegu University
Gyeongsan, Gyeongbuk, KOREA 712-714
Email: youngjin@cs.umn.edu

Guanlin Lu and David H. C. Du†
†Department of Computer Science and Engineering
University of Minnesota – Twin Cities
Minneapolis, Minnesota 55455, USA
Email: {lv, du}@cs.umn.edu

***Abstract*—Reliability in deduplication storage has not attracted much research attention yet. To provide a demanded reliability for an incoming data stream, most deduplication storage systems first carry out deduplication process by eliminating duplicates from the data stream and then apply erasure coding for the remaining (unique) chunks. A unique chunk may be shared (i.e., duplicated) at many places of the data stream and shared by other data streams. That is why deduplication can reduce the required storage capacity. However, this occasionally becomes problematic to assure certain reliability levels required from different data streams. We introduce two reliability parameters for deduplication storage: chunk reliability and chunk loss severity. The chunk reliability means each chunk's tolerance level in the face of any failures. The chunk loss severity represents an expected damage level in the event of a chunk loss, formally defined as the multiplication of actual damage by the probability of a chunk loss. We propose a reliability-aware deduplication solution that not only assures all demanded chunk reliability levels by making already existing chunks sharable only if its reliability is high enough, but also mitigates the chunk loss severity by adaptively reducing the probability of having a chunk loss. In addition, we provide future research directions following to the current study.**

***Keywords*-storage; deduplication; reliability; loss severity;**

I. Motivation

A typical chunking-based deduplication process breaks up a given data stream into smaller chunks of variable lengths and computes a hash value of each chunk called chunk ID with a cryptographic hash function like SHA-1. A hash index table should effectively maintain a large number of chunk IDs with their storage locations. Only if the chunk ID of a chunk is not found in the hashed index table, the deduplicated (unique or unshared) chunk is written into the deduplication storage. Otherwise, the shared chunk will be eliminated from further physical write. Recent deduplication design has mainly focused on its write performance by: 1) improving the duplication detection and chunk existence querying efficiency with efficient chunking, faster hash indexing, locality-preserving index caching, and efficient bloom filters; and 2) compressing the unique chunks and performing (fixed-size) large writes through containers or similar structures.

However, few research efforts have been made to reliability issues of the deduplication storage [1], [2]. Indeed, data reliability and deduplication are pursuing the opposite goals. The data reliability protects its associative information against failures by generating more redundant information, while the deduplication eliminates any redundant information from given data sets. Most deduplication storage systems have managed these two issues in a separate manner; that is, to provide demanded reliability for an incoming data stream, deduplication storage eliminates any duplicates (shared chunks) from the data stream first by sharing already existing chunks and then applies the erasure coding [2] for the remaining (unique) chunks. Sharing chunks is the best way to optimize storage capacity. However, it occasionally entails dissatisfaction of the demanded reliability levels from data streams. HYDRAstor [3] determines a data redundancy level by using supernode cardinality. The supernode represents a logical storage node that consists of a number of physical storage nodes. The supernode cardinality represents how many physical storage nodes reside in a supernode. After duplication variable-length chunks are organized into a synchrun similar to the container [4]. After being erasure-coded, each synchrun is written reliably into the multiple physical storage nodes in a supernode. Similarly, R-ADMAD [5] stores variable-length chunks into a fixed-size object until the object becomes full. Then, it distributes the object over multiple storage servers by using the erasure coding.

Recent studies [1], [2] have highlighted significance and uniqueness of reliability issues in the deduplication storage. Bhagwat et al. [1] showed that a (critical) chunk loss would damage multiple data streams (files) sharing the chunk. They measured the importance of each chunk (called chunk weight) quantitatively by examining the number of data streams that share it (or by the amount of shared data). Based on the number of data streams effected from a given chunk loss (data loss severity), they determined how many duplicates of the shared chunk should be maintained to reduce the severity of the chunk loss. The duplicate number grows logarithmically as the chunk sharing increases.

Li et al. [2] proposed that deduplication should guarantee initially requested data reliability by using more robust erasure coding (more redundant information), because unique chunks are likely to be distributed over a larger number of disks. Given unique chunks of a data stream, they provided a series of well-established formulas to configure the parameters (k, m) of the erasure coding to meet the initial data reliability requirement. Here k and m respectively represent the number of data and parity fragments. The parameters k and m will be discussed further during performance analysis.

