Reducing impact of data fragmentation caused by in-line deduplication

Michal Kaczmarczyk, Marcin Barczynski, Wojciech Kilian, and Cezary Dubnicki
9LivesData, LLC
{kaczmarczyk, barczynski, wkilian, dubnicki}@9livesdata.com

Abstract

Deduplication results inevitably in data fragmentation, because logically continuous data is scattered across many disk locations. In this work we focus on fragmentation caused by duplicates from previous backups of the same backup set, since such duplicates are very common due to repeated full backups containing a lot of unchanged data. For systems with in-line dedup which detects duplicates during writing and avoids storing them, such fragmentation causes data from the latest backup being scattered across older backups. As a result, the time of restore from the latest backup can be significantly increased, sometimes more than doubled.

We propose an algorithm called context-based rewriting (CBR in short) minimizing this drop in restore performance for latest backups by shifting fragmentation to older backups, which are rarely used for restore. By selectively rewriting a small percentage of duplicates during backup, we can reduce the drop in restore bandwidth from 12-55% to only 4-7%, as shown by experiments driven by a set of backup traces. All of this is achieved with only small increase in writing time, between 1% and 5%. Since we rewrite only few duplicates and old copies of rewritten data are removed in the background, the whole process introduces small and temporary space overhead.

Categories and Subject Descriptors E.5 [*Files*]: Backup/ recovery; H.3.1 [*Information Storage and Retrieval*]: Content Analysis and Indexing – Indexing methods

General Terms Algorithms, Performance

Keywords deduplication, fragmentation, chunking, backup, CAS

1. Introduction

Deduplication has already been a hot topic in storage for about a decade. The effectiveness of such approach in reducing both time needed to perform backups and storage space required to save them has been widely described and tested [9, 24, 32]. Today, storage systems with data deduplication deliver new records of backup bandwidth [17, 30] and market is being flooded with various dedup solutions proposed by many vendors [1, 12, 15, 16, 26, 31, 35]. In practice, deduplication has become one of indispensable features of backup systems [2, 4].

Traditionally, performance of a deduplication system is described by the data deduplication ratio, maximal write performance and restore bandwidth. Actual performance achieved by a regular customer in his working environment can often differ from these numbers for various reasons [22, 27–29, 36]. In particular, the read bandwidth is usually very good for an initial backup saved to an empty system, but deteriorates for subsequent backups. The primary reason for this is *data fragmentation* caused by in-line deduplication which results in data logically belonging to a recent backup scattered across multiple older backups. This effect increases with each backup, as more and more of its data is actually located in previous backups. Depending on the data set, its characterization and backup pattern, our experiments show a decrease of read performance from a few percent up to more than 50%. As our data sets cover not more than 15 consecutive backups, we expect this percentage to be much higher when more backups are performed.

The increasing fragmentation of subsequent backups can be completely avoided with so-called post-process (off-line) forward-pointing deduplication. In such approach a backup is written without any deduplication, and later the dedup done in the background preserves the latest copy of a block [21, 36]. As a result, the latest backup is never fragmented, and the fragmentation increases with backup age. Since the latest backup is the most likely to be restored, this solution looks promising. Unfortunately, it suffers from many problems, including (1) an increased storage consumption because of space needed for data before dedupli-

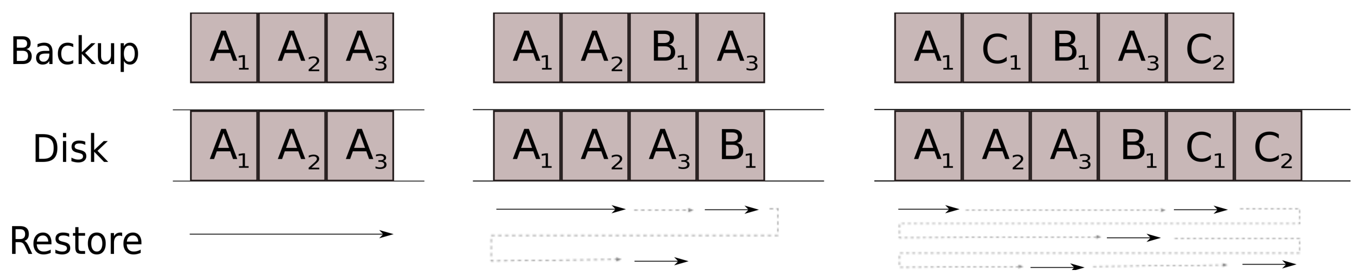

Figure 1. Backup fragmentation process caused by duplicate elimination.

cation, and (2) a significant reduction in write performance of highly duplicated data because writing new copies of duplicates is usually much slower than deduplicating such data in-line [17, 20]. The latter problem occurs because writing new data requires transferring it across the network and committing it to disk, whereas hash-based dedup needs only comparison of a block hash against hashes of blocks stored in the system.

Our goal is to avoid the reduction in restore performance without reducing write performance and without affecting deduplication effectiveness. In other words, the ideal deduplication solution should achieve high write bandwidth of the in-line approach and high restore performance without any read speed reduction caused by fragmentation.

The remainder of the paper is organized as follows: the next section provides the motivation for this work and gives a few examples of the magnitude of the problem. Section 3 looks closer at the nature of the problem of fragmentation and identifies solution requirements. In Section 4 we describe in detail assumptions about backup systems to which our solution is applicable. Section 5 presents the solution called context-based rewriting (CBR in short), followed by its discussion and potential extensions. Section 6 contains the assumptions and the methodology used in performance experiments. Evaluation of our algorithm, including discussion of performance results is given in Section 7. Related work is discussed in Section 8 together with other solutions to the fragmentation problem. Finally, Section 9 contains conclusions and directions for future work.

2. Motivation

2.1 Problem introduction

To illustrate the problem, let's assume full backup of only one filesystem is saved every week to a system with backward-pointing deduplication. In such system the oldest copy of the block is preserved, as is the case with in-line deduplication, because the new copy is not even written.

Usually, a filesystem is not modified much between two backups and after the second backup many duplicates are detected and not stored again. In the end, the first backup is placed in continuous storage space and all the new blocks of the second backup are usually stored after the end of currently occupied space (see Figure 1).

Such scenario is continued during following backups. After some number of backups, blocks from the latest backup are scattered all over the storage space. This results in large number of disk seeks needed for reading the latest backup and in consequence, a very low read performance (see the restore process of the last backup in Figure 1).

This process can be very harmful to critical restore, because the above scenario is typical to in-line deduplication and leads to the highest fragmentation of the backup written most recently – the one which will most likely be needed for restore when user data is lost.

