Bimodal Content Defined Chunking for Backup Streams

Erik Kruus
NEC Laboratories America
kruus@nec-labs.com

Cristian Ungureanu
NEC Laboratories America
cristian@nec-labs.com

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

Abstract

Data deduplication has become a popular technology for reducing the amount of storage space necessary for backup and archival data. Content defined chunking (CDC) techniques are well established methods of separating a data stream into variable-size chunks such that duplicate content has a good chance of being discovered irrespective of its position in the data stream. Requirements for CDC include fast and scalable operation, as well as achieving good duplicate elimination. While the latter can be achieved by using chunks of small average size, this also increases the amount of metadata necessary to store the relatively more numerous chunks, and impacts negatively the system's performance. We propose a new approach that achieves comparable duplicate elimination while using chunks of larger average size. It involves using two chunk size targets, and mechanisms that dynamically switch between the two based on querying data already stored; we use small chunks in limited regions of transition from duplicate to non-duplicate data, and elsewhere we use large chunks. The algorithms rely on the block store's ability to quickly deliver a high-quality reply to *existence queries* for already-stored blocks. A chunking decision is made with limited lookahead and number of queries. We present results of running these algorithms on actual backup data, as well as four sets of source code archives. Our algorithms typically achieve similar duplicate elimination to standard algorithms while using chunks 2–4 times as large. Such approaches may be particularly interesting to distributed storage systems that use redundancy techniques (such as error-correcting codes) requiring multiple chunk fragments, for which metadata overheads per stored chunk are high. We find that algorithm variants with more flexibility in location and size of chunks yield better duplicate elimination, at a cost of a higher number of existence queries.

1 Introduction

Duplicate elimination (DE) is a means to save storage space. CDC techniques [25, 27, 24, 15, 3, 5] are well-established methods that use a local window (typically 12–48 bytes long) into data to reproducibly separate the data stream into variable-size chunks that have good duplicate elimination properties. Such chunking is probabilistic in the sense that one has some control over the average output chunk size given random data input. A "baseline" CDC algorithm has as primary parameters a single set of minimum, average and maximum chunk lengths, and it generates chunks of the desired size range by inspecting only the input stream. A baseline algorithm may also have less influential parameters, such as a backup cut-point policy to deal with the situations when the maximum chunk size has been reached without encountering a good cut point. In typical DE methods, one simply breaks apart an input data stream reproducibly, and then emits (stores, or transmits) only one copy of any chunks that are identical to a previously emitted chunk.

As the average chunk size of such baseline CDC schemes is reduced, the efficiency of deduplication increases. CDC schemes with average chunk sizes of around 8k have been used [25] and shown to result in reasonable deduplication. However, in storage systems, smaller chunk sizes come with costs:

- higher metadata overheads, as each chunk needs to be indexed;
- higher processing cost, which is proportional to the number of data packets processed;
- and lower compression ratio for each chunk, as compression algorithms tend to perform better on larger input.

For distributed deduplicating storage systems using error correcting codes (ECC) capable of protecting against

disk *and* node failure [12], these drawbacks are significant. Metadata needs to be associated with each ECC component of a chunk, and the indexing information used to find a block given a content hash needs to be stored redundantly; this results in higher per chunk overhead than other systems. Additionally, network costs increase as more chunks are processed. Thus, it is desirable to produce large chunks without unduly lowering the duplicate elimination ratio (DER), which we define as the ratio of the size of input data to the size of stored chunks. Note that the DER as defined takes into account both deduplication among chunks and individual chunk compression, but excludes metadata storage costs. The effect of the metadata costs can be trivially calculated; for a given metadata overhead $f \equiv {}^{\text{metadata size}}/_{\text{average chunk size}}$, the DER is reduced to ${}^{\text{DER}}/_{(1+f)}$.

In order to achieve our goal, we exploited the nature of the data stream composition produced by repeated backups. Policroniades et al. [26] noted that on real filesystems most file accesses are read-only, files tend to be either read-mostly or write-mostly, and that a small set of files generates most block overwrites. During repeated backups, entire files may be duplicated, and even when changed, the changes may be localized to a relatively small edit region. Here, a deduplication scheme must deal effectively with long repeated data segments, where our assumption for fresh data is that it have a high likelihood of reoccurring in a future backup run. The nature of the backup data led us to propose the following two principles governing possible CDC improvements for such streams:

P1. Long stretches of unseen data should be assumed to be good candidates for appearing later on (i.e. at the next backup run).

P2. Inefficiency around "change regions" straddling boundaries between duplicate and unseen data can be minimized by using shorter chunks.

In this paper, we propose algorithms that perform better than baseline algorithms under the assumption that P1 and P2 hold, and the system provides an efficient *existence query operation* that allows one to check whether a tentative chunk has been encountered in the past. By a "better" duplicate elimination algorithm, we mean one that produces a larger average chunk size than a baseline CDC algorithm while obtaining comparable DER.

P1 is justified by the fact that the amount of data modified between two backups is a small percentage of the total, and is concentrated in relatively few regions of change. P1 may in fact not be justified for systems with a high rollover of content. P1 implies that an algorithm should produce chunks of large average size when in an extended region of previously unseen data. The data is in a change region if in some vicinity of it there exist both chunks that were encountered in the past, and chunks that were not. Variations in vicinity sizes, and in how small the unseen data in a change region is chunked lead to different variants of the bimodal algorithms. Note that P2 is somewhat counter-intuitive, since it involves speculatively injecting undesirable small chunks into the storage system while providing no guarantee of an eventual storage payoff. Nevertheless, we present real-world evidence that this strategy may benefit scenarios storing many versions of an evolving data set.

Note that our bimodal chunking algorithms avoid problems with historical approaches that use resemblance detection [10, 11, 6, 4] or storage of sub-chunk information [5], whose implementations can suffer from slow speed and/or large amounts of metadata. We assume that the existence queries can be answered accurately, but discuss in Section 3.3 the effect of false positives (as could arise from the use of Bloom filters). Recently, a promising approach for efficient deduplication has been described [4] in which first a similar set of already stored chunks can be quickly selected, and then deduplication is performed within that localized environment. From the point of view of the entire system, this amounts to having a small rate of false negatives: chunks that already exist may be stored again. However, their results show that in practice the effect of these false negatives is minimal, and that they retain sufficient stream locality for good deduplication. We expect that our bimodal algorithms would also perform well in their setting, since both the fast querying algorithm and our bimodal chunking algorithms are exploiting assumptions about stream locality.

The paper is structured as follows. In Section 2 we describe baseline CDC algorithms and introduce two types of bimodal chunking improvements: splitting-apart and amalgamation algorithms. In Section 3 we begin by describing our data sets and testing tools, after which we present the results of applying the algorithms and interpret the results. We establish a performance limit for bimodal algorithms as well as briefly discussing engineering aspects. We also show that our assumptions P1 and P2 do not quite hold for our data set, yet the algorithms produced chunk sizes 2–4 times larger than those produced by a baseline algorithm with a comparable DER. Section 4 contains related work and Section 5 presents conclusions and future work.

2 Method

2.1 Using chunk existence information

Two approaches exist. In one, a breaking-apart algorithm first chunks everything with large chunks, identi-

fies change regions of new content, and then re-chunks data near boundaries of this change region at a finer level. In such an approach, a small insertion/modification of an input stream likely renders an entire large chunk non-duplicate. Were this large chunk re-chunked smaller, later occurrences of a short region of repeated change could be more efficiently bracketed.

In a slightly more flexible approach, a building-up algorithm can initially chunk at a fine level, and combine small chunks into larger ones. A building-up chunking algorithm can query for candidate big chunks at more positions, and more finely bracket such a single inserted/modified chunk. In both cases, at any point in the input stream, a decision must be made whether to emit a small chunk or a big chunk, so we refer to these algorithms as *bimodal* chunking algorithms, as opposed to the (unimodal) baseline CDC approaches.

