Providing High Reliability in a Minimum Redundancy Archival Storage System

Deepavali Bhagwat[1] Kristal Pollack[1] Darrell D. E. Long[1] Thomas Schwarz, S.J. [1,2]
Ethan L. Miller[1] Jehan-François Pâris[3]

[1]*Storage Systems Research Center, University of California, Santa Cruz, CA*
[2]*Computer Engineering Department, Santa Clara University, Santa Clara, CA*
[3]*Department of Computer Science, University of Houston, Houston, TX*

Abstract

Inter-file compression techniques store files as sets of references to data objects or chunks that can be shared among many files. While these techniques can achieve much better compression ratios than conventional intra-file compression methods such as Lempel-Ziv compression, they also reduce the reliability of the storage system because the loss of a few critical chunks can lead to the loss of many files. We show how to eliminate this problem by choosing for each chunk a replication level that is a function of the amount of data that would be lost if that chunk were lost. Experiments using actual archival data show that our technique can achieve significantly higher robustness than a conventional approach combining data mirroring and intra-file compression while requiring about half the storage space.

1. Introduction

Archival digital data continues to accumulate at an astounding pace. It will increase ten-fold between 2006 and 2010 to over 27 exabytes in the commercial and government sectors [16]. As digital data accrues at ever-increasing rates, organizations also face increasing regulatory pressure to retain data for long periods of times and may be required to retrieve data occasionally. In this context, maintaining the availability of archived data becomes part of the due diligence that organizations are expected to exercise.

To reduce the costs incurred for storing such large volumes of archival data, this data is compressed using various compression techniques. Several companies [7, 8, 11] already use various forms of compression for their archival storage solutions. Our project, Deep Store [34], uses both intra-file and inter-file compression to reduce redundancies. One such inter-file compression technique used by Deep Store is chunk-based inter-file compression [17]. In this technique files are split into variable-length chunks and stored. If any redundant chunks are found, they are stored as references rather than as duplicates. In many cases, this method achieves excellent compression ratios [33].

While archival systems require good compression, they must also ensure that data is preserved over long time periods. Compression techniques, while they save storage space, also have the potential to reduce reliability. For example, when inter-file compression is used, dependencies are introduced between files that share the same chunk. If such a shared chunk is lost, a disproportionately large amount of data becomes inaccessible because of the loss of all the files that share this chunk. As a result, some chunks are much more important than others and need to be protected at a higher level to maintain good overall reliability. In this paper, we consider the effects of inter-file chunk-based compression on the reliability of the archival system. Our approach to improving reliability is to add redundancy strategically by *selectively* replicating chunks. We have developed heuristics that weigh the importance of a chunk and use this weight to prescribe the level of replication for the chunk. A part of the storage space saved by compression is thus reinvested in better protecting the important chunks. As a result, we achieve even better data reliability than mirrored (degree of mirroring = 2) Lempel-Ziv (LZ) compressed [35] files, while still using about half of the storage space of mirrored LZ-compressed files and with replication/mirroring as the means to introduce redundancies.

We can also improve reliability by using other redundancy introducing techniques such as erasure correcting codes used in RAID levels 5 and 6, and by introducing different data placement, failure detection and recovery disciplines. We do not consider these here mainly because they offer intricate trade-offs between speed of recovery, ease of recovery, and computational and storage overhead.

To focus our efforts, our analysis assumes constant device failure rates, constant repair rates, and independence of failures. We only investigated replication as a redundancy strategy and used a simple concept of *robustness*, in which we measured the amount of data loss caused by the loss of a

Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation
of Computer and Telecommunication Systems (MASCOTS '06)
0-7695-2573-3/06 $20.00 © 2006 IEEE

small percentage of devices in addition to a standard failure model with all usually made simplifying assumptions. We will investigate other redundancy strategies in the future.

2. Deep Store: An Overview

Deep Store [34] is a large-scale archival storage system that stores volumes of immutable data efficiently, with high reliability and accessibility. It incorporates methods for inter-file and intra-file compression to utilize storage space very efficiently. Deep Store uses three techniques to reduce storage demands: content-addressable storage [11], delta compression [2, 9] and sub-file chunk-based compression [17]. Other storage systems such as Venti [23], EMC Centera [11], StorageTek's Intellistore [28], Nexsan's SATABeast [18], Avamar's Axion [4], and Permabit [21] also use content addressable storage. In content-addressable storage, a single feature or hash, also called *content address* is computed over an entire file and this hash is used to find identical files already in the archive. Two files with the same content address are likely to be identical, but the system still must check for possible collisions. If the files are identical, the system only stores a reference to the existing file rather than storing the file again. In delta compression, the system first searches for a file similar to the file currently being stored, and then stores only the differences between the current file and the stored file. A pointer to the stored file and metadata for reconstructing the current file are stored with the differences.