Drawbacks of the previous reliability solutions for deduplication storage can be summarized as follows: *First, they managed a reliability issue (assuring a demanded reliability level) separately from a typical deduplication process.* Even in the two previous works [1], [2], the deduplication process is performed independently of the reliability management. However, we can easily see that the reliability issue should be taken into account carefully whenever the deduplication process determines if a chunk is to be eliminated (to be shared) or not. For example, suppose that a newly incoming chunk is identical to an already existing chunk in the underlying storage. However, it is found that the reliability level of the already existing chunk is lower than the demanded reliability level of the incoming chunk. In that case, the deduplication process should not allow the incoming chunk to share the already existing chunk. *Second, the previous solutions focused on only one of the reliability-related parameters for deduplication storage: either the probability of losing a chunk (chunk reliability) or severity of a chunk loss (chunk loss severity).* Eventually, the reliable deduplication should be able to manage both the chunk reliability and the chunk loss severity, while optimizing storage capacity in use.

II. Reliable Deduplication

A. Our Deduplication Storage

We assume that the deduplication storage under investigation consists of a deduplication appliance and underlying storage (consisting of multiple independent disks), as depicted in Figure 1. The deduplication appliance resides between a host and the underlying storage. It takes in charge of chunking incoming data streams and eliminating duplicate chunks. It also includes a data encoding/decoding mechanism (erasure coding) to reliably store/restore data (chunks) into/from the underlying storage. Each data stream is a byte stream representing a series of dataset. Data streams can have different reliability requirements (to be defined later as data stream reliability). In Figure 1, one data stream (A) is stored with additional redundant information (`ar`) to tolerate any single chunk loss (disk failure). On the contrary, a chunk loss (`b0` or `b2`) can corrupt the other data stream (B). Notice that the chunk `b1` is eliminated by the already existing chunk `a0`, and the chunk `a0` has been stored more reliably than requested by the chunk `b1`.

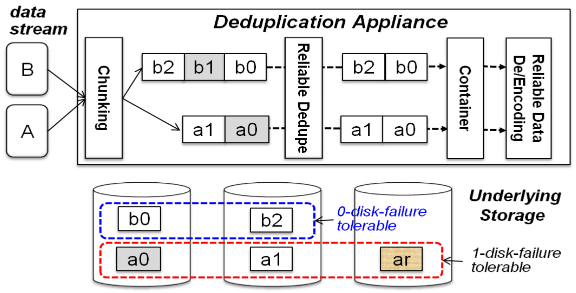


Figure 1: Our reliable deduplication storage example

The write of an incoming data stream to the underlying storage begins by breaking up the stream into variable-size chunks. A deduplication process subsequently eliminates any duplicated copies of chunks (called shared chunks) each of which has been stored already in the underlying storage. Any remaining chunks after the deduplication are called unique (or deduplicated) chunks. Different from previous reliable solutions [1], [2], our reliable deduplication process not only assures a demanded reliability level when storing a chunk (to be defined later as chunk reliability), but also maintains an expected severity level of a chunk loss (to be defined later as chunk loss severity) as low as the case of no-deduplication. The subsequent section will provide more detailed descriptions on this. Note that each incoming data stream has a distinct open container in the memory as in the previous work [4]. When an in-memory container becomes full of chunks, the container is written into the underlying storage to assure the reliability levels demanded by all the chunks (to be defined later as container reliability).

B. Reliability Parameters for Deduplication Storage

We denote by DS_i a data stream and denote by c_i^j the j-th chunk of DS_i (sequence id before deduplication). We define a sharing level of chunk c_i^j, denoted by S_i^j, as how many times the chunk c_i^j is shared by data streams. Obviously, $S_i^j = 1$ if c_i^j has no sharing.

Demanded Reliability for Data Streams (DSR_i): Each user of data streams can request different levels of data reliability, where some data streams are to be stored more reliably than the others. We reasonably assume that the data encoding module provides a set ($\mathcal{R}$) of m reliability levels through the underlying storage, denoted by $\mathcal{R} = \{r_i | 0 \leq i \leq m-1\}$, where $0 \leq r_i \leq r_{i+1} \leq 1$. Note that r_0 and r_{m-1} represent the lowest and highest reliability levels, respectively. Thus, a reliability level requested by a data stream is one of the predefined reliability levels $\mathcal{R}$. Denote the demanded reliability for DS_i (briefly data stream reliability) by DSR_i, where $DSR_i \in \mathcal{R}$.

