Improving Duplicate Elimination in Storage Systems

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

Elimination of redundant data has become a critical concern in the design of storage architectures. Content addressable storage engines eliminate data at the block level by mapping data blocks with the same content to the same physical storage location. Intelligent object partitioning techniques leverage block level content addressing in order to improve duplicate elimination. In this paper, we propose a novel object partitioning technique – *fingerdiff* that is designed to improve storage consumption of existing object partitioning techniques while at the same time reducing associated costs. We present a detailed evaluation of *fingerdiff* and other existing object partitioning schemes, and we show that *fingerdiff* meets its design goals as it improves the effectiveness of block level duplicate elimination while reducing overhead costs.

Categories and Subject Descriptors: [**Distributed Storage**]: Duplicate Elimination

Additional Key Words and Phrases: Rabin's fingerprints, content based addressing, distributed storage systems

1. INTRODUCTION

Storage systems can use content-based addressing to reduce the cost of storage and simplify the process of storage management. Such systems use the hash of an object's contents to effectively serve as a key to identify the object. Content addressability also introduces a unique, transparent and scalable way to name and identify data objects, and to store and search data in a distributed storage system. Because two objects with the same content will have the same key, it is straightforward to eliminate or avoid maintaining duplicate copies of the same content. Eliminating unnecessary redundancy in this fashion improves storage utilization and also reduces the cost of associated storage management.

Duplicate elimination is particularly useful for applications that require periodic backup or archival capability where high-levels of redundancy with previously written versions is expected. Increasing the regularity with which such applications are required to backup all data imposes storage space demands on the storage subsystem that necessitate the use of intelligent redundancy elimination techniques. Systems such as Venti [Quinlan and Dorwards 2002] or Oceanstore [Kubiatowicz et al. 2000] are examples of storage architectures that critically rely on content-based addressing to reduce storage consumption and management costs. These systems are intended to operate seamlessly supporting a variety of storage applications ranging from file systems to databases.

Typically, the unit of storage in a content-addressable system is a *block*; ap-

plication objects such as files are divided into a series of blocks, with each block having a hash key derived from its content. This key is used to map a block to a persistent location. A metadata block for a given version of an object contains enough information required to retrieve the object from the store when required. In a content-addressable store, this information would consist of the hash keys of each block, along with the offset of each block in the object.

When a new object is written into a content-based store, it is partitioned into blocks and each block is stored based on content. When this object is modified and a consequent version is written, the partitioning algorithm again creates blocks and writes them into the store. Some of these blocks will be the same as blocks written previously and the rest will be new. Old blocks are not written to the store because of content-based addressing, while new blocks are written as usual. If the object is modified several times, several versions of the object written written to the store, each time a new version is written, there will be some blocks that are new and add to overall storage space consumption and some blocks that are old and therefore do not consume extra space. Deciding an efficient partition of an object that minimizes the size of the new blocks written for each version is critical to exploiting storage benefits that arise from a content-addressable storage system. Apart from total storage space, there are other overheads associated with writing and storing each block on the storage system. These overheads depend largely on the nature of the storage system itself, and their quantification cannot be generalized. In general, we believe partitioning techniques that create a fewer number of blocks will have smaller overheads than those that create more blocks. However, there is a cutoff that reveals the inherent tension between these two concerns: a system that supports very small block sizes to maximize duplicate elimination may incur unacceptable overheads in terms of storage management, while a system that supports very large block sizes to minimize management overheads may exhibit poor storage utilization characteristics.

We now look at practical techniques that have been deployed in order to store objects into a realistic content-addressable store.

An object may be divided into either *fixed-size* or *variable-sized* blocks. A fixed block-size strategy which we refer to as *fixed-sized chunking* (or *FSC*) is found in systems such as Venti and Oceanstore. In such systems, a fixed block size is chosen *a priori*, independent of the content of the objects being stored, and objects are partitioned into blocks of that size.

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. In spite of this limitation, fixed-size chunking techniques can be effectively utilized in certain scenarios. For example, these techniques are appropriate when it is known that changes are in-place substitutions rather than insertions or deletions, or when changes are appended at the end of the object rather than at the beginning. However, in scenarios where there are arbitrary deletions and insertions, *FSC* techniques can be inadequate in meeting desirable redundancy elimination goals.

Problems with the *FSC* technique can be alleviated to an extent by intelligently

partitioning objects into *variable-sized blocks* or *chunks*. Variable-sized block strategies can potentially localize the effect of changes made to the object, by restricting modifications to the blocks that lie in the vicinity of the changes. This can be achieved by varying the size of the block that contains the changes so as to partition the remaining parts of the object at the same points as before. As a result, blocks outside the change vicinity can be subject to duplicate elimination.

Such a partitioning technique will have to identify the regions of the object that have changed in the current version, and then restrict the new chunks to those regions, making sure that all other chunks remain the same. To do so, it will also have to remember the points in the previous version where the object was partitioned, and partition the current version at the same relative points.

Variable-sized block techniques can use the content of the object to determine partitioning points. One such technique, which we refer to as *Content defined chunking* (*CDC*) employs Rabin's fingerprints to choose these partition points. 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 with every version to regions where changes have been made, keeping all other chunks as is. As a result *CDC* can be expected to outperform *FSC* techniques in terms of storage space utilization on a content-based storage backend. *CDC* is used in the Low-Bandwidth File System (LBFS) [Muthitacharoen et al. 2001] to reduce network traffic between a file server and a client by transferring only those chunks that are new across the network, and in the Pastiche p2p backup system [Cox et al. 2002] to identify backup buddies that share maximum data with a given peer in order to reduce the amount of data to be transferred over the network for purposes of backup. A technique similar to *CDC* is also proposed in [Brin et al. 1995] to identify plagiarism of documents in a digital library.

While *CDC* is a variable-block sized technique, the "variability" of chunks sizes is actually limited, in the sense that most chunk sizes are expected to be within a margin of error from an expected chunk size that is a parameter to the partitioning algorithm. We therefore refer to *CDC* as a *limited* variable-sized block technique. Thus, the size of new chunks is actually dependent on the expected chunk size parameter that the algorithm employs, and not on the size of the changes that have occurred between the last version and the current version of the object.

In particular, a modification of even one byte will result in a new chunk, the size of which varies depending upon this system-defined parameter. To reduce the size of the new chunk, the expected chunk size parameter can be reduced. However You and Karamanolis [You and Karamanolis 2004] show that reducing the expected chunk size to less than 256 bytes can be counter productive as the overheads associated with the additional metadata cost per chunk nullifies the effect of storage savings obtained because of smaller average chunk sizes. It is argued that that optimal storage space using *CDC* is obtained when the expected chunk size is between 256 and 1024 bytes. Apart from storage space overheads associated with maintaining metadata information about each chunk (e.g., the hash key map), small chunk sizes can lead to other overheads as well. For example, in a distributed

storage environment, recording small chunks is likely to result in more network communication on both reads and writes, and may induce greater fragmentation overheads since the chunk size used by the storage system may be smaller than a physical disk block. Most real systems therefore assume a fairly large chunk size. For example LBFS assumes an expected chunk size of 8KB and Pastiche assumes an expected chunk size of 16KB.

In order to overcome the tension between high overheads and better duplicate elimination associated with smaller chunk sizes, we propose a chunking scheme that improves upon *CDC*. Our technique, which we refer to as *fingerdiff*, relaxes the restriction that chunk sizes reside within a fairly narrow size range. Instead, the algorithm leverages information about the content of the data being written to determine partitioning points. Like *CDC*, the challenges in deploying a *fingerdiff* strategy concern tradeoffs between management overheads and redundancy elimination.

The remainder of this document is organized as follows. Section 2 places our work in the context of previous efforts. Section 3 illustrates the design and implementation of the various block encoding schemes. Section 4 establishes the experimental framework that we employ to compare the effectiveness and performance of the different techniques. Section 5 presents performance results. We discuss the results in section 6 and conclude in section 7

2. BACKGROUND AND RELATED WORK

Fingerprints have been proposed to identify similar documents [Manber 1994; Broder 1997; 2000] 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 [Quinlan and Dorwards 2002; Kubiatowicz et al. 2000; 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 boundary regions 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; 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 network traffic by sending only those blocks that are new across the network, and in Pastiche [Cox et al. 2002] in order to identify backup buddies in a peer-to-peer system. However *CDC* is equally applicable in a content-based

storage system, where applications have control over how objects are to be divided into blocks that are recorded by the storage system.

