Proceedings of the General Track: 2004 USENIX Annual Technical Conference

Boston, MA, USA
June 27–July 2, 2004

Alternatives for Detecting Redundancy in Storage Systems Data

Calicrates Policroniades and Ian Pratt
Computer Laboratory
Cambridge University
Cambridge, UK, CB3 0FD
{name.surname}@cl.cam.ac.uk

Abstract

Storage systems frequently maintain identical copies of data. Identifying such data can assist in the design of solutions in which data storage, transmission, and management are optimised. In this paper we evaluate three methods used to discover identical portions of data: whole file content hashing, fixed size blocking, and a chunking strategy that uses Rabin fingerprints to delimit content-defined data chunks. We assess how effective each of these strategies is in finding identical sections of data. In our experiments, we analysed diverse data sets from a variety of different types of storage systems including a mirrored section of sunsite.org.uk, different data profiles in the file system infrastructure of the Cambridge University Computer Laboratory, source code distribution trees, compressed data, and packed files. We report our experimental results and present a comparative analysis of these techniques. This study also shows how levels of similarity differ between data sets and file types. Finally, we discuss the advantages and disadvantages in the application of these methods in the light of our experimental results.

1 Introduction

Computer systems frequently store and manipulate several copies of the same data. Some applications may generate versions of a document stored as separate files, but whose content differs only slightly. Software development teams, file synchronisers [1, 28, 29], backup systems [19], reference data managers [9], and peer to peer systems [8, 11, 17, 26] deal with large quantities of identical data. Efficient data management solutions may be created if system designers are aware of the amount of redundant data seen in diverse data sets.

Although saving disk space can be useful, over the past few years there has been a constant reduction in the cost of raw disk storage. Some may argue that the disk space savings obtained by suppressing identical portions of data are of minimal significance. However, apart from the benefits obtained from disk block sharing, there are other factors that need to be considered:

- Storage systems may exploit data duplication patterns to optimise the use of storage space and bandwidth. Single Instance Storage (SIS) [3] explores the content of *whole files* to implement links with semantics of copies instead of storing a file with the same content several times. Backup systems such as Venti [19] store duplicated copies of *fixed size* data blocks only once. LBFS [18], Pasta [16, 17], Pastiche [8], and the Value-Based Web Caching algorithm (VBWC) [23] find identical portions of data using Rabin fingerprints. In this method, data is divided into *content-defined* chunks in order to exploit cross-file data duplication. Additional details of the content-defined chunking method will be presented in the section 3 of this paper.

- File systems may obtain improved caching performance if they are aware of contents shared between files. In this way, it would be possible to provide better hit ratios for a given cache size. A potential size reduction of the main memory file cache may have important performance effects.

- In mobile environments, devices are often *limited in storage and bandwidth*. Furthermore, there are factors such as *energy consumption and network costs* associated with data transmission that can become critical [2, 18]. Under certain circumstances, it might be desirable to perform significant computation to reduce the number of bits transmitted over low-bandwidth or congested links.

In summary, three different methods are frequently used to eliminate duplicated data among files: whole file content hashing, fixed size blocking, and a chunking strategy that uses Rabin fingerprints to delimit content-defined portions of data. Due to the lack of a practical comparative study, the typical performance of each method and their suitability for different data profiles are not clear. In this work, we evaluate the effectiveness of these three methods to discover data redundancy and show the potential benefits of their employment under diverse data profiles. We have developed a set of programs to evaluate each of the approaches, analyse data redundancy patterns, and expose practical trade offs. We explored different collections of real-world data sets to determine how sensitive these methods are to different data profiles:

- **Mirrored section of sunsite.org.uk**[1]. This data is a subset of an Internet archive and its size is over 35 GB. Compressed and packed data were common in this data set.
- **Users' personal files.** The data analysed in this collection of files is held in 44 home directories of different users in the Cambridge University Computer Laboratory. The size of this data set is approximately 2.9 GB.
- **Research groups' files.** This data set contains collections of files associated with different research projects of the Computer Laboratory. This is a data set with a potentially high level of data duplication because it stores software development projects, shared documents, and information accessed and manipulated by groups of people. The size of this data set was 21 GB.
- **Scratch directories.** The 100 GB of information explored in this section represents the largest and most diverse data set analysed. We thought that this collection of files is a good example of a data set where no obvious interrelationship is previously known.
- **Software distributions.** To explore the sharing patterns of highly correlated data in different states, we explored five successive Linux kernel distributions in three different formats: packed and compressed (.tar.gz), uncompressed but still tarred (.tar), and uncompressed and untarred.

We had a special interest in the method that uses Rabin fingerprints to delimit chunks of identical data because it has been used in Pasta [16, 17], an experimental large-scale peer to peer file system developed at the Computer Laboratory. We have plans to incorporate an optimised version of Pasta into the XenoServers [22] project. This study enabled us to assess the advantages and drawbacks of using the content-defined data chunking strategy as a compression and replica management tool in the next implementations of the file system. The results directly helped us to understand the impact of these techniques in Pasta, and we believe our experimental results may also be useful to other parties.

The next section of this paper presents an overview of the related work. In section 3 we introduce the reader with the methodology used to measure the levels of identical data in large collections of files. Our experiments and results are presented in section 4. Finally, in section 5 we conclude with a discussion of our findings.

2 Related Work

A number of strategies to discover similar data in files have been explored in different systems. Unix tools such as `diff` and `patch` can be used to find differences between two files and to transform one file into the other. Rsync [28] copies a directory tree over the network into another directory tree containing similar files. It saves bandwidth by finding similarities between files that are stored under the same name.

The Rabin fingerprinting algorithm [20] has been employed with different purposes such as fingerprinting of binary trees and directed acyclic graphs [4, 13], or as a tool to discover repetitions in strings [21]. However, we are interested in how Rabin fingerprints can be used to identify identical portions of data in storage systems. In general, Rabin fingerprints have been used for this purpose in two ways: to sample files in order to discover near-duplicate documents in a large collection of files, or to create content-defined chunks of identical data.

Manber [15] employs Rabin fingerprints to sample data in order to find similar files. His technique computes fingerprints of all possible substrings of a certain length in a file and chooses a subset of these fingerprints based on their values; the selected fingerprints provide a compact representation of a file that is then used to compare against other fingerprinted files. Similarly, Broder applies resemblance detection [5] to web pages [6] in order to identify and filter near-duplicate documents. Rabin fingerprints of a sliding window are computed to efficiently create a vector of shingles of a given web page. Consequently, instead of comparing entire documents, shingle vectors are used to measure the resemblance of documents in a large collection of web pages. The techniques used by Manber and Broder have been adapted by Spring and Wetherall [27] to eliminate redundant network traffic. However, they used Rabin fingerprints as pointers into data streams to find regions of overlapping content before and after the fingerprinted regions.