Chunk Reliability (CR_i^j): A data stream consists of a series of variable-length chunks. By storing each chunk of a data stream with the demanded data stream reliability, we can assure the demanded reliability level of the data stream (DSR_i). Thus, we can define the chunk reliability of c_i^j, denoted by CR_i^j, as DSR_i. Sometimes, CR_i^j needs to be larger than DSR_i because of chunk sharing. However, it should not be smaller than DSR_i. Thus, the following relation is valid: $CR_i^j \geq DSR_i$, where $CR_i^j \in \mathcal{R}$.

Container Reliability (CTR_i): Recall that each data stream has a distinct in-memory container for physical writes. It implies that each container includes a series of chunks from a single data stream, DS_i. Thus, the container should be reliably stored, so as to assure the reliability levels demanded by all the chunks in the container. Thus, the demanded reliability level of the container, denoted by CTR_i, can be defined as: $CTR_i = max\{CR_i^k | c_i^k \text{ is stored in the container}\}$, where $CTR_i \in \mathcal{R}$.

Chunk Loss Severity (CLS_i^j): Deduplication storage users are mostly satisfied with reduced storage usage by eliminating duplicates from their underlying storage. Surprisingly, it has been known that 70% of digital data in our digital universe are duplicate. As a result, a higher probability exists that a chunks is shared by multiple data streams [6]. However, losing any of the shared chunks is likely to be more severe than that of the other (non-shared) chunks. As in the previous work [1], a severity level can be measured by examining how many data streams (files) are lost when a chunk is lost. In Figure 1, for example, if we cannot recover the chunk `a0`, the two data streams, A and B, will be lost. On the contrary, losing any of the other chunks will corrupt only a single data stream. Thus, it is reasonable that the shared chunk should be protected more reliably than the other chunks [1].

We define a level of chunk loss severity, denoted by CLS_i^j, as an expected damage level in the presence of a chunk loss. Formally, the expected damage level is expressed as the multiplication of actual damage by the probability of having the damage (a chunk loss): $CLS_i^j = S_i^j(1 - CR_i^j) = S_i^j \cdot CF_i^j$, where the actual damage represents the current sharing level of c_i^j (S_i^j), and the probability of having the damage represents the probability of losing c_i^j ($(1 - CR_i^j) \equiv CF_i^j$). For simplicity, CF_i^j and DSF_i are used instead of $(1 - CR_i^j)$ and $(1 - DSR_i)$, respectively. Note that, without deduplication, $CLS_i^j = (1 - DSR_i) \equiv DSF_i$ for all i, j.

Users of data streams would like to believe that their data streams are stored reliably and separately from the others, so that any chunk loss in the others' data streams does not affect the reliability of their data streams. It implies that we should provide a level of chunk loss severity (expected damage level) as low as (not higher than) the case in that no deduplication is used (no chunk sharing). For this purpose, Bhagwat et al. [1] maintained a more number of duplicate copies for the shared chunk (logarithmically) as its sharing level increases.

C. Problem Description

Given a set of data streams $\{DS_i | 0 \leq i \leq n-1\}$, associative reliability demands $\{DSR_i | 0 \leq i \leq n-1\}$, and a predefined set of reliability levels $\mathcal{R} = \{r_i | 0 \leq i \leq m-1\}$, we would like to devise a reliable deduplication solution to effectively meet the following requirements:

- *first, guarantee the demanded reliability levels of the data streams, i.e.,* $CR_i^j \geq DSR_i$ for all i, j, while performing the data deduplication (allowing chunk sharing), and
- *second, keep the chunk loss severity of each chunk not to be higher than the chunk loss severity of the no-deduplication case* (no chunk sharing); *i.e.,* $CLS_i^j \leq DSF_i$ for all i, j.

