Anchor-Driven Subchunk Deduplication

Bartłomiej Romański
9LivesData, LLC
Warsaw, Poland
romanski@9livesdata.com

Łukasz Heldt
9LivesData, LLC
Warsaw, Poland
heldt@9livesdata.com

Wojciech Kilian
9LivesData, LLC
Warsaw, Poland
wkilian@9livesdata.com

Krzysztof Lichota
9LivesData, LLC
Warsaw, Poland
lichota@9livesdata.com

Cezary Dubnicki
9LivesData, LLC
Warsaw, Poland
dubnicki@9livesdata.com

ABSTRACT

Data deduplication, implemented usually with content defined chunking (CDC), is today one of key features of advanced storage systems providing space for backup applications. Although simple and effective, CDC generates chunks with sizes clustered around expected chunk size, which is globally fixed for a given storage system and applies to all backups. This creates opportunity for improvement, as the optimal chunk size for deduplication varies not only among backup datasets, but also within one dataset: long stretches of unchanged data favor larger chunks, whereas regions of change prefer smaller ones.

In this work, we present a new algorithm which deduplicates with big chunks as well as with their subchunks using a deduplication context containing subchunk-to-container-chunk mappings. When writing data, this context is constructed on-the fly with so-called anchor sequences defined as short sequences of adjacent chunks in a backup stream (a stream of data produced by backup application containing backed up files). For each anchor sequence, we generate an anchor – a special block with set of mappings covering a contiguous region of the backup stream positioned ahead of this anchor sequence. If anchor sequences have not changed between backups, the mappings created with the previous backup are prefetched and added to the deduplication context. It is of limited size and fits in the main memory unlike solutions which require keeping all subchunk mappings for the entire backup stream. At the same time, the context provides most of mappings needed for subchunk deduplication. Compared to CDC, the new algorithm results in up to 25% dedup ratio improvement achieved with almost 3 times larger average block size, as verified by simulations driven by real backup traces.

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, chunking, backup, CAS, CDC

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SYSTOR '11 May 30 – June 1, Haifa, Israel
Copyright 2011 ACM 978-1-4503-0773-4/11/05 ...$10.00.

1. INTRODUCTION

The primary value of deduplication is obviously in reduction of required storage space, which translates into reduced probability of data loss and significant operational savings due to reduced power consumption and overall lower facility costs. However, for these savings to materialize, we need to consider total storage space necessary to keep not only backup data, but also all types of metadata (e.g. backup application metadata, intermediate level metadata like filesystems and proper back-end metadata used for location of data and deduplication). Adding metadata to the picture is not only necessary to estimate the savings from deduplication, but often changes the relative utility of various deduplication alternatives in a particular storage system.

Today, the standard technique for dividing backup data into chunks for deduplication is content defined chunking (CDC) [21, 26]. It produces variable-length chunks using Rabin's fingerprints [28] computed over small fixed-size rolling window to select chunk boundaries. Chunk is cut when the fingerprint has one of the predefined values. This approach allows for detection of unchanged chunks after the original stream of data is modified with insertions and deletions, as unchanged cut points are recalled on the next chunker run. Additional advantage of CDC is that it does not maintain any state. Moreover, to work well, CDC does not need to know association between data streams and backups, nor the backup stream boundaries (i.e. where one backup stream ends and another one starts). This fits nicely with the reality of commercial setups, where a standard backup application usually assumes "dumb" storage system, which makes communication of this information to this system impossible.

Although the chunks produced by CDC are of variable length, their actual sizes are clustered around the expected chunk size value which is a parameter of the CDC algorithm. When using CDC, for any given sequence of backup streams

and a particular storage system, there exists an optimal expected chunk value which delivers the best deduplication ratio. This selection depends on the type and the frequency of data modifications in subsequent streams, as well as on metadata overhead imposed on chunk representation. Using small chunks is not optimal because of metadata overhead, moreover, small chunks also tend to negatively impact performance. On the other hand, larger chunks are also often sub-optimal when granularity of modifications in subsequent backups is fine-grained. These tradeoffs suggest selection of a moderate expected chunk size which should work well for a given storage system. However, we can often do better than that because granularity of change in backup stream is not constant even with one sequence of backups (not to mention across multiple series of unrelated backups). That is, in multiple backups, we have long stretches of data which do not change for long periods of time for which large chunk sizes are better, interspersed with regions of change preferring smaller chunk sizes.

In this work, we describe a new algorithm called anchor-driven subchunk deduplication. The new algorithm deduplicates on two levels – with large chunks and their subchunks. Deduplication with large chunks is done using global index of large chunk hashes. The dedup with subchunks is done by using a limited deduplication context of subchunk-to-container-chunk mappings. This context is read from the storage system and updated dynamically in the process of writing backup data. The context is small and can be kept in the main memory. The new approach does not require the storage system to know backup stream boundaries and relations, and allows for subchunk dedup across all streams. At the same time, the context keeps most of subchunk mappings needed for effective dedup. We have verified these claims by simulating the new algorithm with a set of backup traces and comparing the results against the numbers achieved with alternative, previously described algorithms. The remainder of this paper is organized as follows. The next section discusses in detail the new deduplication algorithm. Section 3 presents its evaluation based on simulation driven by a set of real backup traces. Related work is discussed in Section 4, whereas conclusions and future work are given in Section 5.

2. THE ALGORITHM

2.1 System model

Our storage system is a simplified version of HYDRAstor [14]. Initially, we assume a single-node system, and only later, in the evaluation we discuss impact of distributed architecture of the modeled system.

The assumed system stores variable-size content-addressable blocks and exports a traditional file system interface to be used by backup applications. In this work, we differentiate between chunks and blocks of data. A chunk is usually a contiguous piece of user data stream with borders defined by a chunker to maximize deduplication opportunity. A block is a base unit stored by the system and contains either a chunk of user data or pointers to other blocks. Pointers are represented as hash addresses and facilitate building trees keeping file system structures. Each block has an associated block-level metadata.

The system supports multiple failure resiliency levels; the higher resiliency implies the more storage overhead. The highest resiliency is usually used for metadata.

The system can be accessed by several backup servers writing and restoring backups with standard backup applications, which are unaware of special capabilities of the storage system like deduplication. In the base HYDRAstor system, backup streams are cut into chunks using the CDC algorithm and written as blocks to the block store. In the simplified model we assume that at any given time, no more than one backup stream is being written.

The system supports on-demand data deletion implemented with per-block reference counting with garbage collection. Interaction with deletion must be taken into account when designing a deduplication algorithm.

In such system there are several types of metadata which should be included in an evaluation of deduplication algorithms: the lowest-level is block metadata including the content-derived address of this block. Next level metadata contains pointers belonging to this block. There is also higher-level metadata associated with file system structures, for example inode table.

2.2 Deduplication with multiple chunk sizes

Two previous attempts at deduplication with multiple chunk sizes inspired our work.

A bimodal [15] approach assumes that backup data streams consist of reasonably long interleaving regions of duplicate and non-duplicate data and that the fine-grain stream modifications in subsequent backup streams affect repeatedly areas around borders between such regions. This algorithm uses two chunk sizes, small and large. By default, all data is chunked with large chunk size and deduplication is attempted for resulting chunks. If a non-duplicate large chunk is adjacent to a duplicate large chunk then such sequence of chunks is considered a transition point and one or a few more adjacent non-duplicate large chunks are re-chunked using smaller chunk size. The resulting small chunks are inserted into the block store to allow for finer-grain deduplication around transition point; while all remaining new chunks are emitted as large. This algorithm does not need any additional database of small chunks; instead, it queries the block store to verify if the block is duplicate before deciding which chunks will be actually cut into smaller chunks and which should be emitted.

