Assuring Demanded Read Performance of Data Deduplication Storage with Backup Datasets

Young Jin Nam*†
*School of Computer and Information Technology Daegu University Gyeongsan, Gyeongbuk, KOREA 712-714 Email: youngjin.nam@gmail.com

Dongchul Park‡ and David H.C. Du‡
†Department of Computer Science and Engineering University of Minnesota—Twin Cities Minneapolis, MN 55455, USA Email: {park, du}@cs.umn.edu

Abstract—Data deduplication has been widely adopted in contemporary backup storage systems. It not only saves storage space considerably, but also shortens the data backup time significantly. Since the major goal of the original data deduplication lies in saving storage space, its design has been focused primarily on improving write performance by removing as many duplicate data as possible from incoming data streams. Although fast recovery from a system crash relies mainly on read performance provided by deduplication storage, little investigation into read performance improvement has been made. In general, as the amount of deduplicated data increases, write performance improves accordingly, whereas associated read performance becomes worse.

In this paper, we newly propose a deduplication scheme that assures demanded read performance of each data stream while achieving its write performance at a reasonable level, eventually being able to guarantee a target system recovery time. For this, we first propose an indicator called cache-aware Chunk Fragmentation Level (CFL) that estimates degraded read performance on the fly by taking into account both incoming chunk information and read cache effects. We also show a strong correlation between this CFL and read performance in the backup datasets. In order to guarantee demanded read performance expressed in terms of a CFL value, we propose a read performance enhancement scheme called selective duplication that is activated whenever the current CFL becomes worse than the demanded one. The key idea is to judiciously write non-unique (shared) chunks into storage together with unique chunks unless the shared chunks exhibit good enough spatial locality. We quantify the spatial locality by using a selective duplication threshold value. Our experiments with the actual backup datasets demonstrate that the proposed scheme achieves demanded read performance in most cases at the reasonable cost of write performance.

Keywords-data deduplication; storage; read performance;

I. INTRODUCTION

Data deduplication (for short, dedupe) that efficiently eliminates duplicates from a large set of data streams has already become a key feature of current commercial backup storage systems [1]. A typical dedupe process begins to divide a given data stream into smaller chunks of variable/fixixed lengths and compute a hash value of each chunk (digested chunk information) through a cryptographic hash function (SHA-1). Generally, dynamic chunking (variable-length chunks) with fingerprinting or its variants outperforms static chunking (fixed-length chunks) for data backup applications, where an average chunk size ranges from 4 to 8KB [2]. A hash index table, also known as a sort of key-value store [3], is required to effectively maintain a large number of hash values with their storage locations. Only if the hash value of a chunk is not found in the hash index table (assuming all hash values are kept in the table), the (unique) chunk is allowed to be written into the storage. Otherwise, the (shared) chunk will be ignored. The collision probability of a cryptographic hash function is relatively low enough compared with the soft error rate of the storage system [4]. The unique chunks are not directly written into the underlying storage because the chunk sizes are not large enough to achieve high write performance. Instead, they are organized into a fixed-sized container [2], where the container size is much bigger than the average chunk size. The unique chunks are initially buffered into the in-memory container that is allocated for each data stream. Once the in-memory container is full of the chunks, it is flushed to the underlying storage. To read a chunk from the storage, it is not allowed to read a small chunk itself; instead the entire corresponding container must be read.

Recent dedupe research has focused mainly on maximizing the efficient duplication detection by using better chunking [5], optimized hash indexing [6], locality-preserving index caching [2], and various bloom filters [7]. Moreover, good write performance has been its primary factor by compressing unique chunks and performing (fixed-length) large writes (2 or 4MB) through containers [2], [8] or similar structures.

The read performance (mainly throughput) in dedupe storage, such as backup systems, has not been spotlighted since it was widely adopted that the occurrence of reads is much lesser than that of writes in such systems. However, the read performance becomes extremely critical when it comes to restoring the entire system from crash [2], [9]. Higher read performance can significantly save the recovery time while accomplishing higher system availability. Thus, a
system may request guaranteed read performance to retrieve each data stream in dedupe storage in order to satisfy a target system recovery time, i.e., a target system availability level.