D. Our Solution

Algorithm 1 provides a core of our reliability-aware deduplication solution. The proposed solution not only assures all demanded chunk reliability levels by making already existing chunks sharable only if its reliability is high enough, but also mitigates the chunk loss severity by adaptively reducing the probability of having a chunk loss. Notice that the algorithm receives each chunk from a data stream as an input (after the variable-length chunking process), and it ensures the inequality conditions given in the two aforementioned requirements.

Algorithm 1 Reliability-aware deduplication

```
Require: a newly incoming chunk $c_i^j$ in $DS_i$
Ensure: $CR_i^j \geq DSR_i$ and $CLS_i^j \leq DSF_i$ for all $i, j$
 1: if ($c_i^j$ has (one or more) duplicate) then
 2:     // pick $c_k^l$ whose $CR_k^l$ is higher than the other instances of
        the same chunk
 3:     if ($CR_i^j > CR_k^l$) then
 4:         store $c_i^j$ into $DS_i$ container // and remove $c_k^l$
 5:         $CTR_i \leftarrow max\{CTR_i, CR_i^j\}$, $S_k^l = 1$
 6:     else
 7:         $S_k^l \leftarrow S_k^l + 1$
 8:         if ($CLS_k^l > DSF_k$) then
 9:             adjust $CR_k^l \leftarrow min\{r_i | r_i \geq 1 - \frac{DSF_k}{S_k^l}, r_i \in \mathcal{R}\}$
10:             store $c_k^l$ into $DS_k$ container // and remove older $c_k^l$
11:             $CTR_k \leftarrow max\{CTR_k, CR_k^l\}$
12:         else
13:             store the pointer information of $c_k^l$
14:         end if
15:     end if
16: else
17:     store $c_i^j$ into open container of $DS_i$
18: end if
19: if (container of any $DS_m$ is full) then
20:     store the container with $CTR_m$
21:     prepare another container, $CTR_m \leftarrow DSR_m$
22: end if
```

To meet the first requirement ($CR_i^j \geq DSR_i$), we have to carefully look into the case that the current (newly incoming) chunk has its duplicate copy (called the already existing chunk). A typical deduplication process simply eliminates the current chunk from further physical write and then maintains proper pointer information of the already existing chunk. Given our reliability problem, however, the current chunk no longer shares the already existing chunk if its demanded chunk (data stream) reliability is higher than the chunk reliability guaranteed by the already existing chunk (line 3). Instead, we should write the current chunk into the underlying storage to assure the demanded reliability level (line 4). As a result, the underlying storage contains more than one copy of the chunk (line 1), and the duplicates have distinct reliability levels. Alternatively, to optimize storage capacity consumption, we can remove the already existing chunk after writing the current chunk into the underlying storage with a higher reliability level. However, there is a concern related to read performance of the deduplication storage. Further discussion is beyond the scope of this paper. If the current chunk matches one or more already existing chunks, we need to choose one of the duplicates. We found that the chunk with the highest reliability is usually the best choice (line 2). Notice that the container reliability (CTR_i) is updated accordingly as the highest reliability level demanded by the chunks stored in the container (line 5). As a result, the CR_k^l (line 3) could be higher than its demanded data stream reliability level, DS_k. Initially, CTR_i is set to DSR_i (line 21).

To satisfy the second requirement ($CLS_i^j \leq DSF_i$), by the definition of $CLS_i^j = S_i^j \cdot CF_i^j$, we can reduce the expected damage level in two different ways: (1) by mitigating the actual damage (S_i^j); (2) by decreasing the probability of having a chunk loss (CF_i^j). Since deduplication performance is negatively affected by the first approach (allowing chunk sharing), we employ the second approach that adaptively increases the chunk reliability (reducing the chunk loss probability). Debatably, the actual damage level remains unchanged in the second approach. We will discuss in our future work. For example, suppose that a chunk c_k^l is shared by two data streams, DS_i and DS_k (line 7). We assume that $DSR_i = DSR_k$, followed by $CR_k^l = DSR_k$. Then, the actual damage level of c_k^l (S_i^j) becomes 2. Since we have $CLS_i^j = 2CF_i^j$, its chunk loss severity (CLS_k^l) becomes two times higher than DSF_i (line 8). Thus, we need to adaptively decrease the current value of CF_i^j by half; that is, the newly adjusted chunk reliability ($CF_i^j = 1 - CR_i^j$) becomes $\frac{DSF_i}{2}$ (line 9). Notice that the adjusted chunk reliability should be provided by one of the predefined set of reliability levels, $\mathcal{R}$. In some cases where a data stream initially demands the highest level of data reliability, an adjusted chunk reliability with increase of chunk sharing cannot be satisfied by any of the predefined set. To avoid this problem, we can put a restriction to the demanded reliability levels by data streams. For instance, given a predefined set of reliability levels $\mathcal{R} = \{r_i | 0 \leq i \leq m-1\}$, a data stream can initially request a data reliability level, ranging r_0 through r_{m-2}. Similarly, we need to store the c_k^l by using a higher chunk reliability level (line 10). As with our solution to the first requirement, it will also make more than one copy. Likewise, we can remove the already existing chunk (as long as its associative read performance will not decrease). When a container of DS_m becomes full of chunks (line 19), it is written into the underlying storage to assure CTR_m (line 20). Next, available is an empty in-memory container initialized as $CTR_m = DSR_m$ (line 21).

