Improving Duplicate Elimination in Storage Systems

Deepak R. Bobbarjung and Suresh Jagannathan
Department of Computer Sciences, Purdue University
and
Cezary Dubnicki
NEC Laboratories America

Minimizing the amount of data that must be stored and managed is a key goal for any storage architecture that purports to be scalable. One way to achieve this goal is to avoid maintaining duplicate copies of the same data. Eliminating redundant data at the source by not writing data which has already been stored, not only reduces storage overheads, but can also improve bandwidth utilization. For these reasons, in the face of today's exponentially growing data volumes, redundant data elimination techniques have assumed critical significance in the design of modern storage systems.

Intelligent object partitioning techniques identify data that are *new* when objects are updated, and transfer only those chunks to a storage server. In this paper, we propose a new object partitioning technique, called *fingerdiff*, that improves upon existing schemes in several important respects. Most notably *fingerdiff* dynamically chooses a partitioning strategy for a data object based on its similarities with previously stored objects in order to improve storage and bandwidth utilization. We present a detailed evaluation of *fingerdiff*, and other existing object partitioning schemes, using a set of real-world workloads. We show that for these workloads, the duplicate elimination strategies employed by *fingerdiff* improve storage utilization on average by 25%, and bandwidth utilization on average by 40% over comparable techniques.

Categories and Subject Descriptors: H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing; H.3.2 [**Information Storage and Retrieval**]: Information Storage

General Terms: Storage Management

Additional Key Words and Phrases: Content-based addressing, duplicate elimination, Rabin's fingerprints

1. INTRODUCTION

Traditional storage systems typically divide data objects such as files into fixed-sized blocks and store these blocks on fixed locations in one or more disks. Metadata structures such as file inodes record the blocks on which a file is stored along with other relevant file-specific information, and these inodes are themselves stored on fixed-sized disk blocks. Whenever an object is modified by either inserts, deletes or in-place replacements, the new blocks in the object are written to disk, and the metadata structure is updated with the new block numbers. However due to the inability to efficiently identify those portions of the object that are actually new in the latest update, a large part of existing data must get necessarily rewritten to storage. Thus, the system incurs a cost in terms of storage space and bandwidth whenever data is created or updated. This cost depends upon the storage architecture, but is proportional to the amount of new data being created or updated. Our solution relies on utilizing local computational and storage resources in order to minimize

the cost of writing to scalable storage networks, by reducing the amount of new data that is written with every update. This also reduces the amount of data that has to be stored and maintained in the storage system, enabling greater scalability.

Recently, systems have been proposed that divide objects into variable-sized chunks* instead of fixed-sized blocks in order to increase the amount of duplicate data that is identified. Techniques that partition objects into variable-sized chunks enjoy greater flexibility in identifying chunk boundaries. By doing so, they can manipulate chunk boundaries around regions of object modifications so that changes in one region do not permanently affect chunks in subsequent regions.

This paper describes *fingerdiff*, a device-level variable-sized object processing algorithm designed to reduce the amount of data that is stored and maintained in storage systems. *Fingerdiff* improves upon the duplicate elimination facilities provided by existing techniques [Muthitacharoen et al. 2001; Cox et al. 2002] by dynamically repartitioning data so as to aggregate unmodified data pieces into large chunks, thus minimizing the size of new chunks written with each update. Like LBFS [Muthitacharoen et al. 2001], *fingerdiff* works by maintaining client-side information in the form of hashes of small pieces of data for objects that have been previously written. However dynamic partitioning allows *fingerdiff* to expand the variability of chunk sizes enabling greater flexibility in chunk size ranges. As a result *fingerdiff* can allow unmodified data regions to contain larger chunks, while breaking up modified data regions into smaller chunks in order to minimize the size of new chunks. Writing only data chunks that are new in the current update reduces the total amount of data that has to be written to the system for every update. Similar techniques have been proposed before in order to reduce bandwidth in a low bandwidth network [Muthitacharoen et al. 2001] and to improve duplicate elimination in content addressable stores [Quinlan and Dorwards 2002; Kubiatowicz et al. 2000; Hong et al. 2004]. *Fingerdiff* not only improves upon the duplicate elimination capability of these techniques, it also reduces management overheads involved in storing and maintaining large volumes of data, thus improving storage system scalability.

1.1 Contributions

Our contributions in this paper are the following:

— We propose a new object partitioning algorithm, *fingerdiff* that improves upon the duplicate elimination capability of existing techniques, while simultaneously reducing storage management overheads.
— Using real-world workloads, we compare storage utilization and other storage management overheads of *fingerdiff* with those of existing techniques. We evaluate the effect of chunk sizes on the performance of these techniques.
— We show that *fingerdiff* improves upon the storage utilization of existing techniques by 25% on average and bandwidth utilization by 40% on average.

The remainder of this paper is organized as follows: Section 2 presents the architecture of the system that we use to evaluate the effectiveness of data partitioning techniques. Section 3 briefly discusses existing object partitioning schemes before presenting the *fingerdiff* algorithm in section 3.2.2.2. Section 4 establishes the experimental framework that we

*Henceforth, we will use the term "chunk" to refer to variable-sized data blocks and the term "block" to refer to fixed sized data blocks.

employ to compare the effectiveness and performance of the different techniques that we discuss. Section 5 presents performance results and section 6 presents a detailed discussion of these results. Section 7 contains related work and conclusions are given in section 8.

2. SYSTEM ARCHITECTURE

We assume a system model that consists of a storage engine that is essentially a chunk store. This chunk store accepts requests to persistently store chunks of data from storage clients. The store satisfies each such request by computing a hash key based on the content of the chunk and storing the chunk in a location based on the value of its key. Next, the chunk store returns the key to the client that wrote the chunk and the client in turn retains the key as a capability or pointer to the chunk.

Such content-addressable storage systems[Cox et al. 2002; Hong et al. 2004; Kubiatowicz et al. 2000; Muthitacharoen et al. 2001; Quinlan and Dorwards 2002] employ the content based hash to uniformly name and locate data blocks. If the hash function used is a robust one-way hash function like SHA-1[National Institute of Standards and Technology, FIPS 180-1 1995], the resulting key is unique with high probability. Therefore, if hashes of two objects are equal, such systems can identify corresponding blocks as duplicates with high probability. Systems such as Venti[Quinlan and Dorwards 2002] and Oceanstore[Kubiatowicz et al. 2000] are examples of storage architectures that rely on content-based addressing to reduce storage consumption and management costs.

Applications running on various clients periodically update data objects such as files to the store using an object server. The object server employs a driver that runs an object partitioning technique such as *fingerdiff*. This driver divides objects into either fixed-sized data blocks or variable-sized data chunks depending on the object partitioning algorithm. Chunks or blocks identified by the driver as new in this update are then written to the chunk store. For this purpose, the chunk store exports a simple chunk read/write API to all application drivers. The driver asynchronously employs one of the chunking techniques that we discuss to divide client data objects into chunks and then writes these chunks to the store.

In case of *fingerdiff*, the application will communicate to the object server apriori the exact specification of an object. The server then maintains in its *fingerdiff* driver, a separate tree for every specified object. Examples of an object specification are a single file, all files in one directory or any group of random files that the application believes will share substantial common data. All updates to a particular object will result in the driver comparing hashes of the new update with hashes in the corresponding tree.

The system model is shown in Figure 1. Multiple clients update data through object servers such as file or database servers. Each object server employs a *fingerdiff* driver that maintains a lookup tree for every specified object. The driver writes new chunks of data to a chunk store upon every update originating from data clients.

Note that *fingerdiff* is not restricted to this architecture. Indeed, *fingerdiff* is also applicable in client-server environments, where both the client and the server maintain a series of hashes for each file that they are processing. This model has been used to demonstrate the efficiency of Rabin fingerprint based chunking technique in the low bandwidth network file system [Muthitacharoen et al. 2001]. The chunk store in our system model can be a centralized or distributed hash table that maps hashes to chunk locations. It can thus provide duplicate elimination across objects and clients, if multiple unrelated clients share the

same data. File server based architectures that maintain hashes on a per object basis will fail to identify duplicates across objects.

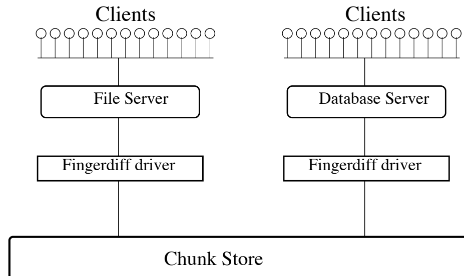

Fig. 1. The storage system model

3. DATA PARTITIONING TECHNIQUES

We first present the design of the *CDC* algorithm, and discuss the different object partitioning techniques used in realistic content addressable stores before proposing the *fingerdiff* technique.