Different systems use blocking strategies that employ the Rabin fingerprinting algorithm to create *content-defined* and *variable-sized* data chunks. Probably the first storage system that used Rabin fingerprints for this purpose was LBFS [18], specially designed to transmit data over low-bandwidth networks. Using the Rabin fingerprinting algorithm, LBFS finds similarities between files or versions of the same file. It avoids retransmission of identical chunks of data by using valid data chunks contained in the client's cache and by transmitting to and from the data server only the chunks that have been modified. LBFS's chunking algorithm was tested on a data set of 354 MB reporting that around 20% of the data was contained in shared chunks.

Blocking in Pasta [16, 17], an experimental peer to peer file system, also exploits the benefits of common information between files. Caching and replica placement are defined by data blocks' content. These blocks are built by computing Rabin fingerprints of the file data over a sliding window. Identical blocks are stored only once and referenced using a shared key. A similar technique is used in Pastiche [8]. In Value-Based Web Caching [23], web proxies index data according to their content and avoid retransmission of redundant data to clients connected over low-bandwidth links. Although all these systems show that improved block sharing levels can be obtained using content-defined chunking strategies, they do not present broad experimental results based on diverse data sets.

Data redundancy in storage systems has also been identified using *fixed size blocking strategies*. Sapuntzakis et al., aimed to reduce the amount of data sent over the network by identifying identical portions of data in memory [25]. They use a hash-based compression strategy of memory aligned pages (i.e. fixed size blocks of data) to accelerate data transfer over low-bandwidth links and improve memory performance.

Venti [19], a network storage system intended for archival of data, aims to reduce the consumption of storage space. It stores duplicated copies of fixed size data blocks only once. Venti reports a reduction of around 30% in the size of the data sets employing this method. Future implementations of Venti may also incorporate a content-based blocking scheme based on Rabin fingerprints.

A different approach to eliminate data redundancy can be seen in SIS [3] for Windows 2000. It saves space on disk and in main memory cache but with a different approach; SIS explores the content of the *whole file* and implements links with the semantics of copies for identical files stored on a Windows 2000 NTFS volume. A user level service, called the *groveler*, is responsible for automatically finding identical files, tracking changes to the file system, and maintaining a database with the corresponding file indexes. When SIS was tested on a server with 20 different images of Windows NT the overall space saving was 58%.

Although many systems have proposed different techniques to manage duplicated data, it has been only lately that practical studies to assess their benefits and applicability have been performed. Building on Manber's observations, Douglis and Iyengar [10] explore duplication in empirical data sets using Delta-Encoding via Resemblance Detection (DERD) and quantify their potential benefits. Their technique generalises the applicability of delta-encoding by choosing an appropriate set of base versions in a large collection of files through resemblance detection.

As part of the design of a storage system specially crafted to manage reference data [9], a comparison on the effectiveness of three duplicate suppression techniques has been done; two of the techniques analysed in this work are similar to the methods we analysed: fixed size blocking and the content-defined chunking algorithm. The third technique, called *sliding blocking*, uses rsync checksums and a block-sized sliding window to calculate the checksum of every overlapping block-sized segment of a file. The sliding blocking technique consistently detected greater amounts of redundant data than the other two strategies.

More recently, Redundancy Elimination at the Block Level (REBL) has been proposed as an efficient and scalable mechanism to suppress duplicated blocks in large collections of files [14]. REBL combines advantageous features of techniques such as content-defined data chunks, compression, delta-encoding, and resemblance detection. Empirical data sets were used to compare REBL with other techniques. REBL presented the smallest encoding size in 3 out of the 5 data sets analysed and consistently performed better than the other techniques. Specifically, the effectiveness of this technique compared to the content-defined chunking strategy varied by factors of 1.03-6.67, whole file compression by 1.28-14.25, sliding blocks [14] by 1.18-2.56, object compression (tar.gz) by 0.59-2.46, and DERD by 0.88-2.91. All these studies [9, 10, 14] used Rabin fingerprints as a tool to eliminate data redundancy. Thus, their experimental results relate highly to ours.

3 Design

In this section we explain the methodology used to quantify duplication in our data sets. In general, we analyse a collection of files and spot identical data among them using the three methods mentioned before: whole file content, fixed size blocks, and Rabin fingerprints. As a result, we collect information to exhibit sharing patterns in the different data sets.

Sharing patterns at a **whole file** granularity are found by calculating the SHA-1 digest of individual files. The first 64 bits of the resulting digest are used as the key to a hash table. Identical files are likely to be indexed using the same hash table entry. We use the hash table to store statistical data of identical files such as number of occurrences and size of the original file. Although Henson [12] warns about some of the dangers of comparing by hash digests, this method can be safely used in our experiments. In our case, false sharing of the key space is not a crucial concern[2].

In order to find similarities using **fixed size blocks** an analogous procedure is employed, but instead of obtaining digests for whole files, they are calculated for contiguous non-overlapping fixed size portions of the files. In this case, hash table entries correspond to unique data blocks.

The third method analysed employs Rabin fingerprints. It offers the advantage that the chunks generated are **defined according to their contents**. The mathematical principles of Rabin fingerprints are well documented [20]. A Rabin fingerprint $f(A)$ is the polynomial representation of some data $A(t)$ modulus an irreducible polynomial $P(t)$. We compute such an irreducible polynomial only once and use the same value in all our experiments in order to find identical pieces of data. The algorithm for computing such a polynomial can be found in [7]. Our implementation follows the principles presented by Broder in [4] which is an extension of the work done by Rabin. Broder uses precomputed tables to process more than 1 bit at a time, particularly, 32 bits are divided into four bytes and processed in one iteration. The value of k, which is the degree of the irreducible polynomial and consequently the length of the fingerprint, should be a multiple of 32.

To divide a file into content-defined chunks of data our implementation incrementally analyses a given file using a sliding window and marks boundaries according to the Rabin fingerprints obtained in this process. Figure 1 depicts this mechanism. The program inspects every w bytes of a sliding window that is shifted over the contents of the file. Although the value of w can be tuned, changing the size of the sliding window does not significantly impact the result. Experimental evidence in [16, 18] shows that by setting $w = 48$ bytes it is possible to discover significant levels of data duplication.

Adding a new byte into the sliding window is accomplished in two parts. Firstly, the value for the oldest byte in the window are subtracted from the fingerprint. Secondly, the terms of the new byte are added to the fingerprint. This is possible since the fingerprint is distributive over addition. When subtracting, precomputed tables are used in order to improve the implementation performance.