2.2 Problem magnitude

To assess severity of the fragmentation problem, we have simulated drop in restore performance of 4 real backup sets saved to a storage system with in-line deduplication. During the experiments, all duplicates (blocks of expected size 8KB created with Rabin fingerprint chunking algorithm [5, 33]) were eliminated in order to maximize deduplication. After each backup, restore bandwidth was measured as a percentage of maximal bandwidth – i.e. the one achieved when only this backup is present in the system.

In all experiments fixed-size prefetch is used, so we can assume that the read bandwidth is inversely proportional to the number of data I/O operations during restore. Although certainly there are systems for which performance is influenced by other factors, we think that correlating achieved bandwidth with the number of fixed-size I/Os allows us to focus on the core of the fragmentation problem and disregard secondary factors like network and CPU speeds.

Figure 2 shows the restore bandwidth for various backup sets, relative to the maximal bandwidth (as defined above) normalized to 100. That is, the number 45 means that only about 45% of maximal restore bandwidth is achieved for a particular backup set and backup number, i.e. restore time of such backup will more than double because of fragmentation across backup streams.

Overall, there is a clear tendency for the fragmentation to spread out and increase over time (with some rare exceptions). However, the drop in read bandwidth strongly depends on backup set: from 45% of the maximal bandwidth after 7 issue repository backups to about 88% of the maximum after 8 wiki backups.

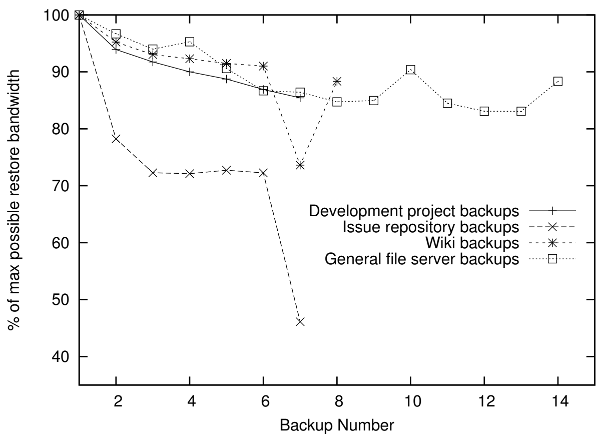

Figure 2. Read bandwidth as a percent of maximal possible bandwidth for each consequent backup (inversely proportional to the number of I/Os needed for restore).

The most important factors causing such variation in results are:

- the characteristics of duplicated/new data pattern – whether the new data is placed rather continuously or in short sequences among duplicates; also whether duplicates appear in the latest backup in long sequences like when originally written, or more randomly (for example because of data deletion),
- number of already backed up versions of a stream – more backups mean more streams to deduplicate against, and in consequence potential for higher fragmentation,

In this work, we do not perform separate analysis of impact of each of these factors on the backup fragmentation, instead we focus on quantifying the read performance drop caused by fragmentation across backup versions, and on effective solution for this problem. Our decision is motivated on one hand by a limited space available for this work; on the other hand both factors mentioned above are related to intrinsic characterization of user data and backup retention policy, both of which usually cannot be changed. However, the investigation of these factors is certainly warranted to estimate severity of the fragmentation problem for a particular user.

3. Deduplication as source of fragmentation

Fragmentation is a natural by-product (or rather waste) of deduplication. It is possible to completely eliminate fragmentation by keeping each backup in a separate continuous disk space with no interference between backups, however in such case there will be no deduplication. Another approach to practically eliminate impact of fragmentation on restore performance is to use big expected block size for deduplication. In such case, when fragmentation happens, it will not degrade restore speed much, because the seek time is dominated by the time needed to read block data. For example, with 4MB expected blocks size, read disk bandwidth of 40 MB/s and 5 ms seek time, a seek on each block read will increase the restore time by about 5%. However, optimal block size for deduplication varies between 4KB and 16KB depending on particular data pattern and storage system characterization (we need to include block metadata in computing the effectiveness of dedup [34]). With much larger blocks, the dedup becomes quite ineffective, so using such big blocks is not a viable option [19, 25, 34].

3.1 Fragmentation impact reduction idea

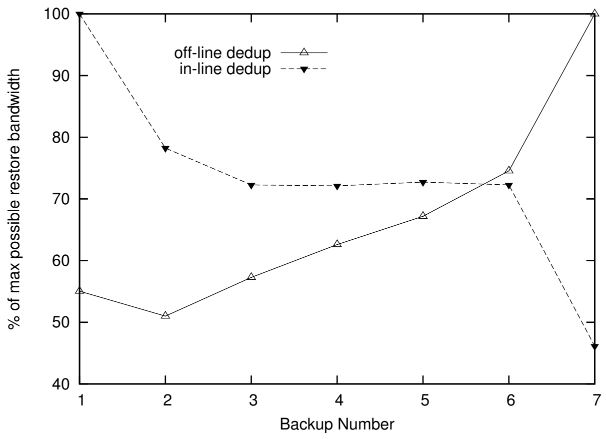

Figure 3. Comparison of backup fragmentation: in-line vs. off-line dedup (restore speed after 7 issue repository backups relative to the maximal restore speed)

In most cases, the latest backup is restored after failure, because users are usually interested in having the most up to date information restored. Based on this, we can try to eliminate fragmentation for the latest backups at the expense of the older ones. An example of such approach is given in Figure 3. It presents the drop in restore performance caused by fragmentation across backup versions as a function of version number in two cases: (1) in-line dedup with fragmentation decreasing with the backup age; and (2) off-line dedup, which results in the latest backup written continuously to disk and fragmentation increasing with the backup age. We'd like to add a defragmentation capability to the in-line dedup to achieve the defragmentation effect similar to the off-line dedup, but without the associated costs.

3.2 Solution requirements

The fragmentation problem requires a solution without negative effects on other important metrics of deduplication system. Such solution should: (1) eliminate reduction in restore bandwidth for the latest backup; (2) introduce very little write performance drop for ongoing backups; (3) not degrade deduplication effectiveness and use little additional space; (4) not require much additional resources like disk bandwidth, spindles and processing power; and (5) offer a range of choices in addressing trade-offs.

4. Backup systems with in-line dedup

To implement our defragmentation solution described in the next section, we assume that backup storage supports the following features:

- *content addressability [32]:* This is a base feature useful for the subsequent features described below.
- *deduplication query based on checking for block hash existence:* It is crucial that this query is hash-based and requires reading metadata only. For our defragmentation solution it is not important if deduplication is attempted against all blocks in the system, or the subset of them (like with sparse indexing [20]). However, we need to avoid reading entire block data to perform dedup, because this would result in fact in reading the fragmented stream and very low total write performance. We note here, that with high fragmentation we may not have enough spindles even to read block metadata fast enough. However, there exist solutions to this problem based on flash memory [7, 23], whereas SSDs are too small and too expensive to hold entire backup data.
- *fast determination of disk-adjacent blocks:* Given two blocks, we should be able to determine quickly if they are close to each other on disk. This can be achieved when each query which confirms block existence in the system returns location of this block on disk.
- *ability to write a block already existing in the system.* This is needed when a decision is made to store a block again in order to increase future read performance. Such rewrite effectively invalidates the previous copy, as the new one will be used on restore. Note that content addressability delivers this feature practically for free.