3.1 Fixed-Sized Partitioning (*FSP*)

A fixed-sized partitioning(*FSP*) strategy employs a fixed block size that is chosen *a priori*, independent of the content of the objects being stored, and objects are partitioned into blocks of that size. Fixed-sized partitioning (*FSP*) is used in content addressable systems such as Venti[Quinlan and Dorwards 2002] and Oceanstore[Kubiatowicz et al. 2000].

As one would expect, the effectiveness of this approach on duplicate elimination is highly sensitive to the sequence of edits and modifications performed on consecutive versions of an object. For example an insertion of a single byte at the beginning of a file can change the content of all blocks in the file resulting in no sharing with existing blocks.

3.2 Variable-Sized Partitioning(*VSP*)

Sensitivity to the nature of object modifications can be reduced by partitioning objects into variable-sized chunks such that the changes made to consecutive versions are localized to a few chunks around the region of change.

Since physical blocks on which data is stored persistently always have a fixed size, the storage engine has to maintain a mapping between a variable sized data chunk, and the one or more fixed-sized physical blocks on which it is stored. This can be done in two ways. The first is by *packing* chunks contiguously in the storage media, and maintaining the physical block number and offset in the physical media where each chunk begins. The second is by assuming a fixed physical block size and storing each chunk in exactly one physical block of that size after *padding* the remainder of the data block with zeros. Both packing and padding strategies have obvious tradeoffs. Padding obviates the need to maintain extra information for each chunk but suffers from internal fragmentation (the space consumed in storing the padded zeros) that can on average be as much as half the size of the fixed block size. In this paper, we assume a packing strategy on the storage

engine, and therefore while calculating storage utilization assume an extra 12 bytes that is required to maintain a block number, offset and size information for each chunk (4 bytes each).

3.2.1 *Content-Defined Chunking (CDC).* One variable-sized technique, which we refer to as *content-defined chunking* (*CDC*) employs Rabin's fingerprints to choose partition points in the object. Using fingerprints allows *CDC* to "remember" the relative points at which the object was partitioned in previous versions without maintaining any state information. By picking the same relative points in the object to be chunk boundaries, *CDC* localizes the new chunks created in every version to regions where changes have been made, keeping all other chunks the same. As a result, *CDC* outperforms *FSP* techniques in terms of storage space utilization on a content-based storage backend[Policroniades and Pratt 2004]. This property of *CDC* has been exploited in the LBFS[Muthitacharoen et al. 2001] and Pastiche[Cox et al. 2002] content addressable systems.

3.2.1.1 *CDC algorithm details:.* The *CDC* algorithm (shown in figure 2) determines partition points based on the contents of the object being partitioned. It assumes a parameter *exp_chunk_size* that determines the average chunk size of all the chunks generated. Chunk sizes, although variable, are expected to be within a margin of error of the *exp_chunk_size*. *CDC* computes fingerprints (typically Rabin's fingerprints) of all overlapping substrings of a given size. In practice, the size of the substring typically varies from 32 bits to 96 bits. Depending on the value of *exp_chunk_size*, *CDC* compares a given number of bits in each fingerprint with a magic value. Whenever a fingerprint is equal to the magic value, the substring corresponding to that fingerprint is marked as a partition point, and the region between two partition points constitutes a chunk. For example, if the expected chunk size is 8KB, *CDC* compares the last 13 bits of each fingerprint with a fixed magic value. Given the uniformity of the fingerprint generating function and since 2^{13} is 8192, the last 13 bits of the fingerprint will equal the magic value roughly every 8KB. As a result all chunks will be of size approximately 8KB.

The storage engine provides packing to support the variable sized chunks generated by *CDC*. For each chunk, the storage engine must maintain a mapping between the chunk's hash key value and a fixed sized physical block number where the chunk can be found, an offset in that block where the chunk begins and the size of the chunk. This mapping enables clients to read a chunk by simply issuing the chunk's hash key.

3.2.1.2 *CDC limitations:.* Notice that the variability of chunk sizes in *CDC* is rather limited. Most chunks are within a small margin of error of the *exp_chunk_size* value. Since this value determines the granularity of duplicate elimination, the storage utilization achieved by *CDC* is tied to this parameter. By decreasing the expected chunk size, we can expect better duplicate elimination since new modifications will more likely be contained in smaller sized chunks. However as You and Karamanolis have shown[You and Karamanolis 2004], reducing the *exp_chunk_size* to fewer than 256 bytes can be counter productive as the storage space associated with the additional metadata needed for maintaining greater number of chunks nullifies the effect of storage savings obtained because of a smaller average chunk size*. Further, other than storage space overheads associated with maintaining metadata information about each chunk (e.g., the hash key map), more

*We observed a similar phenomenon in our results as well.(Figure 7)

```
1  Procedure CDC
2  INPUTS: File f, Integer exp_chunk_size
3  OUTPUT: List L of chunks
4  BEGIN
5      List L := empty;
6      chunkMask := calculateMask(exp_chunk_size);
7      foreach byte position X in f do
8          window := substring(f,X,substring_size);
9          fp := fingerprint(window);
10         if (fp & chunkMask = magic_value)
11         then
12             mark X;
13         endif
14     endfor
15     mark last position in f
16     firstpos := 0;
17     foreach byte position X that is marked do
18         chunk := substring(f,firstpos, X-firstpos);
19         firstpos := X;
20         L.add(chunk);
21     endfor
22     return L;
23 END
```

Fig. 2. Fingerprint based chunking algorithm

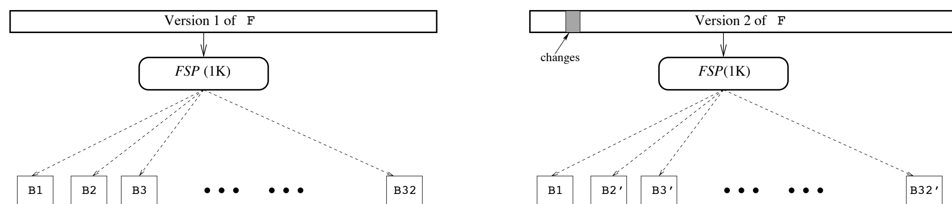

Fig. 3. An example of *FSP* being employed to encode two consecutive versions of a file.

number of chunks can lead to other system dependent management overheads as well. For example, in a distributed storage environment where nodes exchange messages on a per chunk basis, creating a greater number of chunks is likely to result in more network communication during both reads and writes.

3.2.2 *Fingerdiff*. *Fingerdiff* is designed to overcome the tension between improved duplicate elimination and increased overheads of smaller chunk sizes by improvising on the concept of variable-sized chunks. It does this by allowing larger flexibility in the variability of chunk sizes. Chunks no longer need to be within a margin of error of an expected chunk size. The idea is to reduce chunk sizes in regions of change to be small enough to capture these changes, while keeping chunk sizes large in regions unaffected by the changes made.

For this purpose, *fingerdiff* locally maintains information about *subchunks* - a unit of data that is smaller than a chunk. Subchunks are not directly written to the storage engine. Instead a collection of subchunks are coalesced together into chunks whenever possible and then the resultant chunk is the unit that is stored. *Fingerdiff* assumes an expected

subchunk size parameter (*exp_sc_size*) instead of the expected chunk size parameter used in *CDC*. *Fingerdiff* seeks to coalesce subchunks into larger chunks wherever possible. A *max_scs* parameter is used to determine the maximum number of subchunks that can be coalesced to a larger chunk.

For example, if an object is being written for the first time, all its subchunks are new and *fingerdiff* coalesces all subchunks into large chunks, as large as allowed by the *max_scs* parameter. If a few changes are made to the object and it is consequently written to the store again, *fingerdiff* consults a local client-side lookup and separates out those subchunks that have changed. Consecutive new subchunks are coalesced into a new chunk and written to the store. Consecutive old subchunks are stored as a chunk or a part of a chunk that was previously written.

To incorporate the notion of chunk-parts, *fingerdiff* expands the number of parameters required to read data from the store. In addition to the hash key value used by *CDC*, *fingerdiff* has to specify both the offset of the chunk-part within the chunk and the size of the chunk-part to the storage backend. However, the packing requirements of the storage backend needed to support variable chunk sizes of *fingerdiff* are the same as those for *CDC*.