Previous work has also compared *CDC* with *FSC* schemes [Policroniades and Pratt]. Not surprisingly, it was concluded that *CDC* outperforms *FSC* with respect to file system 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 documents is applied in order to identify candidate pairs for encoding. In [Douglis et al. 2004], the authors 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. 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 [Ziv and Lempel 1977; Lelewer and Hirschberg 1987] 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 or whole file compression technique would.

3. DATA CHUNKING TECHNIQUES

We first present the design of a *CDC* algorithm, and then propose the design for our *fingerdiff* algorithm. Subsection 3.5 discusses details about the implementation.

For the remainder of this article, we assume a system model that consists of a distributed storage engine that accepts variable-sized blocks from storage clients for persistent storage. The engine stores these chunks by computing a hash key based on the content of each chunk and storing each chunk in a specified location based on the value of the key. It 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. We make no particular assumptions about the architecture of the storage engine (for example, whether it is centralized or distributed), or the techniques it employs to ensure persistence and availability of the data sent to it. For example, our techniques and analysis hold regardless of whether the storage engine uses replication or erasure codes [Berlekamp 1968] to guarantee availability. However, we do assume that chunks are never overwritten.

Storage clients periodically write data objects such as files to the store using

a client-specific driver. The driver uses one of the chunking techniques that we discuss to divide client data objects into chunks and then writes these chunks * to the store.

3.1 CDC

We first briefly illustrate the *CDC* technique using fingerprints to partition files into variable sized data chunks. *CDC* determines partition points based on the contents of the object being partitioned. It assumes a parameter *expected_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 *expected_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 *expected_chunk_size*, *CDC* compares a given number of bits in each fingerprint with a magic value. For example, if the expected chunk size is 8KB, *CDC* compares the last 13 bits of each fingerprint with a fixed magic value. The reasoning behind this is that given the uniformity of the fingerprint generating function, 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.

3.2 Fingerdiff

The effectiveness of duplicate elimination achieved by *CDC* is tied to the *expected_chunk_size* parameter. As we decrease the expected chunk size we can expect better duplicate elimination since the changes made will be more likely to be contained in smaller sized chunks. However decreasing the expected chunk size increases the number of chunks generated, and induces overheads to manage these chunks that may eventually outweigh benefits accrued by improved duplicate elimination.

Fingerdiff on the other hand manages to nullify this tension between improved duplicate elimination and increased overheads of smaller chunk sizes.

Essentially, *fingerdiff* expands on the concept of variable-sized chunks 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.

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 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

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

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 database and separates out those subchunks that have changed. Consecutive new subchunks are coalesced into 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 this notion of chunk parts, *fingerdiff* expands the interface used to read and write data from the store. Typically, a hash key along with the size of the chunk would be enough to read a chunk. This interface is now extended to ⟨*hash-key, offset, length*⟩ where offset indicates the beginning of the requested chunk part in the stored chunk.

3.3 Example

To illustrate the difference between the *FSC*, *CDC* and *fingerdiff* techniques 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 means of inserting a few bytes at a region near the beginning of the file.

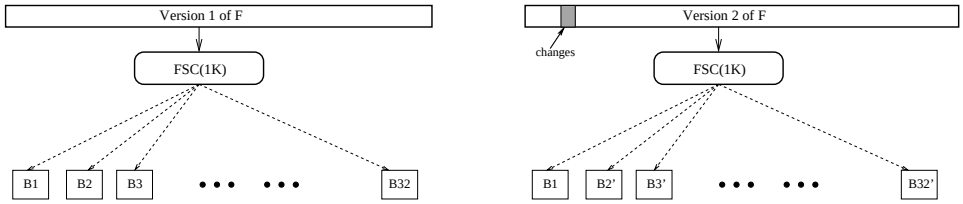

Fig. 1. An example of *FSC* being employed to encode two consecutive versions of a file.

First consider the two versions of *F* being stored using a *FSC* technique with a fixed size of 512 bytes. Figure 1 illustrates the process for the first and second versions of the file. For the first version, the *FSC* 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 *FSC* 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 2 shows the improvement obtained when *FSC* is substituted with *CDC* and *fingerdiff*. For this example we employ a *CDC* algorithm parameterized by an *expected_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 2 (a) *F* is being encoded using *fingerdiff* for the first time. The *CDC* algorithm is called and let us 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 therefore 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 2 (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, *CDC* 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 *FSC*; in *FSC* 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 chunk part of old chunk C1. 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. Information about these chunks are maintained on the client and are not written to the store. The output of *fingerdiff* after having written two versions of the file F to the store contains only 3 chunks, as opposed to *CDC* whose output contains 33 chunks.

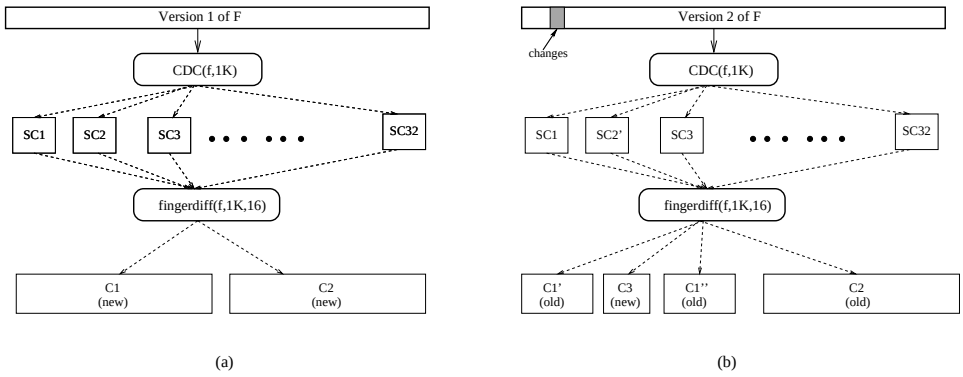

Fig. 2. An example of *fingerdiff* being employed to encode two consecutive versions of a file.

3.4 Algorithms

The *CDC* algorithm (shown in Figure 3) assumes an *exp_chunk_size* parameter and depending on its value calculates a *chunkMask*. For example an 8k expected chunk size value would result in a *chunkMask* of 13 bits (on average every 8kth fingerprint will have its last 13 bits equal to the magic value)

It computes the Rabin fingerprints of each overlapping substring of a fixed size *substring_size* of the file. It chooses a particular byte position as chunk boundaries if the the last *chunkMask* bits of the fingerprint of the substring starting at that byte position equals a fixed magic number. In our algorithm, we choose this magic number to be zero. Once chunk boundaries are identified, the corresponding chunks are added to a list. Each of the chunks in the list are written as a single data entity

```
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. 3. Fingerprint based chunking algorithm

(not shown in the algorithm). Figure 3 hides several implementation details; in particular, it hides the fact that the fingerprints can be computed easily on sliding window of file bytes. The algorithm can be further modified to mark a chunk boundary if a chunk boundary is not marked after a certain *maxval* bytes have been processed in order to avoid pathological cases [Muthitacharoen et al. 2001].

Note that while describing the *CDC* algorithm we used the term "chunk" to describe both the unit of data identified by the fingerprinting algorithm and the unit of data that is written to the data store.

To describe the *fingerdiff* algorithm, we slightly modify this terminology: we introduce a new term called a *subchunk* which is a data unit that is identified by *CDC*. A *chunk* on the other hand is defined as a series of contiguous subchunks that can be written to the data store as one data entity.

The *fingerdiff* algorithm operates with two parameters; an *exp_sc_size* parameter, expected subchunk size, that 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 emitted 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