Many systems with in-line deduplication like DataDomain [37] and HYDRAstor [9] support the above features; for other systems such features or their equivalents can be added. As a result, the algorithm described in the next section can be seen as generic and adoptable to a wide range of systems with in-line deduplication.

5. CBR – context-based rewriting

5.1 Block contexts

Our algorithm utilizes two fixed-size contexts of a duplicate – its disk context and stream context. The stream context of a block in a stream is defined as a set of blocks written in this stream immediately after this block, whereas its disk context contains blocks immediately following this block on disk (see Figure 4). When the intersection of these two contexts is substantial, reading of blocks in this intersection is very fast due to prefetching. In fact, this is quite often the case especially for an initial backup.

The problem of low read bandwidth appears when the disk context contains little data from the stream context. This occurs because of deduplication when the same block is logically present in multiple stream locations with different neighbours in each location. Possible cases include:

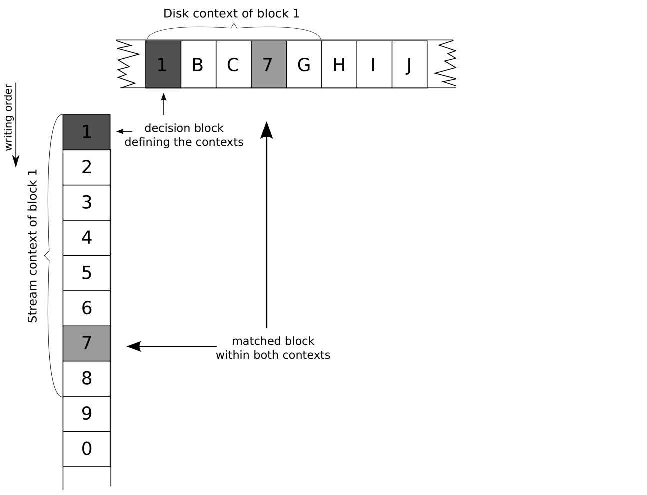
Disk context of block 1
1
B
C
7
G
H
I
J
writing order
Stream context of block 1
1
2
3
4
5
6
7
8
9
0
decision block defining the contexts
matched block within both contexts

Figure 4. Disk and stream contexts of a block

- a global duplicate - i.e. present in different logical backups,
- an internal duplicate - i.e. present in the same stream,
- an inter-version duplicate - i.e. present in one of previous versions of the same stream, for example, previous week backup of the same file system.

In general, the most common is the last case, because of repeated backups of the same data. Moreover, the fragmentation caused by inter-version duplicates in general increases with each backup. Therefore, we focus on this case in the remainder of this work.

The effect of internal duplication is sometimes substantial, as discussed in Section 6.3, but usually it does not get worse with increased number of backups. In all our experiments we consider the effect of fragmentation caused by internal duplication (i.e. each I/O measurement presented in this work includes this effect), but we do not focus on solving this problem.

5.2 The algorithm overview

The basic idea is to rewrite highly-fragmented duplicates, i.e. blocks which stream context in the current backup is significantly different from their disk context. With such rewriting we attempt to make both contexts similar. After rewriting, the new copy of the block can be used for reading and the old copy is eventually reclaimed in the background.

We'd like to rewrite only a small fraction of blocks, because each rewrite slows down backup and consumes additional space until the old copy of the block is reclaimed. By default this parameter, called *rewrite limit*, is set to 5% of blocks seen so far in the current backup.

The algorithm iterates in a loop over the backup stream being written deciding for each encountered duplicate if it should be rewritten. The current duplicated block to be processed by the algorithm we call *the decision block*.

Since decisions whether to rewrite duplicates are made on-line (without future knowledge, except for the stream context), we can always make a sub-optimal choice for a given duplicate: for example by deciding to rewrite it, although such rewrite ”credit” may be better saved for another duplicate later in the stream; or by deciding not to rewrite a duplicate with a hope that a better candidate may appear later in the stream; but such candidate may in fact never materialize. Therefore, the challenge in our algorithm is to make good rewrite decisions.

5.3 Reaching rewrite decisions

To guide rewriting, we introduce a notion of rewrite utility of a duplicate and maintain two rewrite thresholds: the minimal rewrite utility (a constant), and the current utility threshold, adjusted on each loop iteration.

5.3.1 Rewrite utility

If the common part of disk and stream contexts of a decision block is small, rewriting of such block gives us a lot, as we avoid one additional disk seek to read a little useful data. In the other extreme, if this common part is large, such rewriting does not save us much, as the additional seek time is amortized by the time needed to read a lot of useful data.

Therefore, for each duplicate in a backup stream, we define its *rewrite utility* as the size of the blocks in the disk context of this duplicate which do not belong to its stream context, relative to the total size of this disk context. For example, the utility of 70% means, that exactly 30% of blocks in the disk context appear also in the stream context.

5.3.2 Minimal rewrite utility

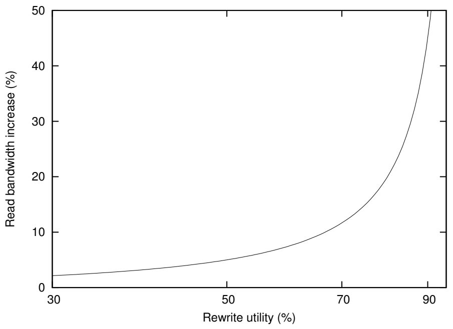

Figure 5. Read bandwidth increase after rewriting 5% of blocks with a given rewrite utility (assuming other blocks are not fragmented).

The minimal utility is a constant parameter of our algorithm ensuring that we avoid rewriting which would improve restore performance only marginally.

We have set minimal utility to 70%. This value may look high, but lower minimal utility is not much useful. Assume a simple case of a backup with 5% of fragmented duplicates, nearly all of them with rewrite utility equal to the minimal rewrite utility. The remaining 95% of blocks are not fragmented (duplicates or not). Moreover, assume that a prefetch of a fragmented block does not fetch any useful data beyond blocks needed to satisfy the rewrite utility of this fragmented block. In such case, rewriting all of the fragmented duplicates improves restore performance by about 12% (see Figure 5), which is in our opinion sufficient to justify the rewriting. If the minimal utility were set to 50%, rewriting all fragmented duplicates in a similar backup would offer only 5% improvement, which is too low in our opinion to justify rewriting.