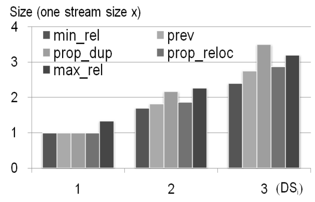
III. Performance Analysis

We assume that our deduplication storage employs erasure coding with parameters (k, m) to store each container reliably into the underlying storage. The k represents the number of data fragments that each container is divided into, and m represents the number of parity fragments generated to tolerate failures. The $k+m$ fragments are written into $k+m$ distinct disks. Any chunk stored in the container can be reconstituted if any k-out-of-$(k+m)$ fragments are available, tolerating at maximum any m fragment losses (disk failures). Our reliable deduplication storage provides three different reliability levels: (1) r_0 with $k = 6$ and $m = 0$; (2) r_1 with $k = 6$ and $m = 1$; (3) r_2 with $k = 6$ and $m = 2$. By having that the MTTF value of a disk is 10^6 hours, we can calculate a predefined set of reliability levels by using the reliability equation provided in the previous work [2]: $\mathcal{R} = \{0.86088(r_0), 0.98719(r_1), 0.99910(r_2)\}$.

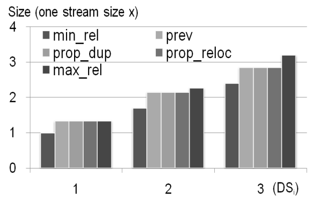
We will analyze the following five solutions: `min_rel`, `max_rel`, `prev`, `prop_dup`, and `prop_reloc`. The `prev` represents a typical reliable deduplication solution that handles the reliability problem separately from the deduplication problem. As previously explained, our solution has two different versions: one keeps the already existing chunk when the current chunk is written into the storage with a higher reliability (`prop_dup`), whereas the other removes the already existing chunk (`prop_reloc`). Moreover, we add two extreme cases: one always uses the lowest reliability level to store a container, r_0 (`min_rel`), and the other always employs the highest reliability level, r_{m-1} (`max_rel`). We will compare the five solutions in terms of storage capacity consumption, a degree of guaranteeing demanded levels of chunk reliability, and a degree of assuring demanded levels of chunk loss severity.

A. Storage Capacity Consumptions

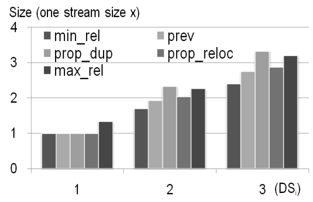
Our analysis assumes the followings: (1) the deduplication storage takes three data streams sequentially (DS_1, DS_2, and DS_3); (2) the sizes of the three data streams are equal; (3) each data stream has 30% deduplication ratio (representing the ratio of the total size of elimitated chunks to the total size of all chunks before deduplication) with its previous data streams; (4) each container has one chunk. Removing the last assumption will increase the consumed storage capacity by the `prop_reloc`; that is, deleting the already existing chunk does not immediately release the storage space occupied by its associative container, because other chunks can still reside in the container. We can use garbage collection to compact partially used container spaces as in log-structured file systems.