In either approach, it is always advantageous to emit an already existing big chunk. If several big chunk emissions are possible, we emit the first-most one. Small chunks are then emitted only for non-duplicate big chunks near (adjacent to, in measurements below) duplicate big chunks. Note that in both schemes, some data may be stored in both small- and large-chunk format. In principle, this loss may be mitigated by rewriting such large chunks as two (or more) smaller chunks. However, for systems with in-line deduplication, rewriting an already emitted big chunk as two or more chunks may be impractical, so we will not consider chunk-rewriting approaches. Nevertheless, this might be possible to implement as a postprocessing step.

We target global duplicate elimination and assume that the block store can be efficiently queried for existence of chunks given a chunk content hash. Our algorithms operate in constant time per unit input, regardless of the number of stored chunks, since they require only a bounded number of chunk existence queries per chunking decision. Implementations of bimodal chunking can vary in the number and type of existence queries required before making a chunking decision. In general, we will find that the more flexibility one has in bracketing change regions and in what boundaries are allowed for large chunks, the better one's performance can be in terms of increasing chunk size.

Note that our approach does not require storing information about finer-grained blocks (e.g. non-emitted small chunks), and thus works well with any block store capable of answering whether a chunk with a given hashkey has already been stored or not. More complicated schemes, in which sub-block information is used, are possible (e.g. *fingerdiff* [5]), but the higher amount of metadata required likely leads to a higher cost of queries and makes more difficult the task of dealing with query latencies, impacting system performance

The heuristics behind our algorithms can be expected to perform well only if the backup stream has properties in line with P1 and P2. Indeed, without a similar-chunk lookup and an indirect addressing method, the first time a largely unmodified big chunk is re-chunked as small chunks, one pays the price of speculatively storing many small chunks that have no guarantee of ever being encountered again. If the small chunks re-occur sufficiently frequently in later backups (i.e. a finer grained delimiting of the duplication range), we can more than recoup the initial loss. In Section 3 we show that although P1 and P2 don't quite hold for our data set, the algorithms worked well, resulting in an average chunk size 2–4 times higher than baseline CDC for comparable DER.

2.2 Baseline rolling window cut-point selection.

Content-defined chunking works by selecting a set of locations, called *cut-points*, to break apart an input stream, where the chunking decision is based on the contents of the data itself. Typically this involves evaluating a bit scrambling function (say, a CRC) on a fixed-size sliding window into the data stream. The result of the function is compared at some number ℓ of bit locations with a predefined value, and if equivalent the last byte of the window is considered a cut-point. This generates an average chunk size of 2^ℓ, following a geometric distribution. For terseness, we will refer to such a chunker as a level-2^ℓ chunker. The probability of identifying a unique cut-point is maximized when the region searched is of size 2^ℓ.

Backup cut-points

For minimum chunk size m, the nominal average chunk size is $m+2^\ell$. For a maximum chunk size M, a plain level-2^ℓ chunker (i.e. chunking algorithm) will hit the maximum with probability approximately $e^{-(M-m)/2^\ell}$, which can be quite frequent. Since chunking at M is no longer content-defined, the deduplication of two similar streams is commonly improved by avoiding this situation. We have adopted a simple approach of choosing a best content-defined "backup" cut-point, chunked at a level $2^{\ell-b}$, to decrease the use of these non content-defined cut-points. The data we present here has used a policy of taking the longest backup cut-point from the highest of $b=$2–3 backup levels; otherwise, we emit a non-content-defined chunk of maximal length. In practice, if one adopts the earliest backup cut-point, other parameters can be varied to increase the average chunk size again. This may result in a small performance improvement. More sophisticated approaches to dealing with chunks of maximum size are also possible [15].

```
1   for (each big chunk) {
2     if (isBigDup)
3       {emit as big; isPrevBigDup=true}
4     else if (isPrevBigDup || isNextBigDup)
5       {rechunk as smalls; isPrevBigDup=false}
6     else {emit as big; isPrevBigDup=true}
7   }
```

Figure 1: A simple breaking-apart algorithm.

2.3 Breaking-apart algorithms

An example of a simple breaking-apart algorithm that re-chunks a nonduplicate big chunk either before or after a duplicate big chunk is detected is shown in Figure 1.

Here the primary pass over the data is done with a large average chunk size, emitting big duplicates in line 2–3. Otherwise, in lines 4–5, a single nonduplicate data chunk after or before a duplicate big chunk is re-chunked at smaller average block size and emitted. Remaining chunks are emitted as big chunks in line 6. One can modify such an algorithm to detect more complicated definitions of duplicate/nonduplicate transitions; e.g., when N non-duplicates are adjacent to D duplicates, re-chunk R big chunks with smaller average size. Here we present results for $N = R = D = 1$, as in Fig. 1. When we varied R we found that similar results for average chunk size and DER could be obtained by simply varying the chunking parameters $\{m, 2^{\ell}, M\}$ of the baseline algorithm instead. Alternatively, one could work with the byte lengths of the chunks to limit the nonduplicate region in which small chunks are emitted adjacent to a nonduplicate/duplicate transition point.

A lookahead buffer is used to support the isNextBigDup predicate. Querying work is bounded by one query per large chunk. This is the fastest of the proposed algorithms. In Fig. 2 we illustrate the operation on a simple example input 2(a). Big chunks (b) are queried for existence (c) and we assume duplicate and non-duplicate tags are assigned as shown. All duplicate big chunks should be stored. Of the remaining chunks, the transition regions (d) are re-chunked at smaller average chunk size. The remaining non-duplicate chunks are re-emitted as big chunks (e). In the final (f) bimodal chunking, chunks 2–6 and 9–11 are of small length. Of these, note that with respect to the byte-level duplication boundaries of the input stream (a), small chunks 2, 3 and 11 are entirely within the duplicate bytes area, and may possess enhanced probabilities of recurring later. In essence, the small transition region chunks can allow the extent of duplicate bytes to be more faithfully represented.

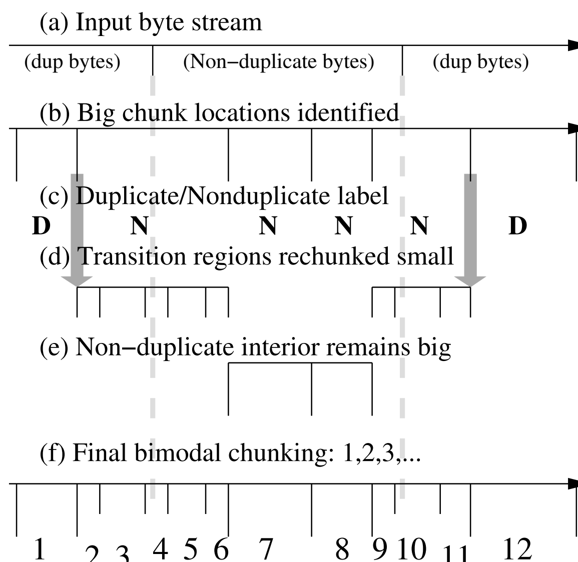


Figure 2: Breaking-apart algorithm steps.

2.4 Chunk amalgamation algorithms

Considerably more flexibility in generating variably-sized chunks is afforded by running a smaller chunker first, followed by chunk amalgamation into big chunks. Consider a simple case where big chunks are only generated by concatenation of a fixed number k of small chunks (Figure 3.) We will call these "fixed-size" big chunks because they are formed from a constant number of variably-sized small chunks during the initial forward search for big duplicates (lines 3–6). Their length in bytes is variable and their chunk endpoints are content-defined. We will call the above algorithms with fixed-size big chunks "k-fixed" algorithms. When the forward search for duplicates fails, lines 7–8 emit k chunks following a duplicate as small chunks when following a duplication region. Otherwise, those k chunks are amalgamated and emitted as a single big chunk in line 9.

A simple extension modifies lines 3–6 to allow variably-sized big chunks (1–k or 2–k small chunks) to be queried at every possible small chunk position during this decision-making process. We will label such extensions as "k-var" algorithms. With fixed-size big chunks we make at most 1 query per small chunk, while for variable-size big chunks we can make up to $k-1$ (or k) queries per small chunk.