Note that there may be backups which suffer significantly from fragmentation, but for which all duplicates have rewrite utility just below the minimal utility. However, to reduce restore bandwidth drop caused by fragmentation for such backups, we would need to rewrite many more blocks than just 5%. Fortunately, we have not encountered any such case in our experiments.

5.3.3 Current utility threshold

As explained earlier, current utility threshold is adjusted on each loop of the algorithm. For this, we introduce *best-5%* set of blocks defined as 5% of all duplicates seen so far in the backup stream with the highest rewrite utility. Note that each rewritable block must be a duplicate, so in some cases, we may keep in this set fewer than 5% of all blocks, because there may be not enough duplicates in the stream.

To establish *best-5%*, we compute utility of rewriting each duplicate seen so far alone, without taking into account actual actions taken by the algorithm. In each loop of the algorithm, we set the current rewrite utility threshold to the utility of rewriting the worst of the *best-5%* blocks. Such selection roughly means, that if this value had been used as the current utility threshold for every decision block from the beginning of the backup up to the current point, and without a limit on the number of rewritten blocks, we would have rewritten all the *best-5%* blocks.

Initially, the current rewrite utility threshold is set to the minimal utility threshold and is adjusted after each loop as described above.

5.3.4 Rewrite decision

If the rewrite utility of the current decision block is not below the maximum of the current rewrite utility threshold and the minimal utility, the decision block is rewritten, subject to the 5% rewrite limit (number of blocks allowed to be rewritten changes as we process more blocks, since the limit is computed based on all stream blocks seen so far). Otherwise, all blocks in the context intersection are not rewritten, i.e. they are treated as duplicates in the current stream and marked to be skipped by future loops of the algorithm.

The decision is asymmetric: rewrite only the decision block or mark all blocks in the intersection as duplicates.

That is, even if the decision block is to be rewritten, we do not want to decide now to rewrite other blocks in the intersection, as they may have their context intersections big enough to avoid rewriting. However, once we decide to keep the decision block as a duplicate, we should keep the remaining blocks in the intersection as duplicates too, to ensure that the read of the decision block will fetch also these additional blocks (i.e. the rewrite utility of the decision block remains low).

Block rewriting does not always guarantee that we increase size of the intersection of stream and disk contexts. For example, the stream context may contain duplicates only and we may decide to rewrite just one of them, because remaining are sequential. In such case, the size of the intersection is not increased. However, the rewritten block will still end up on disk close to other new or rewritten blocks. When such blocks are prefetched, they will most likely survive in read cache, reducing number I/Os needed for restore, so rewriting can be still beneficial.

5.4 The algorithm details

5.4.1 Computing the context intersection

The stream context (5MB by default) of the decision block is filled by delaying the completion of this block write until enough write requests are submitted to fill in this buffer. For each request, we resolve its duplicate status by issuing a new modified dedup query based on secure hash of the data (i.e. SHA-1) [10, 11] (if we do not have such information already filled in by one of the previous loops of the algorithm). In case a duplicate is detected, the modified query returns the location of this block on disk, as discussed earlier. While filling in the stream context, we check if a given block is a duplicate appearing already in the disk context of the decision block by comparing distance on disk between two blocks. In such way, the intersection of its disk context and its stream context is determined.

5.4.2 Adjusting rewrite utility threshold

Since tracking utilities of *best-5%* is impractical, we keep a fixed number of utility buckets (for example 10000). Each bucket is assigned disjoint equal sub-range of rewrite utilities, all buckets cover entire utility range, and each bucket keeps number of blocks seen so far with utility in this bucket range. With such structure, we can approximate the rewrite utility of the worst of the *best-5%* blocks with reasonable accuracy – within the range of utility assigned to each bucket.

5.4.3 Background operations

The CBR algorithm requires a background process removing the old copies of the rewritten blocks. This can be done together with other maintenance tasks already run from time to time like deletion, data scrubbing and data sorting [9]. Until this removal is performed, the additional space used by rewrites temporarily reduces the deduplication. As the percentage of such data is limited and the maintenance tasks are usually performed quite often, such solution is acceptable.

5.4.4 Modifications to read operation

If data blocks are content-addressable, both old and new copies have the same address, so pointers to data blocks do not have to be changed when the old copy is replaced with the new one. To ensure good performance of the latest backup restore, we need to modify the read procedure to access the latest copy of a block.

5.5 Discussion

The CBR algorithm is clearly on-line, as we look only at blocks seen so far.

It is optimal in some cases, for example when utility of rewriting each duplicate in a stream is the same; or when such utility is decreasing with increased position of duplicates in the stream.

A bad case is when at the beginning of a stream there are many duplicates with low, but above-minimum utility of rewriting interspersed across the stream, followed later by many duplicates with high utility which are close to each other at the end of the stream. In such case, our algorithm will rewrite initially low-utility duplicates and only later, a few (e.g. 5%) high-utility ones; whereas with future-knowledge we would have re-written only high-utility duplicates. This case can be addressed if we adjust the minimal utility for the current backup based on statistics from the previous backup.

There is a simple modification of the CBR algorithm which seems to eliminate its cost and preserve the advantages: first identify the blocks to be rewritten, and rewrite them later in the background, after backup is finished. This does not work well however, because rewriting would require reading the fragmented blocks, which could be extremely slow (exactly because they are the most fragmented). In the in-line version of the CBR we get these blocks almost for free when a user is writing the data.

6. Experimental Methodology

We have used 4 sets of full backup traces to evaluate the effect of fragmentation and effectiveness of our algorithm. Their characterization is given in Table 1.

6.1 Base model

We assume a simple storage system that consists of a continuous space (something like a one large disk) which is empty at the beginning of each measurement. Moreover, only one user filesystem is backed up into the system every week, and the chunking is done with variable block size of expected size 8KB produced by the Rabin fingerprinting [33].

In this way, we give the priority to evaluate the severity of the problem and efficiency of its solution without obscuring the experiments with many architectural assumptions.

data set name	number of backups	avg. one backup size	avg. duplicate blocks*	avg. internal duplicate blocks
Development project	7	13.74 GB	96%	18%
Issue repository	7	18.36 GB	86%	23%
Wiki	8	8.75 GB	98%	18%
General file server	14	77.59 GB	78%	16%

Table 1. Data sets characteristics (* – data excluding first backup)

6.1.1 Simulating read algorithm

We simulated a simple read algorithm, with the prefetch size of 2 MB and per-stream LRU read cache of 512 MB. We stop prefetching in case a block to be prefetched is already in the cache (so we may prefetch less than 2MB). The cache size was selected to keep only a small fraction of each backup (at most 6%). Selection of prefetch size is explained in Section 7.3.