Fingerdiff algorithm [12] assumes that it is aware to which dataset a given backup belongs to. For each dataset this algorithm maintains a database of all small chunks encountered but not necessarily emitted in the latest backup. This database is in addition to the emitted chunk metadata maintained by the block store. Fingerdiff detects duplicates on small chunk level and coalesces them as much as possible (with a limit of maximum possible number of small chunks for all new data).

These two approaches have significant disadvantages. Bimodal "forgets" small chunks in deduplicated large chunks, which is especially important on borders of regions of change. Fingerdiff requires a substantial database of all small chunks seen so far in the latest backups, even though for data which does not change often it is not useful. This database may also not fit in the main memory, seriously affecting the performance. Moreover, fingerdiff assumes that storage system is able to identify relation between backup streams, which often today is not the case, as explained earlier. Finally, fingerdiff will not detect duplicates on the small chunk level across unrelated backup streams.

2.3 Simplified subchunk deduplication

The new algorithm called anchor-driven subchunk deduplication addresses the serious shortcomings of the previous approaches described above.

In an overview, the deduplication with subchunks looks as follows. The base deduplication is done with a relatively large expected chunk size (for example, 64 KB), to ensure good performance and to keep the size of the global chunks database manageable. When a big chunk is found not to be a duplicate, we try to deduplicate its subchunks. They are defined within a given chunk by running the CDC algorithm with a lower expected block size. For example, the expected subchunk size can be 8 KB when the expected chunk size is 64 KB, resulting in 8 subchunks within 1 chunk on average.

Assume that when writing a backup stream, we can create a global database of all subchunk-to-container-chunk mappings. With this assumption, the chunker first produces large chunks, and the system checks for chunk duplicate by issuing query to the block store, as in the base system described above. For each subchunk of a new, non-duplicate chunk, we can next look for a relevant mapping in the subchunk database. If such mapping is found, the subchunk is deduplicated and the address from this mapping is used. The subchunks which are not deduplicated in this process can be coalesced and emitted as one large block, as illustrated in Figure 1.

This simple ideal method of deduplication with subchunks is not practical, as it requires fast access to a huge global database of subchunk-to-container-chunk mappings, which moreover needs synchronization with chunk and subchunk additions and deletions. To make deduplication with subchunks realistic, we introduce a notion of subchunk deduplication context. Compared to global subchunk database, this context keeps only subset of subchunk-to-container-chunk mappings limited to those which are most likely to be useful for deduplication of the currently written stream. The details how this context is constructed and maintained are given below.

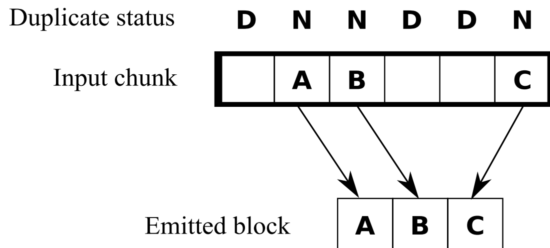

Figure 1: Coalescing of adjacent new subchunks.

2.4 Subchunk deduplication context

When deduplicating a backup stream with subchunks, we usually do not need access to all previously generated subchunks. Instead, most deduplication can be done with subchunks from the previous version of this backup which are "close" in the backup stream to the current position in the backup stream being written. Using this observation, we construct a subchunk deduplication context as a local RAM cache keeping subchunk-to-container-chunk mappings for such subchunks. The deduplication context is built on-the-fly, i.e. when writing a backup stream, with the help of the so-called anchors.

When writing a backup stream, we look for short sequences of 1 to 3 adjacent chunks for which a special anchor hash has some predefined number of trailing bits equal to 0. These sequences are called anchor sequences. The anchor hash for one-chunk anchor sequence is the hash of this chunk itself; for a multi-chunk anchor sequence, the anchor hash is defined as the hash of individual chunk hashes of each of anchor sequence chunks. Note that we control frequency of anchor sequences with the number of trailing bits in this special hash which must be 0; the fewer bits, the more frequent anchor sequences.

The continuous part of backup stream between anchor sequences is called an anchor window (see Figure 2). Its size is usually several tens of MBs and the entire backup stream is covered by disjoint anchor windows. For each anchor window, there is one associated mapping pack created, containing subchunk-to-container-chunk mappings for all subchunks in this window. Each mapping keeps hashes of subchunk and the chunk containing it, together with the position of the subchunk in the big chunk. These packs together approximate one big global subchunk-to-container-chunk mapping database; however it does not have to support fast single mapping retrieval, so it can be stored in the block store.

For each anchor sequence, we store an anchor which is a special type of block addressed with anchor hash and keeping fixed set of pointers [1]. These pointers point to mapping packs corresponding to windows ahead in the stream with respect to this anchor sequence, as illustrated in Figure 2.

When an anchor sequence and the part of the backup stream covered by mappings pointed by this anchor have not changed between backups, the new anchor is identical to the old one, and the new one will be deduplicated. If data in this part of the stream has changed, but anchor sequence blocks are not modified, the new anchor will overwrite the old one and the old one will be reclaimed. New mapping packs will be stored, and the old ones will be reclaimed if there is no more anchors pointing to them. Reclamation of old mapping packs does not affect readability of old backups containing subchunks described in such packs; it only disables deduplication against such subchunks. Additionally, we need a process of periodic sweep of all anchors (for example, once a week), to remove anchors which have not been used for any deduplication since the last sweep.

2.5 Deduplication with the context

During subsequent backups, when an anchor created previously is found, we retrieve pointers associated with it and load pointed mapping packs to update the deduplication context. In such way, we bring into the deduplication context mappings for data which is likely ahead of the current position in the stream, so there is a good chance that we can use these mappings for deduplication soon.

Note that mappings kept in mapping packs (and consequently in the deduplication context) are not synchronized

[1]Note that based on an anchor hash, we need to be able to retrieve these pointers, so we need an additional addressing mode beyond content-addressability. In HYDRAstor this is achieved with so called searchable blocks, see [14] for details.

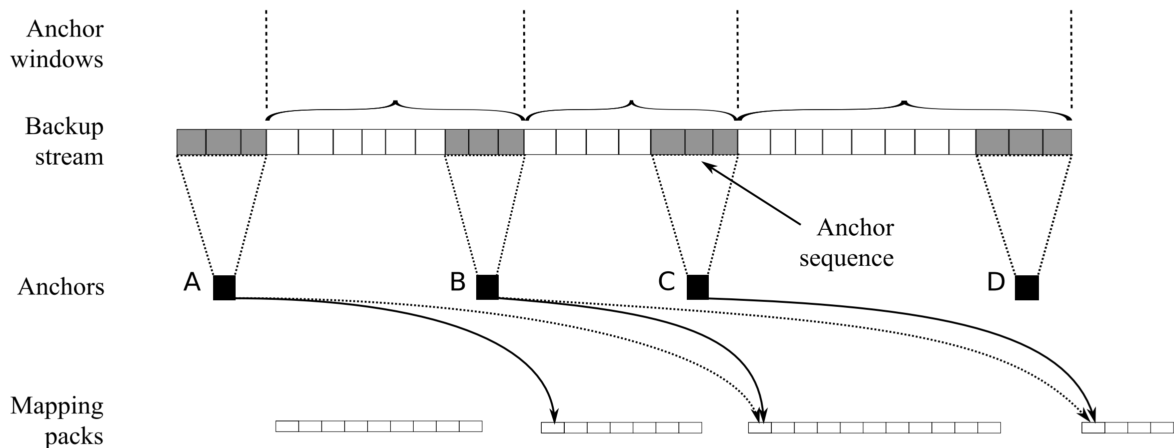

Figure 2: Anchors with associated entities. Anchor sequences are shaded.

with respect to chunk and subchunk deletions. This means, that when attempting subchunk deduplication we need to issue one more query to the block store to verify that the container chunk still exists there and contains the requested subchunk. This step is needed because mappings may be stale and garbage collection may remove unused subchunks. Only if the data to be deduplicated is present in the block store, the subchunk is deduplicated and the address from mapping is used.