In addition, since the dedupe storage has limited capacity, it occasionally needs to stage the backed data streams stored in the underlying storage to the archive storage such as a virtual tape. This requires reconstruction of their original data streams because the archive storage operations are typically stream-based. In fact, this staging frequency is remarkably higher than the user-triggered data retrieval frequency. Recently, long-term digital preservation (LTPD) communities [10] also have placed great emphasis on the importance of read performance in dedupe storage. One of LTPD requirements is that the repository needs effective mechanisms to detect bit corruption or loss [10]. Some of primary storage has started to equip the dedupe feature [11], where reads are definitely as much important as writes. One example is to store images of virtual machines on shared network storage.

As duplicate data (shared chunks) increases, read performance of data dedupe storage to retrieve a data stream gets worse in general because reading a data stream needs to retrieve both unique and shared chunks, whereas the writes store only the unique chunks (Figure 3 in Section IV-A). Read performance drops because shared chunks are likely to have been physically dispersed over different containers in the underlying storage called chunk fragmentation. Figure 1 illustrates an example of the chunk fragmentation with a data stream having 11 incoming chunks. Assume that the chunks marked with 10, 11, 12, 13, 14, and 15 are unique chunks. On the contrary, the chunks marked with 300, 400, 500, 200, and 201 are shared chunks duplicated with the chunks stored in containers 5, 7, 9 and 2, respectively. We also simply assume that each container has four chunks. If none of the eight incoming chunks are shared with the already existing chunks in the in-storage containers, the data stream read requires to retrieve three containers from the storage. In this example, however, the data stream read needs to access six different containers. If the read cache in dedupe storage can hold only a single container in memory, the read needs to retrieve seven containers (two accesses to the container 31). Obviously, this would degrade read performance considerably. In practice, the actual degradation becomes even worse because much smaller portion of the container would be accessed.

II. PRELIMINARIES

Before presenting the proposed scheme, we provide a few assumptions and definitions to clarify our descriptions.

Dataset & data streams: A backup dataset consists of successive versions of data streams. A data stream represents a stream of bytes corresponding to a full data backup at a given time. Assume that the data stream of the  

version always begins from one, i.e.,  

In this paper, we use six different backup datasets traced in actual data backup environments [4]. Each dataset consists of 2 or 5 successive versions of data streams. Each data stream has been already chunked with variable-length chunking [12] that uses an average chunk size of 8KB. Then, each data stream  

has not seen before in the dedupe storage, i.e., it has no already existing chunk in the underlying storage, it is called a unique chunk. Otherwise, it is called a shared chunk that has an already existing chunk in the underlying storage.

Containers & read cache: Chunks are organized into a container before being written into the underlying storage. Likewise, when being read from the storage, the associated container is read from the underlying storage. The container size is usually fixed and much larger than the average size of a chunk (2MB or 4MB). The container size is denoted by  

represents the number of the in-memory containers that are managed according to the LRU policy (in our dedupe storage). The read cache size denoted by  

represents the number of the in-memory containers that are allocated only for chunk reads and are different from the in-memory container for chunk writes. On each chunk read, we look up the read cache first. If the chunk is found in one of the in-memory containers in the read cache, we read the chunk from the cached in-memory container. Otherwise, we need to read an associated container from the storage and put the container into the read cache after evicting one of the least accessed in-memory container from the read cache based on the LRU policy. A more efficient cache replacement strategy than the LRU policy may be employed, but designing such a policy is beyond the scope of this paper.

Spatial locality: In backup datasets, the read sequence of chunks is very likely to be its write sequence. Thus, in many cases the dedupe storage preserves a given spatial locality by storing the incoming chunks into a series of larger-sized chunks. The first
containers [2]. As a result, reading a single container from the underlying storage may give a series of read cache hits for chunk reads of a data stream. However, suppose that only small portion of the cached container is used like container 5, 7, and 9 in Figure 1. Then, it may cause frequent container replacements in the read cache, consequently decreasing the read performance of the dedupe storage due to the increased number of retrieving containers from the underlying storage.

Read and write performance: Read performance of a data stream refers to the average throughput to read the entire chunks belonging to the data stream from the storage. The average read throughput can be computed by dividing the total size of all the chunks in the data stream by the elapsed time to read them all from the underlying storage. The read cache might be a great deal of help on this data stream read.

Similarly, write performance of a data stream refers to the average throughput to write the entire chunks belonging to the data stream to the underlying storage. The average write throughput can be calculated by dividing the total size of all the chunks in the data stream by the elapsed time to write them all from the underlying storage. The dedupe might help to increase the write throughput by eliminating many shared chunks from their actual storage write and therefore shortening the total write time.