6.1.2 Stream context size

We use 5MB as stream context size, because it is big enough to allow the CBR to be effective and small enough so increase in write latency due to filling this context is small. Assuming a backup systems achieving 100 MB/s for a single stream [37], it will take not more than 50ms to fill in the context. Note that this delay will be introduced only for non-duplicates, which already have a significant latency.

We experimented with bigger stream context sizes, but they did not improve effectiveness of defragmentation much. The one noticeable effect was however further reduction in number of duplicates rewritten.

6.1.3 Cost of rewriting

When evaluating the cost of our solution, we need to estimate how much we slow down the backup because of rewrites. Since the CBR rewrites duplicates as non-duplicates, we need to establish how much such operation costs. For this we have modified the write path of a commercial backup system HYDRAstor [9, 26] to avoid checking for duplicates and compared the resulting bandwidth to the bandwidth of unmodified system when writing 100% of duplicates. As a result, the latter was 3 times bigger than the former. Based on this, we use factor of 3 slowdown for writing a block vs. deduplicating it. For example, 5% blocks rewritten cause up to 15% slowdown. Since all rewritten blocks are duplicates, the actual slowdown depends on percentage of duplicates in the original stream – the higher the percentage, the higher the slowdown, and 15% slowdown is achieved when all blocks in the stream are duplicates.

6.2 Omitted factors

To keep the analysis simple, we disregarded some factors:

- global deduplication – as mentioned earlier, it is not a focus of our work.
- low-level placement of data on disk – the actual data fragmentation in the storage system is actually greater than in our model because of how data is stored on disk. We disregarded this factor, because we want to study impact of fragmentation caused by deduplication only.
- many disk drives/servers – if a restore can use bandwidth of all available disks, the fragmentation problem in systems with many disks is at least as serious as with one disk only. On the other hand, if we need only a limited-bandwidth restore, availability of multiple disks reduces such severity, because we have many spindles to perform seeks and they will not necessarily lower the restore performance. In this work we assume the former case, because solving the fragmentation problem with many spindles is not cost-effective. As a result, it is sufficient to study fragmentation with one disk only.

6.3 Internal stream duplication

Internal stream duplication – when a block appears more than once in the current backup stream – can impact the restore bandwidth in two very different ways: (1) positively – when the duplicate data is already in the cache and disk operation is not needed to read it, and (2) negatively – when the data was already evicted from the cache. In such case downloading it from disk again may cause reading old disk context, much different than the current stream context. Such data will be put into cache but may be never used in the future (apart from the requested block).

In our traces, internal duplication occurs at relatively high rates – see Table 1. As a result, the effects described above were sometimes clearly visible; from 12% increase of the restore bandwidth of the initial backup to 47% reduction. We include this effect in all our experiments, since the computation of the maximal achievable restore bandwidth takes into account internal fragmentation. Clearly, restore slowdown caused by internal fragmentation remains a problem and needs to be addressed, but it is outside of our focus. Some possible options to address this problem is caching the data in a more intelligent way and duplicating data on the disk for better performance [18].

7. Evaluation

7.1 Meeting the requirements

The CBR algorithm presented above optimizes future read bandwidth of the latest stream, as illustrated by Figure 6. Defragmentation with our algorithm is very effective for all traces, because the resulting restore bandwidth of the latest backup is within a few percent of the optimal restore

data set name	B/W before defragmentation as % of optimal	B/W after defragmentation as % of optimal	% of blocks rewritten	write time increase
Development project	85.43%	92.57%	1.16%	3.13%
Issue repository	46.12%	95.58%	2.61%	4.78%
Wiki	88.34%	95.48%	1.41%	3.96%
General file server	88.34%	95.77%	0.52%	0.94%

Table 2. Impact of CBR defragmentation on the last backup of each backup set. Optimal bandwidth of restore without inter-version fragmentation normalized to 100.

which is achieved after complete defragmentation (for example by off-line deduplication). Without defragmentation, bandwidth of restore from the latest backup with in-line dedup is below 50% of the optimal for the issue repository backup; and 85%-88% for other backup sets. The biggest increase in fragmentation occurs for backups 2 and 7 of issue repository. This is caused most likely by data deletion, because these backups are the only ones significantly smaller than their predecessors.

Assuming eliminating duplicates immediately is 3 times faster than writing them as new blocks, the CBR ensures, that the write bandwidth is reduced by not more than 15% since we rewrite not more than 5% of all blocks. As seen in Table 2, our algorithm actually rewrites much less than the target 5%. This is because we are very conservative since our minimal rewrite utility is set high at 70% and we always observe the 5% limit while processing backup streams. As a result, the CBR reduces write bandwidth by only 1%-5% (see write time increase column in Table 2).

Our algorithm does not use additional space except for rewritten duplicated blocks, therefore the additional space consumption is below 5% of all blocks. Actual number is much lower – between 0.52% and 2.61%. Old copies of the blocks are removed in the background, for example as part of deletion process running periodically, so this space consumption is temporary. Additional disk bandwidth consumption is also limited to writing rewritten blocks.

The algorithm is also easily tunable by setting the percent of blocks to be rewritten. The higher the percentage, the better restore performance at the expense of bigger drop in write performance and more disk space required for storing temporarily old copies of the rewritten blocks.

7.2 Setting rewrite limit

To select the best value for the rewrite limit, we performed experiments varying this limit from 1% to 7% while keeping the minimal rewrite utility unchanged at 70%. The results for the latest backup in each backup set are given in Figure 7. Setting this limit to low values like 2% or even 1% works well for all sets except issue repository, for which the rewrite limit of 5% offers the lowest reduction in restore bandwidth. Increasing this limit beyond 5% does not give us anything and may increase the backup time significantly, so we decided to set this limit to 5% for all experiments.

7.3 Disk context/prefetch size

For reading data which is continuous on disk, bigger prefetch size is generally better. As Figure 8 shows, this is not true for fragmented data. Here, in almost all cases, the best fixed size prefetch is 2MB. One exception is issue repository backup, for which 512KB prefetch is better. Since we compare here using different prefetch sizes, extrapolation of performance based on number of I/Os only cannot be done (comparison result in such case depends on how much data a disk can read in a seek time). Therefore, we have measured disk characteristics (seek time 6.75ms, disk bandwidth 84MB/s) to be able to reason about achieved performance.

Based on those results, we have decided to use 2MB prefetch. It is possible that with the CBR some other prefetch size is better, so our estimation of efficiency of this algorithm is conservative for all sets except issue repository. We have done so because we extrapolate performance based on number of I/Os, and with different prefetch sizes it would be not possible without additional assumptions.