```
1  Procedure fingerdiff
2  Inputs: File f, Integer exp_sc_size , Integer max_scs
3  Output: List CL of Chunks
4  BEGIN
5      ChunkList CL := empty;
6      SubChunkList SL := CDC(f,exp_sc_size);
7      SubChunk SC := SL.next();
8      Type currentChunkType := lookup(SC);
9      while SL ≠ empty do
10         Chunk C := new Chunk();
11         if (currentChunkType = new)
12         then
13             C.type := new;
14             while (currentChunkType = new and chunkSizeof(C) < max_scs) do
15                 C.add(SC);
16                 SC := SL.next();
17                 currentChunkType := lookup(SC);
18             endwhile
19         else
20             C.type = old;
21             while (currentChunkType = old and isContiguous(SC) do )
22                 C.add(SC);
23                 SC := SL.next();
24                 currentChunkType := lookup(SC);
25             endwhile
26         endif
27         if (C.type = new)
28         then
29             foreach Subchunk SC in C do
30                 size := sizeof(SC);
31                 offset := getOffset(C,SC);
32                 AddToLocalLookup(SC,C, offset,size);
33             endfor
34         endif
35         CL.add(C);
36     endwhile
37     return CL;
38 END
```

Fig. 4. The *fingerdiff* algorithm

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, 17 and 24 uses an auxiliary data structure that records information about subchunks. If a match is found it returns the type as *old*; otherwise it returns the type as *new*.

(2) The *isContiguous* function called on line 21 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, 16 and 23 has the effect of removing the next subchunk from the list and returning that subchunk.

(4) The *chunkSizeof*(*C*) function called on line 14 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. 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 26 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*. Line 21 ensures 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 27 through 34 add the 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 is read from the store using the superchunk-key, and indexed into using the offset and size information.

3.5 Implementation of *fingerdiff*

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. The lookup procedure maintains a tree in which each node consists of information about one subchunk. This information 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 given lookup tree is read from disk whenever its corresponding object is being chunked. At any point of time, the tree of a given object will contain information about all the subchunks of all the versions of that object that have been written so far. We refer to this lookup tree as a local database in the following sections. 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.

4. EXPERIMENTAL FRAMEWORK

A major 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 are also interested in understanding the overheads associated with each technique. Subsection 4.1 elaborates on the different characteristics of each chunking technique that we measure and why they are significant. Subsection 4.2 discusses the benchmarks we use to measure the characteristics of each chunking technique.

4.1 Characterizing the different chunking techniques

The following characteristics of the chunking technique are measured and used to differentiate between them.

(1) *Storage utilization:* For each technique that we study, we measure the storage space utilization on a storage engine during and after a series of consecutive versions have been written to the engine. The storage space includes the sum of the size of both data and metadata chunks plus the size of one pointer per chunk. This pointer has to be maintained on the storage backend to retrieve the chunk whenever necessary. We assume a SHA-1 hash pointer and therefore calculate the pointer size to be 20 bytes. In Section 5, we also explore the effect of storing more than one pointer per data block.

(2) *Number of chunks and distribution of chunk sizes:* Apart from overall storage space usage, we also measure the total number of chunks generated by the different chunking techniques. Measuring this quantity is significant because there are overheads associated with each chunk that vary among different storage systems. For example, in systems that store more than one replica of each chunk, an additional pointer needs to be stored for each replica. In peer-to-peer systems retrieving an object composed of multiple chunks will result in routing and lookup overheads per chunk. Further a storage system that has lesser chunks to store and maintain will have better lookup and update performance than a system that manages more chunks.
The chunk size distribution is also relevant as it affects read and write bandwidth in network storage architectures. Most storage systems are moving toward increasing block sizes to improve bandwidth performance of data. From this perspective we want data to be in relatively large chunks.

(3) *Encoding latency:* The latency of chunking the first and last version of each version series is measured and reported. Chunking latency is is due to the additional work done by the chunking technique while attempting to chunk data objects intelligently so as to optimize storage space.

(4) *Database size:* A local database is maintained by *fingerdiff* to determine chunking sizes for objects. The sum of all individual lookup sizes is the size of the database. We trace the size of the database for four *fingerdiff* instantiations as they are used to write consecutive versions to a storage engine. We show that though the size of the database grows over time, the percentage space saved per byte on the backend also grows with time. We discuss methods to limit the

size of this database.

(5) *Read overheads:* We model the number of seeks (i.e., non-contiguous disk accesses) required on a variable sized block device to read back a given version of a file that has been written by our chunking techniques. We report these numbers for each version of each of the benchmarks, and for each chunking technique that we use.

4.2 Real life traces

We obtained a set of consecutive versions of real programs. These include the following

—*gnu emacs:* A series of 8 emacs versions from 20.1 through 21.3.

—*gnu gcc:* A series of 21 gcc versions from 2.95.0 through 3.4.1.

—*Linux kernel:* A series of 11 versions of the Linux kernel from 2.6.0 through 2.6.3.