Rabin fingerprints for each window frame are calculated and if the value obtained matches the r least significant bits of a constant, a breakpoint is marked. These

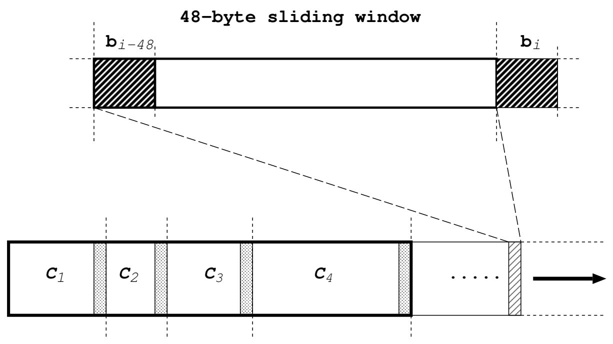

Figure 1: Chunks' boundaries are found using a 48-byte sliding window that incrementally analyses the file's content and computes Rabin fingerprints. Shaded boxes represent the 48-byte regions that generated a boundary. The light striped rectangle corresponds to the current 48-byte window. At each step the byte in the oldest position of the sliding window (b_{i-48}) is subtracted from the fingerprint and the next byte in the file (b_i) is added to the fingerprint.

breakpoints are used to indicate chunk boundaries. In order to avoid pathological cases (i.e many small blocks or enormous blocks) our implementation forces a minimum and a maximum block size.

Once a boundary has been set, the SHA-1 digest corresponding to the chunk's content is calculated. Similar to the other two techniques, the first 64 bits of the SHA-1 digest are used as the key for accessing the hash table that stores statistical information about identical data chunks.

Two kind of values are calculated and reported in our experimental results: percentage of identical data and storage savings. To calculate the **percentage of identical data** in shared blocks we add for each duplicated block, the product of its size and its number of occurrences; then, present this value as a percentage of the original data set size.

To calculate **storage space savings** we add the size of every unique block in the hash table; replicated blocks are counted only once. We present this value as a percentage of the original data set size. Additionally, storage space savings are compared with the space used by simply **tarring and compressing** the whole collection of files in the different data sets. We used the standard `tar` and `gzip` utilities for this purpose.

4 Repeated Data in Empirical Data Sets

This section presents the experimental results used to evaluate the benefits and performance of three different duplicate suppression methods in each of the data sets mentioned.

4.1 Mirror of sunsite.org.uk

In order to find commonality in data that resembles a standard Internet archive, we ran our programs on a 35 GB section of sunsite.org.uk. The total number of files in the data set was 79,551, with an average file size of 464 KB. Table 1 shows a partial characterisation of this data set. It shows the 15 most popular file-name extensions and their percentage of the total number of files. The 15 most popular file extensions account for over 82% of all files. Table 1 also indicates the 15 file extensions that use the most storage space and the percentage of the total space they consume; collectively, they cover almost 97% of the whole data set. Packed and compressed files (e.g., rpm, gz, bz2, zip, and Z) represent an important part of the data set (around 24.4 GB). A detailed analysis of compressed files is presented in section 4.1.1.

	Popularity		Storage Space	
Rank	Ext.	% Occur.	Ext.	% Storage
1	.gz	32.50	.rpm	29.30
2	.rpm	10.60	.gz	20.95
3	.jpg	7.54	.iso	20.40
4	.html	4.83	.bz2	6.26
5	.gif	4.43	.tbz2	5.65
6	–	4.16	.raw	4.44
7	.lsm	3.74	.tgz	2.66
8	.tgz	2.90	.zip	2.53
9	.tbz2	2.35	.bin	2.00
10	.Z	2.12	.jpg	0.94
11	.asc	1.84	.Z	0.65
12	.zip	1.59	.gif	0.43
13	.rdf	1.39	.tif	0.31
14	.htm	1.21	.img	0.21
15	.o	1.06	.au	0.19
Total	——	82.26	——	96.92

Table 1: Data profile of our mirrored section of sunsite.org.uk. Files without extension are denoted by the – symbol.

We ran our implementation of the content-defined chunking method over the data using different expected chunk sizes and sliding window lengths. Although in all the subsequent experiments the maximum chunk size permitted was set to 64 KB, the minimum chunk size was fixed to $1/4$ of the expected chunk size; the expected chunk size is set as a parameter in the algorithm. By fixing the maximum chunk size to 64 KB despite changes in the expected chunk size, we aimed to maximise the opportunities of finding identical portions of data in larger chunks. We enforce minimum and maximum chunk lengths to avoid pathological cases such as very small or large chunks. Table 2 shows the amount of information in shared chunks with two different window sizes and three different expected chunk sizes. The last column of Table 2 illustrates the percentage of identical data that was found using fixed size blocks.

Table 2 suggests that the size of the window does not significantly influence the commonality levels. Other studies [16, 18], which use similar techniques to ours, report similar findings. Therefore, in all subsequent experiments we fixed the window size to 48 bytes. We also observe that when the expected size of the chunks is shorter, the percentage of shared data is larger because the probability of finding similar chunks also increases. However, the expected size of the blocks represents a trade-off between the size of the hash table needed to maintain a larger number of entries for each of the unique chunks and the potential storage space saved (see section 4.6).

The percentage of identical data in whole files was only 5%. Therefore, it was possible to find a considerable amount of similar data in partially modified files. On the other hand, the percentages of identical data found using the content-defined chunking method presented only slight differences when compared with those obtained using fixed size blocks. These findings suggest that a storage system handling this kind of data could easily select a fixed size blocking scheme without losing significant storage space savings.

	% of Data in Identical Blocks		
Size	48 B window	24 B window	Fixed Size
8 KB	16.43	16.12	12.13
4 KB	20.27	19.72	17.25
2 KB	25.00	24.18	22.23

Table 2: Percentages of identical data in a 35 GB section of sunsite.or.uk.

Figure 2 shows the distribution of block sizes under the content-defined chunking method when the expected chunk size was set to 8 KB. In particular, we obtained an average block size of 9.2 KB. The algorithm generated 3,730,576 blocks of which 7.6% have at least one identical copy. It is also possible to appreciate the impact of pathological cases on the distribution: the chunks under the minimum block size (2 KB) correspond to files that are shorter than the minimum permitted or to final portions of files. Moreover, the peak at 64 KB corresponds to chunks inserted because of the maximum size allowed.