3.2.2.1 *Example:*. To illustrate the difference between *FSP*, *CDC* and *fingerdiff*, we consider an example where these three techniques are employed to chunk two consecutive versions of a file `F`. The second version has been modified from the first version by inserting a few bytes at a region near the beginning of the file.

First consider the two versions of `F` being stored using a *FSP* technique with a fixed size of 1KB. Figure 3 illustrates the process for the first and second versions of the file. For the first version, the *FSP* algorithm creates 32 new blocks `B1` through `B32` each of which are exactly 1K bytes. The second version of the file includes some changes (which are insertions) that are restricted in the region of block `B2`. As a result, when *FSP* is run on this version, all blocks `B2` through `B32` have been changed into new blocks `B2'` through `B32'` respectively. Changing just a few bytes at the beginning of the file `F` results in the generation of many new blocks. Figure 5 shows the improvement obtained when *FSP* is substituted with *CDC* and *fingerdiff*. For this example we employ a *CDC* algorithm parameterized by an *exp_chunk_size* of 1K bytes, and a *fingerdiff* algorithm that uses a subchunk size of 1K bytes and a *max_scs* parameter of 16. In Figure 5 (a) `F` is being encoded using *fingerdiff* for the first time. When the *CDC* algorithm is called, assume that it returns a series of 32 subchunks `SC1` to `SC32` with an average expected size of 1K bytes. Assume each of these subchunks are marked *new*. The algorithm coalesces these 32 subchunks into two chunks `C1` and `C2` (because *max_scs* is 16) each of which has an expected size of 16K bytes. These two chunks are also marked as *new*, and supplied to the storage system. In Figure 5 (b), `F` has been modified and the changes are introduced in a region that corresponds to subchunk `SC2` in the original version. When this file is again partitioned with *CDC*, it returns a series of 32 chunks as before; however only the subchunk `SC2` is now replaced by `SC2'` because of a modification in this region. This marks an improvement of *CDC* over *FSP*; in *FSP* all the blocks following `B2` would be new.

Fingerdiff coalesces these subchunks into larger chunks depending on whether they are old or new. It finds that `SC1` is an old subchunk and records it as a chunk `C1'` which is a part of old chunk `C1`. We call such parts as *chunk-parts*, where each chunk-part contains

```
25 Procedure fingerdiff
26 Inputs: File f, Integer exp_sc_size , Integer max_scs
27 Output: List CL of Chunks
28 BEGIN
29    ChunkList CL := empty;
30    SubChunkList SL := CDC(f,exp_sc_size);
31    SubChunk SC := SL.next();
32    Type currentChunkType := lookup(SC);
33    while SL ≠ empty do
34          Chunk C := new Chunk();
35          if (currentChunkType = new)
36          then
37                C.type := new;
38                while (currentChunkType = new and
39                numSubchunks(C) < max_scs) do
40                      C.add(SC);
41                      SC := SL.next();
42                      currentChunkType := lookup(SC);
43                endwhile
44          else
45                C.type = old;
46                while (currentChunkType = old and
47                isContiguous(SC) do )
48                      C.add(SC);
49                      SC := SL.next();
50                      currentChunkType := lookup(SC);
51                endwhile
52          endif
53          if (C.type = new)
54          then
55                foreach Subchunk SC in C do
56                      size := sizeof(SC);
57                      offset := getOffset(C,SC);
58                      updateLookup(SC,C, offset,size);
59                endfor
60          endif
61          CL.add(C);
62    endwhile
63    return CL;
64 END
```

Fig. 4. The *fingerdiff* algorithm

one or more subchunks but not a whole chunk. It finds that `SC2'` is a new subchunk which was not seen before and therefore writes this as a new chunk `C3`. It finds that `SC3` through `SC16` are old subchunks that belong to old chunk `C1` and therefore coalesces these into chunk `C1''` which is a partial chunk that is part of old chunk `C1`. Similarly, it coalesces subchunks `SC17` through `SC32` as old chunk `C2`. Note that `C1'` and `C1''` are parts of an old chunk `C1`, and start at an offset in `C1`. This offset has to be maintained along with the key and size of `C1` in order to read these parts from the store. Since only `C3` is new, it is the only chunk written to the store. The remaining chunks are all either old chunks that were previously written or parts of old chunks that were previously written to the store. The output of *fingerdiff* after having written two versions of the file `F` to the store contains

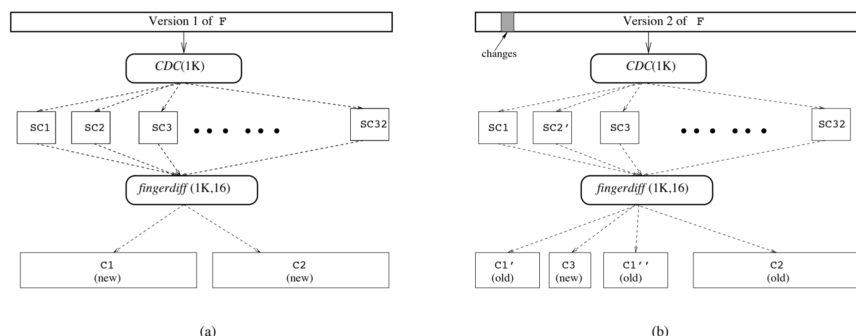

Fig. 5. An example of *fingerdiff* being employed to encode two consecutive versions of a file.

only 3 chunks, as opposed to *CDC* whose output contains 33 chunks. The storage savings is due to the fact that the backend has to maintain metadata for only 3 chunks in *fingerdiff* as opposed to 33 chunks in *CDC*. In our experiments, we show that this difference can be crucial.

3.2.2.2 *The fingerdiff algorithm:*. The *fingerdiff* algorithm operates with two parameters; an *exp_sc_size* parameter that is the expected subchunk size, which is similar to the *exp_chunk_size* parameter used by *CDC*, and a *max_scs* parameter that is the maximum number of subchunks that can be contained in one chunk. A *subchunk* is therefore contained in a chunk at a given offset. The chunk that contains a subchunk is referred to as the subchunk's *superchunk*.

The algorithm is illustrated in Figure 4. It takes as input a file *f* that has to be chunked and the parameters, *exp_sc_size* and *max_scs* and returns a list of chunks or chunk-parts. Once the chunks are returned, those chunks that are marked *new* are written to the store. All the chunks and chunk-parts are recorded in a metadata block using their ⟨*chunk-key, size, offset*⟩ information. Depending on the design of the application, this metadata block can also be written to the store and its key can be maintained as a pointer to this particular version of the file.

The algorithm description hides the following details:

(1) The *lookup* procedure called on lines 8, 18 and 26 uses an auxiliary data structure that records information about subchunks. If a match is found, the lookup procedure returns the type as *old*; otherwise it returns the type as *new*.
(2) The *isContiguous* function called on line 23 ensures that the current subchunk being processed is contiguous with the previous subchunk that was processed; i.e they have the same superchunk and that the current subchunk appears immediately after the previous subchunk that was processed in that superchunk. In case some subchunk appears in multiple superchunks, the algorithm maps it to the first superchunk it appeared in. By checking for the order of subchunks in a superchunk, the *isContiguous* function ensures that this mapping is never changed.
(3) The *SL.next*() function called on lines 7, 17 and 25 has the effect of removing the next subchunk from the list and returning that subchunk.
(4) The *numSubchunks*(*C*) function called on line 15 returns the number of subchunks

currently present in chunk *C*.

The algorithm begins by invoking *CDC* (line 6) with an expected chunk size value equal to the *exp_sc_size* to obtain a sequence of subchunks. In practice the list of subchunks can be greater than what can be returned in one procedure call. The *fingerdiff* implementation can handle this by calling *CDC* in batches, retrieving subchunks for a portion of the object per batch.

The key intuition here is that the implementation can assume a lower *exp_sc_size* value than the expected chunk size assumed in an implementation of *CDC*. This is because after calling *CDC*, *fingerdiff* will merge the resultant subchunks into larger chunks wherever possible before writing them to the store. Lines 11 through 28 coalesce contiguous subchunks into chunks that are either *new* or *old* depending on whether or not the local lookup for them succeed. Line 14 ensures that the number of subchunks in a new chunk does not exceed *max_scs*. Lines 22 and 23 ensure that old subchunks are coalesced only if they belong to the same superchunk and if they again appear in the same order as they did in their superchunk. Lines 29 through 36 add information about the new subchunks to a client-local data structure that is consulted by the lookup procedure.