To limit the possibility for two duplicate input streams to remain out-of-synch for extended periods, it is possible to introduce *resynchronization cut-points*: whenever the cut-point level of a small chunk exceeds some threshold (r higher than the normal chunking threshold ℓ), a big chunk can terminate there, but may never contain the resynchronization point in its interior. In this

```
1   void process( small chunks buf[0 to 2k−1] ){
2     for( pos=0; pos<=k; ++pos ) { //fwd search
3       if isBigDup(buf[pos to pos+k−1]) {
4         emit any smalls buf[0] to buf[pos−1]
5         emit big @ buf[pos to pos+k−1]
6         isPrevDupBig=true; return }
7     if( isPrevDupBig ) { emit k smalls
8       isPrevDupBig=false; return }
9     emit big @ buf[0 to k−1]; isPrevDupBig=true
10  }
```

Figure 3: A simple chunk amalgamation algorithm, in which k contiguous small chunks constitute a big chunk. Big duplicate chunks are always desirable (lines 2–6). Small chunks can only be emitted either in line 4, upon detecting an ensuing transition to duplicate data, or in line 7 when exiting a region of duplicate data. Regions considered fresh data (line 9) are emitted as big chunks.

fashion, two duplicate input streams can be forcibly re-synched after a resynchronization cut-point in algorithms that do not have sufficient lookahead to do so spontaneously. This mechanism can protect against certain malicious inputs, but will lower the average chunk size. A second means to favor spontaneous resynchronization is to use a hierarchy of backup cut-points (parameter b of Section 2.2).

In our test code, we also allowed some algorithms of theoretical interest. We maintained Bloom filters for many different types of chunk emission separately: small chunks and big chunks, both emitted and non-emitted. One benefit (for example) is to allow the concept of 'duplicate' data region to include both previously emitted small chunks as well as non-emitted small chunks (that were emitted as part of some previous big chunk emission). An algorithm modified to query non-emitted small chunks (i.e. the small chunks that were not emitted because they were part of some big chunk) can detect duplicate data at a more fine-grained level, at the cost of additional storage for such sub-chunk metadata. Nevertheless, when resources are more plentiful, implementations such as *fingerdiff* adopt such an approach and obtain substantial compression improvements [5].

Figure 3 shows the algorithm as applied in this paper. The length of the lookahead buffer is of minimal size and gives the behavior that transition regions are never covered by more than k small chunks. It is also quite reasonable to extend the lookahead to $3k-1$ chunks, and allow up to $2k-1$ small chunks to precede an upcoming duplicate big chunk, as depicted in Fig. 4

The logic of breaking apart and amalgamation algorithms (Figs. 2 and 4) is highly similar. For amalgamation input 4(a), small chunks (b) are used to form big chunks that are defined here as exactly 3 consecutive

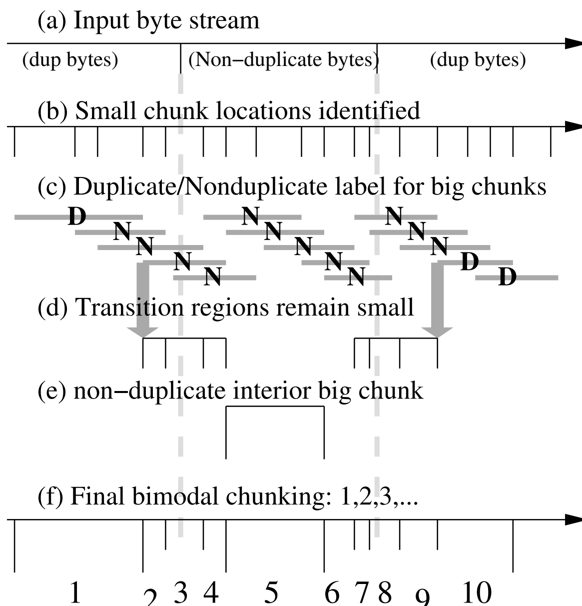


Figure 4: "k-fixed" amalgamation algorithm steps. We assume fixed-size big chunks are constituted of precisely three small chunks in this example.

small chunks. Big chunks are queried in 2/4(c) and first-most-occurring duplicate big chunks are emitted. Of the remaining chunks, transition regions 2/4(d) are emitted as small chunks. The remaining non-duplicate interior chunks are re-emitted as a series of big chunks inasmuch as possible 2/4(e), with one straggling small chunk left over at the end in 4(e). The final chunk emission 4(f) has small chunks 2–4 and 6–9. With the byte-level duplication points as in 4(a), small chunks 2 and 9 lie entirely within the span of duplicate bytes, and may have enhanced potential for deduplication.

Querying work is larger for amalgamation algorithms than for breaking-apart. Breaking apart uses one query per big chunk, whereas k-fixed amalgamation uses up to k queries per big chunk (one per small), and k-var amalgamation for big chunks consisting of 2–k small chunks uses up to $k(k-1)$ queries per big chunk. The increased number of existence queries for k-var amalgamation may be unattractive for practical implementations.

3 Results and Discussion

3.1 Test data

We used a data set for testing consisting of 1.16 Terabyte of full Netware backups of hundreds of user directories over a 4 month period. For privacy reasons, we had no idea what the distribution of file types was, only that it was a large set of real data, typical of what might be seen

in practice. Some experiments were also conducted using an additional 400 GB of incremental backups during this same period, but the results reported here include only the data from the full backups.

In order to study the behavior of the algorithms on data sets with characteristics different from our 1.16 TB data, we also analyzed data sets similar to those of Bobbarjung et al. [5], consisting of tar files for consecutive releases of several large projects. Their work targeted improvements for very small chunk sizes (< 1KB), while we target large chunk sizes.

3.2 Simulation tools

We have developed a number of tools for offline, anonymized, analysis of very large customer data sets. The key idea was to generate a binary "summary" of the input data, storing fine-grained information about potential chunk-points that could later be reused to generate any coarser-grained re-chunking. For every small chunk generated with expected size 512 bytes, we stored the SHA-1 hash of the chunk, as well as the chunk size and actual cut-point level ℓ (# of terminal zeroes in the rolling window hash). The summary data was obtained by running with minimum chunk size 1 byte and maximum chunk size 100k, with expected chunk size 512 bytes. This chunk data was sufficient to re-chunk our input data sets. Data sets that generate no chunk-points at all (e.g. all-zero inputs) are better handled by reducing the maximum chunk size used for generating the summary stream.

Our utilities also stored local compression estimates, generated by running every fixed-size chunks (ex. 4k, 8k, 16k, 32k) through LZO and storing a single byte with the percent of original chunk size. Then, given the current file offset and chunk size, we could estimate the compression at arbitrary points in the stream. Using piecewise constant or linear approximations for the estimated size of compressed chunks yielded under 1% errors in compressed DER for our large dataset. In this fashion, the 1.16 Terabyte input data could be analyzed as a more portable 60 GB set of summary information (a sequence of several billion summary chunks, involving over 400 million distinct chunks). Such re-analyses took hours instead of days. We also stored, to a separate file, the duplicate/nonduplicate status of every summary stream chunk as it was encountered. This allowed us to investigate the size distribution of nonduplicate and duplicate segments of input data, as well as efficiently ascertaining which small-chunk decisions would *later* generate duplicate chunks.

To answer existence queries we used in-memory Bloom filters of up to 2 Gigabytes in length. The summary streams and Bloom filters allowed us to quickly simulate a large number of chunking algorithms on up to 1.5 Terabytes of original raw data using a single computer. We were also interested in knowing the limits of coalescing small chunks into large chunks. Since an exact calculation is prohibitive, a simple approximation was obtained by coalescing all always-together chunk sequences into single chunks. Other tools allowed us to consult an oracle in order to maintain statistics about the future re-encounter probabilities of different types of chunks.