An anchor does not have to point to mappings packs that directly follow its anchor sequence. Instead, an anchor can point to mappings packs well ahead in a backup stream (as in Figure 2). This enables fetching necessary mappings before writing and chunking data which can use them for deduplication.

Storing pointers to multiple mapping packs with each anchor has two major advantages. First, we can prefetch the deduplication context mappings covering longer region than just one anchor window. Second, since a new backup is modified version of the previous backup, individual anchor sequences can disappear in the new backup. Storing multiple pointers with a single anchor results in storing pointers to a given mapping pack in multiple anchors. This in turn reduces chances of missing useful mappings because of a removed anchor sequence.

To recap, the following actions happen while a backup is being written: (1) subchunk deduplication is performed using the deduplication context as described above; (2) anchors are generated for the current backup, and (3) the deduplication context is updated with detection of anchors created previously. Upon detection of an anchor sequence corresponding to an existing anchor, all packs pointed by this anchor are read into a local cache building deduplication context. Since we know which packs are already in the deduplication context, each pack is loaded only once. At the same time, a new value of the anchor is computed and eventually emitted replacing the old one.

2.6 Advantages of subchunk deduplication

In terms of storage overhead, all mapping packs together occupy space comparable to fingerdiff per-backup databases. The big difference is that mapping packs are small and can be combined to build deduplication context which fits in the main memory, whereas fingerdiff database for a huge backup is large and will not fit there. Additionally, for fingerdiff we need to know relations among backup streams and stream boundaries, whereas anchor-based approach does not require this knowledge.

Unlike in fingerdiff, the deduplication is not limited to one stream, because anchors are stored globally in the block store, so it is possible to identify similar parts of a backup stream coming from different backup servers or different clients (for example, operating system files in case of backup of workstations). In such case, subchunk deduplication works across unrelated backups.

Compared to bimodal, the new approach allows for more extensive searching for duplicate data. Unlike bimodal, the new algorithm checks for deduplication using subchunks encountered previously in both non-duplicated and duplicated chunks, as the subchunk deduplication context contains all these mappings. Bimodal checks only for subchunks of non-duplicate chunks adjacent to at least one duplicate chunk in the stream being written.

The metadata of a block with subchunks delineated is much smaller than when each subchunk is stored as a separate block because subchunks share a lot of block metadata, for example encryption status and most of the address of data location on disk.

Additionally, new metadata containing mappings from subchunks to container chunks do not need to have as high resiliency as block metadata. It is sufficient that these mappings are kept with resiliency equal to default resiliency of user data. If these mappings are lost, the worst case would be decreased deduplication effectiveness, but on a very limited scale, as certainly the level of user data resiliency should not result in likely massive failures.

Using a local context instead of global subchunk database reduces a chance of detecting subchunk duplicates if the same small piece of data exists in many independent backup streams. A good example of this case is a short mail stored in many mailboxes. However, for most backups this effect is not substantial, so the proposed subchunk deduplication should result in deduplication effectiveness close to this delivered by the CDC with the expected block size equal to

the subchunk size, but without associated, potentially very significant, metadata overhead. And even for the worst case of mail backups, the new algorithm behaves reasonably well, as shown by experiments in Section 3.

2.7 Details and refinements

After backup removal, it may happen that some subchunks are dead and need to be also removed, while others are alive and need to be preserved. For this reason, the garbage collection algorithm needs to be changed to allow identification of dead chunks and reclamation of their space. To facilitate subchunk-within-chunk location after space reclamation, we need to keep a bit vector with each block metadata indicating which of the original subchunks are still present. Moreover, each subchunk needs to get a small reference counter (a few bits) to allow subchunk reclamation. Such counter can be small, because in rare cases when it overflows, such subchunk will not be reclaimed until the entire block is reclaimed.

To enable subchunk-based deduplication, we extend the address format with a subchunk selector. There are multiple forms of subchunk selector possible. One is just a subchunk order number. For example, with 8 subchunks in 1 chunk on the average, it is enough to extend the address with 4 bits, allowing for addressing of the entire chunk and up to 15 individual subchunks.

One possible refinement is adjacent subchunk coalescing which is done when a large chunk cannot be deduplicated, but multiple adjacent subchunks can. This can be determined based solely on mappings, without additional queries to the block store. In such case we generate only one pointer to a range of subchunks. To allow this, the subchunk selector is changed into subchunk range selector which contains two subchunk numbers – in the example above, we would extend the address by 1 byte instead of 4 bits.

3. EVALUATION

In this section, we evaluate the proposed subchunk algorithm (further referred to as *subchunk*) against CDC and bimodal using three datasets described below.

3.1 Distributed system

Up till now, we have assumed a single node system, as this assumption allowed for simplified description of our algorithm. In reality, HYDRAstor is a scalable distributed system. Such system is more likely than a centralized one to store backups with wide variations of deduplication patterns, so there is a stronger motivation for not using one expected chunk size for all stored data.

The modeled system uses erasure codes for data resiliency. On writing, if a chunk being written is found not to be a duplicate, such new chunk is compressed and erasure-coded and the obtained fragments are stored on different block store servers. For user data, the system supports multiple resiliency levels, while keeping the overall number of fragments produced by erasure coding of each block constant and equal to 12. The default level introduces 33% overhead and allows to survive 3 node and disk failures.

A loss of one block with pointers may incapacitate multiple files or even file systems [11], as such blocks can be also deduplicated. Therefore, all system metadata are kept in multiple copies. The number of copies is equal to number of erasure code fragments, i.e. 12. Beyond making all metadata practically indestructible, such approach speeds up important system operations like rebuilding data resiliency and deletion, as multiple metadata copies are readily available for parallelization of background tasks even after many node failures.

3.2 Setup and metrics

Since direct operation on real backup data would have taken too long, we used a special chunking tool introduced in [15], to generate compressed backup traces. The tool dumps all potential cut points and hashes of data chunks between them. Such preprocessing greatly reduces the size of test data and the evaluation time.

We define *DER (duplication elimination ratio)* for a given dataset and a given algorithm as the total size of all data stored in the system divided by the total disk space needed. DER can be calculated without or with metadata included. We call the former the *data-only DER*, and the latter the *real DER*.

Data-only DER is simple and well defined, and, unlike real DER, it does not depend on the actual storage system characteristics. On the other hand, in real DER calculation all factors that can influence storage needed by a real system are taken into account. That includes data redundancy level, compression ratio and the space needed for all kinds of metadata. The value of real DER strongly depends on an actual system model. The metadata overhead of our model is described in Section 3.6

3.3 Test data

Our experiments are based on 3 datasets described in Table 1.

Name	Wiki	Netware	Mail
Number of backups	5	14	32
Avg. backup size	25 GB	78 GB	34 GB
Total size	125 GB	1086 GB	1087 GB

Table 1: Dataset characteristic

The *wiki* dataset consists of 5 official XML snapshots of English Wikipedia. Files contain only the newest versions of regular pages. Change history, special pages, images, etc. are not included. Snapshots were created quite rarely, on average once a month. Because of long periods between backups, low number of backups and the characteristic of wiki data (fine, frequent changes), deduplication ratio measured on this dataset is very low. Real DER varies from 1.07 to 1.50 depending on the algorithm chosen.

The next dataset, *netware*, represents typical backup data. This is the same dataset that was used in [15]. It consists of 14 full weekly Netware backups of user directories. Real DER measured on this dataset varies from 3.18 to 3.67 depending on the used algorithm.