Chunk indexing (or hash index table): The dedupe storage needs an efficient hash index table also called chunk indexing to determine if the incoming chunk has been seen before or not. For this, we assume that the dedupe storage has enough memory space to keep all the chunk index information (20-byte chunk ID and its associated chunk metadata including LBA information). Besides, it is assumed that this indexing process is fast enough compared with the storage read and write.

III. THE PROPOSED SCHEME

The proposed scheme consists of two key components: (1) cache-aware chunk fragmentation level (CFL) monitor to predict degraded read performance and (2) selective duplication to enhance the read performance by improving the current level of chunk fragmentation. Figure 2 depicts the overall architecture of our proposed scheme. It is assumed that incoming data streams have been already chunked according to variable-length chunking. The CFL monitor examines if read performance of the current data stream is worse than demanded read performance, which is expressed in terms of a CFL value (\(CFL_{\text{req}}\)). (We will investigate a correlation between the CFL value and its corresponding read performance in Section IV-A.) If the current CFL becomes lower than the \(CFL_{\text{req}}\), the CFL monitor changes the typical deduplication (for short, dedupe) to the selective duplication.

The typical dedupe eliminates all shared chunks from the storage write, whereas the selective duplication allows shared (non-unique) chunks to be written into the storage unless the accessed shared chunks have a certain level of spatial locality quantitatively defined by threshold \(p\). To fulfill this process, the selective duplication requires to use extra memory space called in-memory temp container. When the current CFL becomes higher than \(CFL_{\text{req}}\), the selective duplication is replaced by the typical dedupe.

A. Cache-aware Chunk Fragmentation Level (CFL) Monitor

The cache-aware chunk fragmentation level (CFL) monitor estimates degraded read performance during a dedupe process by using a performance indicator called cache-aware Chunk Fragmentation Level (CFL). The CFL is composed of two parameters: optimal chunk fragmentation (for short, OCF) and cache-aware current chunk fragmentation (for short, CCF). The OCF follows the same definition in our previous work [9]. However, the definition of the CCF is significantly different from its previous definition in [9]. That is, the previous CCF did not consider a read cache that is closely related to the read performance. The CFL of the current data stream retaining its \(k\)-th incoming chunk is denoted by \(CFL_i^k\) and defined as follows:

\[
CFL_i^k = \min \left\{ 1, \frac{OCF_i^k}{CCF_i^k} \right\},
\]

where \(OCF_i^k\) and \(CCF_i^k\) represent the optimal chunk fragmentation and the cache-aware current chunk fragmentation of \(DS_i^k\), respectively. Note that the CFL value is calculated for each data stream (version), and it varies whenever a new chunk of the data stream arrives.

The OCF of \(DS_i^k\) is computed by dividing the total size of all arrived (unique and shared) chunks by the container size \(S_{cs}\). We denote it by \(OCF_i^k\) and compute it as follows:

\[
OCF_i^k = \frac{\sum_{m=1}^{k-1} s_m + s_k}{S_{cs}},
\]
where $s_m$ is the size of chunk $c_m$. In other words, the $OCF_i^j(k)$ represents how many containers must be retrieved to read all the chunks in $DS_i^j(k)$ on the assumption that no chunks are deduplicated. We assume that chunks are not compressed in containers. The time complexity to compute $OCF_i^j(k)$ from $OCF_i^j(k-1)$ is extremely low; i.e., $O(1)$.

The CCF represents how many containers need to be actually retrieved to read all the chunks in the $DS_i^j(k)$ through the read cache. In order to obtain the CCF value, the CFL monitor executes the read cache replacement algorithm (LRU policy in our scheme) while deduplicating the incoming chunks. The rationale of this approach lies in (1) the read sequence of chunks for $DS_i^j(k)$ is the same as the current write sequence, and (2) the data stream to the data storage will not occur before the data is fully written into the storage. This implies that the data stream write can employ the read cache resource dedicated to the data stream read. In fact, this LRU read cache does not need to store chunk data, instead it just maintains the list. As a result, it can considerably reduce the CPU overhead. The CCF of the $DS_i^j(k)$ (denoted by $CCF_i^j(k)$) can be computed as follows:

$$CCF_i^j(k) = \begin{cases} 
CCF_i^j(k-1) & \text{if } cont(c_k) \in \text{cache} \\
CCF_i^j(k-1) + 1 & \text{otherwise},
\end{cases}$$