Because of intended use at customer sites, the tools were also used to evaluate faster alternatives to Rabin Fingerprinting [7, 29] to select cut-points. Using a combination of boxcar functions and CRC-32c hashes allowing input streams to be chunked at memory bandwidth and represented a considerable time savings when generating chunking summaries. We verified that using a faster rolling window (operating essentially at memory bandwidth) had no effect upon DER, corroborating Thaker's [31] observation that with typical data even a plain boxcar sum generated a reasonably random-like chunk size distribution. He explained this as a reflection of there being enough bit-level randomness in the input data itself, making a high-quality randomizing hash function unnecessary in practice. We verified that choice of rolling window function had no little impact upon DER measurements for our 1.16 TB dataset.

3.3 DER of different chunking algorithms

Within a given algorithm, there are several parameters, such as minimum m and maximum M chunk size, and trigger level ℓ, which can generate different behavior. Breaking-apart and amalgamation algorithms also have other parameters, such as k (the number of small chunks in a big chunk) and an optional resynchronization parameter r (defining a coarser-grained chunking level $2^{\ell+r}$ across which no big chunk may extend). When an algorithm was run over the entire 1.16 Terabyte data set or its summary, we measured the DER as the ratio of input bytes to bytes within stored chunks. Bytes within stored chunks could be reported raw, or as compressed size estimates. We used an LZO compressor to derive compression values; however, other compressors should display qualitatively similar behavior. Compression is relevant because most archival systems store data in compressed format. We explored a wide space of parameters for amalgamation (fixed- and variable-size big chunks) and breaking-apart algorithms on this data set. We show plots assuming zero metadata overhead initially and will give an illustration of the effects of metadata upon the DER later.

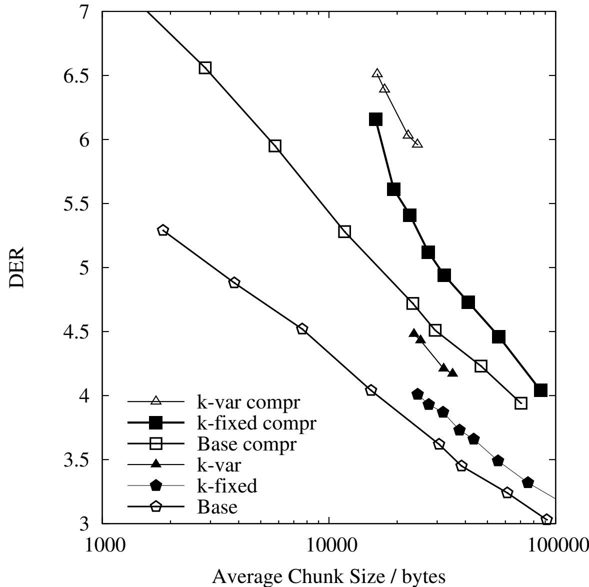


Figure 5: Performance of two amalgamation chunking algorithms, k-fixed and k-var, compared to a baseline chunking algorithm "Base", over a range of chunk sizes. The top 3 "compr" curves are the same data as the lower three traces, but DER and chunk sizes are reported assuming compressed chunk storage.

Performance of bimodal amalgamation chunking

Figure 5 compares two bimodal amalgamation algorithms "k-fixed" and "k-var" with standard baseline chunking algorithms "Base". For each of these 3 chunking algorithms, raw DER values and chunk sizes are in the bottom 3 traces, while corresponding DER using stored compressed chunk sizes appears in the upper 3 traces. Comparing the two sets of three traces, we note for compressed storage the traces are more highly sloped, which reflects the rapid initial rise in compression efficiency as chunk size is increased. Linearity in the raw DER traces indicate some scale-independent statistical behavior in our large archive dataset: this is not the case for some small test datasets that we present later.

In this and later figures, precise parameter settings of a particular algorithm are usually not influential, serving to move measured points along the same general curve. Since precise parameter settings are not crucial, the parameters we do describe should be viewed as examples of reasonable settings.

The "Base" baseline chunking traces shown in Fig. 5 varied the minimum, nominal average and maximum chunk sizes $\{m, m+2^{\ell}, M\}$, often maintaining a 1:2:3 ratio for these values. We consulted $b = 3$ levels of backup cut-points if maximum chunk size was encountered.

The "k-fixed" traces of Fig. 5 use an amalgamation algorithm, running with fixed-size big chunks (i.e. a big chunk consists always of k small chunks). Half these runs maintained a 1:2:3 ratio for min:avg:max, with $k = 8$ and $r = 4$. Two used $k = 4$ instead, and two did not use resynchronization points. Investigating more parameter settings showed that minor variations in chunking parameters typically lay along the same curve: the algorithm was robust to parameter choices. We found a broad optimal region for k from 8 to 12, and suggest that resynchronization points be either unused or maintained at $r \gtrsim 3$.

The algorithm labelled "k-var" in Fig. 5, at an additional querying cost, allows variable-sized big chunks that use any number 1–k of small chunks. It also used Bloom Filter queries for small chunks which were previously encountered but emitted only as part of a previous big chunk as finer-grained delineators of change regions. In spirit the "k-var" traces of Fig. 5 might be viewed as a lower bound for what more sophisticated algorithms using sub-chunk information (such as *fingerdiff* [5]) or chunk rewriting approaches could achieve.

Later, we will show that the extensions to the "k-var" algorithms provide only slightly better performance. This suggests that the most important algorithmic difference between fixed- and variably-sized big chunks lay in the increased flexibility of generating and recognizing large chunks. Nevertheless, algorithms in this "k-var" class require more existence queries so they are not algorithms of choice.

Note that the "k-fixed" algorithm of Fig. 5 can already maintain average compressed chunk sizes up to 3–4$\times$ as large as a baseline chunker at small chunk sizes (e.g. DER 6.1 at 16100 bytes using $k = 4$ and no resynchronization, as compared to an interpolated 4700 bytes for "Base compr"). For uncompressed storage systems, we see that k-fixed bimodal amalgamation algorithms uniformly yielded $\approx$50% increase in average uncompressed chunk size, even at the largest (96k) chunk sizes presented.

Our implementation used a look-ahead buffer of $2k$ small chunks and in-memory Bloom filters for speed. As noted before, a lookahead buffer of $3k - 1$ chunks is also a reasonable choice. In practice, however, to maintain streaming performance very much larger look-ahead buffers may be necessary, since answering existence queries is likely to require asynchronous network or disk operations of high latency.

Our use of Bloom filters in answering existence queries led us to question the impact of false positives. For the "k-fixed" amalgamation algorithm, we found all benefits of bimodal chunking over the baseline were negated by $\approx$2.5% false positives. Falsely identified duplicate/nonduplicate transitions should be avoided. So techniques such as a hierarchy of more accurate Bloom filters [39] may be useful. Alternatively, in other work,

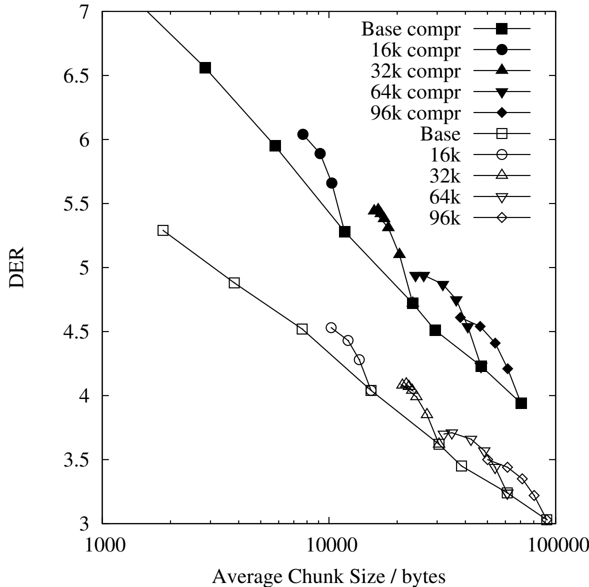


Figure 6: Breaking-apart chunking algorithms compared with baseline performance.

we have adapted efficient hash table implementations [19, 16, 23] to take full advantage of SSD R/W characteristics (possibly in conjunction with fingerprint approaches) to provide fast, exact answers to existence queries.

Variants of amalgamation algorithms, that prioritize equivalent choices of big chunk if they occurred, were found to offer no significant performance improvement. In fact, several such attempts work badly when run on actual data, often for rather subtle reasons.