—*gnu gdb:* A series of 9 gnu gdb versions from 5.0 through 6.3.

Table I enumerates the characteristics of the first and last version of each of the four benchmarks.

We measure the characteristics of seven instantiations of the different techniques described thus far (explained in the next section) for each benchmark. We assume a chunk store that accepts chunks and stores them based on their content. We then write all the versions of the each series one at time. At each point, we measure the characteristics discussed in this section.

benchmark	**emacs**		**gcc**		**Linux**		**gdb**	
	First (20.1)	Last (21.3)	First (2.95.0)	Last (3.4.1)	First (2.6.0)	Last (2.6.10)	First (5.0)	Last (6.3)
Version number	1	7	1	20	1	10	1	9
Size of gzipped version (MB)	13	20	13	36	40	44	12	17
Size of tarred version (MB)	46	73	56	191	179	196	56	88
Size of all files (MB)	44	71	51	164	167	183	53	84
Number of files in version	1967	2553	2771	21817	15007	16448	3771	5255

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

5. RESULTS

We consider three *CDC* and four *fingerdiff* technique instantiations for our measurements. Using each of these seven instantiations we write each version of the four benchmarks systems to a content addressable store. We compare different characteristics of each chunking instantiation. We weigh the benefits and overheads of each instantiation and identify scenarios where employing one has clear advantages over the rest.

The seven instantiations considered are:

—*cdc-8k* : A content defined chunking (*CDC*) strategy with *exp_chunk_size* of 8KB.

—*cdc-2k* : A *CDC* instantiation with *exp_chunk_size* of 2KB.

—*cdc-256* : A *CDC* instantiation with *exp_chunk_size* of 256 bytes.

—*fingerdiff(2k-32k)* : A *fingerdiff* strategy with *exp_sc_size* 2KB and *max_scs* of 16. This results in variable sized chunks whose expected size can range from 2KB to 32KB.

—*fingerdiff(512-32k)* : A *fingerdiff* strategy with *exp_sc_size* 512 bytes and *max_scs* 64. This results in variable sized chunk whose expected size can range from 512 bytes to 32k.

—*fingerdiff(256-32k)* : A *fingerdiff* strategy with *exp_sc_size* 256 bytes and *max_scs* 128. This results in variable sized chunk whose expected size can range from 256 bytes to 32k.

—*fingerdiff(128-32k)* : A *fingerdiff* strategy with *exp_sc_size* 128 bytes and *max_scs* 256. This results in variable sized chunk whose expected size can range from 128 bytes to 32k.

We chose to exclude *FSC* based instantiations as it has been well documented elsewhere [Policroniades and Pratt] that *CDC* instantiations exploit commonality of data better than *FSC* instantiations.

In order to understand the nature of both *CDC* and *fingerdiff*, we chose a range of instantiations for each of these two techniques, varying the *exp_chunk_size* for *CDC* and the *exp_sc_size* for *fingerdiff*. For *CDC*, we experimented with *exp_chunk_sizes* of 8K, 2K and 256 bytes respectively. We expect that as the *exp_chunk_size* of *CDC* is decreased, we would observe better storage utilization due to decreased chunking granularity. While this was true for *exp_chunk_sizes* from 8K to 256 bytes, we observed for all benchmarks that reducing it to 128 bytes did not improve storage utilization over the 256 byte instantiation. This was found to be because the overhead of generating an excessive number of chunks, and the cost of storing each chunk (a 20 byte pointer is used to maintain each chunk) for the 128 byte instantiation was greater than the benefits accrued due to reduced chunking granularity. It should be noted therefore that we found the *cdc-256* instantiation to be the best *CDC* instantiation in terms of storage utilization. For *fingerdiff*, we used *exp_sc_size*s of 2K, 512 bytes, 256 bytes and 128 bytes. For each instantiation, we also varied the *max_scs* parameter, so that in all cases, the maximum expected size of a chunk would be 32K. Recall that *fingerdiff* coalesces unchanged subchunks, allowing us to have small chunking granularities, without generating excessive number of chunks. As a result, unlike in the case of *CDC*, we observed that *fingerdiff(128-32k)* improves storage utilization over *fingerdiff(256-32k)*. We conclude that *fingerdiff* is capable of reducing chunking granularity in order to improve duplicate elimination without incurring excessive overheads of *CDC*. However for each benchmark reducing the *exp_sc_size* of *fingerdiff* does increase some of the overheads and we measure and explain these overheads. More detailed discussion of results are presented in the following subsections.

5.1 Storage utilization

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. Figure 5 compares the storage utilization achieved on account of duplicate elimination while storing consecutive versions of (a) gnu emacs, (b) gnu gcc, (c) the Linux kernel and (d) gnu gdb for all seven instantiations.

The "Total data size" plot indicates the storage that would have been necessary in a storage system which provides no duplicate elimination; i.e it is the sum of the sizes of each version written so far. The difference between this plot and the plot of any chunking instantiation is a measure of the storage space saved by that instantiation.

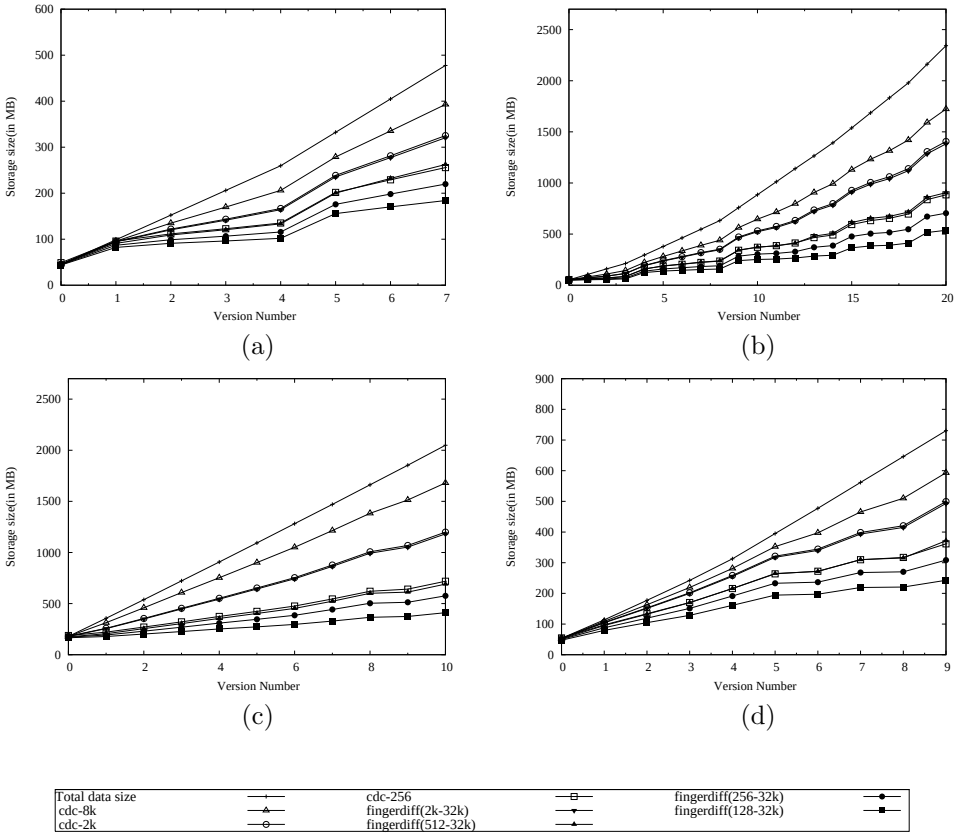

Fig. 5. Comparison of the total storage space consumed by the six chunking technique instantiations while writing a series of (a) emacs, (b) gcc (c) Linux kernel and (d) gdb versions to a content addressable chunk store. The "total storage size" plot is the storage space for each benchmark on a store that does not provide any duplicate elimination.

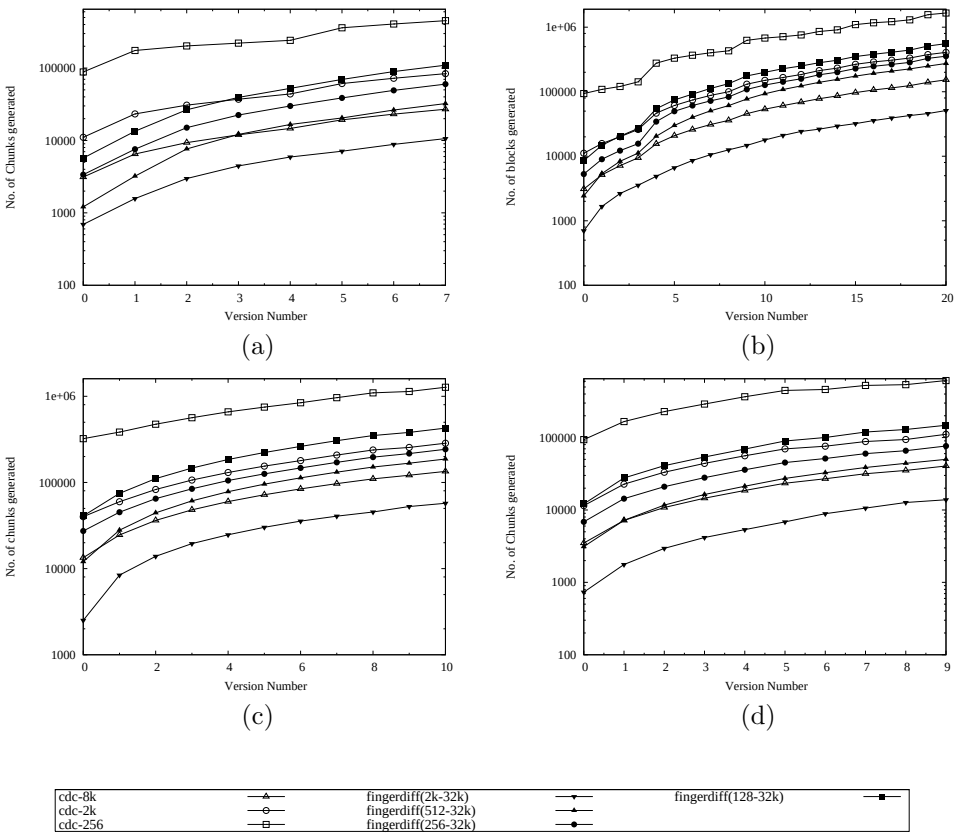

Fig. 6. Comparison of the number of chunks generated by the the six chunking technique instantiations while writing a series of (a) emacs, (b) gcc (c) Linux kernel and (d) gdb versions to a content addressable chunk store. Note that the Y axis is a log plot.