We focused our attention on the set of files that account for 97% of the total storage space. We explored the levels of similarity of these 15 kinds of files. We fixed the expected chunk size to a value of 4 KB. Table 3 shows the results and compares them against the percentage that can be obtained if we consider whole file contents. We found remarkable patterns in our results. Firstly, it is very difficult to exploit similarity in com-

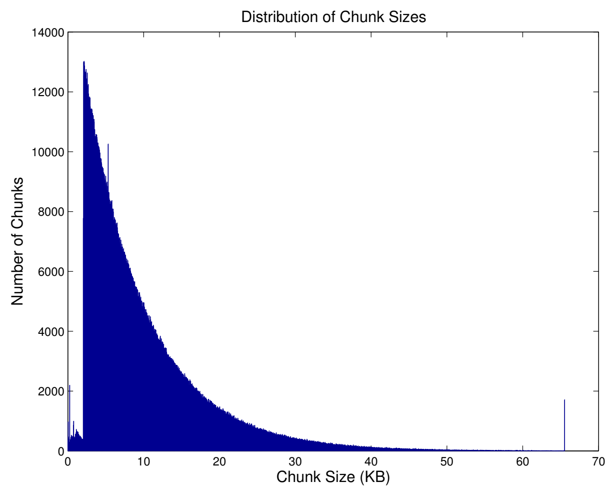

Figure 2: Distribution of chunk sizes obtained from sunsite.org.uk using an expected chunk size of 8 KB and a 48-byte sliding window size.

pressed files; practically all the duplicated chunks were contained in identical files. The behaviour of compressed data will be further investigated in the section 4.1.1.

Secondly, .iso and .img files presented the highest difference in percentage of similarity against the whole file column. These findings suggest that variations between files can be efficiently isolated using the content-defined chunking method, whereas under the whole file approach, even a small change in the file leads to storing a new almost-identical file. The negative effects of this situation are intensified in large files such as ISO image files in which the average size was 280 MB. All other files showed only slight increments if they are compared against the value obtained for the whole file scheme.

Keeping only one copy of the information saves storage space. Using the best scenario, which was the content-defined chunking method with an expected chunk size of 2 KB, our experimental results indicate that a file system would store only 30 GB of unique data instead of the original 35 GB, representing around 14% of storage savings. Duplicate suppression proved to be somewhat more efficient in saving storage space than the tar-compressed version of the data set; the tar.gz file for this data set claimed around 33.3 GB of disk space.

We consider that an explanation to these numbers may be found in the detailed analysis of the files that conform this data set. As has been pointed out before, a large amount of data is already compressed (see table 1); approximately 24.4 GB correspond to rpm, gzip, bz2, zip, and Z files. Note that archives in rpm files are compressed using gzip's deflation method. To a certain extent, redundant data in these files has already been removed as part of the LZ77 [31] compression technique. Compressing compressed data with the same algorithm normally results in more data, not less.

Format	% of Identical Data	
	Content-defined chunks	Whole file content
.rpm	9.08	7.07
.gz	6.71	5.29
.iso	31.26	0.54
.bz2	8.33	8.32
.tbz2	5.02	5.02
.raw	0.47	0.0
.tgz	13.55	13.55
.zip	2.79	1.43
.bin	2.57	0.0
.jpg	0.49	0.28
.Z	3.40	3.14
.gif	2.73	2.69
.tif	95.29	95.29
.img	33.84	8.69
.au	0.0	0.0
all ext.	20.27	5.03

Table 3: Detailed similarity pattern in our mirror of sunsite.org.uk. A 4 KB expected chunk size and a 48-byte sliding window size were used in the content-defined chunking method. The last row of the table shows the values obtained when all files in the data set were analysed.

However, it may still be possible to argue that if the content-defined chunking method was able to remove duplicated chunks of data from the data set, the compressed version of the data would do it as well, resulting in a smaller tar.gz file. The gzip compression algorithm replaces repeated strings in a 32 KB sliding window with a pointer of the form (distance, length) to the previous and nearest identical string in the window. Distances are limited to the size of the sliding window (i.e. 32 KB) and lengths are limited to 258 bytes. As a consequence, redundancy elimination occurs within a relatively local scope; identical portions of data across files will be detected only if files are positioned close in the tar file. In contrast, the content-defined chunking method is able to find data redundancy across distant files in the data set and, in this particular case, to save more storage space than the compressed tar file.

4.1.1 Compressed Data

Compressed files (e.g., gz, bz2, zip, and Z) constitute an important segment of information within our sunsite data set: around 14 GB correspond to compressed files. In this experiment we measured the potential storage space savings that might be obtained if once data is decompressed, the content-defined chunking strategy is used

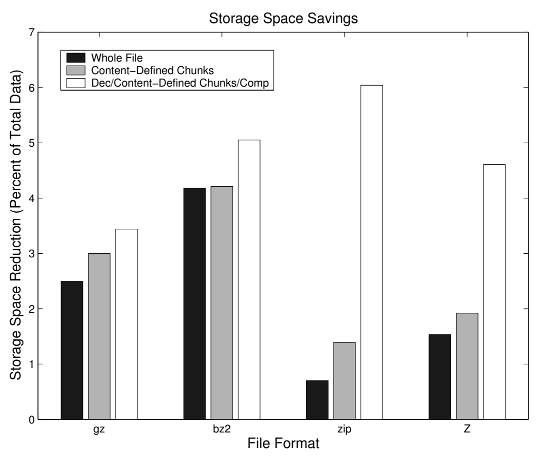

Figure 3: Storage space reductions using three different methods to eliminate duplication in compressed data. Original sizes of the data sets: gz=8.13 GB, bz2=2.4 GB, zip=1 GB, and Z=260 MB.

to suppress duplicates, and then compression is applied again on the resulting set of unique chunks. We decided to use the content-defined chunking method to eliminate data redundancy because it proved to be the most efficient strategy to find data similarity over the four main categories of compressed files. Tools such as `zcat`, `bzcat`, and `unzip` were used to decompress the files. The output stream generated by all these utilities was set as the input to our redundancy elimination program that used a 4 KB expected chunk size.

Figure 3 compares the storage space savings obtained in each of the four categories of compressed files using three different methods to eliminate duplication. The first method suppress duplication from the original collection of compressed files using the whole-file approach. The second method removes similar chunks of data from the original compressed files using the content-defined chunking method. Finally, in the third strategy the files are decompressed, redundancy is removed using the content-defined chunking method, and the resulting unique chunks are compressed again. Although the storage space savings are maximised using the third method, it barely outperforms the result obtained with the content-defined chunking strategy on the original files especially in the cases in which the original data set is of considerable size (gz and bz2 formats).

When duplication is removed from the uncompressed version of the files, the value obtained for the zip category is substantially different to those seen in the other two techniques. It also contrasts with the pattern observed in the other three data sets in which the differences between columns are fairly small. It seems that redundancy elimination specially helped zip files. In general, `zip` is used to deflate one file at a time to then include it into a single object; it limits any potential size reduction to intra-file compression. In contrast, the other compression tools also remove inter-file data duplication (e.g., from all the files in a tar) which finally reduces the benefits of redundancy suppression due to the content-defined chunking method. We conclude that the improvement seen in zip files can be attributed to the ability of the content-defined chunking method to eliminate redundancy within a broader scope (i.e. inter-file redundancy); a gap that zip compression fails to address.