The last dataset, *mail*, consists of 32 daily snapshots of mailboxes of 52 consultants at 9LivesData. This is a very specific dataset. A typical user mailbox does not change much daily, so deduplication measured on this dataset should be very high (close to the number of backups). However, a single snapshot of a user's mailbox is stored as a tar archive usually containing thousands of very small files, one message per file. Order of these files in a tar archive depends

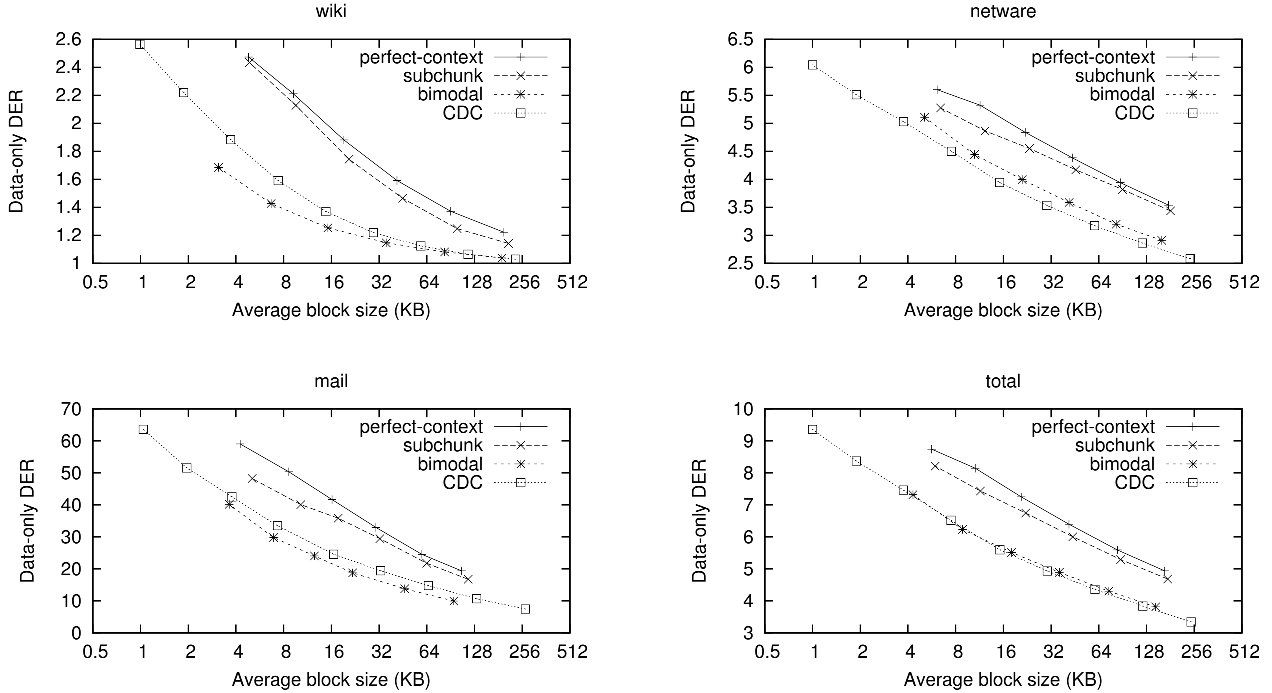

Figure 3: Data-only DER as a function of average block size

on tar and filesystem internals and changes slightly from time to time. If an expected chunk size spans over multiple files, even slight permutations in their order can adversely affect many duplicate elimination algorithms. Moreover, the changing order of data can potentially result in a very high data fragmentation. Real DER measured on this dataset varies from 13.14 to 18.45 depending on the algorithm.

An artificial dataset further referenced as *total* was built by concatenating all 3 datasets. Real DER for this dataset varies from 4.38 to 5.06.

3.4 Policies tested

All algorithms have been tested in many possible configurations.

The first algorithm, CDC, was tested with the expected chunk size set to all powers of 2 from 1 KB to 256 KB. In all experiments we are using CDC variant based on a combination of boxcar and CDC-32. Window size is equal to 64 bytes and the maximum chunk size is set to $\frac{3}{2}$ of the expected chunk size. The minimum chunk size is half of the expected chunk size – our CDC chunker immediately after cut point skips half of the expected chunk size and then looks for the next cut point on the level of half of the expected chunk size.

For bimodal, described in Section 2.2, we have used big chunks with expected chunk size varying from 8 KB up to 256 KB, and the expected subchunk size that is always 8 times smaller.

The proposed anchor-based deduplication was configured with the expected chunk size set to all powers of 2 from 8 KB to 256 KB, and the expected subchunk size always 8 times smaller. Anchor sequence length was set to 1 chunk. The average distance between anchor sequences was configured to 1024 chunks (on average 8192 subchunks). Deduplication context size was limited to $500,000$ entries (capable to store mappings from about 64 packs), and pointers to 16 mapping packs (covering on average 1 GB of real data stream) were stored with each anchor. Such configuration provides reasonable balance between the DER achieved by the algorithm and the resources needed to run it.

The algorithm labeled as *perfect-context* works similarly to the *subchunk* but, instead of using anchor-based mechanism, it puts all encountered mappings directly in the deduplication context, which in this case is unlimited. Such algorithm is impractical, but its deduplication is an upper bound for the subchunk approach.

3.5 Comparison using data-only DER

Figure 3 presents data-only DER as a function of an average block size. This is for all 3 datasets and for all algorithms. Each chart represents a different dataset, each curve – a different algorithm, and each point – a different configuration (different expected chunk size). The idea behind such presentation is that a desired duplicate elimination algorithm should not only achieve high DER, but also maintain high average block size, which is necessary to keep metadata overhead and performance overhead on acceptable levels. The average block size is defined as the total size of all unique blocks stored in the system divided by the number of such blocks (even if a single block appears in many backup streams, it is still counted as one). The average block size is slightly shorter than the expected chunk size (which is a configuration parameter), because our chunker cuts artificially potentially long blocks if they exceed $\frac{3}{2}$ of

the expected chunk size. Without such restriction, the average block size would be very close to the expected chunk size.

Naturally, using a smaller block size results in better deduplication if metadata is not included. In almost all cases, data-only DER is nearly linear in logarithm of an average block size.

Regardless of the dataset, the proposed algorithm performs better than CDC and the bimodal. While maintaining big average block size, it performs deduplication on a (much finer) subchunk level resulting in higher DER. In case of perfect-context algorithm this is cleanly visible. The deduplication achieved by CDC with X KB chunks is almost equal to the deduplication achieved by the perfect-context algorithm with X KB subchunks. Such relation is satisfied regardless of the expected chunk size used by the perfect-context algorithm, only the expected subchunk size matters. However, the perfect-context algorithm needs to maintain a huge index of all subchunks which requires a lot of resources and is not easily implementable.

DER achieved by the proposed subchunk algorithm depends mostly on the performance of the anchor-based mechanism used for prefetching mappings. The better the anchors work, the higher deduplication. As can be seen, the anchor-based subchunk deduplication is quite close to the perfect-context algorithm proving that anchor-based mechanism for prefetching mappings works reasonably well. We define the context hit ratio as the number of duplicated subchunks found by the anchor-based mechanism divided by the number of duplicated subchunks found by the perfect-context algorithm. Indeed, in the basic configuration (64 KB chunks and 8 KB subchunks) context hit ratio is quite high (81% for wiki dataset, 87% for netware, and 99% for mail).

As expected, bimodal performs better than CDC on the netware data. This is the dataset used in [15], and our results are consistent with theirs. Surprisingly, on other datasets DER achieved by bimodal is slightly worse than DER achieved by CDC. This can happen in case of many small changes in random places. Big chunk is re-chunked into small chunks only if it is a neighbor of a duplicated chunk. If a block is chunked in one backup and does not change in the next backup stream, such block has to be re-chunked every time in order not to lose some deduplication opportunity. Such situation does not happen very often in netware traces but quite often in wiki and mail resulting in slightly worse performance of bimodal.