All four benchmarks indicate that among the seven instantiations, *fingerdiff(128-32k)* consumes the least and *cdc-8k* the most storage. Note that *fingerdiff(2k-32k)* and *cdc-2k* have similar storage consumption for all benchmarks. This is because they both use the same chunking granularity, and although *fingerdiff(2k-32k)* generates lesser chunks than *cdc-2k* (figure 6) the effect of the reduced number of chunks in terms of storage space is not significant enough for the difference to be visible in the graphs. However reducing chunking granularities below 2K changes this relation. For example, observe that for all benchmarks, *fingerdiff(256-32k)* consumes far less storage than *cdc-256*. (In fact *cdc-256* consumes more storage than even *fingerdiff(512-32k)*.) As noted earlier this is due the metadata overheads of the massive number chunks generated by *cdc-256* adversely impacting its storage consumption. On the other hand reducing chunking granularity to 128 bytes continues to reduce storage consumption of *fingerdiff* further validating our claim that *fingerdiff* allows us to decrease chunk sizes without incurring the overheads of *cdc*.

The gcc and emacs results have sharp spikes at some points (Version 4 for emacs and version 14 for gcc). This is the effect of multiple version series of gcc and emacs being grouped together. When a gcc 3.0 version is written after a series of 2.95 versions, the amount of duplicates encountered drops drastically resulting in an unusually steep increase in storage consumption at this point. Such steep points represent the end of one version series and the beginning of another within a benchmark.

5.2 Chunks and chunk size distribution

Figure 6 shows the number of chunks that were generated by each chunking technique when writing the four benchmarks to a content addressable store. From the system point of view we would like to have as few chunks as possible to reduce the overheads 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.

Cdc-256 and *fingerdiff(2k-32k)* generate the maximum and minimum number of chunks respectively for all four benchmarks. Note that the log scale is used for the Y axis to accommodate the excessively large number of chunks generated by *cdc-256*. A *fingerdiff* instantiation with a lower *exp_sc_size* generates more chunks than one with a higher *exp_sc_size*; but much fewer than a *cdc* instantiation with the same *exp_chunk_size*. For example while *fingerdiff(256-32k)* generates more chunks than *fingerdiff(512-32k)*, it generates far fewer chunks than *cdc-256*.

These results reflect the tension between storage consumption and associated overheads of *CDC* due to large number of chunks. Among the *CDC* instantiations *cdc-256* provides better duplicate elimination than *cdc-2k* and *cdc-8k*; however it also generates the maximum number of chunks. The *fingerdiff* instantiations attempt to strike a balance. For example *fingerdiff(2k-32k)* gives us approximately the same duplicate elimination as *cdc-2k* but generates far fewer chunks. *Fingerdiff(256-32k)* not only consumes lesser storage space than *cdc-256* but generates fewer chunks as well. Similarly *fingerdiff(512-32k)* consumes lesser space and generates fewer chunks than *cdc-2k*.

As storage designers are moving toward increasing block sizes in order to improve system throughput, we deemed it important to understand the distribution of chunk sizes generated by the *fingerdiff* instantiations. Figure 7 shows the distribution of chunk sizes in terms of the percentage of total bytes on the backends that belong to a chunk of a certain size range. We measure this distribution after writing the last version of each benchmark for all four *fingerdiff* instantiations. (The distribution for *CDC* was expected to be constant, with all chunks expected be of size *exp_chunk_size* with a margin of error.) Observe that in each of the four benchmarks series, and for three of the four *fingerdiff* techniques (*fingerdiff(128-32k)* being the exception) more than half of all data is still contained in chunks of size 16k or higher even after the last version of each series has been written. For *fingerdiff(128-32k)* more than 45% of all data resides in chunks of size 16k or higher after the last version of each benchmark is written. In contrast, if *cdc-128* were to be employed instead of *fingerdiff(128-32k)*, all data would reside in chunks of size 128 bytes.

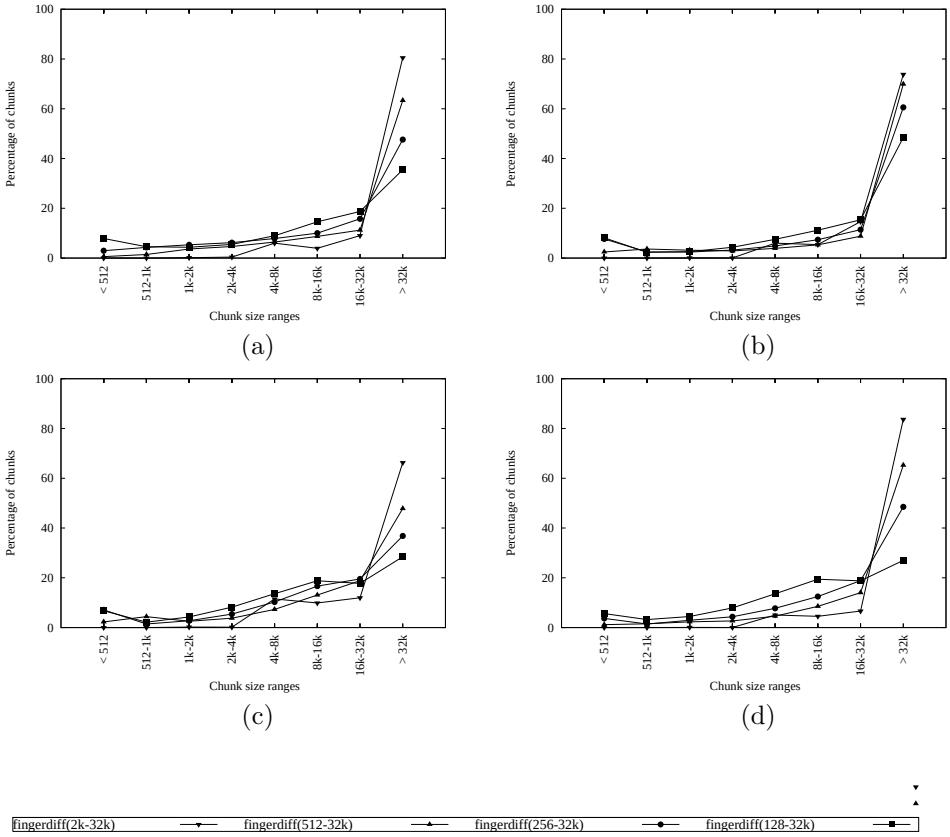

Fig. 7. Percentage of total bytes belonging to chunks of a certain size range after writing the first and last versions of the (a) emacs, (b) gcc ,(c) Linux and (d) gdb series using the four *fingerdiff* instantiations.

5.2.1 *Erasure coded stores.* As mentioned earlier, the overheads imposed on the storage system due to each chunk generated depends on the system architecture. In this section we analyze these overheads for a specific kind of storage system – erasure coded stores. 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] for archival storage. 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. The key property of erasure codes is that the original data block can be re-assembled(decoded) by obtaining any m of the encoded n fragments. The value $r = m/n$ is called the rate of encoding and the value $1/r$ gives a measure of redundancy introduced. Note that we can increase the availability guarantees that we provide by increasing proportionately the values of m and n, but as long as the rate of encoding, r is kept constant,

the aggregate storage space consumed by the data is constant[Weatherspoon and Kubiatowicz 2002a]. In contrast, replication necessitates a linear increase in the aggregate storage space consumed in order to achieve the same increase in availability guarantees [Weatherspoon and Kubiatowicz 2002b].

However for each fragment that is stored in a node there is at least one reference to that fragment. More fragments dispersed over more nodes will automatically mean more storage consumed by these references. For example in [Goldberg and Yianilos 1998], the authors propose each block to be fragmented into 65536 fragments to guarantee that the block lasts until eternity. Assuming the standard 20 byte SHA-1 pointer per fragment, this would mean a 65536 * 20 = 1310720 byte overhead per each block, irrespective of the size of the block. From this perspective, we would want the number of chunks generated by the chunking technique to be as less as possible.

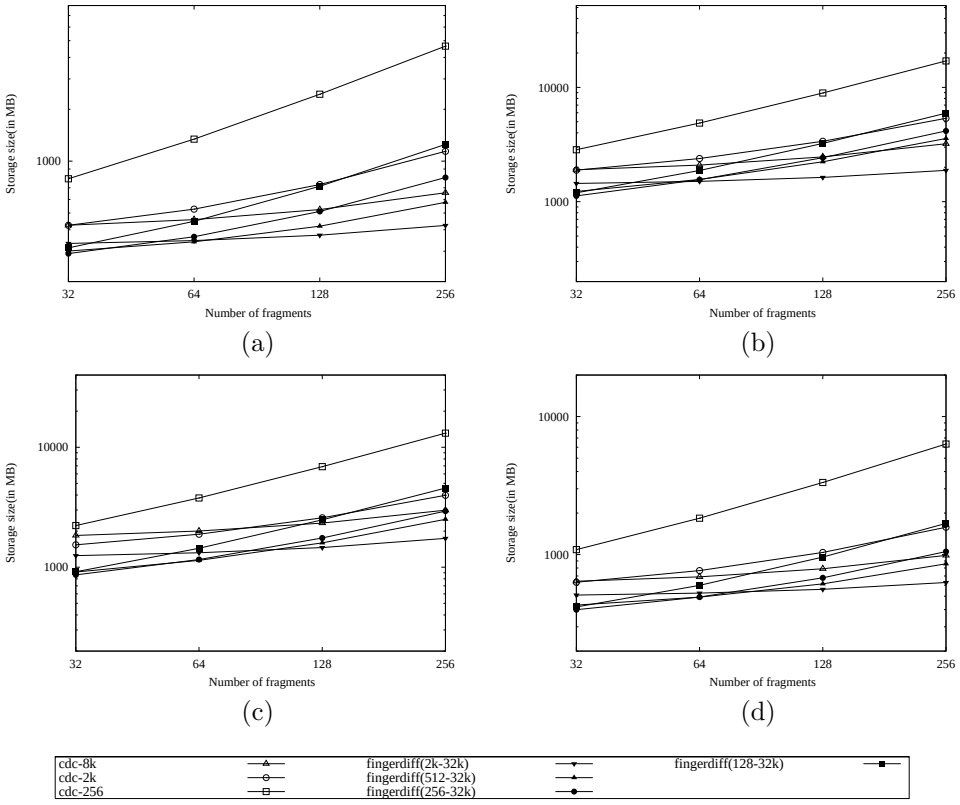

Fig. 8. Comparison of storage overhead of each of the chunking technique instantiations as the number of nodes used to disperse fragments of generated chunks is increased exponentially

For this set of experiments, we assume an erasure coded back end as opposed to the previously assumed chunk store. Figure 8 measures the growth in storage space