Small chunk statistics, using an oracle

Using knowledge of the full set of small chunk emissions we investigated the statistics of the smaller transition region chunks, which bore out premise P2 for an amalgamation algorithm using fixed-size big chunks. For example (not shown in figures), for $k = 8$ small chunks in a transition region between two duplicate big chunks, the bordering small chunks have around 88% chance of being encountered subsequently, dipping to 86% for central small chunks. For one-sided duplication transitions, we found that the small-chunk duplication chance decayed from ~75% to ~67%. Bimodal chunking with $k = 32$ showed small-chunk duplication probability declining from 86% adjacent to the duplicate big chunk to 65% at the furthest small chunk. These experimental results agree with earlier expectations based on Fig. 4 assuming good future duplication of byte-level duplication regions and, say, a uniform location for the start of the byte-level non-duplicate region in 4(a) with respect to the small chunk transition region 4(d).

Performance of bimodal breaking-apart chunking

In Figure 6 we present results with a breaking-apart algorithm, which uses one query per large chunk, compared to the baseline algorithm. Most runs retain baseline $m : m + 2^{\ell} : M$ settings in a 1:2:3 ratio. Beginning with a baseline chunker we consecutively divided these settings by two to generate a series of small chunkers, which were used in the breaking apart algorithm of Fig. 1. A few additional points vary R, the size of transition region that gets re-chunked, but do not depart substantially from the breaking-apart curves for $R = 1$. We note that reasonable performance is obtainable by choosing a small chunker with average chunk size about 4–8 times smaller than the original baseline chunker.

Comparing Figs. 5 and 6, we see that a carefully tuned breaking apart algorithm can be competitive with the performance of amalgamation algorithms with fixed-size big chunks, particularly in the regime of chunk sizes $\gtrsim$40k. The practical benefit of breaking-apart over the "k-fixed" amalgamations of Fig. 5 is a reduction in the number of existence queries by a factor of k.

Effect of non-zero metadata overhead

One approach to accounting for metadata effects is to pretend that it simply increases the average stored block size by some number of bytes. Another instructive approach is to consider the the metadata effects on the oft-reported DER values. For example, with a metadata overhead of 800 bytes per chunk, we can use the known total amount of input bytes (which is a constant 1.16 TB in Figs. 5 and 6) to transform the DER value of each measurement, while still reporting the average size of the chunk.

In Figure 7, we have simply scaled the DER values of the empty symbols, which are traces taken from Fig. 5, by reducing their DER by $1 + f$. Here $f \equiv {}^{\text{metadata size}}/_{\text{average chunk size}}$ is the metadata overhead, and the transformed traces are plotted with solid symbols. The DER reduction can be quite dramatic at low chunk sizes where metadata overhead is a substantial fraction of the stored chunk size. We see that including metadata magnifies the DER improvement relative to a baseline chunker of equivalent average chunk size. The figure motivates maintaining average chunk sizes much larger (preferably $\gtrsim 20\times$) than the per-chunk metadata overhead.

Data	# of versions	Baseline chunk size / bytes	Baseline DER	Amalgamation chunk size / bytes	Amalgamation DER	Compressed size of 16k records / 16k
gcc source	20	4952	4.68	13742	4.59	0.37
gdb source	10	6184	4.14	15225	4.05	0.35
linux source	10	6921	3.51	16804	3.52	0.40
emacs source	10	7525	3.23	17265	2.95	0.46

Table 1: Comparison of DER (w/ LZO) achieved by baseline chunkers and amalgamation algorithms. The average input chunk size of the baseline chunker was 16k with allowed sizes 8k–24k and two backup levels. The amalgamation used large chunks composed of exactly $k = 8$ small chunks. Values of chunk size and DER reflect chunks stored in compressed LZO format. The average compressibility of fixed-length 16k records of input data (no deduplication) are in the last column.

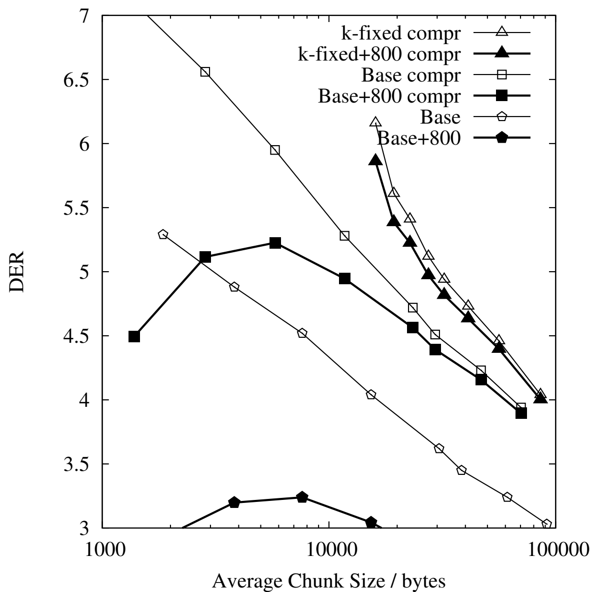


Figure 7: Two baseline and one "k-fixed" amalgamation algorithm curves (open symbols) from Fig. 5 have been transformed (solid symbols) to reflect 800 metadata bytes per chunk.

Performance using source code archives

We also analyzed data sets consisting of tar files for consecutive releases of several large projects. The compressed chunk size and DER under one set of baseline conditions and an amalgamation algorithm based upon these small chunks is shown in Table 1. We see that amalgamation has increased the average chunk size of stored chunks by a factor of around 2.5, with a worst case decrease in DER of 8%.

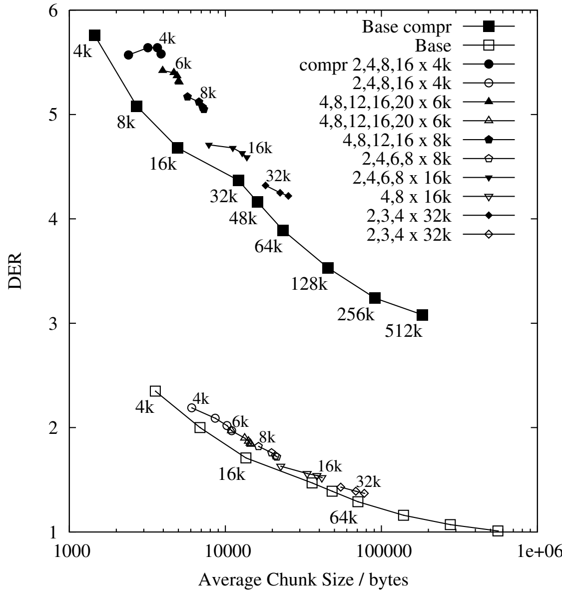
A picture of the performance of baseline and "k-fixed" amalgamation on these source archives is offered by Fig. 8, which shows DER curves with compression (top curves) and without (bottom). Corresponding to various baseline chunkers, we ran "k-fixed" amalgamate algorithms as in Fig. 5 for k values between 2 and 20. Recall that $k = 8$ was suggested to be a reasonable value for the large dataset. Improvements in DER and chunk size are much worse for these small archive datasets, when compared with the 1.16 TB dataset of Fig. 5.

The baseline chunkers all display uncompressed DER that approaches 1.0 as average chunk size rises, showing that at large chunk sizes, DER can be obtained primarily by using compression. These data sets have small file sizes and quite scattered change sections (i.e. property P1 for filesystems may not apply well when the density of changes is large and somewhat uniform). The DER (w/o LZO) points are usually above (better) the smooth Baseline curve, but do not show significant improvement. The improvement is better when storage of compressed chunks is considered. The emacs data set consistently shows the smallest improvements from amalgamation, as well as the least duplicate elimination (2.0 at 4k average chunk size, 4.12 compressed) and least compressibility (fixed-size 16k chunks were compressed to 46% of their original length).