3.6 Comparison using real DER

The picture looks significantly different when all kinds of metadata are included. In the system model described in Section 3.1, each block is stored as 12 fragments created with erasure coding. For most data, 9 are original fragments and 3 are redundant fragments, however blocks with pointers are kept in 12 copies. Compression level is globally estimated as a linear function of logarithm of a block size (about 0.77 for 64 KB blocks and about 0.80 for 8 KB blocks). Each block stored has 112 bytes of metadata associated with it. All metadata are stored in 12 copies which results in 1344 bytes of metadata overhead per block. In a real system, blocks with pointers can also be deduplicated resulting in less metadata overhead (especially for small blocks), but in the model, we emulate only the worst-case scenario where blocks with pointers are not deduplicated. Therefore, there must be a pointer for each chunk appearing in any backup stream (no matter if this chunk is a duplicate or not), and all pointers are also stored in 12 copies.

The subchunk algorithm needs extra space for storing mapping packs. A single mapping contains SHA-1 of a subchunk (20 bytes long), an index and SHA-1 of a whole chunk. Since usually a few consecutive subchunks belong to the same chunk, SHA-1 of a whole chunk can be stored once for a few subchunks. Thus, a single mapping can be estimated from above by 32 bytes. Mapping packs can be stored with low resiliency – in case of data loss, DER will slightly drop to the level of whole-chunk CDC, but the system will remain fully functional. Therefore, in our experiments we have assumed that the resiliency of mapping packs is the same as the resiliency of user data.

Figure 4 presents real DER as a function of an average block size stored in a system. When all kinds of metadata are included in DER calculations, lowering block size results in better deduplication only until some point, after which extra space needed for storing metadata exceeds space gained by better deduplication.

For each dataset and for each algorithm there is some optimal block size resulting in the highest possible real DER. With a given algorithm (and without lowering metadata overhead, which is usually not an easy task), we cannot get better DER. Of course, the optimal block size varies heavily depending on the chosen dataset. Using different expected chunk sizes for various datasets needs extra manual configuration and may break global deduplication – duplicates cannot be found between two streams chunked with different chunking policies.

For CDC, the optimal expected chunk size is equal to 8 KB for wiki (DER=1.20), 16 KB for netware (DER=3.18), and 32 KB for mail (DER=14.89). Average chunk size equal to 16 KB sounds as a reasonable global choice for all these datasets. On artificial total dataset it achieves DER=4.39.

Bimodal performs well on netware dataset – for 32 KB big chunk and 4 KB subchunk it achieves DER=3.38 (6% improvement over CDC). However, it performs rather poorly on wiki (DER=1.07 at 32 KB big chunks) and mail (DER= 13.14 at 64 KB big chunks). On the total dataset expected big chunk size equal to 32 KB is the best choice (DER=4.38).

The performance achieved by the proposed subchunk algorithm is significantly higher. For the wiki dataset, the optimal average block size is 16 KB resulting in DER=1.50 which is a 25% improvement over the optimal CDC. For the netware dataset, the optimal average block size is 32 KB resulting in DER=3.67 which is a 15% improvement over CDC. Also for the mail dataset the optimal subchunk algorithm uses 64 KB blocks and gives DER=18.45 which is a 24% improvement over the optimal CDC. For the total dataset, the subchunk algorithm performs best with the expected block size equal to 64 KB (DER=5.06, a 15% improvement over CDC). As chunk size becomes very small, the performance of the subchunk algorithm gets worse and close to CDC, or even below CDC on mail traces. This is because for small chunks (e.g. 8 KB) subchunk is very small (e.g. 1 KB) which leads to very short average block size and results in too much metadata.

3.7 Low metadata overhead

Although our research is focused on systems with high metadata overhead such as HYDRAstor, the proposed al-

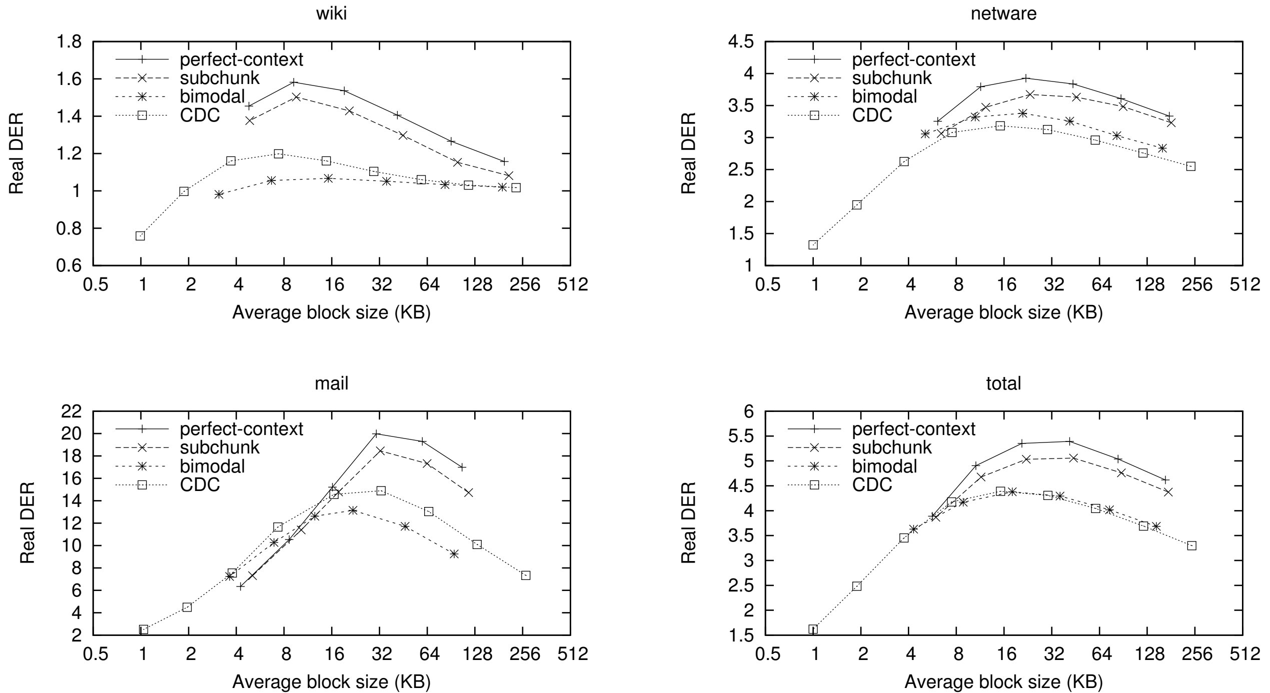

Figure 4: Real DER as a function of average block size

gorithm can be used also for systems with very different characteristics. We have tested the algorithm with a system model with minimal metadata overhead, which is probably the worst case for the algorithm. In this model all data and metadata is stored with only 33% overhead and the pointer size is reduced from 20 to 8 bytes.

Figure 5 presents the results. For wiki dataset the proposed algorithm is much better than CDC for all tested block sizes. For netware and mail datasets the proposed algorithm works better than CDC for block sizes equal to or higher than 16 KB. If we consider all datasets together (total), the new algorithm offers better dedup for block sizes of 8 KB and longer. However, the best dedup is provided by very short block sizes (1 KB and 2 KB). For those, we have not tested our algorithm, as the collected traces do not contain cut points for subchunks shorter than 1 KB. In general, for shorter blocks the extra overhead of storing mapping packs is higher than the benefits from performing the deduplication at the subblock level. We note here, that very short blocks may be optimal for dedup, but usually they are not used because they negatively impact the performance and require big indices for their location.