Once *fingerdiff* returns, the encoder program only writes the new chunks to the store. The old chunks are remembered as a ⟨*superchunk-key,offset,size*⟩ tuple. To read an old chunk, the superchunk-key, offset and size information is provided to the backend to exactly read the chunk or chunk-part required.

3.2.2.3 *Implementation.* In order to compare *fingerdiff* with other object partitioning techniques, we implemented a chunk store that records the hash of each chunk, along with its size and offset in a packing based storage system. We also implemented a file client that reads files and directories and writes it to a file server. The file server implements an object partitioning technique. This technique is either *FSP, CDC* or *fingerdiff*. In case of *fingerdiff*, the file server maintains object specific tables, where each table contains hashes of all subchunks seen in all previous versions of a given object. This table is pulled into memory, whenever a corresponding file is being updated. The subchunks are computed using a *CDC* implementation that identifies chunk boundaries by computing Rabin's fingerprints on a sliding window of 32 bit substrings of the file. For each partitioned object (e.g. file), there is a tree containing information about all the subchunks of all the versions of that object that have been written so far. The information about each subchunk includes:

—The hash of the subchunk.

—The hash of the subchunk's superchunk.

—The offset of the subchunk in its superchunk.

—The size of the subchunk.

The tree itself is indexed using the hash of the subchunk. All hashes are computed using an implementation of the standard SHA-1 algorithm. The tree is stored persistently on disk. Another tree is used to maintain a mapping between the object being chunked and its corresponding lookup tree.

A lookup tree is read from disk whenever its corresponding object is being chunked. Maintaining a separate lookup tree for each object improves the time to lookup information about subchunks of each object, but it does eliminate the possibility of cross-object duplicate elimination. However, note that if separate clients write the same object through

different file servers (and different *fingerdiff* drivers) to the storage repository, content-addressing at the storage engine will still ensure only one set of chunks for the identical objects are stored. Only in scenarios where unrelated clients modify the same object in different ways is it possible for the storage system to not identify duplicates in an efficient manner.

3.2.2.4 *Lookup management in fingerdiff*. We use the hash of a subchunk as a key into the lookup structure in order to obtain information about that subchunk. The lookup operation introduces overheads in *fingerdiff* both in terms of the space required for the lookup and the time needed to read and insert subchunk information. As we reduce the expected subchunk size of *fingerdiff* more subchunks are created, increasing the lookup size.

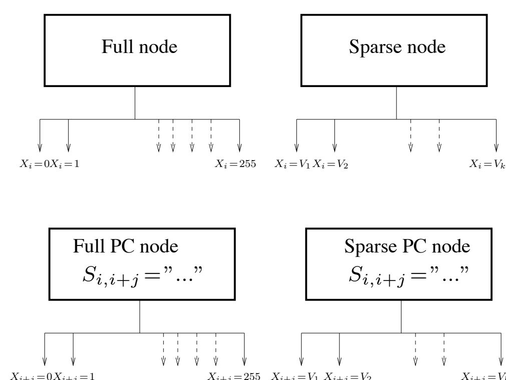

Fig. 6. Different i^{th} level nodes that make up the *fingerdiff* tree. In case of ith level full and sparse nodes, the ith byte X_i of the key is used to decide the node to lookup at the $i+1^{th}$ level. In case of PC nodes the $i+j^{th}$ byte X_{i+j} is used to make this decision, provided the substring $X_{i,i+j}$ of the key is same as the substring $S_{i,i+j}$ stored in the PC node.

We originally implemented the data structures for *fingerdiff* lookup using a classical in-memory hash table that given a hash key returns subchunk information. However we found that the time and space overheads of this approach were considerable due to the random distribution of SHA-1 hash-key values and the rapid growth in the amount of information that had to be stored. Essentially, related or similar subchunks can have completely different hash-key values due to the uniform hashing of the SHA-1 function. Consequently, two or more hash keys for unrelated subchunks may contain common substrings. The amount of commonality and the range in the 20 byte key at which this commonality occurs depends solely on the contents of data being written and changes dynamically with object updates. We would like to dynamically adjust to this commonality in key space in order to avoid storing repeated substrings in the lookup structure without increasing the time to perform the search.

We designed an in-memory data structure for *fingerdiff* lookup by taking this observation into account. This data structure is a variant of a digital search tree that given the hash of a subchunk returns subchunk information. The tree contains different types of nodes at each level. Based on the ith byte of the key and the node at the ith level, the algorithm decides which node to consult at the $i+1^{th}$ level. This node will be a child of the ith node. For

20 byte SHA-1 hashes, the tree has 20 levels. The leaf nodes in this tree contain subchunk information.

Nodes can be of two types – *full* nodes and *sparse* nodes. Full nodes are nodes that contain children for 128 or more possible values of the byte at a particular level. Since a byte has a possible maximum of 256 values, some children of a full node may be null values, indicating that the corresponding subchunks do not exist. Sparse nodes are nodes that contain $k < 128$ children. Each child corresponds to a particular byte value that has been seen at this node and at this level. The children are sorted based on the values $V_1, ...V_k$ of the byte that have been seen at this level. A new byte value at this level not belonging to the set $V_1, ...V_k$ indicates that the corresponding subchunk is new.

In cases where a subtree is linear, i.e. where nodes in several consecutive levels of the tree have only one child, the entire substring corresponding to that path is stored at the root of the subtree. Such a root node is called a path compressed or PC node. PC nodes are also either full nodes or sparse nodes depending on the number of bytes seen at the $i + j^{th}$ level of a ith level PC node with a substring of length j.

	No. of Versions	Version No. or Snapshot date.		Size of tarred version (MB)		No. of files in version	
		First	Last	First	Last	First	Last
Sources							
gcc	20	2.95.0	3.4.0	56	191	2771	21817
gdb	10	5.0	6.3	56	88	3771	5255
emacs	8	20.1	21.3	46	73	1967	2553
linux	10	2.6.0	2.6.9	179	196	15007	16448
Binaries							
gaim	365	01/01/04	12/31/04	46	47	140	143
Databases							
freedb	11	02/01/2003	01/01/2004	177	295	102627	155153

Table I. Characteristics of the first and last version of each benchmark

4. EXPERIMENTAL FRAMEWORK

An important goal of this work is to measure the effectiveness of chunking techniques including *fingerdiff* in eliminating duplicates in a content addressable storage system with specific emphasis on applications that write consecutive versions of the same object to the storage system. But apart from storage space utilization, we also measured the bandwidth utilization, the number of chunks generated and other chunk related management overheads for different chunking techniques.

4.1 Benchmarks

We used three classes of work loads to compare *fingerdiff* with *CDC*. The first one, **Sources**, contains a set of consecutive versions of source code of real software systems. This includes versions of gnu gcc, gnu gdb, gnu emacs and the linux kernel. The second class, **Databases** contains periodic snapshots of information about different music categories from the Freedb database obtained from www.freedb.org. Freedb is a database of compact

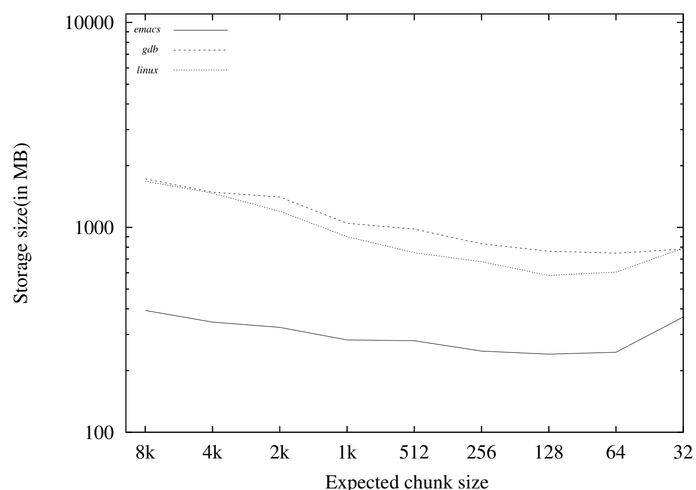

Fig. 7. Backend storage space consumed after writing all versions of a benchmark for different expected chunk sizes of *CDC*. The X-axis is a log 10 scale

disc track listings that holds information for over one million CDs. Freedb allows an indexing structure, whereby to lookup CD information, clients can calculate a nearly unique disc ID and then query the database. For our experiments, we obtained 11 monthly snapshots of *freedb* during the year 2003 for the jazz, classical and rock categories. These snapshots were created by processing all the updates that were made each month to the *freedb* site. The third class, **Binaries** contains executables and object files obtained by compiling daily snapshots of the *gaim* internet chat client being developed at http://sourceforge.net taken from the cvs tree for the year 2004. While the **Sources** and **Databases** classes of benchmarks contains versions at well defined release or update points, the *gaim* benchmark contains all data that existed at the end of the work day for all days of the year 2004. As a result, the *gaim* benchmark has a total of exactly 365 snapshots. Therefore the *gaim* benchmark represents a different object modification characteristic from the rest: each modification in *gaim* is incremental in nature while modifications are more frequent, whereas each modification in the other benchmarks contain all the changes of a new release, but modifications are few and far between. Table I enumerates the characteristics of the first and last version of each of our benchmarks.