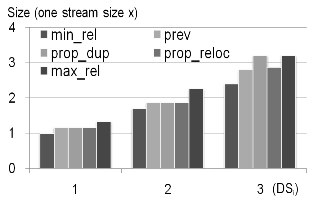
(a) P_0 (up & up)



(b) P_1 (down & down)



(c) P_2 (up & down)



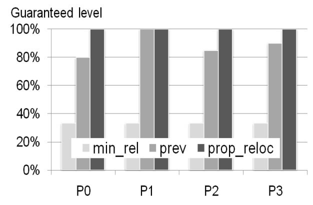
(d) P_3 (down & up)

Figure 2: Storage capacity consumptions with the four combinations of reliability levels

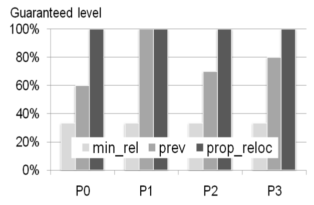
Figure 2 shows our analysis results of the storage capacity consumed by the five solutions. We employed four different combinations of reliability levels for the three data streams: P_0, P_1, P_2, and P_3, where $P_0 = \{r_0 \rightarrow r_1 \rightarrow r_2\}$ (up & up), $P_1 = \{r_2 \rightarrow r_1 \rightarrow r_0\}$ (down & down), $P_2 = \{r_0 \rightarrow r_2 \rightarrow r_1\}$ (up & down), and $P_2 = \{r_1 \rightarrow r_0 \rightarrow r_2\}$ (down & up). Given P_2, for example, we have $DSR_1 = r_0$, $DSR_2 = r_2$, and $DSR_3 = r_1$. Obviously, the `min_rel` and `max_rel` respectively make the lower and upper bounds in terms of storage capacity consumptions. In most cases, the `prop_reloc` consumes the storage capacity almost as same as typical reliable deduplication storage (`prev`), while always guaranteeing the demanded chunk reliability levels.

B. Guaranteeing of Demanded Chunk Reliability Levels

Figure 3 shows a degree of assuring the demanded chunk reliability levels for the five solutions with the same environment used for the previous (storage capacity consumption) analysis. The `prev` does not guarantee the requested chunk reliability levels all the time. As pointed out, the `prev` unconditionally eliminates all duplicate chunks by making already existing chunks being shared without any consideration of demanded chunk reliability. In case of P_1, the `prev` guarantees the demanded reliability levels, where the initially stored chunks provide the highest reliability level. We could observe similar results with an increased deduplication ratio ($DR = 60\%$).



(a) $DR = 30\%$

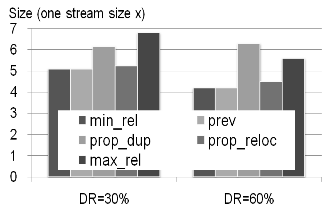


(b) $DR = 60\%$

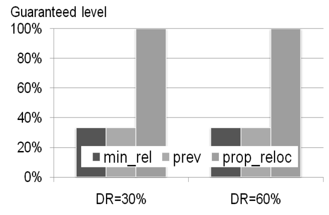
Figure 3: Percentage of assuring the demanded chunk reliability levels with the four reliability combinations (its environment is the same as Fig. 2 and the results of `prop_dup` and `max_rel` are equal to that of `prop_reloc`)

C. Guaranteeing of Chunk Loss Severity Levels

Finally, we analyze how well the demanded chunk loss severity levels are satisfied with different solutions. For this analysis, we assume that the demanded reliability levels of the three data streams remain the same (r_0). However, as a chunk's sharing level grows, its chunk reliability should be increased accordingly to meet the predefined chunk loss severity level (equal to the chunk loss severity level of the non-deduplication case). Figure 4 clearly shows that the `prev` cannot deliver the requested chunk loss severity level.