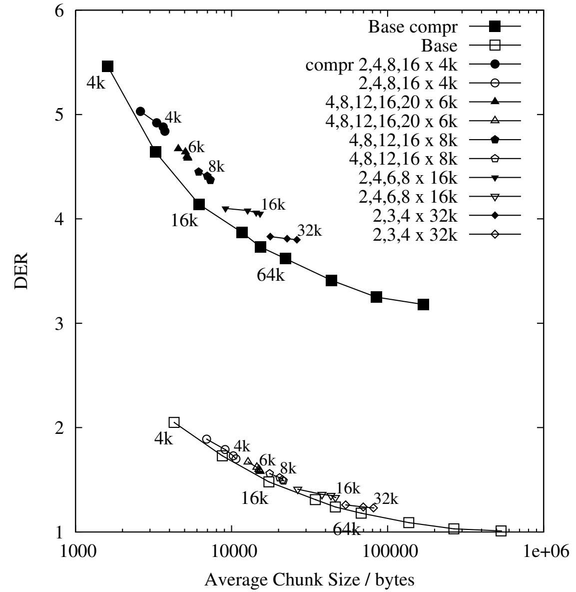
Even though there is no reason that tar files of source code releases should concentrate most change regions into a small subset of files, amalgamation still shows modest DER vs. chunk size improvement with respect to baseline CDC chunking. Lightly degraded DER was achieved with average chunk sizes larger by factors of $2.5\times$ (see Table 1) in these data sets, as compared to a factor of 3–$4\times$ in the actual 1.16 TB archival data set.

Optimal "always-together" chunks

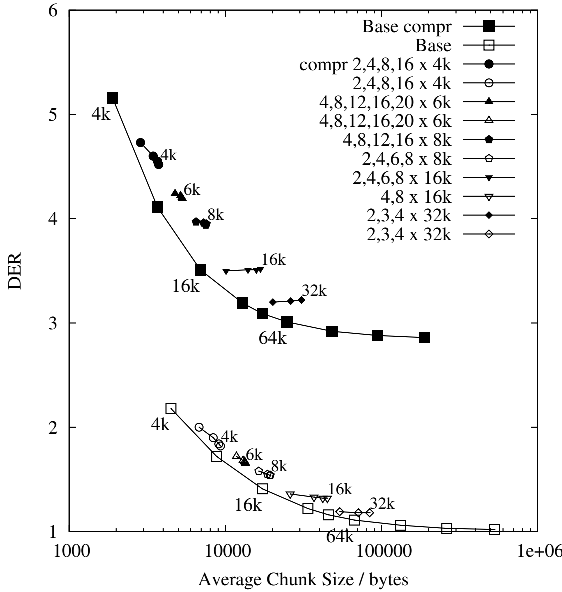
For our 1.16 TB data set, it is also interesting to consider what a good theoretical amalgamation of small chunks would be. A simple set of optimization moves is to always amalgamate consecutive chunks that always occurred together. This will not affect the DER at all, but will increase the average chunk size. Iterating this pro-



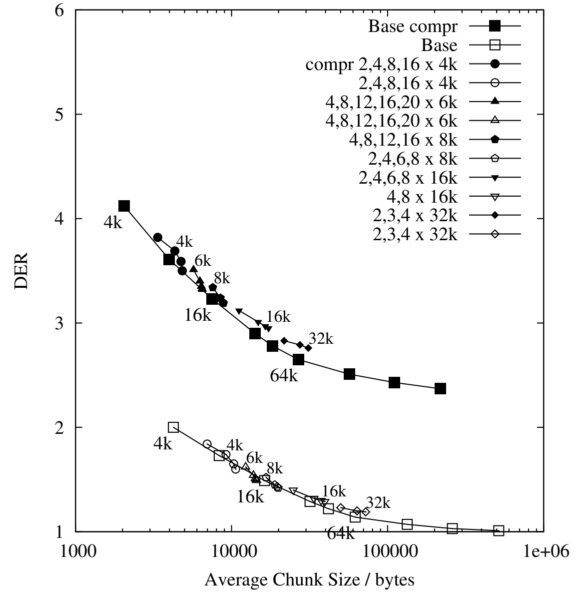
(a) DER vs. chunk size: gcc dataset



(b) DER vs. chunk size: gdb dataset



(c) DER vs. chunk size: linux dataset



(d) DER vs. chunk size: emacs dataset

Figure 8: Duplicate elimination versus stored chunk size measurements on consecutive source code releases. Baseline and bimodal *k*-fixed chunking were performed, yielding results for uncompressed storage (lower traces, open symbols) and compressed storage (upper traces, solid symbols). Chunk compression used the default LZO settings. Bimodal series denoted in the legends as "$k_1, k_2, \ldots$ x *N*k" amalgamate a fixed number, *k*, of chunks output from the baseline chunker with *N*k average chunk length.

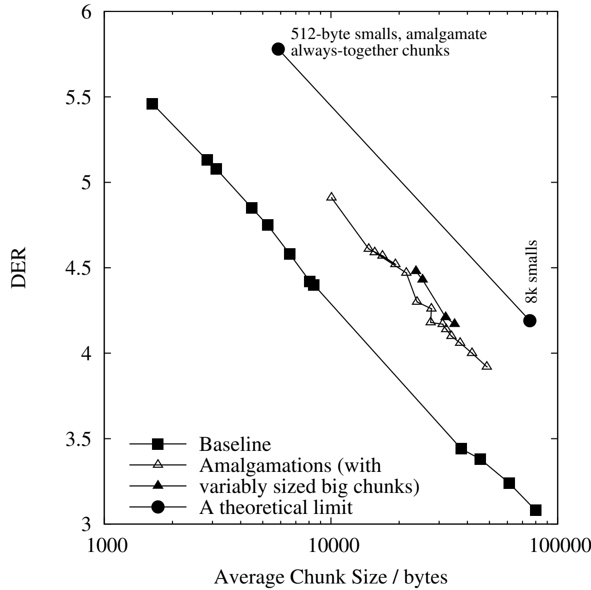


Figure 9: Baseline and k-var amalgamation are compared with theoretical chunk size limits determined by amalgamating every set of chunks which always co-occurred in our 1.16 Terabyte data set. k-var amalgamation results (triangles) cover a wide range of parameters chunking parameters. Solid triangles in Figs. 5 and 9, using extensions to the basic algorithm, are included here for comparison.

duces that longest possible strings of chunks that always co-occurred and increases the average chunk size. This parallelized calculation is lengthy and non-scalable.

Using "future knowledge" to amalgamate all always-together chunks was done for input chunk sequences of 512 and 8192 average size to produce two isolated points in Fig. 9. Analyzing the raw summary stream, with chunks 512 bytes long on average, increased the average uncompressed stored chunk size from 576 to 5855 bytes (i.e. the average number of always-co-occurring small chunks was around 10 for this data set). Similarly, the other theoretical calculation increase the average chunk size from around 8k to 75k bytes, once again nearly a factor of $10\times$ improvement in uncompressed chunk size.

In practice, amalgamating often- or always-together chunks opportunistically may be a useful background task to optimizing storage. This experiment provides an easily-defined theoretical bound against which we can judge how well our simple algorithms based on duplicate/nonduplicate transition regions were performing: $10\times$ improvement can be achieved, with such an oracle.