5. RESULTS

Different instantiations of *FSP*,*CDC* and *fingerdiff* are possible depending on the fixed block size of *FSP*, the *exp_chunk_size* of *CDC* and the *exp_sc_size* of *fingerdiff* as discussed so far. We use the following terminology to define *CDC* and *fingerdiff* instantiations:

— A *cdc-*x instantiation is a content defined chunking strategy with an *exp_chunk_size* of x bytes;

— A *fd-*x instantiation is a *fingerdiff* instantiation with a *exp_sc_size* of x bytes and *max_scs* of 32 KB.

	Sources				**Databases**	**Binaries**	**Total**
benchmark	*gcc*	*gdb*	*emacs*	*linux*	*freedb*	*gaim*	
cdc-2k	1414	501	327	1204	396	225	4067
cdc-256	866	363	258	708	348	245	2788
cdc-128	828	344	259	629	369	301	2731
cdc-64	859	358	281	692	442	447	3079
cdc-32	979	500	457	985	644	527	4090
fd-2k	1400	498	324	1195	370	213	3999
fd-256	799	336	239	644	396	196	2611
fd-128	680	293	220	520	317	208	2238
fd-64	579	255	199	469	291	244	2038
fd-32	498	255	221	543	290	246	2052
% saving	40	26	25	23	17	13	25

Table II. Comparison of the total storage space consumed (in MB) by the ten chunking technique instantiations after writing each benchmark on a content addressable chunk store. The last column gives the % savings of the best *fingerdiff* technique over the best *CDC* technique for each benchmark.

A storage driver partitions each version or snapshot of the benchmark using one of the chunking instantiations and writes each chunk asynchronously to a content-based storage backend. Asynchronous chunking ensures that applications do not have to wait for the chunking operation to be completed upon each write.

We chose to exclude *FSP* based instantiations from our experiments as it has been well documented[Policroniades and Pratt 2004] that *CDC* instantiations exploit commonality of data better than *FSP* instantiations.

We calculate storage utilization of a chunking technique instantiation for a particular benchmark by storing consecutive versions of the benchmark after chunking it into variable sized chunks using that instantiation.

The total storage space is calculated by adding the space consumed by the benchmark data on the chunk store backend (*backend storage utilization*) and the lookup space required for a given benchmark on the object server (*local storage utilization*). The backend storage space consists of data and metadata chunks for the benchmarks along with the cost of storing a pointer for each chunk. We calculate this cost to be 32 bytes(20 bytes for SHA-1 pointers plus 12 bytes to maintain variable-sized blocks through packing). The local lookup space is used on the driver to support *fingerdiff* and *CDC* chunking. This lookup is a tree that maps hashes of subchunks of an object to information about that subchunk. This tree resides in disk persistently, but is pulled into memory when an object is being updated and has to be partitioned. As can be expected, this tree grows as more versions of the object are written to the store. We measure the size of the tree for all our *fingerdiff* instantiations. The lookup space is measured as the total space occupied by the lookup tree for each benchmark in the local disk.

Note that if a replication strategy is used for improved availability, the *backend storage utilization* will proportionately increase with the number of replicas but the *local storage utilization* will remain constant for any number of replicas.

We observed that the *backend storage utilization* of *CDC* peaked at an expected chunk size of either 128 bytes or 256 bytes for all benchmarks. (Figure 7 illustrates this phenomenon for three benchmarks in **Sources**.) This is because, as we decrease the expected chunk size of *CDC* in order to improve chunking granularity, the number of chunks generated increases, which in turn increases the cost of maintaining a pointer per chunk (32 bytes). As a result, the increased duplicate elimination due to improved granularity is offset by the cost of storing an increased number of chunks.

Note that the functionality of *fingerdiff* necessitates a local lookup; On the other hand *CDC* can function without one. The use of a local lookup in a *CDC* technique will impose a local lookup space overhead, whereas a *CDC* technique without one will incur extra bandwidth overhead as *every chunk* will have to be sent to the chunk store over the network.

For our experiments we assume that *CDC* techniques also maintain a local lookup to avoid incurring a heavy bandwidth overhead of transferring every chunk to the server. We compare the total storage consumed (*backend storage utilization* + *local lookup utilization*) of five *fingerdiff* instantiations and five *CDC* instantiations. We limit the *CDC* instantiations for which we show results to *cdc-2k*, *cdc-256*, *cdc-128*, *cdc-64* and *cdc-32*. We compare these with five *fingerdiff* instantiations namely *fd-2k*, *fd-256*, *fd-128*, *fd-64* and *fd-32*. Note that many more instantiations are possible, but we limit our presentation in order to reduce the clutter in our tables and graphs, while ensuring that the broad trends involved with changing chunk sizes are clear.

5.1 Total storage space consumed

	Sources				**Databases**	**Binaries**
benchmark	*gcc*	*gdb*	*emacs*	*linux*	*freedb*	*gaim*
cdc-2k	412	112	84	289	221	193
cdc-256	1880	677	496	1472	1046	1257
cdc-128	3324	1280	971	2467	1799	2310
cdc-64	5799	2325	1864	4579	3616	4589
cdc-32	10204	5231	4804	10202	8226	5760
fd-2k	51	13	10	57	31	22
fd-256	360	76	60	244	419	69
fd-128	642	174	121	515	706	119
fd-64	732	240	203	562	881	231
fd-32	1177	551	398	1059	1309	234

Table III. Comparison of the number of chunks (in thousands) generated by the seven chunking technique instantiations while writing the different benchmarks to a content addressable store.

The storage space consumed by each chunking technique reflects the amount of storage space saved by leveraging duplicate elimination on the store. The technique which best utilizes duplicate elimination can be expected to consume the least storage space. Table 2 compares the total (backend+local) storage utilization achieved on account of duplicate elimination after individually storing all our benchmarks for all ten chunking instantiations.

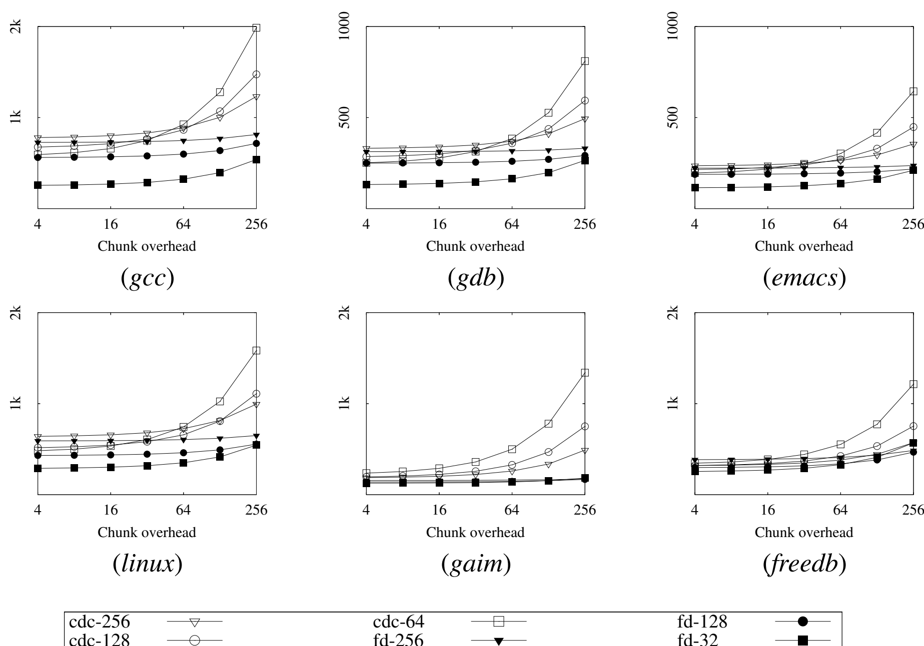

Fig. 8. Comparison of the total network traffic (in MB) consumed by six of the ten chunking technique instantiations after writing each benchmark on a content addressable chunk store. The X-axis of each graph is a log plot which gives the chunk overhead; i.e the overhead in bytes associated with transferring one chunk of data from the driver to the chunk store. The network traffic measured is between the object server and the chunk store. The Y-axis gives the total network traffic generated in MB after writing each benchmark to the chunk store.