It seems that the comparatively slight storage savings obtained by decompressing files to eliminate redundancy, and compressing the result anew may not be enough to justify the computational overhead of the whole process. Douglis and Iyengar also analyse commonality patterns in compressed data [10]; they reach a similar conclusion to ours.

4.2 Users' Personal Files

The data analysed in this section was held in 44 home directories of different users of the Computer Laboratory. Although the total amount of data processed was only 2.9 GB, this data set presented high diversity in the kind of files stored; we collected 1,756 different file-name extensions in 98,678 files with an average file size of 31 KB. However, the profile of the data set follows a clear pattern. Table 4 shows the most common file-name extension in terms of popularity and storage space. Apart from the files without extension, home directories are mainly used to store files related to word processing and source code development. The 15 file-name extensions showed in Table 4 under the column related to storage space account for over 69% of the whole data set.

In this case the percentage of identical data in whole files was 12.80%. Table 5 shows the result of running our implementation of the content-defined chunking algorithm over the data set using different expected chunk sizes. It also shows the percentage of identical data found when we explored the files using only fixed size blocks. Although the performance of the content-defined chunking algorithm was better, significant storage space savings can also be obtained by using fixed size blocks.

Our results indicate that using the content-defined chunking method and an expected chunk size of 2 KB, which is the best case, a storage utility would maintain only 2.3 GB of the original 2.9 GB. This represents a storage space reduction of 20.6%. However, the compressed tar version of the data set used only 1.4 GB of disk space; considerably outperforming our best duplicate suppression scenario. Similar storage saving ratios were obtained in the next two data sets (research groups'

	Popularity		Storage Space	
Rank	Ext.	% Occur.	Ext.	% Storage
1	–	19.47	.ps	17.24
2	.eps	4.83	–	11.73
3	.obj	3.98	.gz	10.66
4	.tex	3.82	.pdf	6.16
5	.c	3.43	.eps	4.58
6	.gz	2.85	.zip	4.13
7	.gif	2.45	.doc	3.13
8	.ps	2.28	.ppt	2.60
9	.dat	2.11	.obj	1.75
10	.html	1.81	.xls	1.53
11	.h	1.68	.tgz	1.29
12	.log	1.61	.tex	1.25
13	.aux	1.30	.c	1.24
14	.java	1.28	.so	1.19
15	.dvi	1.26	.txt	1.04
Total	——	54.16	——	69.52

Table 4: Profile of the data in 44 home directories of the Cambridge University Computer Laboratory. Files without extension are denoted by the – symbol.

	% of Data in Identical Blocks	
Size	Content-defined chunks	Fixed size blocks
8 KB	24.16	17.22
4 KB	26.76	18.05
2 KB	29.30	19.25

Table 5: Percentages of identical data obtained in 44 home directories of users of the Cambridge University Computer Laboratory. A 48-byte sliding window size was used in the content-defined chunking method.

files and scratch directories) when their corresponding compressed tar files were generated.

4.3 Research Groups' Files

This data set represents a collection of files stored by work groups. We analysed the information of different research groups in the Computer Laboratory. It presents high degrees of similarity because it contains software projects, documents, and information shared among groups of people. This is an ideal environment to save storage space or to reduce the amount of data transmitted based on suppressing identical portions of data. The data set contained a total of 708,536 files in 21 GB of disk space and a 32 KB average file size. Table 6 illustrates the main sections of information arranged by file-name extension popularity and storage space used. Although we found 2,820 different file extensions, our list with the 15 most popular extensions covers more than 66% of the files. Moreover, the percentage of data within the 15 extensions that use the most storage space accounts for over 63% of the total size of the data set.

	Popularity		Storage Space	
Rank	Ext.	% Occur.	Ext.	% Storage
1	.c	15.82	–	15.56
2	.h	14.51	.gz	10.20
3	–	13.90	.ps	8.01
4	.html	4.07	.c	6.66
5	.o	3.45	.a	3.29
6	.c,v	2.79	.pdf	3.15
7	.h,v	2.11	.o	2.97
8	.py	1.95	.eps	2.41
9	.gif	1.56	.tgz	2.18
10	.S	1.40	.h	1.93
11	.gz	1.23	.0	1.82
12	.if	1.14	.html	1.40
13	.m	1.01	.taz	1.29
14	.eps	0.97	.5	1.24
15	.s	0.85	.tar	1.23
Total	——	66.76	——	63.34

Table 6: Profile of the data stored by different research groups of the Cambridge University Computer Laboratory. Files without extension are denoted by the – symbol.

The percentage of identical data in whole files was 25%. It clearly demonstrates an increment over the previous data sets. Table 7 shows the percentages of data in shared blocks for the other two methods: content-defined chunks and fixed size blocks for different expected block sizes. The rates of commonality obtained under the fixed size approach also present high levels of commonality although they are substantially behind the content-defined chunks' percentages.

Under this ideal scenario, not only due to the high amount of identical data contained in whole files but also due to the potential relationships between the files analysed, the use of Rabin fingerprints proved its efficiency. Our results indicate that using an expected chunk size of 2 KB, a storage system would hold only 14 GB in unique blocks in contrast with the 21 GB of the original data set. This means a reduction in storage space of around 33%. Once more, the compressed collection of files (i.e. tar.gz format) used less storage space than the duplicate suppression techniques; it claimed only 8.9 GB of disk space.

4.4 Data Stored in Scratch Directories

The 100 GB of data explored in this section represents the largest data set studied. It contained a total of 1,959,883 files with an average file size of 55 KB. Ta-

Size	% of Data in Identical Blocks	
	Content-defined chunks	Fixed size blocks
8 KB	37.01	28.77
4 KB	39.61	29.62
2 KB	44.59	32.94

Table 7: Percentages of identical data found in several research groups' directories of the Cambridge University Computer Laboratory. A 48-byte sliding window size was used in the content-defined chunking method.

Rank	Popularity		Storage Space	
	Ext.	% Occur.	Ext.	% Storage
1	.c	16.47	.log	33.41
2	.h	15.05	–	24.04
3	–	13.14	.gz	2.76
4	.o	6.55	.c	1.52
5	.0	4.00	.txt	1.46
6	.d	3.99	.o	1.24
7	.gz	1.94	.ps	0.81
8	.3	1.89	.a	0.70
9	.S	1.47	.pdf	0.69
10	.py	1.26	.ul2	0.60
11	.s	1.25	.xls	0.57
12	.ih	1.09	.dl1	0.56
13	.al	0.92	.il1	0.55
14	.1	0.84	.prof	0.55
15	.ast	0.81	.frag	0.53
Total	——	70.67	——	69.99

Table 8: Profile of the data stored in scratch directories in machines of the Cambridge University Computer Laboratory. Files without extension are denoted by the – symbol.

ble 8 gives a partial characterisation of the data set. Once more, it presents the information organised in two main columns according to file-name extension popularity and storage space used. This time the top 15 files, in terms of storage space, account for almost 70% of the total size. It is notable that a large portion of the data set is contained in files without extension or with the .log extension; they represent more than 57% of the whole data set. Apart from this fact, the information was evenly distributed over the whole set of files.