Since for the issue repository backup set, the best tested prefetch size is 512KB, we simulated both a system with and without the CBR algorithm using this prefetch size. In such case, the fragmentation caused restore performance drop of 25% (instead of 45% for 2 MB prefetch). However, the CBR algorithm was still able to reduce this drop to about 5%. The CBR with 2MB prefetch size also reduced this drop, but to 12% (100% here corresponds to the restore bandwidth achieved with 512KB prefetch when no inter-version fragmentation is present).

7.4 Effect of compression

So far we have assumed that the backup data is not compressible. If we keep the prefetch size constant and equal to 2MB, the compressible data results in fragmentation problem magnitude increased and the CBR algorithm delivering even better relative improvements in restore bandwidth. For example, for 50% compressible data, the drop in restore performance increases from 12-55% to 15-65%, whereas the CBR defragmentation reduces this drop to 3-6% (instead of 4-7% without compression).

Obviously, selecting different prefetch size based for example on compressibility of data could change these results, but we assume one fixed prefetch size for all types of data (and optimized for incompressible data).

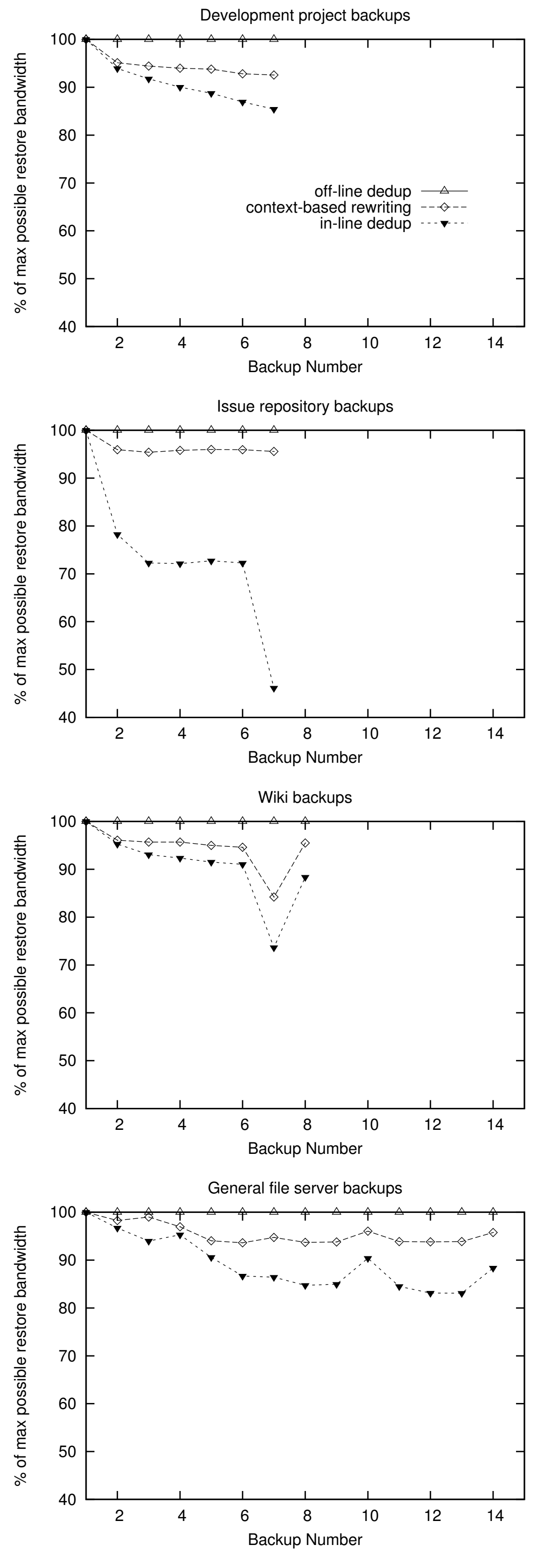

Figure 6. Percentage of maximum possible restore bandwidth (i.e. without inter-version fragmentation) after each completed backup, in 3 cases: (1) in-line dedup with the CBR, (2) in-line dedup without any defragmentation and (3) off-line dedup.

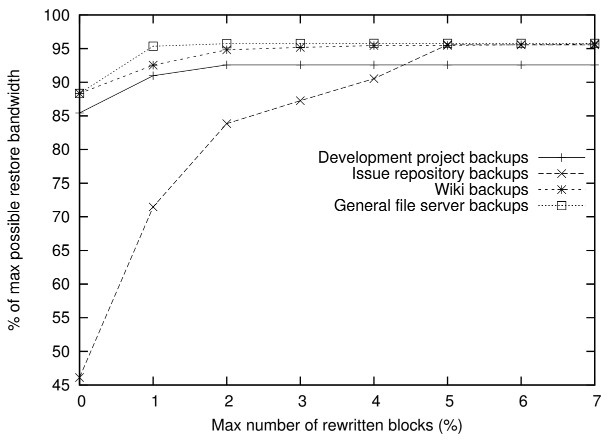

Figure 7. Impact of rewrite limit on restore bandwidth.

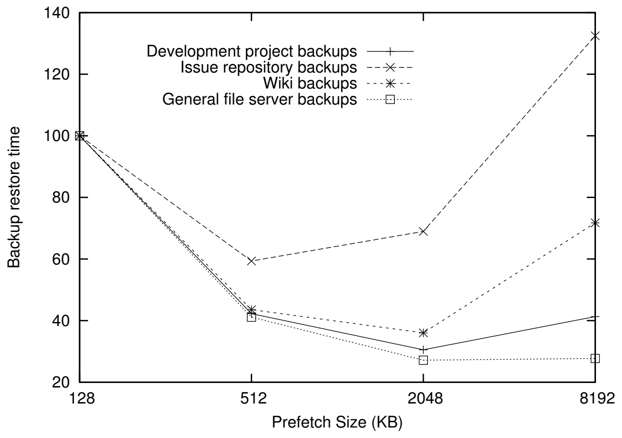

Figure 8. Relative read time of the latest backup for each backup set, before defragmentation, as a function of prefetch size.

7.5 Potentially unnecessary rewrites

With future knowledge, some of the rewrites may be not necessary, because blocks in the disk context of the decision block which look bad (i.e. they are not in the stream context), in fact may appear later in the stream. To estimate how many such rewrites occur, we extend stream context size from the standard 5MB to cover entire future part of the stream.

We define a rewrite as potentially unnecessary, when the rewrite utility computed with such extended stream context is below the minimal rewrite utility. It turns out that the fraction of potentially unnecessary rewrites is relatively high, 30-44% for the latest backup, depending on a backup set. To avoid some of these rewrites, we can increase the standard stream context size. For example, increasing it from 5MB to 20MB reduces this fraction to 22-37%.