For all benchmarks (except *gaim*) either *fd-32* of *fd-64* consumes the least and *cdc-32* the most storage. In case of *gaim fd-256* consumes the least storage. Among the *CDC* instantiations, either *cdc-128* or *cdc-256* gives the best storage utilization. Decreasing the chunk size of *CDC* to 64 or 32 increases total storage consumption for all benchmarks.

However for most benchmarks, reducing the expected subchunk size of *fingerdiff* to 64 or 32 bytes helps us to increase the granularity of duplicate elimination without incurring the storage space overheads of too many small chunks. The last column (% savings) in table 2 gives the savings achieved by the best *fingerdiff* (in most cases *fd-32* or *fd-64*) instantiation over the best *CDC* instantiation (either *cdc-128* or *cdc-256*). In spite of the large number of hashes for subchunks maintained in *fingerdiff* drivers, *fingerdiff* improves the storage utilization of the best *CDC* . For example, *fd-32* improves backend storage utilization of the best *CDC* by a significant percentage for all benchmarks that we measured. This improvement varied from 13% for *gaim* to up to 40% for *gcc*. The last row in table 2 gives the total storage consumed after writing all the benchmarks to the chunk store. Here, we observed that *fd-64* gives the best storage utilization. It improves upon the storage utilization of the best *CDC* technique (*cdc-128*) by 25%.

5.2 Number of chunks

From the storage system point of view we would like to have as few chunks as possible to reduce the management cost associated with each chunk. These overheads include at least one 20 byte pointer per chunk. Depending on the storage architecture, the overheads could also involve one disk request per chunk on reads, and one network request per chunk from either a client to the server or a peer to another on reads and writes. Table 3 shows the number of chunks(in thousands) that were generated by each chunking technique after all our benchmarks were written to a content addressable store.

Cdc-32 and *fd-2k* generate the maximum and minimum number of chunks respectively for both the *emacs* and *gaim* benchmarks.

As expected, the trend we observe here is that as we reduce the *exp_chunk_size* for *CDC* and the *exp_sc_size* for *fingerdiff*, the number of chunks generated increases.

These results reflect the inherent tension between storage consumption and chunk overhead, i.e trying to improve granularity of chunking inevitably increases the number of chunks generated. *Fingerdiff* however resists this trend more strongly than *CDC*. As a result we have *fingerdiff* instantiations that strike a better balance between the two attributes. For example, for all benchmarks *fd-256* gives us better storage utilization than any *CDC* instantiation, while generating fewer chunks than *cdc-256*.

5.3 Total network bandwidth consumed

Once the object server identifies the chunks that are new in each update, it sends each new chunk to the chunk store along with necessary metadata for each chunk. In our model, this metadata must include the size of the chunk (necessary to support variable sized chunks), imposing an overhead of 4 bytes for every chunk that is sent. Based on this we calculated the average bandwidth savings of the best *fingerdiff* technique over the best *cdc* technique for all benchmarks to be 40%.

However other models might require extra metadata. For example, a model akin to the the low bandwidth file system[Muthitacharoen et al. 2001] where the server also maintains object information might require the client to send the file descriptor along with each chunk. Peer to peer architectures might require the client to check the existence of each hash with the chunk store[Cox et al. 2002]. In general, chunking techniques that generate more chunks will send more traffic over the network, the exact amount of which will depend on the network protocol and the system model. Figure 7 illustrates the amount of network bandwidth consumed by different instantiations for all benchmarks for a varying amount of metadata traffic overhead per chunk. For each benchmark the per-chunk overhead is varied from 4 bytes to 256 bytes. Observe that for all benchmarks, a chunk overhead as low as 4 bytes results in substantial bandwidth savings for the best *fingerdiff* instantiations over all the *CDC* instantiations. Note that to preserve clarity of our graphs, we plot only 3 instantiations from *fingerdiff* and 3 from *CDC*. However note that we do plot *cdc-128* and *cdc-256* which formed the most efficient *CDC* instantiations for all benchmarks. Also observe that the instantiations that generate more number of chunks (i.e the *CDC* instantiations) consume more bandwidth as the per-chunk overhead is increased from 4 to 256. We conclude that *fingerdiff* substantially improves upon the bandwidth utilization of *CDC*.

5.4 Erasure coded stores

We have shown that when increasing the variability of chunk sizes, *fingerdiff* generates fewer number of chunks than *CDC* for a given level of duplicate elimination. It therefore reduces management overheads associated with storing and maintaining every chunk in the system.

Systems such as Oceanstore [Kubiatowicz et al. 2000] and Intermemory[Goldberg and Yianilos 1998] propose the use of erasure codes[Berlekamp 1968; Blomer et al. 1995; Weatherspoon and Kubiatowicz 2002] instead of replication[Lv et al. 2002] for guaranteeing data availability. In such systems, a data block is divided into m equally-sized fragments and these m fragments are encoded into n fragments (where $n > m$). These n fragments can be dispersed across n or less nodes in a potentially distributed system. Unlike replication, erasure codes allows for increase in availability without a proportional blowup in storage space of data. Availability can be improved by increasing both m and n, but as long as the ratio m/n is kept constant, the space consumption of data remains the same. However for each fragment that is stored, there is at least one reference to that fragment. In a content based storage where we use SHA-1 pointers for reference, we would need at least 20 bytes per each new fragment.

Since erasure coded stores maintain metadata per each fragment, the overall size of metadata is much greater than in regular storage systems. Further this size increases not just with the size of the data, but also with the number of chunks used to represent that data. Since *fingerdiff* generates fewer chunks than *CDC* while partitioning the same data, we expect *fingerdiff* techniques to incur smaller overheads in erasure coded stores. Figure 8 measures the growth in storage space for six chunking technique instantiations (three *CDC* and three *fingerdiff*) and for all benchmarks, as we increase the number of encoded fragments for each block from 8 to 64. For each of the graphs, the Y-axis is a log 10 plot that measures the total storage space in MB for a given number of encoded fragments. Observe that for all benchmarks, the instantiation that results in the most number of chunks(*cdc-64*) experiences the fastest rate of growth in storage space as we increase the number of fragments per block. We conclude that as the number of encoded fragments is increased, eventually the instantiation which generates more chunks will consume more storage than techniques which generate fewer chunks. In general, *fingerdiff* provides a given level of duplicate elimination by generating fewer chunks than *CDC*. This makes it more efficient in erasure coded stores.

6. DISCUSSION

It has been well-documented that *CDC* provides better duplicate elimination than *FSP* techniques[Policroniades and Pratt 2004]. However, as we have shown in Figure 7, the backend storage utilization of *CDC* peaks for a particular chunk size, making it impossible to improve storage utilization over this peak value through *CDC* alone. We have shown that *fingerdiff*, by coalescing smaller chunks into larger ones wherever possible, breaks this barrier and allows far greater storage utilization than the best *CDC* instantiation. We have also shown the conflicting nature of the two characteristics associated with storage systems– the total storage space and the number of chunks generated. In order to further highlight this conflict, and show the role of *fingerdiff* in balancing the two, we plot one against the other. Figure 10 plots backend storage consumption as a function of the number of chunks generated by three *CDC* and three *fingerdiff* instantiations for all our benchmarks. Each