where $cont(c_k)$ represents the container to which the chunk $c_k$ belongs. Only if the container retaining the demanded chunk $c_k$ is not found in the LRU list, the number of containers to be retrieved increases by one. Otherwise, we do not need to retrieve the container due to cache hit. The time complexity of updating the CCF is also low since the number of read cache entries is likely to be small in practice. Assuming (1) each data stream has its own cache, and (2) the cache entry size is big enough that is equal to the container size (2-4MB), the time complexity would be $O(S_{cache})$, the number of in-memory containers in the read cache for each data stream (4 or 8). The LRU list is reset every time a new data stream (version) of each dataset begins. By definition, $OCF_i^j(k)/CCF_i^j(k)$ can be greater than 1 when a single data stream hits the read cache many times. In this case, we simply set the value to 1 because its read performance remains high enough.

### B. Selective Duplication

The selective duplication enhances the read performance by improving the current level of chunk fragmentation. It judiciously writes shared (non-unique) chunks into the storage together with unique chunks when the shared chunks do not exhibit high spatial locality.

Algorithm 1 shows how our selective duplication works. If the incoming chunk $c_k$ is unique, it is stored in the in-memory container (lines 7–8). When the in-memory container is full of the chunks, all the chunks in the in-memory container are flushed to the underlying storage. If the incoming chunk is a shared chunk in the storage, the selective duplication (1) writes it into the in-memory container (causing actual write) or (2) eliminates it from actual write like a typical dedupe. Before making this decision, the selective duplication temporarily stores the incoming shared chunk $c_k$ in the temporary in-memory container (called in-memory temp container) as long as its container ID is equal to the container IDs of the chunks in the temp container (line 10). In our design, both container sizes are identical. If the incoming chunk already exists in the in-memory temp container, it is just overwritten. Only when the container ID of the incoming (newly arrived) shared chunk is different from that of chunks in the in-memory temp container, the selective duplication checks if the total size of all the shared chunks in the in-memory temp container is greater than $p$ percentage of the container size ($S_{cache}$). If the container is read from the storage, at least $p$ percentage of the container needs to be accessed for the data stream read (we will investigate a reasonable value of $p$ with actual backup datasets later). If the total size is smaller than $p$ percentage of the container size, all the shared chunks buffered in the in-memory temp container are written to the in-memory container together with unique chunks (line 14). Otherwise, the selective duplication simply eliminates all the shared chunks like the typical dedupe (line 16). Next, it stores the incoming (newly arrived) shared chunk $c_k$ into the in-memory temp container and sets the container ID information of the in-memory temp container to the container ID of the $c_k$ (lines 16–17).

<table>
<thead>
<tr>
<th>Algorithm 1 Selective duplication with threshold $p$.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Require:</strong> incoming chunk $c_k$ in $DS_i^j(k)$</td>
</tr>
<tr>
<td>1: // $CT_{norm}$: in-memory container</td>
</tr>
<tr>
<td>2: // $CT_{temp}$: in-memory temp container</td>
</tr>
<tr>
<td>3: // $id(C_T_k)$: container ID of $C_T_k$</td>
</tr>
<tr>
<td>4: // $id_{temp}$: container ID of the chunks stored in $CT_{temp}$</td>
</tr>
<tr>
<td>5: // $sz_{temp}$: the total size of all the chunks in $CT_{temp}$</td>
</tr>
<tr>
<td>6: // $S_{cache}$: container size</td>
</tr>
<tr>
<td>7: if $c_k$ is a unique chunk then</td>
</tr>
<tr>
<td>8: store $c_k$ into $CT_{norm}$</td>
</tr>
<tr>
<td>9: else</td>
</tr>
<tr>
<td>10: if $id(C_T_k)$ is equal to $id_{temp}$ then</td>
</tr>
<tr>
<td>11: store the shared chunk $c_k$ into $CT_{temp}$</td>
</tr>
<tr>
<td>12: else</td>
</tr>
<tr>
<td>13: if $sz_{temp}$ is smaller than $p$ percentage of $S_{cache}$ then</td>
</tr>
<tr>
<td>14: store all shared chunks in $CT_{temp}$ into $CT_{norm}$</td>
</tr>
<tr>
<td>15: end if</td>
</tr>
<tr>
<td>16: reset $CT_{temp}$; store the shared chunk $c_k$ into $CT_{temp}$</td>
</tr>
<tr>
<td>17: set $id_{temp}$ to $C_T_k$</td>
</tr>
<tr>
<td>18: end if</td>
</tr>
<tr>
<td>19: end if</td>
</tr>
</tbody>
</table>
CFL reaches the $CFL_{req}$ (actually HWM). This case is expected to give the highest read performance, while its write performance becomes worse than the other thresholds.