We explored this large data set with an 8 KB expected chunk size which enabled us to reduce the potential number of chunks generated. All the percentages of similarity dropped. Although the percentage of similar data in fully identical files was 14%, the value obtained using the content-defined chunking strategy was only slightly over 20%. The advantages of using the content-defined chunking method are minimal if we consider that the fixed size blocking scheme offered a value of 17.52%. According to our experiments, storage space used in unique chunks for this data set would be 88.26 GB and 91.59 GB, using the content-defined chunking method and fixed size blocks respectively. The compressed tar version of this data set claimed 49 GB of disk space.

We investigated separately the two main categories of files in our data set in terms of storage space used: .log files and files without extension. Using the content-defined chunking strategy .log files and files without extension showed values of around 0.3% and 30% of identical data respectively. Files without extension represent a considerably large amount of data that is difficult to characterise. However, they presented better levels of correlation compared to those obtained for the whole data set (20%). When the fixed size blocking strategy was used on files without extension, the percentage of identical data reached a value of 24.5%.

On the other hand, files with the .log extension offered extremely low levels of similarity and none of the .log files analysed were identical. It negatively influenced the sharing levels of the whole data set. Pathological cases such as these may be difficult to foresee and handle given the limited information that file names in these two categories provide about their contents; no relationship can be inferred *a priori* just by looking at the file names.

4.5 Software Distributions

In this experiment, we looked at five successive Linux kernel source distributions. Initially, we ran the content-defined chunking method over one of the kernels (2.5.34) and then added successive kernel versions one by one; we recorded the results at each step. Furthermore, we analysed the three possible states of the distributions (tar.gz, .tar, and raw files).

When the files were in the tar.gz format the percentage of data in shared chunks was 0% in all cases. Table 9 presents the values obtained in the other formats (tar and raw files) with two different expected chunk sizes. The amount of information shared was substantially greater in tar files when the expected chunk size was smaller. Once more, this result suggests that a smaller expected chunk size increases the likelihood of content overlap.

Furthermore, the last column of Table 9 shows the values obtained when we explored the whole file content of the original files. Comparing these values with those obtained in the 4 KB expected chunk size over raw files, which was the best scenario, in none of the cases the difference is greater than 2%. This lead us to the conclusion that, although the content-defined chunking method efficiently found identical portions of data, their main source was in wholly identical files. As would be expected, no similarity was found among the files in their tar.gz and tar formats when the whole file technique was used.

	8 KB		4 KB		File
Version	tar	raw	tar	raw	raw
2.5.34	1.49	2.39	2.26	3.18	1.5
+2.5.35	43.42	94.46	57.41	95.43	93.8
+2.5.36	44.17	96.33	58.24	96.94	95.12
+2.5.37	44.62	97.09	58.85	97.71	95.31
+2.5.38	44.99	98.33	59.25	98.70	96.86

Table 9: Sharing pattern percentages in a succession of five Linux kernel distributions. A 48-byte sliding window size was used in this experiment.

Figure 4 shows the discrete cumulative distribution function of chunk occurrences that was obtained considering the five kernels in their raw state and using an expected chunk size of 4 KB. These results indicate that the kernel distributions are very similar. When an ordinary file system holds different versions of Linux kernels in its primal state, it is storing the same information almost as many times as versions it holds. Storing a Linux kernel in its primal state adds around 145 MB of data but at least 95% of this information is already contained in chunks of precedent versions of the kernel. Therefore, a hypothetical storage utility would add only 7 MB of new data if reuses the chunks already stored. However, distributing patches of the kernels in their compressed format continues being the most efficient method of propagation. For example, the largest patch in our set of kernels accounts for only 977 KB. Even if we uncompress this patch file, its size is smaller than the size of non identical chunks that were found using the content-defined chunking method. Using the content-defined chunking method a hypothetical storage utility would transmit 7 MB in new chunks while the size of the uncompressed patch is only about 3.7 MB.

However, if these five kernels are decompressed and untarred, then processed with the content-defined chunking method in order to eliminate repetitions, and finally compressed again, the resulting size is only 38 MB. This value is considerably smaller than the original 171 MB used by five tar.gz kernel files, and only slightly over the 34 MB of an individual compressed kernel.

4.6 Associated Overheads

System designers considering employing any of these techniques to reduce data duplication must be aware of computational and storage overheads. Computational overheads are due to the calculation of SHA-1 digests and Rabin fingerprints. Additional CPU time and memory is spent in maintaining the data structures that keep track of the chunks generated (a hash table in our case) and their reference counters.

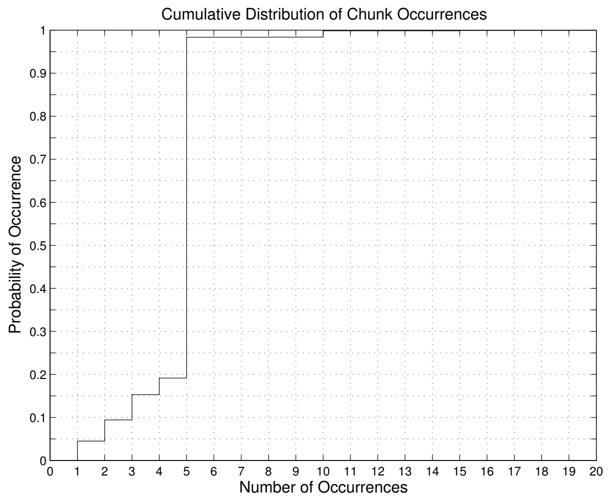

Figure 4: Discrete cumulative distribution function of chunk occurrences of five uncompressed and untarred kernels.

To compare the computational overhead in the three methods analysed, we created a 300 MB file containing random chunks of data taken from the data set corresponding to the Research Groups' files and then ran each of the methods on it. All the experiments were performed using a network-isolated machine with an Intel Pentium III 500 MHz processor. The content-based chunking method involves the generation of Rabin fingerprints and SHA-1 digests of the chunks. It took around 340 CPU seconds to process all the file. Around 76% of the total execution time was spent in tasks related to the computation of fingerprints over a sliding window. The fixed size blocking method took approximately 71 CPU seconds to compute all the necessary SHA-1 digests. Finally, the whole file approach used a total of 62 CPU seconds to calculate the digest. However, the reader should notice that our goal was to analyse data sharing patterns and potential storage space savings in diverse data sets; the prototypes were not implemented having performance as a compelling factor.