for each of the seven chunking technique instantiations and for each benchmark as we increase the number of nodes among which the fragments are dispersed upon from 32 to 256. Because of the nature of the erasure codes, the overall size of data remains a constant; however the metadata size increases since an additional pointer is required to maintain every fragment (as before we assume a 20 byte pointer for each fragment). The x-axis indicates the number of fragments and therefore number of nodes and the y-axis shows the total storage consumed by both data data and metadata in MB. Note both x and y axes are drawn on a logarithmic scale. We start by measuring the storage space consumed by all versions of a given benchmark on a back end that stores each fragment of chunks on 32 nodes and then increase the number of nodes and fragments to 256.

For all four benchmarks, *fingerdiff(128-32k)* consumes the least space when every chunk is coded into 32 fragments. This is because as we saw in the previous experiments this instantiation of *fingerdiff* gives us the best storage utilization. However observe that as we increase the number of fragments, *fingerdiff(2k-32k)* eventually outperforms all the other techniques for all four benchmarks. This is simply because, *fingerdiff(2k-32k)* generates chunks that are significantly fewer than the rest. As we increase the number of nodes on which fragments are stored, the overhead of storing each additional fragment dominates the cost of storage making the duplicate elimination obtained by the various techniques insignificant. Also observe that storage space consumed by the instantiations that generates more chunks (*cdc-256*) increases more rapidly than instantiations which generate fewer chunks. This indicates that as we increase the number of fragments used to store data, eventually the technique which generates more chunks will consume more storage than techniques which generate fewer chunks, i.e the cost of storing increasing metadata will eventually be greater than the benefit of data saved due to duplicate elimination. Clearly the benefits of duplicate elimination are valid only in systems that do not erasure code blocks into extremely large number of fragments as is proposed in [Goldberg and Yianilos 1998].

5.3 Chunking Latency

Table II presents the latency of chunking the first version and the last version for all four benchmarks and for of each of the seven instantiations. For *CDC*, the chunking time includes only the time to compute Rabin's fingerprints on fixed size substrings of each file of the version, and to identify chunk boundaries in the file. For *fingerdiff*, the chunking time includes the time spent in identifying chunk boundaries as in *CDC*, and also the time required to compute hashes on subchunks, look up the local database for hashes, and update the database if necessary.

fingerdiff(128-32k) and *cdc-8k* spend the maximum and minimum amount of time respectively chunking each version. The chunking latency of *CDC* increases with decreasing *exp_chunk_sizes*, although only slightly. Recall that the same substring size is used to compute Rabin's fingerprints for all *CDC* instantiations resulting in only marginal differences between their chunking latencies. On the other hand the size of the version being chunked impacts the chunking latency of *CDC* more directly as more fingerprints have to be computed for larger versions.

Note that both *CDC* and *fingerdiff* instantiations spend the same amount of time computing fingerprints because the same substring size is used for all in-

stantiations. *Fingerdiff* instantiations spend more time chunking than their *CDC* counterparts because of the overheads involved with looking up subchunks in the database. These overheads include hashing each subchunk, looking up the hash in the database, checking for subchunks that can be coalesced, and then inserting new hashes into the database if necessary. The time required for database operations depends on the size of the database, which in turn depends on the subchunk size of the *fingerdiff* instantiation. *Fingerdiff(128-32k)* has the greatest sub chunk size granularity among the *fingerdiff* instantiations we measure. As it has to compute the most subchunk hashes and perform the most database operations, it has the maximum chunking latency. Using the same argument, *fingerdiff(2k-32k)* has the least chunking latency among the *fingerdiff* instantiations we measure, which is still more than any *CDC* instantiation.

benchmark	**emacs**		**gcc**		**Linux**		**gdb**	
instantiation	First	Last	First	Last	First	Last	First	Last
cdc-8k	2.8	4.6	3.4	10.2	11.7	12.0	3.3	5.3
cdc-2k	2.9	4.7	3.6	12.2	11.8	12.9	3.5	5.6
cdc-256	3.3	4.8	3.8	13.0	11.9	13.4	3.9	5.9
fd(2k-32k)	6.1	8.7	7.7	21.9	20.8	24.3	7.2	11.2
fd(512-32k)	6.8	9.5	8.1	25.1	21.6	26.2	7.9	11.8
fd(256-32k)	7.9	11.1	9.5	28.2	24.5	30.2	9.2	13.1
fd(128-32k)	11.1	14.2	12.8	37.9	33.2	41.5	12.4	17.2

Table II. Chunking latency in seconds measured while writing the first and last version of each benchmark for all chunking technique instantiations.

5.4 Database size

As mentioned previously, a local database is used on the client to support *fingerdiff* chunking. This database contains a tree that maps hashes of subchunks to information about that subchunk. This database resides in disk persistently, but is pulled into memory when an object that is mapped to it is being chunked. As can be expected, this database grows as more versions are written to the store. We measure the size of the database for all our *fingerdiff* instantiations. We also measure the efficiency of the database; we define efficiency as the number of bytes gained from duplicate elimination on the back end per each byte of the database, assuming only one client has written this data to the back end.

Figure 9 shows the results for the (a) emacs, (b) gcc (c) Linux kernel and (d) gdb benchmarks respectively. These graphs show that the database size grows with every new version in each of the four traces. They also indicate that the smaller the size of the *exp_sc_size* of the *fingerdiff* instantiation, the faster the growth of the database. Having said that, observe that the storage savings on account of duplicate elimination per byte of database size also grows more rapidly for instantiations with smaller values of *exp_sc_size*. So although *fingerdiff(128-32k)* requires a larger local database, it also saves more space per byte of its database than the other *fingerdiff* instantiations that we measure.