IV. PERFORMANCE EVALUATION

We implement our dedupe storage simulator on the basis of the DiskSim simulator. The underlying storage includes 9 individual disks (IBM DNE5-309170W), each of which provides storage capacity of 17,916,240 blocks (8.54GB). It is configured as RAID0 to provide better performance and enough storage space to accommodate all the chunks in the backup datasets. The stripe unit size is set to 32KB. We base our chunk indexing (hash index table) on sparsehash originally named google-sparsehash [13]. For read and write performance measurements, we ignore the elapsed time to execute the typical dedupe or our scheme because it is much smaller than storage I/O time. For the dedupe, we use a fixed sized container of 2MB. Then, each container read (write) accesses 64 stripe units, where each individual disk serves about 7–8 stripe unit reads (writes). Our experiments focus on a single data stream, not multiple data streams in parallel. Each data stream has an in-memory container for chunk writes and a LRU-based read cache containing a number of in-memory containers. To read a chunk, the dedupe storage reads a container from the storage and caches it into the read cache after evicting one of the in-memory containers in the read cache based on the LRU policy.

We employ six backup datasets traced in actual data backup environments [4]. The ds-1, ds-2, and ds-3 datasets were obtained from all Exchange server data. The ds-4 contains system data for a revision control system. The ds-5 includes data from the /var directory in the same machine. The ds-6 contains data from home directories with several users. For the experimental purpose, all the datasets except the last one (ds-6) were truncated to be 20GB in size and to have about 1.900K chunks in total. Each dataset contains chunked data streams by using variable-length chunking with an average chunk size of 8KB for data privacy. Each chunked data stream (hereafter, data stream) in the datasets consists of a sequence of chunk records each of which specifies the key chunk information including 20-byte chunk ID (SHA-1 hash value), its LBA information, dataset and version IDs, and a chunk size (not compressed size). The dedupe gain ratio (DGR) represents the ratio of the stored data stream size to the original data stream size. Most datasets except ds-3 contains many duplicated chunks.

Table I shows the DGR variation over different versions in each dataset. In many datasets (ds-1, ds-2, ds-3, and ds-4), the first version does not contain many shared chunks (less than 1–10%). However, successive versions except ds-3 have many shared chunks. Thus, each dataset shows small DGR values (0.29–0.50). In ds-5 and ds-6, the first version contains a significant number of shared chunks (small DGR).

Table I: Variation DGR in successive versions (data streams) of each backup dataset.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>ver-1</th>
<th>ver-2</th>
<th>ver-3</th>
<th>ver-4</th>
<th>ver-5</th>
<th>avg. DGR</th>
</tr>
</thead>
<tbody>
<tr>
<td>ds-1</td>
<td>0.999</td>
<td>0.025</td>
<td>0.069</td>
<td>0.036</td>
<td>0.312</td>
<td>0.29</td>
</tr>
<tr>
<td>ds-2</td>
<td>1.000</td>
<td>0.280</td>
<td>0.247</td>
<td>0.149</td>
<td>0.206</td>
<td>0.37</td>
</tr>
<tr>
<td>ds-3</td>
<td>0.996</td>
<td>0.952</td>
<td>0.977</td>
<td>0.973</td>
<td>0.966</td>
<td>0.97</td>
</tr>
<tr>
<td>ds-4</td>
<td>0.005</td>
<td>0.554</td>
<td>0.636</td>
<td>0.208</td>
<td>0.206</td>
<td>0.50</td>
</tr>
<tr>
<td>ds-5</td>
<td>0.841</td>
<td>0.033</td>
<td>0.025</td>
<td>0.119</td>
<td>0.026</td>
<td>0.20</td>
</tr>
<tr>
<td>ds-6</td>
<td>0.544</td>
<td>0.223</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>0.38</td>
</tr>
</tbody>
</table>

In many cases (ds-1, ds-2, ds-5, and ds-6), a large number of shared chunks are consistently observed from the second version through the last version. However, in ds-4, only the fourth and fifth versions possess a considerable amount of shared chunks. The ds-3 consists mostly of unique chunks, showing a linearly increasing pattern.