SHA-1 computations have been made extremely efficient. Commercially available hardware and operating systems' cryptographic services can be used to compute SHA-1 digests at very high speeds. Furthermore, Rabin fingerprints of a sliding window can be computed efficiently in software due to their algebraic properties, especially if the internal loop of the Rabin fingerprinting method is coded in assembly language [4]. An efficient implementation and computation of Rabin fingerprints on real-life data sets have already been reported [6].

Storage overhead is correlated to the number of unique blocks produced and the data structure used to keep track of their number of occurrences. As mentioned before,

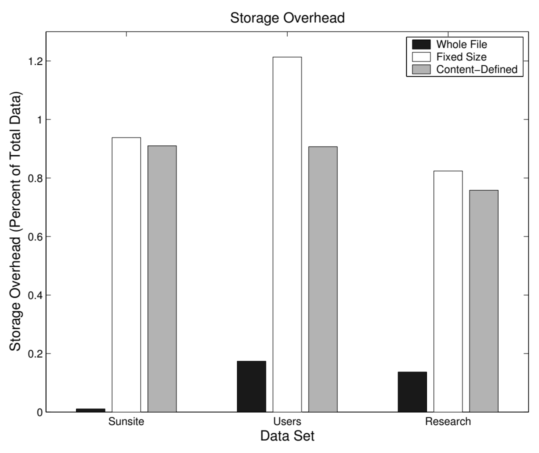

Figure 5: Storage overhead in three different data sets using a 4 KB expected chunk or fixed block size. Original data sets sizes: Sunsite=35 GB, Users=2.9 GB, and Research=21 GB

we used a hash table with this purpose. Figure 5 shows the storage space needed to store the hash table. The overhead has been computed for a 4 KB expected chunk size and fixed block size. It also compares these values with the overhead introduced by the whole-file strategy in which only one digest per file is required. In all the cases the amount of extra space required is small compared with the total size of the data set, but would pose a significant burden were it to be stored in memory.

5 Summary and Discussion

We found remarkable patterns in our results. The content-defined chunking algorithm was the best strategy to discover redundancy in the data sets studied. It consistently reported the largest amounts of data duplication. However, the fixed size blocking strategy also revealed useful levels of similarity. In the sunsite data set these values were considerably close to those obtained using the content-defined chunking method. For other data sets our results are similar to the numbers reported in [9] which revealed a more important difference in duplicated data detected by the content-defined chunking method. As may be expected, the whole-file approach was always at the bottom of the ranking.

The content-defined chunking strategy is specially efficient with potentially correlated data. We obtained high levels of similarity when the program was executed on data held by research groups (44.59%), and expanded source code distributions and software projects (98.7%). When this method was used on more diverse data sets, such as our sunsite.org.uk mirror and scratch directories, the similarity levels dropped (25% and 20% respectively), and were noticeably closer to sharing levels found using a fixed size block approach (22.23% and 17.53% respectively). The whole file content approach reported modest levels of similarity with exceptionally high values in research groups' data (25%) and expanded source code distributions (96.86%). In data sets with a not-so-evident correlation such as the sunsite.org.uk mirror the similarity levels plummeted (5%).

In terms of storage space savings, the tar.gz version of the data sets consistently outperformed the other techniques. However, for systems that need to access and update separate files, probably in a distributed environment, compression is not easy to implement effectively. Chunk or block-based strategies such as the explored in this paper might be a better option in this domain.

In general, packed (i.e. rpm) and compressed data presented low levels of similarity; compression algorithms have already removed a degree of redundancy in the data set. Apparently, the storage space savings that can be obtained by decompressing a large number of arbitrarily selected files in order to remove data duplication from their expanded versions and finally compressing only non-identical chunks (see figure 3) does not justify the extra computational effort involved in this process.

The data structures needed to keep track of the extra information (SHA-1 block digests and reference counts) introduced a very small amount of storage overhead. The amount of extra storage required depends on the number of unique blocks managed by the hash table. As was expected, the whole file approach created only a very small number of unique entries in the hash table (i.e. one per file in the data set). The fixed size and content-defined methods produced comparable amounts of storage overhead, considerably greater than those exhibited by the whole file approach (see figure 5). A practical study of computational overheads incurred in each of the methods remains to be done. Our analysis, together with other experimental results [9], simply point out that there is an important amount of extra computation that has to be considered when using the content-based chunking method.

File access patterns should also be taken into consideration. Whole file content and fixed size blocking strategies present the disadvantage that file updates may lead to the recomputation of SHA-1 digests for large amounts of data. File updates under the whole file technique create the need to recompute the SHA-1 digest for the whole file. Using the fixed size blocking approach, any update that causes a shift at any position of the file will invalidate the SHA-1 digests for the rest of the blocks. As a consequence, reference counters of these blocks have to be decremented and SHA-1 digests have to be computed for the new blocks. However, a fixed size blocking scheme may offer the advantage that blocks can be page-aligned and consequently improve memory performance as is pointed out by Sapuntzakis et al. [25]. On the contrary, updates in the content-defined chunking scheme are self-contained into the blocks where they occurred, thus SHA-1 digests are recomputed only for the modified blocks.

In light of our results, we consider there is no one method able to perform satisfactorily on all data sets. The extra processing and storage space required for each of the techniques, together with usage patterns and typical workloads of specific data sets can be decisive factors when deciding to employ these techniques. The fixed size blocking method offers high processing rates which make it a good candidate for interactive contexts. Despite the fact that the recomputation of SHA-1 digests could represent an inconvenience, especially under workloads in which file updates are common, our experimental results showed that the similarity patterns seen in passive data sets were relatively close to those obtained using the content-defined chunking strategy.

The question seems to be whether in practice the majority of the common blocks would remain valid after file updates and how often these updates occur. File access patterns [24, 30] indicate that file updates present significant locality: only a very small set of files is responsible for most of the block overwrites. Files tend to have a bimodal access, they are either read-mostly or write-mostly. Finally, an important percentage of files, even under different workloads, are accessed only to be read [24]. Considering this experimental evidence we feel persuaded to believe that, in the general case, an important number of blocks will remain valid for their lifetime.