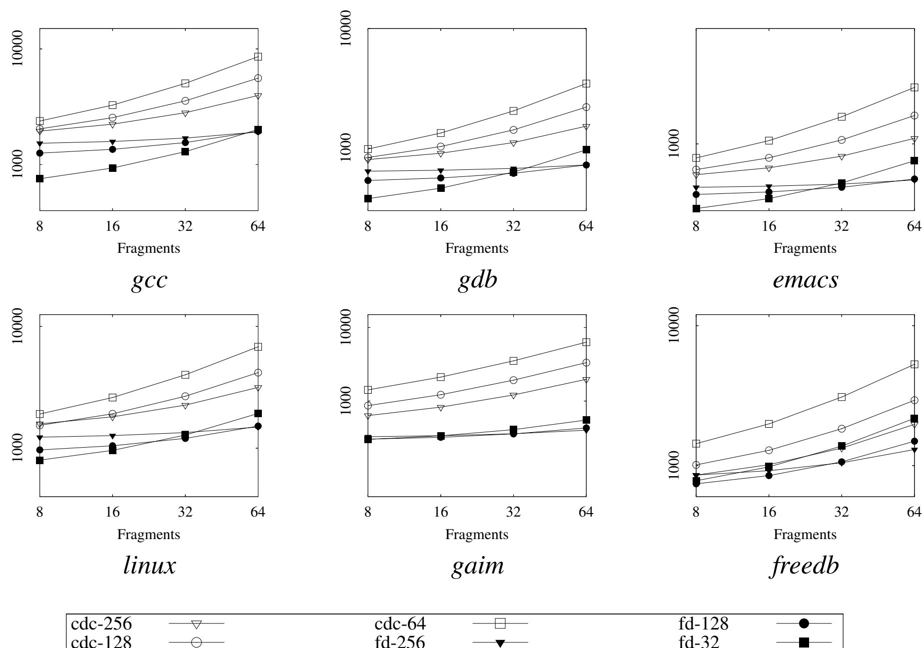

Fig. 9. Storage space consumed after storing all versions of all benchmarks on an erasure coded store as the number of fragments each block is encoded into (i.e n value) is increased from 8 to 64($m = n/2$). The X-axis is a log 2 plot that indicates the number of fragments that each block is encoded into, and the Y-axis is a log 10 plot that shows the total storage consumed by both data and metadata (pointer references) in MB.

point in the lines of figure 10 represents a version release in the corresponding benchmark. In case of *gaim* and *freedb*, each point represents a end of month snapshot. In this graph a line going up (parallel to Y-axis) indicates storage space growth, whereas a line going wide (parallel to X-axis) indicates growth in the number of chunks. A shorter line implies that the corresponding instantiation controlled the rate of growth of both storage space and number of chunks better than one with a longer line. The shortest line for all the graphs are those of *fingerdiff* instantiations; *fd-256* for *gaim* and *emacs*, and *fd-128* for the rest, emphasizing our point that *fingerdiff* finds a better balance between the two conflicting attributes.

The improved storage efficiency of *fingerdiff* comes with a cost. A local lookup that maintains information about each of the subchunks that have been written so far must be maintained. Note, however, that the lookup need not be maintained with the same availability and persistence guarantees as data on the storage end. Losing information stored in the lookup to a disk failure will not result in catastrophic loss of data; at worst, it will result in lower storage utilization on the backend because of sub-optimal duplicate elimination.

In our implementation, a separate lookup is maintained for every object that is being updated. While this ensures that no single lookup becomes too large, it does not allow for the *fingerdiff* driver to identify duplicates across two different objects. However chunks

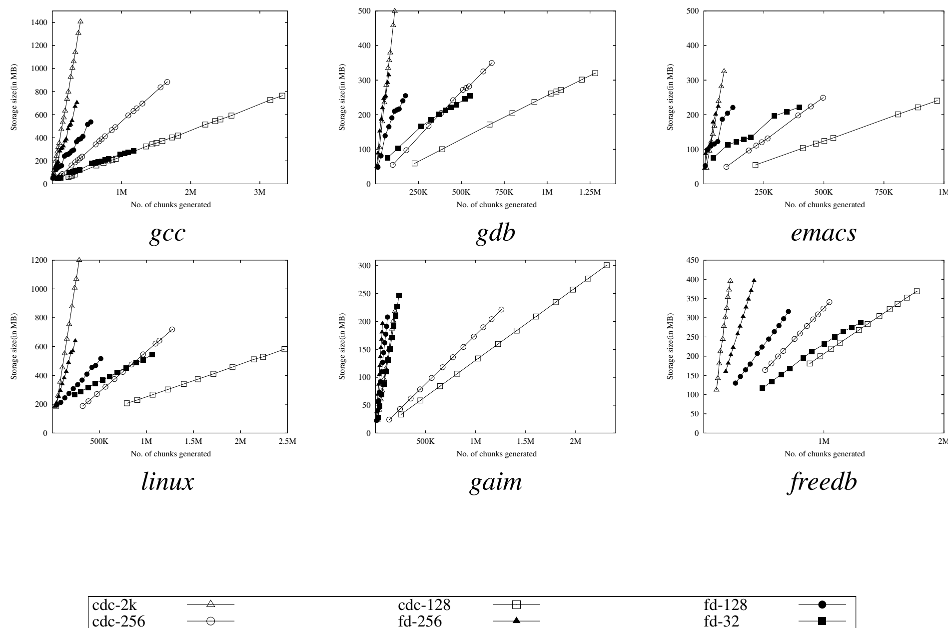

Fig. 10. The storage utilization as a function of the number of chunks generated while writing all the benchmarks to a content addressable chunk store.

belonging to identical objects that enter the system via different drivers will still be likely to get eliminated in the chunk store because of its content based nature. Only in rare cases where different applications modify identical objects in separate ways and then send the respective updates to the store via different drivers will it be possible that the storage system will fail to identify duplicate data in an efficient way. Also note that in our storage model, having one large lookup for all objects will allow for such cross-object duplicate suppression and also eliminate the need for the chunking instantiation to be aware of which object is being updated. But such a lookup structure will grow quickly and will have to be efficiently managed both in memory and in disk. We are currently working to ensure that such a structure can work in bounded memory and that different parts of the lookup can be paged in and out of disk efficiently. This in itself is an interesting problem because the lookup is based on hashes of subchunks, and since uniform hashing ensures that related subchunks have totally unrelated hashes, information for related subchunks are dispersed throughout the lookup structure making it difficult to page them collectively.

The SHA-1[National Institute of Standards and Technology, FIPS 180-1 1995] hashing function that we use ensures that the probability of collision is much lower than that of a mechanical disk failure in storage systems[Quinlan and Dorwards 2002]. However it is conceivable that SHA-1 can be broken in the future, making it easier to provide two independent data chunks with different content that have the same hash. The alternative for content-based storage systems will be to employ hashing algorithms such as SHA-224, SHA-256 and SHA-512 that generate larger hash keys than SHA-1 and further reduce the probability of finding collisions. Larger key values will have larger metadata overheads per chunk. Since *fingerdiff* provides a given level of storage utilization while generating fewer

chunks than other chunking techniques, we believe that *fingerdiff* based systems will pay a smaller penalty in terms of storage utilization while transitioning from SHA-1 to more complex hash functions.

7. RELATED WORK

Fingerprints have been proposed to identify similar documents[Broder 1997; 2000; Manber 1994] in a large set of unrelated documents. Similarity detection has various applications in domains such as copy-detection[Shivakumar and García-Molina 1995] and web clustering[Broder et al. 1997]. Among fingerprinting techniques, a specific type, known as Rabin's fingerprints[Rabin 1981] has been used extensively for implementing fingerprint-based software systems. The chief advantage of Rabin fingerprints is that they are very easy to compute over a sliding window of substrings in a document. Thus the cost of computing fingerprints for an entire document containing l substrings is much less than l times the cost of computing the fingerprint of one substring.

Duplicate elimination (sometimes also referred to as duplicate suppression elsewhere), differs from this area of research as it aims to eliminate redundancy due to identical (and not similar) objects or blocks by comparing hashes of the object's or block's content[Hong et al. 2004; Kubiatowicz et al. 2000; Quinlan and Dorwards 2002; W. J. Bolosky and Douceur]. In these schemes, objects are hashed in their entirety or divided into fixed sized blocks (*FSC*) and each block is then hashed. Fingerprints can be used to identify not only documents, but also offsets inside documents that determine where blocks can be divided. Once blocks have been identified, they can be hashed using robust hashing algorithms such as SHA-1[National Institute of Standards and Technology, FIPS 180-1 1995]; this hash can then be used for duplicate elimination. Such content defined chunking(*CDC*) schemes are used in the LBFS file system[Muthitacharoen et al. 2001] to reduce bandwidth requirements between storage clients and servers by reducing the amount of data that has to travel across the network. LBFS maintains state information on the client side and uses a technique similar to *cdc-8k* in order to identify and send only those chunks that are new in the modified version. There is a direct correlation between the amount of bandwidth that can be saved in such systems and the amount of storage space that can be gained due to duplicate elimination. Since we have shown that *fingerdiff* significantly improves the storage utilization over *CDC*, we believe that using *fingerdiff* over *CDC* in systems such as LBFS can further reduce bandwidth requirements of the network.