A. Correlation between CFL and Read Performance

Figure 3 depicts the variations of read performance when the typical dedupe retrieves each version of backup datasets from the dedupe storage. The read cache is configured with different sizes: $S_{cache} = 2, 4,$ and 8. In most cases (ds-1 through ds-4), the read performance of the first version is reasonably high because most incoming chunks are unique. They are written into the underlying storage through the in-memory container. However, from the second version, their read performance gets worse gradually or rapidly depending on how fast their DGR drops. The ds-4 exhibits high read performance in all versions since its DGR value always remains high, i.e., most chunks are written into the underlying storage without any chunk dedupe. The read performance of ds-5 and ds-6 is low from the first version due to their small DGR values (0.841 and 0.544). It implies that there exist many shared chunks from the first version. We observed that the read cache of $S_{cache} = 4$ and 8 works reasonably in most datasets. When $S_{cache} = 2$, however, the read performance drops considerably. Thus, we adopt $S_{cache} = 4$ for the remaining of our experiments.

Figure 4 plots all the pairs of CFL and its read performance obtained in Figure 3. Regardless of the cache size, we observe that both CFL and its read performance have a strong correlation, i.e., the read performance degrades almost linearly as the CFL decreases. The read performance of the larger container size ($S_{cs} = 4$MB) is slightly higher than that of the smaller container size ($S_{cs} = 2$MB). CFL = 0 means that a data stream read needs to retrieve an extremely large number of containers, implying its performance approaches to almost zero. On the contrary, when CFL = 1, the read performance is almost the same as the sequential read performance. Based on the correlation graphs given in Figure 4(a) and (b), we can convert demanded read performance into a corresponding CFL value. For instance, the demanded read performance of 20MB/s (or 25MB/s) with $S_{cs} = 2$MB can be mapped into $CFL_{req} = 0.6$ (or 0.7). In practice, the CFL
monitor represents the $CFL_{req} = 0.6$ by using two different water mark values, i.e., LWM = 0.6 and HWM = 0.7.

B. Read Performance of the Proposed Scheme ($CFL_{req}$, $p$)

Given different read performance demands ($CFL_{req} = 0.6, 0.7,$ and $0.8$), we explore a reasonable selective duplication threshold $p$ to assure the demanded performance by using $p = 0, 1, 3, 5, 10,$ and $100$. This experiment sets $S_{cache} = 4$ and $S_{cs} = 2MB$.

Figure 5: Given $CFL_{req} = 0.6$ (demanded read performance of 20MB/s), read performance of the proposed scheme to retrieve each version of the backup datasets, where $p = 0, 1, 3, 5, 10, 100,$ and $S_{cs} = 2MB$.

Figure 5(a)–(f) present the variations of read performance when our scheme retrieves each version of the six backup datasets for $CFL_{req} = 0.6$, corresponding to demanded read performance of 20MB/s. In most cases except $ds-2$, our scheme with $p \geq 3$ (3% of the container size, i.e., 60KB, for the selective deduplication threshold) meets the demanded read performance. The typical dedupe ($p = 0$) cannot meet the performance demand, while our scheme with $p = 100$ provides the best read performance. With $p = 1$ (20KB for the threshold), the proposed scheme cannot assure the demanded performance. In $ds-2$, the performance demand can be assured when $p \geq 10$ (more than 10% of the container size, i.e., 600KB, for the threshold). Examining the distribution of the total sizes of all shared chunks in the in-memory temp container has provided clear evidence that the $ds-2$ requires to have higher $p$ to guarantee the demanded read performance. Due to the page limitation, we omitted the detailed results.

Figure 6(a)–(f) present that given demanded CFL, $CFL_{req} = 0.6, 0.7, $ and $0.8$, how successfully the proposed scheme ($p = 3$) can maintain the CFL at different versions.
of each backup dataset. The y-axis represents the average of all observed $CFL_i^k(k)$ values. Given $CFL_{req} = 0.6$, the selective duplication threshold of $p = 3$ works effectively except $ds-2$. However, as the demanded CFL value increases ($CFL_{req} = 0.7$ and 0.8), the $p$ value needs to be gradually increased to meet the demanded read performance. In other words, our scheme has to perform chunk defragmentation more aggressively (more duplication on shared chunks) to meet the higher read performance demand.