However, even with these potentially unnecessary rewrites, we rewrite a very small fraction of all stream blocks.

8. Related work and alternative solutions

The fragmentation of the latest backup can be avoided with post-process deduplication. Unfortunately, this simple solution has a few serious problems, discussed in Section 8.1. Some vendors, like EMC try to fight the fragmentation with time and resource consuming housekeeping processes [22, 37]. The description of this process has not been published, but one possible approach is to selectively rewrite subset of duplicates in the background, i.e. in a way similar to our CBR approach, but done off-line. More on such algorithm is given in Section 8.2. Other systems like HYDRAstor [9, 26] use bigger block size (64KB) which reduces the severity of the fragmentation problem, but at the expense of reduced deduplication. Finally, we can eliminate fragmentation by deduplication with logical objects. In early versions of EMC Centera [13], the unit of deduplication was entire file, which worked well for Centera's target market, i.e. archiving, but is not the right solution for backups, because file modifications make such dedup ineffective.

8.1 Off-line deduplication

One simple solution which satisfies some of the requirements above is already present on the market and is called off-line deduplication [27, 28, 35]. In its simplest form, all data from the current backup are stored continuously on disk, and the deduplication is done in the background in such a way that the blocks from the latest backup are a base for eliminating duplicates from older backups [21, 27]. As a result, the currently written stream has no fragmentation and older backups are fragmented proportionally to their age. Even though the algorithm was most probably not designed to deal with fragmentation, it is very effective for eliminating it in recent backups. However, since deduplicating a block is usually much faster than sending it over a wire and storing it on disk, off-line deduplicating systems may be slower than in-line deduplicating systems (or require more spindles and network bandwidth to avoid such problem).

The percentage of duplicates in a backup depends on data patterns, but usually is much higher than 75%, as little data is modified between backups. As explained in section 6.1.3, deduplication without writing the data can be 3 times faster than writing the data first and then deduplicating it in the background. Therefore, writing a backup stream with 75% of duplicates costs 300 time units with off-line deduplication and only 175 time units using a system with in-line dedup, even if such system does a dedup query for each block. As a result, using off-line dedup results in backup window more than 70% larger. This is clearly not acceptable, as backup window usually cannot be extended much.

8.2 Comparison of defragmentation solutions

Beyond off-line deduplication and the in-line CBR, there is at least one more option – to perform the context-based rewriting in the background, i.e. off-line, mentioned already in Section 5.5. Such solution does not affect backup writing at all, but it needs a long time to complete reading the fragmented data and rewriting them in the background. Additionally, a restore attempted before block rewriting is completed will still suffer from low bandwidth.

The comparison of all mentioned alternatives is presented in Table 3. We note here that storage consumption of both off-line options can be improved by staging, i.e. by running the process of removing the duplicates (or rewriting some of them) in parallel but little behind the process of backup writing. Staging however requires more resources like available disk bandwidth and spindles.

Aspect	Off-line dedup	In-line CBR	Off-line CBR
duplicated stream write bandwidth	low	high	high
read bandwidth	maximal	close to maximal	low after backup, later close to maximal
additional disk space	low (with staging)	low (not more than 5%)	low (with staging)
additional disk bandwidth and spindles	high (write all duplicates and remove)	low (no duplicates written except for 5% rewrites)	medium (additional reads)

Table 3. Comparison of defragmentation solutions.

8.3 Remaining related work

In general, there is not much work about the problem of data fragmentation even though it is known for some time already [21, 22, 27–29, 36].

A few papers investigated improving metadata read for faster duplicate elimination. Zhu et al. [37] describes a solution with Bloom Filter and stream-oriented metadata prefetch, whereas Lillibridge et al. [20] argues that sparse indexing (eliminating duplicates only within previously selected few large segments) is better due to smaller memory consumption. These solutions assume streaming write pattern, whereas SSD can be used for elimination of random duplicates [7, 23]. Such approach makes the fragmentation problem even harder, as more fine-grained duplicates can be detected. Additionally, none of the above techniques solves the problem of reading the fragmented data and in all cases fragmentation increases with subsequent backups.

If we relax the requirement on defragmentation solution of not degrading deduplication effectiveness, we can try to deduplicate only within a subset of data, therefore potentially reducing fragmentation. Besides sparse indexing, such approach is possible with extreme binning [3], with large segment similarity like in ProtectTier [1], subchunk deduplication [34], and with multi-node systems restricting dedup to one or a subset of nodes like Pastiche [6] and DataDomain global extension [8, 14]. Unfortunately, even if we consider very few (2-3) segments of previous backups to deduplicate the current segment against, those segments may already be not continuous on disk, because they may contain also duplicates from other, older segments.

9. Conclusions and future work

In this work we have described data fragmentation in systems with in-line deduplication and quantified impact of fragmentation caused by inter-version deduplication on restore speed. The problem is quite severe, and depending on backup characteristics may result in restore speed drop of more than 50%, as shown by experiments driven by set of real backup traces, with 7-14 backups in each set. For larger sets, with backups spanning many months or even years, we expect this problem to be much bigger. Moreover, it affects latest backups the most, which are also the most likely to be restored.

To deal with the problem, we have proposed the context-based rewriting algorithm (CBR in short) which rewrites selected few duplicates (not more than 5% of all blocks). The new algorithm improves restore speed of the latest backups, while resulting in fragmentation increased for older backups, which are rarely used for restore. Old copies of the rewritten blocks are removed in the background, for example during periodic deletion and space reclamation process which is already required in storage systems with deduplication.

Our trace-driven simulations have shown that rewriting a few selected blocks reduces write performance a little, but practically eliminates the restore speed reduction for the latest backup (within 4-7% of the optimal bandwidth).

For future work, we plan to investigate impact of deletion on fragmentation, and letting the CBR algorithm be less conservative (allowing it to exceed temporarily the desired fraction of rewritten blocks).

Additionally, a number of interesting extensions of the CBR algorithm are possible: (1) a simulation of reading during writing can suggest which blocks should not be rewritten as they would be present in system caches; (2) extending stream context to include blocks written before a given block should make context analysis more precise and lower the number of rewritten blocks; (3) avoiding too frequent rewriting (for example of popular blocks with all bits zeros) by using timestamps - recently rewritten blocks are not rewritable for some time; (4) adjusting prefetch size based on effectiveness of recent prefetches; (5) setting minimal utility of the current backup by using previous backup statistics.

We plan to research the impact of these extensions on algorithm effectiveness. Moreover, we plan to evaluate the proposed solution in a commercial backup system.

This paper concentrates on fragmentation caused by the inter-version duplication within one backup set. Even though this is the dominant type of fragmentation, two more sources of the problem are still to be examined. Both internal and global deduplication may result in significant drops in restore performance. Addressing these issues would make another step in critical restore optimization.