In the remainder of this paper we continue to focus our attention on systems with high overhead like HYDRAstor.

3.8 Reading performance

Deduplicating backup data may lead to backups which are no longer stored as contiguous streams but instead they consist of many small chunks of data scattered throughout the system. Such fragmentation results in more disk operations (seeks) needed during reads.

Highly scalable storage systems typically have a lot of resources and more disk seeking usually will not affect negatively the performance, especially if only one or just few streams are read at the same time. On the other hand, when designing a duplication algorithm, its impact on data fragmentation should be evaluated. Data fragmentation is a broad subject, mostly out of scope of this paper, but we present here results of a basic evaluation to show that the proposed algorithm does not result in a worse fragmentation than alternative solutions.

To evaluate disk reading performance, we have simplified model of a storage system assuming that all blocks are stored in a single contiguous file placed on a single physical disk. All new (not duplicated) blocks are written at the end of the file, while duplicated blocks are simply skipped. In this model we do not take into consideration storing any metadata. We simulate reading in the following way. Blocks are requested one by one in the order of appearing in the backup stream. When the block is being fetched from disk, we extend the read operation to read also a number of following blocks. The expected single read size is 256 KB. All blocks that fit in this limit are loaded in a read cache and the next block is loaded if and only if half of it fits in the limit (this, a bit strange, condition can avoid problem of *rounding down* the number of prefetched blocks). Size of the cache is limited to 1 GB. If a block is already present in the cache, we do not need an extra IO operation. We use the average number of IO operations necessary to read 1 MB of data from the last (the most fragmented) backup as a measure of fragmentation.

In case of the subchunk algorithm, the model is slightly different. We follow the same strategy, but instead of using big blocks, we operate on subchunks because the system is

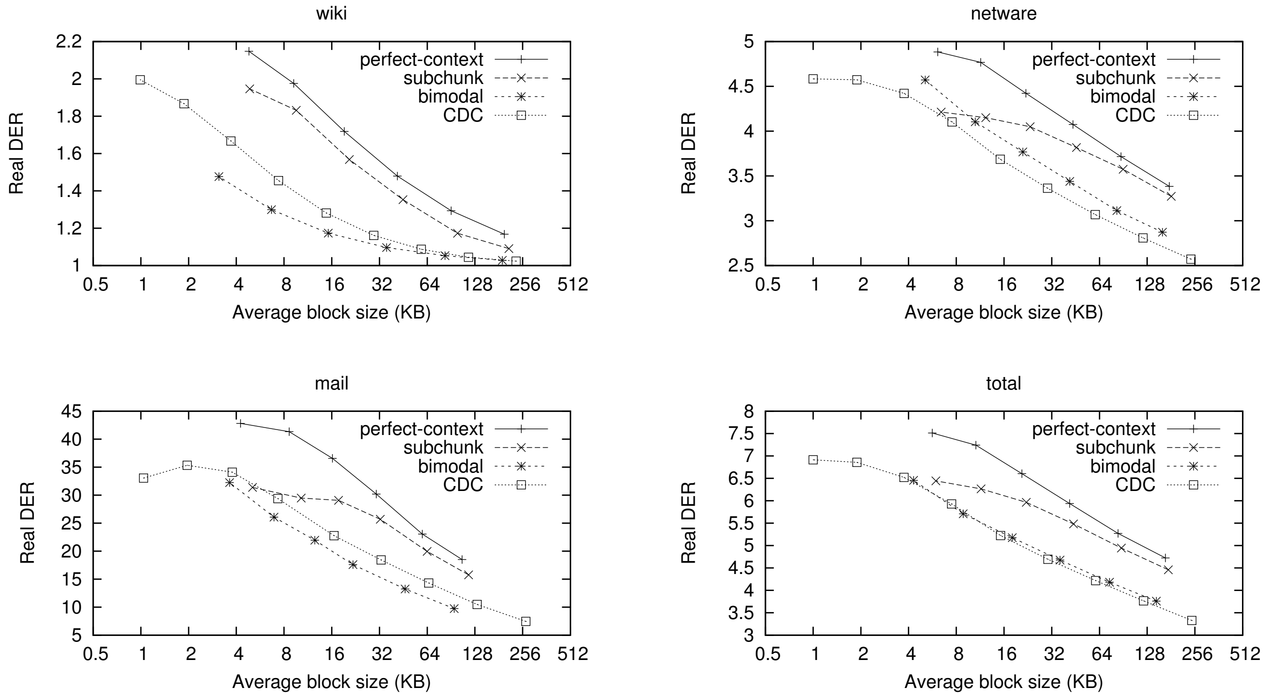

Figure 5: Real DER as a function of average block size on a system with low metadata overhead

capable of reading individual subchunks.

To compare all algorithms, for each of them we have chosen the best settings in terms of real DER (64 KB chunks and 8 KB subchunks for subchunk algorithm, 32 KB big chunks and 4 KB small chunks for bimodal and 16 KB chunks for CDC). Figure 6 presents real DER for each algorithm against the average number of disk operations needed to read 1 MB from the last (the most fragmented) backup.

The results strongly depend on the dataset. For netware, the subchunk algorithm results in the lowest fragmentation and the highest real DER, which is the best case. This is a good news as this trace should reflect a typical backup data. Generally, there should be a trade-off between deduplication and fragmentation. Better dedup should result in more fragmentation but as we can see there are exceptions.

For mail and wiki traces the subchunk algorithm fragmentation is always between the other two algorithms fragmentation, but the real DER is always the highest. For these traces we pay in fragmentation for what we get in the improved deduplication.

Another factor that can potentially affect the read performance is the average chunk size defined as the total size of all data streams stored in the system divided by the total number of pointers. In average chunk size calculation duplicates are counted several times, unlike in average block calculation. Shorter average chunk means that for the same amount of data more pointers have to be stored and processed.

Tables 2 and 3 show the average block size and the average chunk size respectively for the best instance of each algorithm, and additionally for bimodal with the 64 KB big chunk. This instance shows effectiveness of bimodal, as the 64 KB big chunk bimodal results in only slightly lower dedup ratio than the best bimodal, but delivers substantially larger average block and chunk sizes. However, the subchunk instance delivering the best dedup ratio still generates larger average blocks and chunks in almost all cases compared even to bimodal with 64 KB chunks.

	netware	mail	wiki	total
subchunk 64k / 8k	45.81	32.37	45.06	43.89
bimodal 64k / 8k	41.61	21.79	35.45	36.10
bimodal 32k / 4k	21.03	12.51	15.22	18.05
CDC 16k	15.15	16.54	14.78	15.20

Table 2: Average block size

	netware	mail	wiki	total
subchunk 64k / 8k	43.72	23.96	23.64	30.44
bimodal 64k / 8k	39.93	20.81	32.62	27.60
bimodal 32k / 4k	20.35	11.43	14.09	14.60
CDC 16k	14.62	15.62	14.50	15.07

Table 3: Average chunk size

The subchunk algorithm tries to emit big chunks by default and generates a block containing less data than a big chunk only after the remainder of such chunk has been deduplicated. Bimodal shares with the subchunk algorithm the first advantage but not the second, as bimodal emits small blocks on the border of change speculatively, in a hope that