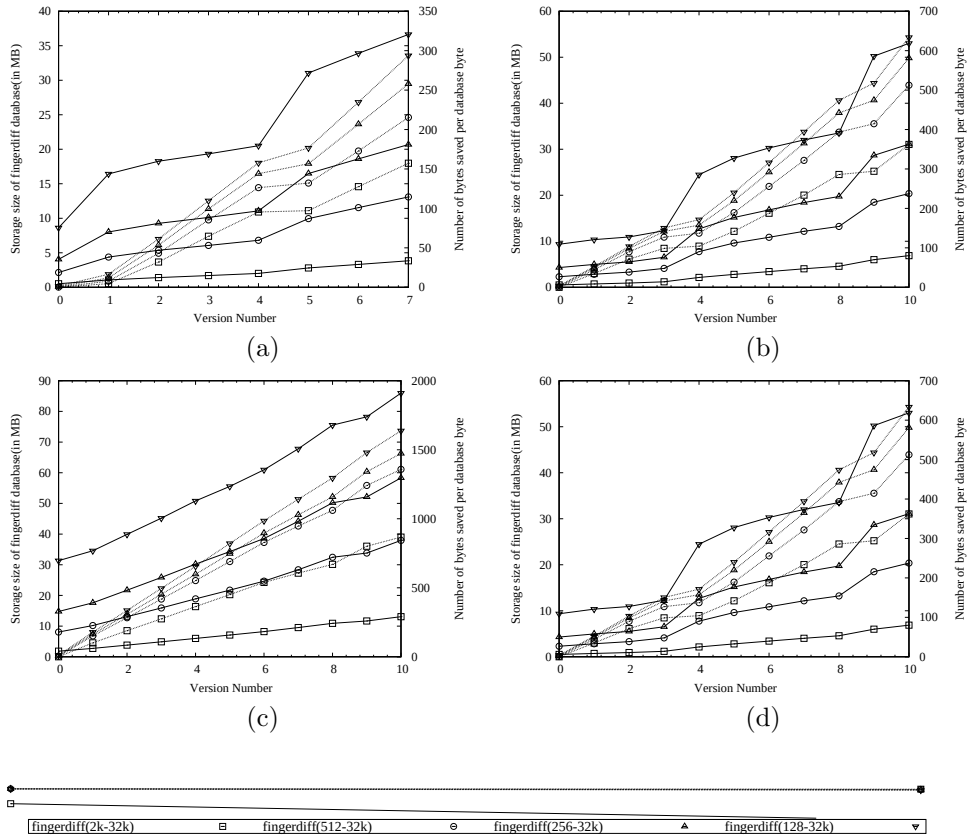

Fig. 9. Comparison of database characteristics of *fingerdiff(2k-32k)* and *fingerdiff(512-32k)* instantiations while writing a series of (a) emacs, (b) gcc (c) Linux kernel and (d) gdb versions to a content addressable chunk store. The solid lines show the absolute database size for each technique; whereas the dotted lines plot the efficiency of the database; i.e the number of bytes saved in the back end per one database byte assuming only one client has written this version to the back end.

5.5 Read overheads

The chunking techniques introduce certain overheads for reading a given version of a file from the store.

In order to read a given version of a file back from the store, the client must request each chunk corresponding to that file from the store, and then group all chunks together in the right order to regenerate the file. A request for each chunk will translate into a read request on some disk system in the storage system, if the chunk is not found in any of the system caches. It is safe to assume that a request for archival data will miss the cache and translate into disk requests. In order to better measure the cost of reading chunks from the storage system and understand the impact of each chunking technique on this cost, we developed a simple model

to calculate the overhead of reading from a disk store. The goal of our model is to calculate the penalty paid while reading a given version from a disk store. Specifically our model assumes that whenever two chunks are read one after the other from disk, and the chunks are not contiguous in disk, one disk "seek" is made to read the second chunk. We report the total number of such "seeks" necessary to read each version of a file after it has been encoded and stored using our seven chunking technique instantiations. Note however that the value we return is not technically the number of disk seeks made by the disk head, but simply a notion of locality of data placement; We assume an ideal value of 1, which is when the entire version is stored contiguously on disk and can be read in just one disk "seek". Hence lesser the number of seeks, the higher the locality and vice-versa. To avoid confusion with actual disk seeks, we will refer to the value returned by the model as "read-overhead".

The specific details of our model are as follows:

—*While writing data*: An infinite disk log is assumed that stores each new data chunk irrespective of its size at the end of the log. This position is recorded as an offset from the beginning of the log. If the chunk already exists it is not stored again. A hash table that maps chunk keys to disk locations (offsets) is maintained to look up each chunk. In the scenario where the first version of a file is written, all chunks will be written contiguously one after the other on disk (assuming no duplicates for the first version). However, when a later version of the file is written, some of the chunks will be old and therefore will not be appended at the end of the log. Only new chunks for this version will be contiguously written on disk.

—*While reading data*: To read a given version of a file, a series of chunks that comprise the file is requested. A disk pointer variable is assumed that keeps track of the position of the disk chunk currently being read. Each time the disk pointer has to be moved to a chunk in the disk that is not contiguous with the last chunk read, the number of seeks is incremented. The final value of the number of seeks is returned.

Figure 10 presents the read overhead of each of the seven chunking technique instantiations while reading every version of the (a) emacs, (b) gcc (c) Linux, and (d) gdb traces back from the store as measured by our model.

Fingerdiff(128-32k) and *cdc-8k* has the maximum and minimum overheads respectively. *Fingerdiff(128-32k)* in its attempt to maximize duplicate elimination produces chunks that are more likely to be already on disk; therefore these chunks are more dispersed throughout the disk than the other instantiations. Reading a version from the disk will therefore result in fewer chunks being contiguous. This is the price paid by *fingerdiff* for attempting to optimize duplicate elimination. *CDC-8k* on the other hand provides poor duplicate elimination and subsequently writes contiguous chunks more often to disk. Another interesting comparison in figure 10 is between *fingerdiff(256-32k)* and *cdc-256*. Recall that both instantiations generate chunks of expected size 256, but *fingerdiff* coalesces them whenever possible to form larger chunks. As a result it generates fewer number of chunks. Hence, *fingerdiff(256-32k)* not only outperforms *cdc-256* in terms of storage space(figure 5),

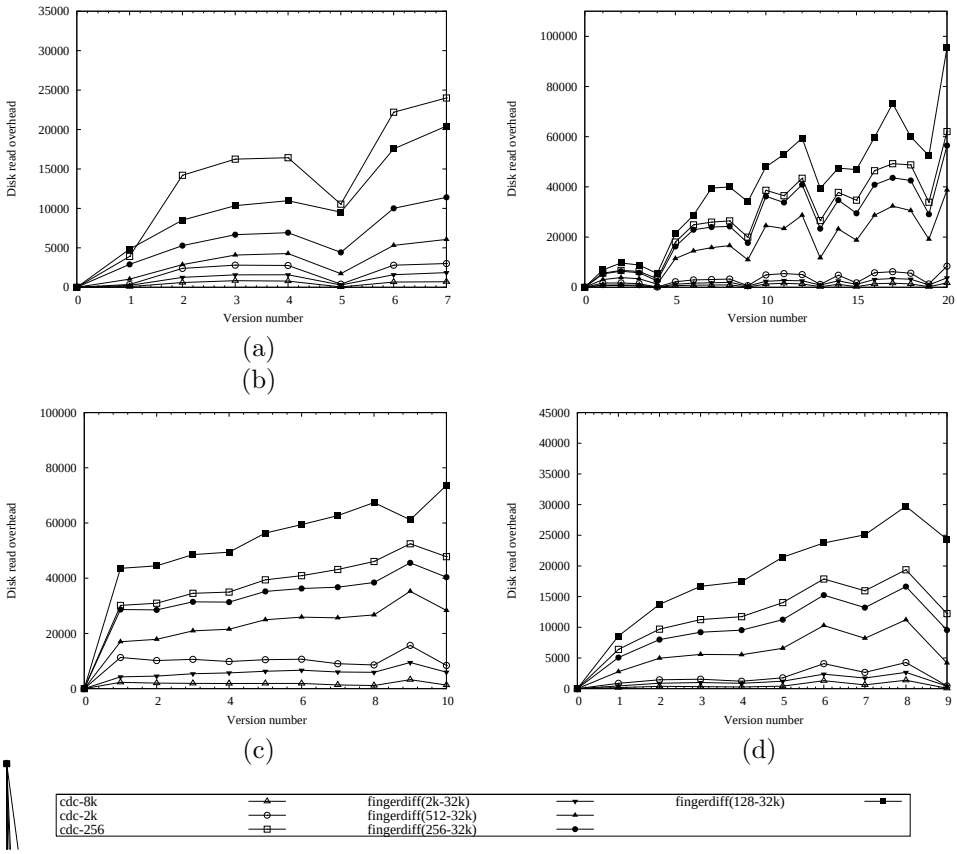

Fig. 10. Comparison of the read overhead while reading back each of the (a) emacs, (b) gcc, (c) Linux kernel and (d) gdb versions for each of the seven chunking technique instantiations. The abrupt fall in read overheads at specific points for all instantiations is because of multiple version sets of the benchmarks being clubbed together into one versions series.

but also in terms of read overhead. Similarly *fingerdiff(2k-32k)* outperforms *cdc-2k* in terms of both storage space and read overheads; clearly a win-win situation and a strong case for the use of *fingerdiff*. The read overhead falls abruptly for all instantiations at certain points in the emacs and gcc traces. This is because the benchmarks we use have multiple version sets clubbed together as one continuous series. For example in figure 10(a), version point 4 on the x-axis corresponds to emacs 20.7 whereas version point 5 corresponds to emacs 21.1. Emacs 21.1 being a new version set, has a disproportionate amount of differences when compared with 20.7. As a result, at this point there is an increase in the amount of data that is new and therefore contiguous on disk – resulting in the decrease in read overhead for all chunking technique instantiations at this point.

6. DISCUSSION

Storage consumption is a direct measure of the ability of chunking techniques to maximize duplicate elimination by leveraging content addressability. Each technique imposes certain overheads that vary depending on the design and architecture of the storage system. In general a technique that generates more chunks imposes greater overheads on the system because of the increased cost of managing and storing greater volumes of metadata on the backends.