References

[1] L. Aronovich, R. Asher, E. Bachmat, H. Bitner, M. Hirsch, and S. T. Klein. The design of a similarity based deduplication system. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, pages 6:1–6:14, New York, NY, USA, 2009. ACM.

[2] T. Asaro and H. Biggar. Data De-duplication and Disk-to-Disk Backup Systems: Technical and Business Considerations, 2007. The Enterprise Strategy Group.

[3] D. Bhagwat, K. Eshghi, D. D. E. Long, and M. Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *Proceedings of the 17th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2009)*, Sep 2009.

[4] H. Biggar. Experiencing Data De-Duplication: Improving Efficiency and Reducing Capacity Requirements, 2007. The Enterprise Strategy Group.

[5] A. Z. Broder. Some aplications of rabin's fingerprinting method. In *Sequences II: Methods in Communications, Security, and Computer Science*, pages 143–152. Springer-Verlag, 1993.

[6] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: making backup cheap and easy. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*, pages 285–298, New York, NY, USA, 2002. ACM.

[7] B. Debnath, S. Sengupta, and J. Li. Chunkstash: Speeding up inline storage deduplication using flash memory. In *2010 USENIX Annual Technical Conference*, June 2010.

[8] W. Dong, F. Douglis, K. Li, H. Patterson, S. Reddy, and P. Shilane. Tradeoffs in scalable data routing for deduplication clusters. In *Proceedings of the 9th USENIX conference on File and Storage Technologies*, FAST'11, pages 15–29, Berkeley, CA, USA, 2011. USENIX Association.

[9] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki. HYDRAstor: a Scalable Secondary Storage. In *FAST'09: Proceedings of the 7th USENIX Conference on File and Storage Technologies*, pages 197–210, Berkeley, CA, USA, 2009. USENIX Association.

[10] C. Dubnicki, C. Ungureanu, and W. Kilian. FPN: A distributed hash table for commercial applications. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, pages 120–128, Washington, DC, USA, 2004. IEEE Computer Society.

[11] D. E. Eastlake and P. E. Jones. US Secure Hash Algorithm 1 (SHA1). RFC 3174 (Informational), September 2001.

[12] EMC Avamar: Backup and recovery with global deduplication, 2008. http://www.emc.com/avamar.

[13] EMC Centera: Content addressed storage system, January 2008. http://www.emc.com/centera.

[14] EMC Corporation: Data Domain Global Deduplication Array, 2011. http://www.datadomain.com/products/global-deduplication-array.html.

[15] EMC Corporation: DataDomain - Deduplication Storage for Backup, Archiving and Disaster Recovery, 2011. http://www.datadomain.com.

[16] Exagrid. http://www.exagrid.com.

[17] D. Floyer. Wikibon Data De-duplication Performance Tables. *Wikibon.org*, May 2011. http://wikibon.org/wiki/v/Wikibon_Data_De-duplication_Performance_Tables.

[18] R. Koller and R. Rangaswami. I/O Deduplication: Utilizing content similarity to improve I/O performance. volume 6, pages 13:1–13:26, New York, NY, USA, September 2010. ACM.

[19] E. Kruus, C. Ungureanu, and C. Dubnicki. Bimodal content defined chunking for backup streams. In *Proceedings of the 8th USENIX conference on File and Storage Technologies*, FAST'10, pages 239–252, Berkeley, CA, USA, 2010. USENIX Association.

[20] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezis, and P. Camble. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *FAST'09: Proceedings of the 7th USENIX Conference on File and Storage Technologies*, pages 111–123, 2009.

[21] J. Livens. Deduplication and restore performance. *Wikibon.org*, January 2009. http://wikibon.org/wiki/v/Deduplication_and_restore_performance.

[22] J. Livens. Defragmentation, rehydration and deduplication. *AboutRestore.com*, June 2009. http://www.aboutrestore.com/2009/06/24/defragmentation-rehydration-and-deduplication/.

[23] D. Meister and A. Brinkmann. dedupv1: Improving Deduplication Throughput using Solid State Drives (SSD). In *Proceedings of the 26th IEEE Symposium on Massive Storage Systems and Technologies (MSST)*, May 2010.

[24] A. Muthitacharoen, B. Chen, and D. Mazires. A low-bandwidth network file system. In *In Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01*, pages 174–187, New York, NY, USA, 2001. ACM.

[25] P. Nath, B. Urgaonkar, and A. Sivasubramaniam. Evaluating the usefulness of content addressable storage for high-performance data intensive applications. In *HPDC '08: Proceedings of the 17th international symposium on High performance distributed computing*, pages 35–44, New York, NY, USA, 2008. ACM.

[26] NEC Corporation. HYDRAstor Grid Storage System, 2008. http://www.hydrastor.com.

[27] W. C. Preston. The Rehydration Myth. *BackupCentral.com*, June 2009. http://www.backupcentral.com/mr-backup-blog-mainmenu-47/13-mr-backup-blog/247-rehydration-myth.html/.

[28] W. C. Preston. Restoring deduped data in deduplication systems. *SearchDataBackup.com*, April 2010. http://searchdatabackup.techtarget.com/feature/Restoring-deduped-data-in-deduplication-systems.

[29] W. C. Preston. Solving common data deduplication system problems. *SearchDataBackup.com*, November 2010. http://searchdatabackup.techtarget.com/feature/Solving-common-data-deduplication-system-problems.

[30] W. C. Preston. Target deduplication appliance performance comparison. *BackupCentral.com*, October 2010. http://www.backupcentral.com/mr-backup-blog-mainmenu-47/13-mr-backup-blog/348-target-deduplication-appliance-performance-comparison.html.

[31] Quantum Corporation: DXi Deduplication Solution, 2011. http://www.quantum.com.

[32] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *FAST'02: Proceedings of the Conference on File and Storage Technologies*, pages 89–101, Berkeley, CA, USA, 2002. USENIX Association.

[33] M. Rabin. Fingerprinting by random polynomials. Technical report, Center for Research in Computing Technology, Harvard University, New York, NY, USA, 1981.

[34] B. Romanski, L. Heldt, W. Kilian, K. Lichota, and C. Dubnicki. Anchor-driven subchunk deduplication. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, SYSTOR '11, pages 16:1–16:13, New York, NY, USA, 2011. ACM.

[35] SEPATON Scalable Data Deduplication Solutions. http://sepaton.com/solutions/data-deduplication.

[36] L. Whitehouse. Restoring deduped data. *searchdatabackup.techtarget.com*, August 2008. http://searchdatabackup.techtarget.com/tip/Restoring-deduped-data.

[37] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.