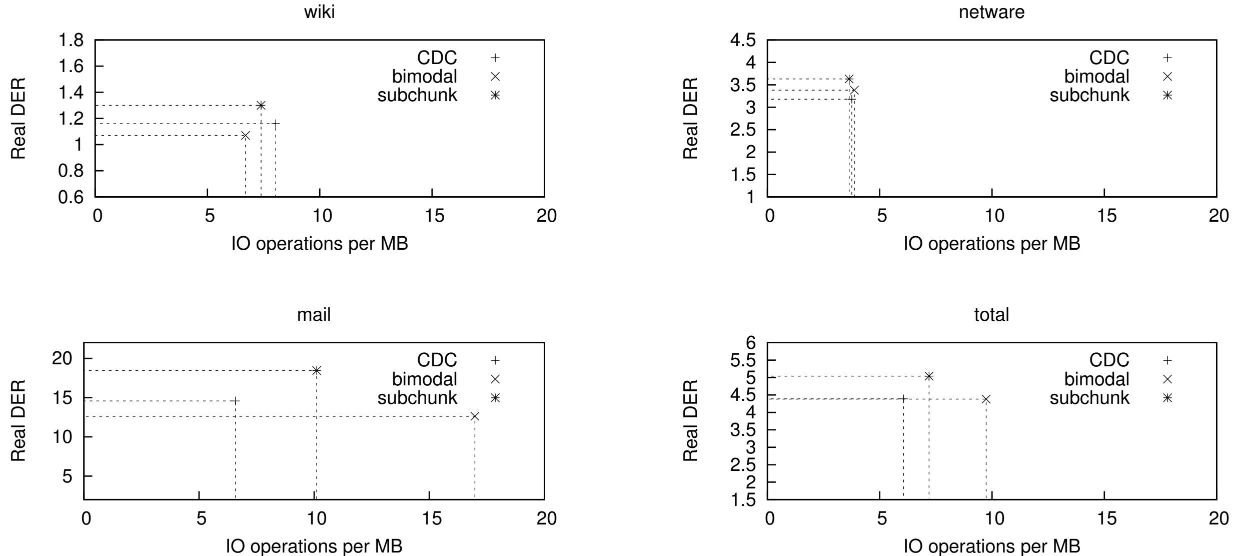

Figure 6: Real DER and the average number of disk operations needed to read 1 MB of data from last backup

they will be used for deduplication later. However, such expected savings sometimes do not materialize. As a result, subchunk algorithm generates a much larger average blocks, compared to those generated by CDC and bimodal. Additionally, because of pointers to subchunks, there is a much bigger difference between the average block size and the average chunk size in case of the subchunk algorithm, than for the other two algorithms.

3.9 Configuration tuning

In this section, we show how real DER is affected by changing a single parameter from the base configuration of the subchunk algorithm (described in Section 3.4 for the 64 KB expected chunk size). To simplify the plots, we present the results for the total dataset only. Usually, the results for individual datasets do not differ significantly. Figure 7 presents the results.

The first parameter tuned is the anchor sequence length – the number of consecutive chunks forming an anchor sequence. Surprisingly, the best results can be achieved for anchors sequences created from a single chunk, as real DER drops nearly linearly with the logarithm of the number of chunks used. Shorter anchor sequences are more likely to appear unchanged in subsequent backups, which is necessary to download the appropriate mapping packs.

The next parameter is the expected distance between consecutive anchor sequences. With changing this value, also the number of mapping packs pointed by each anchor has been adjusted to always point mappings corresponding to 1 GB of real data. Setting more frequent anchor sequences results in better deduplication, but, since each anchor sequence generates a read and write operations, too frequent anchor sequences can reduce performance. We decided to set the expected distance between consecutive anchor sequences to 1024 chunks (on average 64 MB of real data for 64 KB chunks).

We have also modified the number of mapping packs pointed by each anchor. Generally, increasing this value results in better deduplication. The sudden drop for the highest value is caused by pruning of the deduplication context. It works as a queue, and due to loading to many mappings, the mappings needed are evicted from the context before being used. Increasing the deduplication context size would help in such case. We have experimented with bigger deduplication contexts, but the results were not significantly better and we kept 1 GB prefetch as a reasonable compromise.

Finally, we have experimented with other expected subchunk sizes but our experiments have not shown any significant improvement.

Besides configuration tuning, we have also tested a few modifications to the algorithm. The first one is coalescing leftovers (blocks made of not duplicated subchunks). Instead of emitting a single leftover block for each chunk, we can join a few of them and emit a single block. Such modification almost does not affect DER – leftovers are quite rare and very unlikely to duplicate, but it allows us to maintain high average block size.

The other one is limiting the number of subchunks inside a single chunk. With default chunker configuration, the maximum number of subchunks is 24, while the expected number of them is 8. If our system requires constant size of metadata records, this can lead to waste of space – we need to reserve about 3 B for each subchunk no matter if it is present or not. However, we can coalesce all subchunks above some limit. Our experiments show that limiting number of subchunks to 12 affects deduplication only slightly (data-only DER drops from 6.01 to 5.95 on total dataset), while limiting to 16 does not affect it at all (data-only DER drops by less than 0.01).

4. RELATED WORK

Deduplication today is an active area of research with a fast path transfer to commercial products.

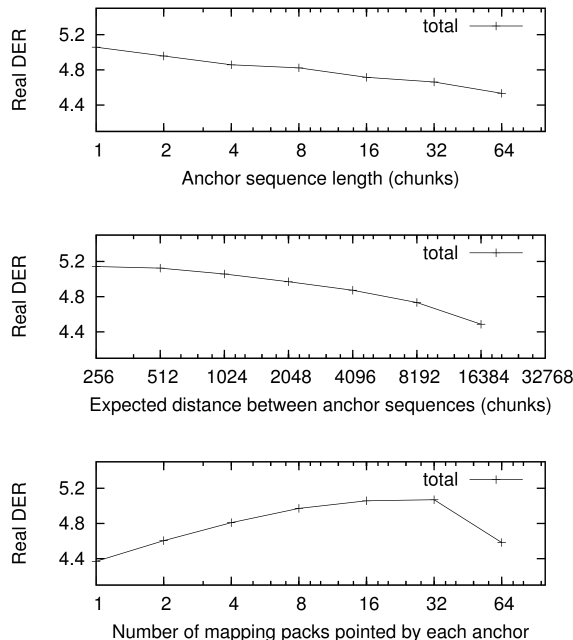

Figure 7: Real DER vs various parameters

One of the early systems with deduplication was Venti [27], which used fixed size CAS blocks, so it was not able to deduplicate shifted contents. CAS was also applied early to detect duplicates of entire objects. EMC Centera [5] is a CAS-based commercial product for archiving. When introduced, Centera computed the content address on whole files, so it was not able to perform sub-file deduplication.

A major improvement in deduplication was introduction of CDC with variable sized blocks. Initially, it was used in networked file systems to reduce bandwidth usage, most notably in LBFS [21] and others [31, 26, 6].

MAD2 [32] is an example of a recent research system employing CDC. Foundation [29] uses CDC to deduplicate versions of entire hard disks in nightly snapshots. CZIP [24] is a generic compression file format using CDC for general use, e.g. in content distribution servers, file transfer or web caching.

Besides exact matching of each chunk hash against a large index, deduplication can be also achieved with an approximate similarity detection. Pastiche [13] is a P2P backup system using CDC and content based encryption. The deduplication is done approximately across multiple nodes by finding a set of backup buddies defined as nodes keeping similar data. RedFS distributed filesystem [9] combines local deduplication for groups of files with finding similar file groups based on vectors of groups known to replica in order to minimize data transferred between replicas.

Extreme binning [10] is an interesting technique for finding duplicates across files using a one-per-file representative chunk defined as the chunk with the minimum hash. These chunks are used to distribute files in bins, and deduplication is limited to a given bin. Bin index is of limited size like our deduplication context. However, our context is stream-oriented as it is built with anchors encountered in the stream, whereas bin content is file-oriented. Moreover, extreme binning assumes knowledge of file boundaries, which the subchunk approach does not require.