For comparison, Fig. 9 also presents a number of amalgamation results with variable-size big chunks (k-1 queries per small chunk). Such amalgamation algorithms

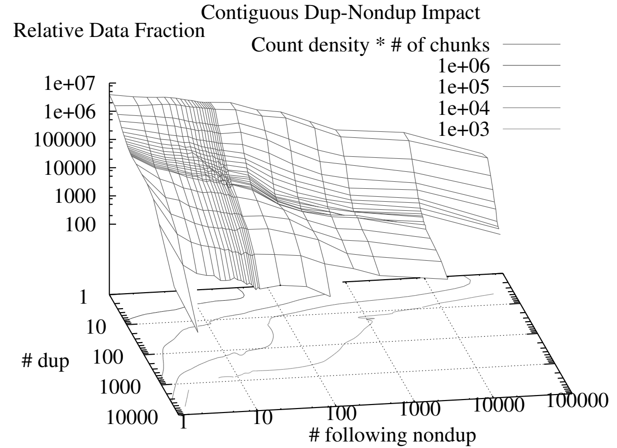


Figure 10: Histogram of number of contiguous duplicate chunks vs. number of subsequent contiguous nonduplicate chunks at the 512-byte expected chunk size. Raw counts have been scaled by the number of chunks to produce histogram values representing the total amount of input data. Note the logarithmic scales: the overwhelmingly most frequent (and still most important with regard to total amount of input data involved) occurrence is one duplicate chunk followed by one nonduplicate chunk.

come almost half-way from the baseline curve to this particular theoretical limit. These runs had a haphazard selection of m, ℓ and M small chunk size settings, use 0–4 resynchronization cut-points (usually zero or 4), and mostly have $k = 8$. Again, noting that the results lie more or less along a common line we conclude that precise values of parameter settings are not vitally important. We also note that performance is on par with the traces labeled "k-var" in Fig. 5 (reproduced here in Fig. 9 as solid triangles). This indicates that the additional complication of using sub-chunk information to delineate change regions was not particularly useful.

3.4 Data characteristics

Size-of-modification distribution

Although originally formulated based on considerations of simple principles P1 and P2, it is important to judge how much our real data departs from such a simplistic data model. We found that the actual data deviated quite substantially from an "ideal" data set adhering to P1 and P2. A simplest-possible data set adhering to P1 might be expected to have long sequences of contiguous nonduplicate data during a first backup session, followed by long stretches of duplicate data during subsequent runs.

We interrogated the anonymized summary stream, as chunked at the 512-byte expected chunk size, using a bit-

stream summary of the "current" duplication status of the chunk. The actual histograms of number of contiguous nonduplicate chunks vs. number of contiguous duplicate following chunks (and vice-versa) showed an overwhelming and smoothly varying preference to having a single nonduplicate chunk followed by a single duplicate chunk. A 2-dimensional histogram of the final contiguous numbers of duplicate/nonduplicate chunks (after 14 full backup sessions) is in Figure 10. The histograms after the first "full" backup was of similar character. Such histograms do not suffice for estimating DER since duplication counts are absent. This analysis found no naive adherence to P1 and P2.

Only a minor fraction of the input stream was data occurring as long stretches of unseen data. Only the earlier oracular results provided direct evidence for P2: small chunks close to duplicate big chunks did indeed have significantly augmented re-emission probabilities. This effect can be predicted simply by assuming a uniform location of the transition region from duplicate to nonduplicate bytes within the large chunk being stored as smaller chunks in Figs. 2(d) and 4(d), and may be the dominant reason why bimodal chunking works for archival data.

This suggests that for input data sets showing such high interspersal of duplicate with nonduplicate chunks, alternate approaches may be able to come closer to the theoretical limit than the algorithms presented in this paper. Nevertheless, even for such data, even simple bimodal chunking heuristics were able to increase average chunk size by a factor of 3 or more.

4 Related Work

For our purposes, the speed of blocking (chunking) was a consideration because we target throughputs of several hundred MB/s. The simplest and fastest approach is to break apart the input stream into fixed-size chunks. This is the approach taken in the *rsync* file synchronization tool [34, 33]. However, consider what happens when an insertion or deletion edit is made near that beginning of a file: after a single chunk is changed, the entire subsequent chunking will be changed. A new version of a file will likely have very few duplicate chunks. Pratt [26] provides good comparison of fixed- and variable-sized chunking for real data. Lufei et al. [22] provides an introduction to options such as gzip, delta-encoding, fixed-size blocking and variable-size chunking. For filesystems, You et al. [36] compares chunking and delta-encoding. Delta-encoding is particularly good for things like log files and email, which are characterized by frequent small changes.

CDC produces chunks of variable size that are better able to restrain changes from a localized edit to a limited number of chunks. Applications of CDC include network filesystems of several types [2, 27], space-optimized archival of collections of reference files [9, 14, 37], as well as file synchronization [32, 15]. By using special rolling window functions in innermost loops, the baseline CDC algorithms can operate very quickly.

Mazières' Low-Bandwidth File System (LBFS) [25, 31] was influential in establishing CDC as a widely used technique. Usually, the basic chunking algorithm is typically only augmented with limits on the minimum and maximum chunk size. More complex decisions can be made if one reaches the maximum chunk size [30, 13, 15] (see Section 2.2).

Alternatives to CDC for compressing data exist and typically have higher cost. An often used technique in more aggressive compression schemes is resemblance detection and some form of delta encoding. Unfortunately, finding maximally-long duplicates [17, 18, 1] or finding similar (or identical) files in small [5] or large (gigabyte) [8, 10, 20, 11, 28] collections is a nontrivial task.

In HYDRAstor [12] and DEBAR [35], existence queries (and *global* deduplication) can be addressed efficiently by consulting a scalable, distributed data structure. Our approach has been to tackle the small chunk size problem directly. A noted in the introduction, a recent alternative approach is to reduce metadata requirements by practicing only *local* duplicate elimination within a suitably large local basin of data. For example, the approach of Brin et al. [6] has been revived in an elegant "extreme binning" approach that distributes information at a large-block level (file-level representative hash) to detect near-similarity, and has been shown to achieve near-optimal deduplication at small-chunk level [4]. Another recent approach describes a sparse indexing approach to determining similar segments of an stream [21].

Bimodal chunking presumes only an existence query for already-stored chunks, and has the potential to provide system improvements of several types. The increase in average chunk size (roughly 2.5× in these data sets, and 3–4× in the 1.16 TB archival data set) decreases the storage cost for metadata describing these chunks. By reducing the number of disk accesses, there are potential increases in read and write speeds as fewer transactions with the storage units are involved. Furthermore, the existence query information can be used in some backup systems to entirely elide network transmission of existing duplicates, which may result in additional write speed improvements or decreased system cost.

5 Conclusion and Future Work

In this paper, we proposed bimodal algorithms that vary the expected chunk-size dynamically. They are able to

perform content-defined chunking in a scalable manner, involving a constant number of chunk existence queries per unit of input. Significantly, these algorithms require no special-purpose metadata to be stored. We show that these algorithms increased average chunk size while maintaining a reasonable duplication elimination ratio. We demonstrated the benefits of the algorithms when applied to 1.16 Terabyte of actual backup data as well as to four sets of source code archives.

Although the statistics of these data sets suggest that they do not conform to our expectations based on principles P1 and P2, the algorithms still perform well, leading us to conjecture that they are robust (applicable to many types of archival inputs). We expect the proposed algorithms will behave best for storage of versioned data in block stores with high metadata cost, but we plan to evaluate them for other data sets.

Under a wide variety of chunking parameters, chunk amalgamation algorithms performed well. They present more flexibility in querying for duplicate chunks than algorithms involving breaking apart chunks within a preliminary large chunking. We also plan to investigate algorithms that use compressibility to govern chunking decisions based on fast entropy estimation.

This work has targeted evaluating a prospective bimodal chunking algorithm that has potential to address real issues in the HYDRAstor storage system and other systems that require large per-chunk storage overhead. The simple algorithms of Figs. 1 and 3 used in the evaluation are in the process of being adapted for inclusion and evaluation in HYDRAstor. Because of the latency of answering existence queries, this requires a larger lookahead buffer and issuing (in a straightforward approach) all possible existence queries. Additionally, current storage systems go to great lengths to avoid disk accesses . For example, both HYDRAstor and Data Domain products address disk access reduction and locality of access issues and both have used Bloom filters to reduce disk the number of disk accesses [38]. Because of the disk bottleneck, efficient mechanisms to reply to existence queries with minimal impact of streaming read and write performance is desired. Implementation, currently underway for the HYDRAstor storage product, may eventually involve new data structures, or even new hardware (particularly SSDs) before bimodal chunking becomes a commercial offering.

6 Acknowledgments

We would like to thank our shepherd, Randal Burns, whose feedback has greatly improved the paper, and the anonymous reviewers for their comments and suggestions. We also wish to acknowledge Krzysztof Lichota for his work developing fast rolling windows, using boxcar functions, to obtain throughputs higher than those achievable with the usual approach of Rabin Fingerprinting [7, 29] to select cut-points.