Overall, the levels of data redundancy that can be identified using the fixed size blocking strategy are respectably high and sometimes close to those obtained using the content-defined chunking method. With file access patterns in consideration, the fixed size blocking strategy seems to be a sensible option for the general case; it is simple, acceptably effective, and quite efficient. We consider that the content-defined chunking method is justified only in contexts in which potential data repetition is high and the costs of not identifying redundant portions of data due to scattered updates throughout the file are high. Suppressing duplication at the file level still seems to be a good option especially where the amount of duplicated data is high and enclosed in a group of well connected machines [3]. System designers should take a decision based on the practical trade-offs between saving storage space, bandwidth consumption, and the computational and storage overheads necessary to support each of these methods; the results presented in this work can assist them in such a decision.

Acknowledgements

We thank Pablo Vidales, Tim Moreton, Tim Harris, the anonymous reviewers, and our shepherd Fred Douglis, for their suggestions and feedback. Calicrates Policroniades has a scholarship of the Mexican government through the National Council of Science and Technology (CONACyT).

References

[1] S. Balasubramaniam and Benjamin C. Pierce. What is a File Synchronizer? In *Proceedings of the 4th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 98–108. ACM Press, 1998.

[2] Kenneth Barr and Krste Asanovic. Energy Aware Lossless Data Compression. In *Proceedings of the First International Conference on Mobile Systems, Applications, and Services (MobiSys)*, San Francisco, CA, USA, May 2003.

[3] William J. Bolosky, Scott Corbin, David Goebel, and John R. Douceur. Single Instance Storage in Windows 2000. In *Proceedings of the 4th Usenix Windows System Symposium*, August 2000.

[4] Andrei Z. Broder. Some Applications of Rabin's Fingerprinting Method. In Renato Capocelli, Alfredo De Santis, and Ugo Vaccaro, editors, *Sequences II: Methods in Communications , Security, and Computer Science*, pages 143–152. Springer-Verlag, 1993.

[5] Andrei Z. Broder. On the Resemblance and Containment of Documents. In *Proceedings of Compression and Complexity of Sequences (SEQUENCES'97)*, 1997.

[6] Andrei Z. Broder. Identifying and Filtering Near-Duplicate Documents. In *Proceedings of Combinatorial Pattern Matching: 11th Annual Symposium*, Montreal, Canada, June 2000.

[7] Calvin Chan and Hahua Lu. Fingerprinting Using Polynomial (Rabin's Method). Faculty of Science, University of Alberta, CMPUT690 Term Project, December 2001.

[8] Landon P. Cox, Christopher D. Murray, and Brian D. Noble. Pastiche: Making Backup Cheap and Easy. *ACM SIGOPS Operating Systems Review*, 36(SI):285–298, 2002.

[9] Timothy E. Denehy and Windsor W. Hsu. Duplicate management for reference data. Research Report RJ 10305 (A0310-017), IBM, October 2003.

[10] Fred Douglis and Arun Iyengar. Application-specific Delta-encoding via Resemblance Detection. In *Proceedings of 2003 USENIX Technical Conference*, pages 113–126, San Antonio, Texas, USA, 2003.

[11] Richard G. Guy, Peter L. Reiher, David Ratner, Michial Gunter, Wilkie Ma, and Gerald J. Popek. Rumor: Mobile Data Access Through Optimistic Peer-to-Peer Replication. In *Proceedings of ER Workshop on Mobile Data Access*, pages 254–265, 1998.

[12] Val Henson. An Analysis of Compare-by-Hash. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, Lihue, Hawaii, USA, May 2003.

[13] Richard M. Karp and Michael O. Rabin. Efficient Randomised Pattern Matching Algorithms. Technical Report TR-31-81, Center for Research in Computing Technology, Harvard University, 1981.

[14] Purushottam Kulkarni, Fred Douglis, Jason LaVoie, and John M. Tracey. Redundancy Elimination Within Large Collections of Files. In *Proceedings of 2004 USENIX Technical Conference*, Boston, Massachusetts, USA, 2004.

[15] Udi Manber. Finding Similar Files in a Large File System. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 1–10, San Fransisco, CA, USA, 1994.

[16] Tim Moreton. Pasta: a Distributed Scalable File System for the Pastry Routing Substrate. University of Cambridge, Computer Laboratory Part II Project, 2002.
`http://www.cl.cam.ac.uk/˜tdm25`.

[17] Tim D. Moreton, Ian A. Pratt, and Timothy L. Harris. Storage, Mutability and Naming in Pasta. In *Proceedings of the International Workshop on Peer-to-Peer Computing, Networking*, May 2002.

[18] Athicha Muthitacharoen, Benjie Chen, and David Maziéres. A Low-Bandwidth Network File System. In *Proceedings of the Symposium on Operating Systems Principles (SOSP'01)*, pages 174–187, 2001.

[19] Sean Quinlan and Sean Dorward. Venti: a New Approach to Archival Storage. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST'02)*, Monterey, CA, USA, 2002.

[20] Michael O. Rabin. Fingerprinting by Random Polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[21] Michael O. Rabin. Discovering Repetitions in Strings. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, pages 279–288. Springer-Verlag, Berlin, 1985.

[22] Dickon Reed, Ian Pratt, Paul Menage, Stephen Early, and Neil Stratford. Xenoservers: Accountable Execution of Untrusted Programs. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, pages 136–141, March 1999.

[23] Sean C. Rhea, Kevin Liang, and Eric Brewer. Value-Based Web Caching. In *Proceedings of the 12th International Conference on World Wide Web*, pages 619–628. ACM Press, 2003.

[24] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A Comparison of File System Workloads. In *Proceedings of 2000 USENIX Annual Technical Conference*, San Diego, California, USA, June 2000.

[25] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the Migration of Virtual Computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[26] Jeff Sidell, Paul M. Aoki, Adam Sah, Carl Staelin, Michael Stonebraker, and Andrew Yu. Data Replication in Mariposa. In *Proceedings of the 12th International Conference on Data Engineering*, pages 485–494, 1996.

[27] Neil T. Spring and David Wetherall. A Protocol-Independent Technique for Eliminating Redundant Network Traffic. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'00)*, pages 87–95. ACM Press, 2000.

[28] Andrew Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, April 2000.

[29] Unison, File Synchronizer.
`http://www.cis.upenn.edu/˜bcpierce`.

[30] Werner Vogels. File system usage in Windows NT 4.0. In *Proceedings of the Symposium on Operating Systems Principles (SOSP'99)*, pages 93–109, 1999.

[31] Jacob Ziv and Abraham Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transaction on Information Theory*, IT-23(3):337–343, May 1977.

Notes

[1] ftp://sunsite.org.uk

[2] The maximum number of blocks obtained for any single data set in all of our experiments was $n \approx 18\text{X}10^6$. We indexed these blocks using the first $b = 64$ bits of their SHA digests. The probability of having one or more collisions is given by $1 - (1 - 2^{-b})^n$. This small probability can be neglected in our experimental results.