![Figure 6](image)

Figure 6: CFL variations of the proposed scheme to retrieve each version of the backup datasets ($p = 3$, $S_{cs} = 2MB$).

Based on the current observation, one possible extension of the current design is to configure the selective duplication threshold $p$ adaptively. For instance, we (1) initially configure $p = 3$; (2) increase $p$ (linearly or exponentially) if the CFL has not been improved by the selective duplication in a given time interval; (3) conversely decrease $p$ down to 3 if the CFL has been improved well and $p > 3$. Alternatively, we can use a control theoretical approached used in [14].

C. Degraded Write Performance of the Proposed Scheme

Given $CFL_{req} = 0.6$, Figure 7(a)-(f) show the variations of associated write performance when our scheme writes each version of the backup datasets, where $p = 0, 1, 3, 5, 10, 100$, and $S_{cs} = 2MB$. In most datasets except $ds-5$ and $ds-6$, the write performance is sustained reasonably high. Obviously, $p = 100$ gives the worst write performance because it writes all the incoming chunks when the current CFL becomes lower than its demanded CFL value (LWM) until the current CFL reaches its HWM. We can observe the same trends of performance results under different configurations of $CFL_{req} = 0.7$ and 0.8, and $S_{cs} = 4MB$.

![Figure 7](image)

Figure 7: Given $CFL_{req} = 0.6$, write performance of the proposed scheme to store each version of the backup datasets, where $p = 0, 1, 3, 5, 10, 100$, and $S_{cs} = 2MB$.

V. RELATED WORK

Zhu et al. in [2] placed an emphasis on that read performance of dedupe storage is critical, in particular, for recovering a data stream in order to reduce a recovery window time. They also proved that read performance noticeably decreased during dedupe process by using synthetic workloads. Since the read sequence of a backup data stream is almost identical with its write sequence, a user might simply expect that the read performance of the data stream will be as high as the sequential read performance. However, in reality, the data stream read in dedupe storage turns out to be no longer sequential because of the existence
of shared chunks physically dispersed over the storage (so-called chunk fragmentation). This chunk fragmentation in a data stream gets more severe with a more number of shared chunks, which inevitably causes considerable degradation of its read performance. Koller and Rangaswami in [15] proposed a selective data replication scheme to increase the read performance by reducing a disk head movement in a single disk. Nam et al. in [9] originally introduced an indicator for the degraded read performance named chunk fragmentation level (also called CFL) and observed a strong correlation between the CFL value and the read performance under backup datasets. However, their CFL has a limitation when a data stream contains many identical (shared) chunks in the same version itself like the datasets of ds-5 and ds-6. That is, by definition of their CFL, since the CCF becomes smaller than the OCF, it makes the CFL higher than one. However, its read performance becomes much worse than the sequential read performance. This problem is mainly caused by not taking into account the read cache effect. Recently, Srinivasan et al. in [16] proposed primary inline dedupe system design (named iDedup). The iDedup exploited both spatial locality by selectively deduplicating primary data and temporal locality by maintaining dedupe metadata completely in memory, not on disk. Unlike our scheme, it was not designed to assure any given read performance; instead it simply tried to enhance dedupe read performance at a best effort manner. Besides, the iDedup does not predict degraded read performance attributed by fragmented chunks.

VI. CONCLUDING REMARKS

In this paper, we emphasized the importance of assuring demanded read performance in dedupe storage in order to help to eventually guarantee a target system recovery time from its crash. For this, we proposed a new dedupe scheme that consists of cache-aware Chunk Fragmentation Level (CFL) monitor and selective duplication. Given demanded read performance (expressed in terms of a CFL value), the CFL monitor computes the current CFL and activates the selective duplication when it becomes worse than the demanded CFL. Then the selective duplication writes shared (non-unique) chunks to the underlying storage only if they do not exhibit enough spatial locality defined by a threshold value (p). By using realistic backup datasets, we demonstrated that the proposed scheme with low selective duplication threshold (p = 3 or 10) could deliver demanded read performance in most backup datasets while achieving write performance at a reasonable level.

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) and by the Ministry of Knowledge Economy, Korea, under the ITRC (Information Technology Research Center) support program (NIPA-2012-H0301-12-3002) supervised by the National IT Industry Promotion Agency.

REFERENCES