References

[1] AGARWAL, R. C. Method and computer program product for finding the longest common subsequences between files with applications to differential compression. United States Patent 20060112264, May 2006.

[2] ANNAPUREDDY, S., FREEDMAN, M., AND MAZIÈRES, D. Shark: Scaling File Servers via Cooperative Caching. In *NSDI '05 Paper [NSDI '05 Technical Program]* (2005).

[3] BARRETO, J., AND FERREIRA, P. A replicated file system for resource constrained mobile devices. In *Proceedings of IADIS International Conference on Applied Computing* (2004).

[4] BHAGWAT, D., ESHGHI, K., LONG, D. D. E., AND LILLIBRIDGE, M. 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)* (Sept. 2009).

[5] BOBBARJUNG, D. R., JAGANNATHAN, S., AND DUBNICKI, C. Improving duplicate elimination in storage systems. *Trans. Storage 2*, 4 (2006), 424–448.

[6] BRIN, S., DAVIS, J., AND GARCIA-MOLINA, H. Copy detection mechanisms for digital documents. In *In Proceedings of the ACM SIGMOD Annual Conference* (1995), pp. 398–409.

[7] BRODER, A. Some applications of Rabin's fingerprinting method. *Sequences II: Methods in Communications, Security, and Computer Science* (1993), 143–152.

[8] CHOWDHURY, A., FRIEDER, O., GROSSMAN, D., AND MCCABE, M. C. Collection statistics for fast duplicate document detection. *ACM Trans. Inf. Syst. 20*, 2 (2002), 171–191.

[9] DENEHY, T., AND HSU, W. Duplicate management for reference data. Technical report RJ 10305, IBM Research, October 2003.

[10] DOUGLIS, F., AND IYENGAR, A. Application-specific Delta-encoding via Resemblance Detection. In *Proceedings of the USENIX Annual Technical Conference* (2003).

[11] DOUGLIS, F., KULKARNI, P., LAVOIE, J. D., AND TRACEY, J. M. Method and apparatus for data redundancy elimination at the block level. United States Patent 20050131939, June 2005.

[12] DUBNICKI, C., GRYZ, L., HELDT, L., KACZMARCZYK, M., KILIAN, W., STRZELCZAK, P., SZCZEPKOWSKI, J., UNGUREANU, C., AND WELNICKI, M. HYDRAstor: a Scalable Secondary Storage. In *Proccedings of the 7th conference on File and storage technologies* (2009), USENIX Association, pp. 197–210.

[13] ESHGHI, K., AND TANG, H. K. A framework for analyzing and improving content-based chunking algorithms. Technical report HPL-2005-30R1, HP Laboratories, 10 2005.

[14] FORMAN, G., ESHGHI, K., AND CHIOCCHETTI, S. Finding similar files in large document repositories. In *KDD '05: Proceeding of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining* (New York, NY, USA, 2005), pp. 394–400.

[15] GUREVICH, Y., BJORNER, N. S., AND TEODOSIU, D. Efficient chunking algorithm. United States Patent 20060047855, March 2006.

[16] HUA, N., ZHAO, H., LIN, B., AND XU, J. Rank-indexed hashing: A compact construction of bloom filters and variants. In *IEEE International Conference on Network Protocols (ICNP 2008)* (Oct. 2008), pp. 73–82.

[17] JAIN, N., DAHLIN, M., AND TEWARI, R. TAPER: Tiered Approach for Eliminating Redundancy in Replica Sychronization. In *USENIX Conference on File and Storage Technologies (FAST05)* (Dec 2005).

[18] JAIN, N., DAHLIN, M., AND TEWARI, R. TAPER: Tiered Approach for Eliminating Redundancy in Replica Synchronization. Tech. rep., Technical Report TR-05-42, Dept. of Comp. Sc., Univ. of Texas at Austin, 2005.

[19] KANIZO, Y., HAY, D., AND KESLASSY, I. Optimal fast hashing. In *28th IEEE International Conference on Computer Communications (INFOCOM)* (Apr. 2009), pp. 2500–2508.

[20] KULKARNI, P., DOUGLIS, F., LAVOIE, J., AND TRACEY, J. Redundancy Elimination Within Large Collections of Files. In *Proceedings of the USENIX Annual Technical Conference* (2004).

[21] LILLIBRIDGE, M., ESHGHI, K., BHAGWAT, D., DEOLALIKAR, V., TREZISE, G., AND CAMBLE, P. Sparse indexing: large scale, inline deduplication using sampling and locality. In *Proceedings of the 7th conference on File and storage technologies* (2009), USENIX Association, pp. 111–123.

[22] LUFEI, H., SHI, W., AND ZAMORANO, L. On the effects of bandwidth reduction techniques in distributed applications. *Proceedings of International Conference on Embedded and Ubiquitous Computing (EUC'04)* (2004).

[23] LUMETTA, S., AND MITZENMACHER, M. Using the power of two choices to improve bloom filters. *Internet Mathematics 4*, 1 (2007), 17–34.

[24] MOULTON, G. H. System and method for unorchestrated determination of data sequences using sticky byte factoring to determine breakpoints in digital sequences. United States Patent 6810398, October 2004.

[25] MUTHITACHAROEN, A., CHEN, B., AND MAZIÈRES, D. A low-bandwidth network file system. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles* (New York, NY, USA, 2001), pp. 174–187.

[26] POLICRONIADES, C., AND PRATT, I. Alternatives for detecting redundancy in storage systems data. In *USENIX Ó4: Proceedings of the USENIX Annual Technical Conference* (2004).

[27] PORTS, D. R. K., CLEMENTS, A. T., AND DEMAINE, E. D. PersiFS: a versioned file system with an efficient representation. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles* (New York, NY, USA, 2005), pp. 1–2.

[28] PUGH, W., AND HENZINGER, M. H. Detecting duplicate and near-duplicate files. United States Patent 6658423, December 2003.

[29] RABIN, M. Fingerprinting by random polynomials. Technical report TR-15-81, Harvard University, 2003.

[30] SCHLEIMER, S., WILKERSON, D. S., AND AIKEN, A. Winnowing: local algorithms for document fingerprinting. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2003), pp. 76–85.

[31] SPIRIDONOV, A., THAKER, S., AND PATWARDHAN, S. Sharing and bandwidth consumption in the low bandwidth file system. Tech. rep., Department of Computer Science, University of Texas at Austin, 2005.

[32] SUEL, T., NOEL, P., AND TRENDAFILOV, D. Improved File Synchronization Techniques for Maintaining Large Replicated Collections over Slow Networks. In *ICDE '04: Proceedings of the 20th International Conference on Data Engineering* (Washington, DC, USA, 2004), p. 153.

[33] TRIDGELL, A. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, April 2000.

[34] TRIDGELL, A., AND MACKERRAS, P. The rsync algorithm. Technical report TR-CS-96-05, Australian National University, Deparment of Computer Science, FEIT, ANU, 1996.

[35] YANG, T., JIANG, H., FENG, D., AND NIU, Z. DEBAR: A Scalable High-Performance De-duplication Storage System for Backup and Archiving. *CSE Technical reports* (2009), 58.

[36] YOU, L., AND KARAMANOLIS, C. Evaluation of efficient archival storage techniques. In *Proceedings of 21st IEEE/NASA Goddard MSS* (2004).

[37] YOU, L. L., POLLACK, K. T., AND LONG, D. D. E. Deep Store: An Archival Storage System Architecture. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering* (Washington, DC, USA, 2005), pp. 804–8015.

[38] ZHU, B., LI, K., AND PATTERSON, H. 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* (Berkeley, CA, USA, 2008), USENIX Association, pp. 1–14.

[39] ZHU, Y., JIANG, H., AND WANG, J. Hierarchical Bloom filter arrays (HBA): a novel, scalable metadata management system for large cluster-based storage. In *Cluster Computing, 2004 IEEE International Conference on* (Sept. 2004), pp. 165–174.