CDC is also used in pastiche[Cox et al. 2002], in order to identify backup buddies in a peer-to-peer system. Previous work has also compared *CDC* with fixed sized chunking schemes[Policroniades and Pratt 2004]. Not surprisingly, it was concluded that *CDC* outperforms *FSC* with respect to storage utilization.

Delta encoding[Ajtai et al. 2000; Hunt et al. 1998; Tichy 1984] is a technique that attempts to encode the difference between two given strings (or objects) in the most efficient way possible. This technique is used extensively in versioning systems such as CVS[Cederqvist 1992], SCCS[Rochkind 1975] and RCS[Tichy 1985]. By storing only the changes made to consecutive versions, delta encoding can reduce storage overheads. Delta encoding has also been extended to pairs of objects that do not share an explicit versioning relationship[Douglis and Iyengar 2003; Ouyang et al.]. In these systems similarity detection on a vast collection of unrelated documents is applied in order to identify candidate pairs for encoding. In[Douglis et al. 2004], Kulkarni et.al combine these techniques to first

eliminate identical objects and blocks; they then identify similar blocks in the remaining set and apply delta encoding on those blocks. A similar tiered approach is taken in [Jain et al. 2005] to efficiently synchronize replicas. Restoring versions in systems that rely on delta encoding however can be complicated as it may involve reading a previous fixed version along with a chain of changes and decoding the required version from the previous version and the delta chain. In this study, we focus on object partitioning techniques that simply divide objects into variable sized blocks. Restoring a given version in such schemes will only involve reading all the individual blocks that comprise that version and reassembling them.

Finally, data compression techniques[Lelewer and Hirschberg 1987; Ziv and Lempel 1977] eliminate redundancy internal to an object and generally reduce textual data by a factor of two to six. We can leverage data compression techniques by compressing chunks that are output by our object partitioning technique. We expect to benefit from compression just as any other object partitioning technique would.

8. CONCLUSIONS

Existing object partitioning techniques cannot improve storage and bandwidth utilization without significantly increasing the storage management overheads imposed on the system. This observation motivated us to discover a chunking technique that would improve duplicate elimination over existing techniques without increasing associated overheads.

We have proposed a new chunking algorithm *fingerdiff* that improves upon the best storage and bandwidth utilization of *CDC* while lowering the overheads it imposes on the storage system. We have measured storage and bandwidth consumption along with associated overheads of several *CDC* and *fingerdiff* instantiations as they write a series of versions of several real-world software systems to a content addressable store. For both these benchmarks, we show that *fingerdiff* significantly improves the storage and bandwidth utilization of the best *CDC* instantiation while also reducing the rate of increase in storage overheads.

Our contention is not that a particular *fingerdiff* technique is the best choice in all content based storage engines. But, by allowing for greater variability of block sizes, and by being able to better localize the changes made to consecutive object versions into smaller chunks, *fingerdiff* is able to minimize the size of new data introduced with every version, while keeping the average size of all chunks relatively large. This in turn allows it to provide the best storage and bandwidth utilization for a given amount of management overhead.

REFERENCES

AJTAI, M., BURNS, R., FAGIN, R., LONG, D., AND STOCKMEYER, L. 2000. Compactly encoding unstructured input with differential compression. In *IBM Research Report RJ 10187*.

BERLEKAMP, E. R. 1968. *Algebraic Coding Theory*. McGraw Hill.

BLOMER, J., KALFANE, M., KARP, R., KARPINSKI, M., LUBY, M., AND ZUCKERMAN, D. 1995. An xor-based erasure-resilient coding scheme. Technical report, International Computer Science Institute, Berkeley, California.

BRODER, A. 1997. On the resemblance and containment of documents. In *SEQUENCES '97: Proceedings of the Compression and Complexity of Sequences 1997*. IEEE Computer Society, 21.

BRODER, A., GLASSMAN, S., MANASSE, M., AND ZWEIG, G. 1997. Syntactic clustering of the web. In *Proc. of the 6th International WWW Conference*. 391–404.

BRODER, A. Z. 2000. Identifying and filtering near-duplicate documents. In *COM '00: Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*. Springer-Verlag, 1–10.

CEDERQVIST, P. 1992. Version management with cvs. http://www.cvshome.org/docs/manual/.

COX, L., MURRAY, C., AND NOBLE, B. 2002. Pastiche: Making backup cheap and easy. In *Proceedings of Fifth USENIX Symposium on Operating Systems Design and Implementation*. Boston, MA.

DOUGLIS, F. AND IYENGAR, A. 2003. Application-specific deltaencoding via resemblance detection. In *Usenix Annual Technical Conference*. 59–72.

DOUGLIS, P. K. F., LAVOIE, J., AND TRACEY, J. M. 2004. Redundancy elimination within large collections of files. In *Usenix Annual Technical Conference*. 59–72.

GOLDBERG, A. V. AND YIANILOS, P. N. 1998. Towards an archival intermemory. In *IEEE Advances in digital libraries*.

HONG, B., PLANTENBERG, D., LONG, D. D. E., AND SIVAN-ZIMET, M. 2004. Duplicate data elimination in a san file system. In *In Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST 2004)*. 301–314.

HUNT, J. J., VO, K.-P., AND TICHY, W. F. 1998. Delta algorithms an empirical analysis. *ACM Transactions on Software Engineering and Methodology 7,* 2, 192–214.

JAIN, N., DAHLIN, M., AND TEWARI, R. 2005. Taper: Tiered approach for eliminating redundancy in replica sychronization. In *Proceedings of the 4th Usenix Conference on File and Storage Technologies (FAST 2005)*.

KUBIATOWICZ, J., BINDEL, D., CHEN, Y., CZERWINSKI, S., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO, B. 2000. Oceanstore: An Architecture For Global Store Persistent Storage. In *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*. Cambridge, MA.

LELEWER, D. A. AND HIRSCHBERG, D. S. 1987. Data compression. *ACM Computing, Springer Verlag (Heidelberg, FRG and NewYork NY, USA)-Verlag Surveys, ; ACM CR 8902-0069 19,* 3.

LV, Q., CAO, P., COHEN, E., LI, K., AND SHENKER, S. 2002. Search and replication in unstructured peer-to-peer networks. In *ICS '02: Proceedings of the 16th international conference on Supercomputing*. ACM Press, New York, NY, USA, 84–95.

MANBER, U. 1994. Finding Similar Files in a Large File System. In *Usenix Winter Conference*. 1–10.

MUTHITACHAROEN, A., CHEN, B., AND MAZIERES, D. 2001. A low-bandwidth network file system. In *Symposium on Operating Systems Principles*. 174–187.

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY, FIPS 180-1. 1995. Secure hash standard.

OUYANG, Z., MEMON, N., SUEL, T., AND TRENDAFILOV, D. Cluster-based delta compression of a collection of files. In *International Conference on Web Information Systems Engineering (WISE)*.

POLICRONIADES, C. AND PRATT, I. 2004. Alternatives for detecting redundancy in storage systems data. In *Usenix Annual Technical Conference*. 73–86.

QUINLAN, S. AND DORWARDS, S. 2002. Venti: a new approach to archival storage. In *Usenix Conference on File and Storage Technologies*.

RABIN, M. 1981. Fingerprinting by Random Polynomials. Tech. Rep. TR-15-81, Center for Research in Computing Technology, Harvard University.

ROCHKIND, M. J. 1975. The source code control system. *IEEE Trans. on Software Engineering 1(4)*, 364–370.

SHIVAKUMAR, N. AND GARCÍA-MOLINA, H. 1995. SCAM: A copy detection mechanism for digital documents. In *Proceedings of the Second Annual Conference on the Theory and Practice of Digital Libraries*.

TICHY, W. F. 1984. String to string correction problem with block moves. *ACM Transactiosn on Software Engineering 2,* 4 (December), 364–370.

TICHY, W. F. 1985. RCS — a system for version control. *Software — Practice and Experience 15,* 7, 637–654.

W. J. BOLOSKY, S. CORBIN, D. G. AND DOUCEUR, J. R. Single instance storage in windows 2000. In *Usenix Annual Technical Conference*.

WEATHERSPOON, H. AND KUBIATOWICZ, J. 2002. Erasure coding vs. replication: A quantitative comparison. In *First International Workshop on Peer-to-Peer Systems* (Cambridge, MA).

YOU, L. L. AND KARAMANOLIS, C. 2004. Evaluation of efficient archival storage techniques. In *proceedings. of the 21st IEEE Symposium on Mass Storage Systems and Technologies (MSST)*.

ZIV, J. AND LEMPEL, A. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory 23,* 3, 337–343.