Figure 11 plots storage consumption as a function of the number of chunks generated by each of the seven instantiations for the four benchmarks. Each point in the lines of figure 11 represents a version in the benchmark. Note that the *cdc-256* instantiation generates the most number of chunks, while the *cdc-8k* instantiation consumes the maximum space.

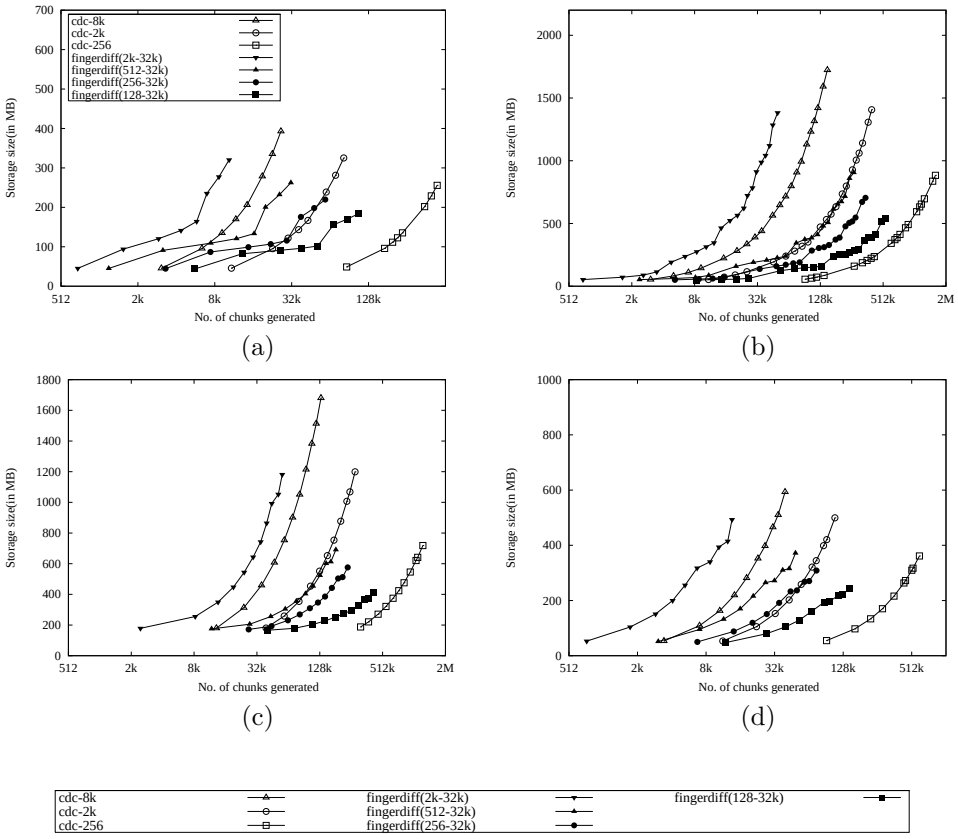

Fig. 11. The storage utilization as a function of the number of chunks generated for the six chunking technique instantiations when writing a series of (a) emacs, (b) gcc, (c) Linux kernel, and (d) gdb versions to a content addressable chunk store. The X-axis is plotted using a log scale in order to accommodate the large number of chunks generated by *cdc-256*.

It has been well-documented that *CDC* provides better duplicate elimination than *FSC* techniques. [Policroniades and Pratt]. Unfortunately, in order to further improve storage consumption of *CDC*, we have to reduce its expected chunk size. Observe that storage consumption of *cdc-2k* is less than that of *cdc-8k*. But this improvement comes at the cost of increased number of chunks generated by the smaller-sized *CDC* instantiation. Observe also in figure 11 that *cdc-2k* generates more than twice as many chunks as *cdc-8k* for all four benchmarks. Further, due to increasing metadata overheads associated with large number of chunks, decreasing the expected size of *CDC* below a certain threshold results in no further improvement in storage consumption. For example in our experiments we observed that *cdc-128* did not improve the storage consumption beyond that of *cdc-256*.

Fingerdiff overcomes these problems by coalescing chunks into larger chunks wherever possible. As a result it manages to provide better duplicate elimination than *CDC* at lower overheads. For example *fingerdiff(256-32k)* consumes less than 90% of the space consumed by *cdc-256* while generating less than half the number of chunks. Also observe that *fingerdiff(128-32k)* and *fingerdiff(256-32k)* lines are the shortest lines in all four graphs of figure 11 representing the fact that these instantiations store the same data by consuming the least space for a given number of chunks generated --the best cost benefit ratio among all chunking technique instantiations.

The improved storage efficiency of *Fingerdiff* comes with a cost. A local database that maintains information about each of the subchunks that have been written so far must be maintained. Note, however, that the database need not be maintained with the same availability and persistence guarantees as data on the storage end. Losing information stored in the database 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. *Fingerdiff* also necessitates a lookup against a database-managed tree to chunk objects. Since this tree resides on disk, it has to be fetched into main memory, resulting in one or more disk accesses per object. Thus, the chunking latency of *fingerdiff* is greater than both *CDC* and *FSC*(table II). This latency can be masked by effectively caching the database in memory, and by writing asynchronously to the store while acknowledging writes to the application.

In addition, the growth of the *fingerdiff* database can be moderated if the algorithm is modified so as to delete subchunk information for those subchunks that are not likely to be accessed. The intuition here is that once a subchunk ceases to be a part of the latest version of an object, the probability that it will become a part of a later version is low. We can exploit this observation to bound the size of the database without adversely affecting storage utilization. Such a technique will also have a positive impact on chunking latency of *fingerdiff* as the overheads associated with database operations will be reduced.

7. CONCLUSIONS

As a result of the limited variability in chunk sizes of *CDC*, we show that attempting to improve storage consumption of *CDC* will result in greater number of chunks and smaller chunk sizes. As storage hardware moves towards increasing disk block sizes

in order to improve system throughput, it is unreasonable to reduce chunk sizes in order to improve duplicate elimination alone. Further increasing the number of chunks impose metadata overheads on the storage engine. These observations motivated us to discover a chunking technique that would improve duplicate elimination over existing *CDC* techniques without increasing its overheads.

We propose a new chunking algorithm *fingerdiff* that is designed to improve upon the storage consumption of *CDC* while lowering the overheads it imposes on the storage system. We have measured storage consumption along with associated overheads of several *CDC* and *fingerdiff* instantiations as they write a series of versions of four different real-world software systems to a content addressable store. For all these benchmarks, we show that *fingerdiff* instantiations improve upon existing *CDC* techniques without increasing 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 chunks introduced with every version, while keeping the average size of all blocks relatively large. This in turn allows it to provide the best storage consumption for the least overhead costs. By choosing the range of fingerdiff chunk sizes that is most suited to their storage architecture, storage managers can significantly reduce the amount of data they have to store and manage.

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.

Brin, S., Davis, J., and García-Molina, H. 1995. Copy detection mechanisms for digital documents. In *Proceedings of the ACM SIGMOD Annual Conference*. 398–409.

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.

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*.

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.

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.

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.

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. Feasibility of data compression by eliminating repeated data in practical file systems. http://www.cl.cam.ac.uk/users/cbp25/fyr.pdf.

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

Weatherspoon, H. and Kubiatowicz, J. 2002b. Erasure coding vs. Replication: A Quantitative Study. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02), Boston, MA, USA*.

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.