(a) Storage capacity consumption



(b) Guaranteed level

Figure 4: Percentage of assuring the demanded chunk loss severity levels and storage capacity consumptions

D. Analysis with Realistic Workloads

We evaluate the performance (storage capacity consumption) of our reliability-aware deduplication solution with realistic backup datasets used in the previous work [6]. Actually, we select two typical backup datasets (fredvar and fredp4) from the previous work [6], each of which contains three full backup versions (DS_1, DS_2, and DS_3). Each version is around 4GB in size. We take each version as an input data stream. The deduplication ratios of the backup datasets (fredvar and fredp4) are 30% and 15% respectively, and the number of unique (deduplicated) chunks are 345,070 and 781,250 accordingly. Notice that the number of unique chunks in fredp4 is two times more than that in fredp4, while each version of the both datasets are equally sized. This implies that for fredvar, each version may contain significantly less shared chunks. We adopt the same reliable deduplication storage model as described in the beginning of this section, which provides three different reliability levels. Due to page size limitation, we only present the result for `prop_reloc`. Under $P_0 = \{r_0 \rightarrow r_1 \rightarrow r_2\}$, the deduplicated data sizes taken at each reliability level are 0.11GB, 0.17GB and 4.45GB for fredvar, and 1.68GB, 3.16GB and 5.32GB for fredp4. The result for fredvar matches the fact that each full backup version of fredvar dataset adds only a small amount of new data. Thus, after placing DS_3 onto devices of the highest reliability level, all shared chunks appeared in the previous data streams DS_1 and DS_2 are relocated into the devices which host DS_3. This explains why the 4.45GB is much larger than the rest two. The result for fredp4 shows much less chunk relocations because of the significantly lower chunk sharing level. In other word, each version of fredp4 brings in remarkably more new data, comparing with fredvar.

IV. Summary and Future Work

In this paper, we introduced the two reliability parameters for deduplication storage: the chunk reliability and the chunk loss severity. We also proposed an effective reliability-aware deduplication solution that not only assures all demanded chunk reliability levels by making already existing chunks sharable only if its reliability is high enough, but also mitigates the chunk loss severity by adaptively reducing the probability of having a chunk loss.

To further our current work, the following research issues should be studied. *First, how to more effectively handle the case when the current chunk demands higher reliability than the already existing chunk?* Presently, the `prop_reloc` seems the best in terms of the storage capacity consumption, the guaranteeing of the demanded chunk reliability, and the assuring of the chunk loss severity. However, it could deteriorate spatial localities of chunks in a data stream by removing chunks from their original locations, eventually reducing read performance of the data stream. Note that the `prop_dup` does not affect read performance, whereas it consumes storage capacity considerably. We believe that combining the two solutions can be more effective.

Second, how to efficiently meet various reliability requirements from multiple concurrent data streams? Our analysis revealed that a sequence of writing data streams with different reliability demands could affect storage capacity consumptions. The `prop_dup` consumed the least storage capacity with P_1 (down & down), because the first written chunk could be shared by the subsequent duplicate chunks that demand lower reliability levels.

Third, how to minimize actual damage of a chunk loss? In order to reduce the chunk loss severity, our current approach simply decreases the chance of losing a chunk. Eventually, we should mitigate the actual damage, for example, by using elaborate chunk placement and relocation over multiple disks.

Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable comments. This work was partially supported by grants from NSF (NSF Awards: 0960833 and 0934396). This research was also supported by the MKE(The Ministry of Knowledge Economy), Korea, under the ITRC (Information Technology Research Center) support program supervised by the NIPA (National IT Industry Promotion Agency) (NIPA-2011-C1090-1131-0009).

References

[1] D. Bhagwat, K. Pollack, D. Long, T. Schwarz, E. Miller, and J. Paris, "Providing high reliability in a minimum redundancy archical storage system," in *Proceedings of the 14th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2006.

[2] X. Li, M. Lillibridge, and M. Uysal, "Reliability analysis of deduplicated and erasure-coded storage," in *Proceedings of HotMetrics*, 2010.

[3] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki, "HYDRAstor: A scalable seconary storage," in *Proceedings of the 7th USENIX File and Storate Technologies*, 2009, pp. 197–210.

[4] B. Zhu, K. Li, and H. Patterson, "Avoiding the disk bottlenext in the data domain deduplication file system," in *Proceedings of the 6th USENIX File and Storage Technologies*, 2008.

[5] C. Liu, Y. Gu, L. Sun, B. Yan, and D. Wang, "R-ADMAD: High reliability provision for large-scale deduplication archival storage systems," in *Proceedings of International Conference on Supercomputing*, 2009, pp. 370–379.

[6] N. Park and D. Lilja, "Characterizing datasets for data deduplication in backup applications," in *Proceedings of 2010 IEEE International Symposium on Workload Characterization*, 2010.