Sparse index [18] also limits deduplication to a few similar multi-megabyte segments, and the similarity is determined with sampling. Unlike extreme binning, no knowledge of file boundaries is assumed. The subchunk approach builds deduplication context on-the-fly with anchors, instead of accumulating large segments of a stream being written for sampling as done by sparse indexing. Moreover, sparse indexing does not deduplicate with multiple levels of chunking. ProtecTier [7] is similar to sparse indexing in using large segments of potentially similar data for deduplication.

Hybrid solutions combining CDC with other techniques like delta compression [35] are also possible. Such systems include REBL [16] and DeepStore [33]. Various deduplication techniques are evaluated in [20, 25, 19]. Compared to CDC, delta encoding resulted in better dedup for fine-grained changes.

Today, there is a multitude of commercial solutions delivering deduplication mostly for backup and archiving markets. Some of these systems like HYDRAstor [14], EMC Avamar [4], Symantec Pure Disk [3], and EMC DataDomain [34] employ CDC and/or secure hashing to detect duplicates. Others like ExaGrid [1], IBM Protectier [23] and SEPATON VTL [2] deliver deduplication with similarity matching and delta-differential techniques.

The work described so far in this section concentrated on chunking with one expected chunk size. Besides fingerdiff [12] and bimodal chunking [15] already discussed in Section 2.2, there is little research on deduplication with multiple chunk sizes. One example is adaptive chunking [17] which advocates switching between CDC and fixed size chunking to minimize necessary processing power on mobile devices.

Our anchor blocks are in a way similar to per-file representative chunks of extreme binning [10] and representative blocks of the Spring network protocol [30] as all are used to limit data against which deduplication is done. The idea of coalescing content pointers has been suggested in a partially content-shared filesystem [8].

Impact of chunk size on deduplication has been evaluated in [22]. The optimal chunk size was found to be small – equal to 1 KB, but the metadata overhead in this work is also small and dominated by the hash size. A larger block size – 8 KB – is used in DataDomain [34]. In our case, even larger blocks are preferred because of bigger metadata overhead, which is used to ensure multiple node failure resiliency of system metadata.

5. CONCLUSIONS AND FUTURE WORK

In this work, we have proposed the subchunk deduplication algorithm which is driven by a dynamically prefetched subchunk deduplication context of a limited size. This context provides most of the mappings needed for an effective deduplication on the subchunk level. Moreover, the context input mappings can be stored as not-so-important metadata, i.e. with low resiliency overhead. The new algorithm additionally reduces effective metadata overhead by using whole chunks when possible and sharing most of metadata among subchunks belonging to the same chunk. As a result, for systems with significant metadata overhead, the subchunk algorithm results in a superior real DER compared to other

approaches like CDC and bimodal while delivering significantly bigger average block and chunk sizes. At the same time, for standard backup traces, the new algorithm results in less fragmentation. For other data streams the fragmentation may be higher, but this is a cost of improved deduplication.

For future work, we plan to evaluate the new algorithm using more backup traces and to study in detail how to address the fragmentation problem without reducing significantly the deduplication ratio.

6. REFERENCES

[1] ExaGrid. http://www.exagrid.com.

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

[3] Symantec NetBackup PureDisk. http://www.symantec.com/business/netbackup-puredisk.

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

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

[6] S. Annapureddy and M. J. Freedman. Shark: Scaling file servers via cooperative caching. In *In Proc NSDI*, 2005.

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

[8] J. Barreto and P. Ferreira. Efficient file storage using content-based indexing. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 1–9, New York, NY, USA, 2005. ACM.

[9] J. Barreto and P. Ferreira. Efficient locally trackable deduplication in replicated systems. In *Middleware'09: Proceedings of the ACM/IFIP/USENIX 10th international conference on Middleware*, pages 103–122, Berlin, Heidelberg, 2009. Springer-Verlag.

[10] D. Bhagwat, K. Eshghi, D. D. E. Long, and M. Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. Sept. 2009.

[11] D. Bhagwat, K. Pollack, D. D. E. Long, T. Schwarz, E. L. Miller, and J.-F. Paris. Providing high reliability in a minimum redundancy archival storage system. In *MASCOTS '06: Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation*, pages 413–421, Washington, DC, USA, 2006. IEEE Computer Society.

[12] D. R. Bobbarjung, S. Jagannathan, and C. Dubnicki. Improving duplicate elimination in storage systems. *Trans. Storage*, 2(4):424–448, 2006.

[13] 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.

[14] 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: Proccedings of the 7th conference on File and storage technologies*, pages 197–210, Berkeley, CA, USA, 2009. USENIX Association.

[15] E. Kruus, C. Ungureanu, and C. Dubnicki. Bimodal content defined chunking for backup streams. In *FAST*, pages 239–252, 2010.

[16] P. Kulkarni, F. Douglis, J. LaVoie, and J. M. Tracey. Redundancy elimination within large collections of files. In *ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 5–5, Berkeley, CA, USA, 2004. USENIX Association.

[17] W. Lee and C. Park. An adaptive chunking method for personal data backup and sharing, February 2010. 8th USENIX Conference on File and Storage Technologies (FAST âĂŹ10) poster session.

[18] 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*, pages 111–123, 2009.

[19] N. Mandagere, P. Zhou, M. A. Smith, and S. Uttamchandani. Demystifying data deduplication. In *Companion '08: Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion*, pages 12–17, New York, NY, USA, 2008. ACM.

[20] D. Meister and A. Brinkmann. Multi-level comparison of data deduplication in a backup scenario. In *SYSTOR '09: Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, pages 1–12, New York, NY, USA, 2009. ACM.

[21] 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.

[22] 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.

[23] A. Osuna, R. Pflieger, L. Weinert, X. X. Yan, and E. Zwemmer. *IBM System Storage TS7650 and TS7650G with ProtecTIER* . IBM Redbooks, 2010.

[24] K. Park, S. Ihm, M. Bowman, and V. S. Pai. Supporting practical content-addressable caching with czip compression. In *ATC'07: 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2007. USENIX Association.

[25] C. Policroniades and I. Pratt. Alternatives for detecting redundancy in storage systems data. In *ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 6–6, Berkeley, CA, USA, 2004. USENIX Association.

[26] D. R. K. Ports, A. T. Clements, and E. D. Demaine. Persifs: a versioned file system with an efficient representation. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems*

principles, pages 1–2, New York, NY, USA, 2005. ACM.

[27] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *First USENIX conference on File and Storage Technologies*, pages 89–101, Monterey, CA, 2002. USENIX Association.

[28] M. Rabin. Fingerprinting by random polynomials. *Tech. Rep. TR-15-81*, 1981.

[29] S. Rhea, R. Cox, and A. Pesterev. Fast, inexpensive content-addressed storage in foundation. In *Proceedings of the 2008 USENIX Annual Technical Conference*, pages 143–156, Berkeley, CA, USA, 2008. USENIX Association.

[30] N. T. Spring and D. Wetherall. A protocol-independent technique for eliminating redundant network traffic. *SIGCOMM Comput. Commun. Rev.*, 30(4):87–95, 2000.

[31] N. Tolia, M. Kozuch, M. Satyanarayanan, B. Karp, T. Bressoud, and A. Perrig. Opportunistic use of content addressable storage for distributed file systems. In *IN PROCEEDINGS OF THE 2003 USENIX ANNUAL TECHNICAL CONFERENCE*, pages 127–140, 2003.

[32] J. Wei, H. Jiang, K. Zhou, and D. Feng. Mad2: A scalable high-throughput exact deduplication approach for network backup services. In *Proceedings of the 26th IEEE Symposium on Massive Storage Systems and Technologies (MSST)*, May 2010.

[33] L. L. You, K. T. Pollack, and D. D. E. Long. Deep store: An archival storage system architecture. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*, pages 804–8015, Washington, DC, USA, 2005. IEEE Computer Society.

[34] 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.

[35] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. In *IEEE Trans. Inform. Theory*, 1977.