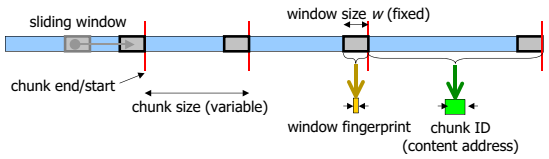

Figure 1. Sliding Window Technique

Our study focuses only on chunk-based compression. Chunk-based compression or *chunking* subdivides a file deterministically into variable-sized blocks or chunks. This technique was first used in the Low-bandwidth Network File System [17]. Chunking is a two step process. First, a file is divided into chunks in a deterministic fashion. Second, the content within every chunk is used to compute its features. Figure 1 shows a data stream or a file represented by the long horizontal rectangles and chunk boundaries indicated by short vertical lines. To divide the file into chunks, starting from the beginning of the file, we examine the its contents as seen through a fixed sized (overlapping) sliding window. At every position of the window, a fingerprint or digital signature of its contents is computed. In practice we use Rabin fingerprints [24] to calculate the digital signature of the contents of the sliding window for their computational efficiency in this scenario. Rabin fingerprinting by random polynomials computes a hash of a fixed size from a binary string of arbitrary length. Rabin fingerprinting functions are of a class of randomized functions that exhibit uniform distribution of results over arbitrary data. In the scope of our work, we select a random function from a set of functions, such as, the set of all irreducible polynomials of a fixed degree. Once selected, the fingerprinting function is retained to produce deterministic results. When the fingerprint meets a certain criteria, such as when the value, modulo some specified integer divisor, is zero; that position of the window defines the boundary of the chunk. This process is repeated until the complete data stream has been broken into chunks. In the second step we compute a digest or hash function over the contents of the chunk using Rabin fingerprints. This digest is the content address of the chunk. This content address can also be computed using functions such as MD5 [25], SHA-1 [19] or higher SHA standards [20]. You *et al.* [33] have evaluated chunking and delta-compression with respect to their storage space efficiency and computational complexity. They conclude that delta-compression and chunking outperform traditional stream compression methods with respect to storage space efficiency. Chunking requires two hashing operations per byte in the input file: one fingerprint calculation of the fixed size window and one chunk digest calculation. Once the file is broken into chunks, only the unique chunks are actually stored. Deep Store identifies a chunk in the same way as it identifies files: using a content address (a hash or digest of the content) to determine if a chunk already exists in the system. After this type of compression, a file consists of a set of references to chunks and the metadata necessary to rebuild the file.

3. Effect of Compression on Reliability

Chunk-based interfile compression can be quite effective for certain types of data. You, *et al.* [33] have characterized this data as files that evolve slowly mainly through small changes, additions, and deletions. One of the data sets for our experiments consists of 9.8 GB of several web sites: those of the University of California at Santa Cruz, Santa Clara University, Stanford University, University of California at Berkeley, BBC, NASDAQ, CERT, CNN, SANS, SUN, CISCO, and IBM as they developed over time. We obtained them from the Internet Archive's Wayback machine [29]. This data is a representative sample of archival data, and will greatly profit from chunk-based compression due to the incremental nature of the changes that it has gone through. Chunk-based inter-file compression stores this data using a storage space of 1.74 GB for chunks

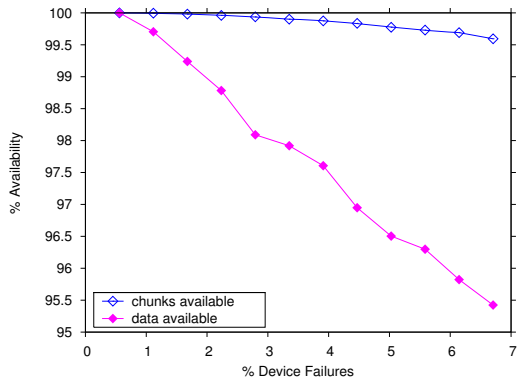

Figure 2. Effect of inter-file dependencies on robustness

and 280 MB for metadata. On the other hand, when each file was compressed using LZ-compression, the total storage space required was 5.6 GB. Clearly, chunk-based compression can use significantly less storage space than LZ-compression.

To study the effect of chunk-based compression on reliability we conducted a pilot experiment using this data. We compressed the files using chunk-based compression, and then mirrored the chunks and stored them evenly across a set of 179 devices. The devices were then randomly selected to fail independently, resulting in the loss of up to 7% of the total devices. Figure 2 shows the availability as a function of device failures in two forms. The first form is the percentage of raw chunks available. The second form is the percentage of original data that could be reconstructed from these available chunks. The data robustness is seen to be significantly lower than the chunk robustness. For example, when 6% of the devices fail, about 99.5% of all chunks are still available, but only 96% of all the data is still available. This increased data loss happens due to inter-file dependencies formed as common chunks are shared amongst multiple files. These inter-file dependencies are shown schematically in Figure 3 where the dependencies are measured by the number of file references to a chunk. If a common chunk is no longer available, all the files that depend on the chunk are lost resulting in a disproportionately large amount of data loss that we see in Figure 2. This increased data loss illustrates how good compression can be detrimental to reliability in the event of device failures, due to inter-file dependencies formed by common chunks shared between multiple files.

Chunk-based compression achieved excellent compression ratios by removing redundancies across files. However, this introduced inter-file dependencies that hampered

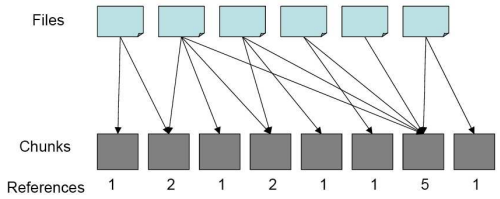

Figure 3. Inter-file dependencies

reliability. Since compression saves significant amount of storage space, some of this savings can be used to regain reliability. A simple way of doing this is to use a higher degree of replication. However, we have used a more discerning approach to do this — one that decides the replication level for a chunk depending on its popularity or importance so that we did not end up defeating our original purpose of efficient storage space utilization.

4. Storage Strategy

The simple experiment in the previous section showed that the loss of a small number of chunks can result in a disproportionately large data loss. To protect against this, our heuristics replicate certain important or popular chunks more aggressively than the others. To accomplish this, we developed some good measures for the importance of a chunk. This measure of importance, or *weight*, is used to determine the number of replicas for each chunk and their distribution across devices.

4.1. Replicas Based on Chunk Weight

The effects of the loss of a chunk can be measured by the amount of data lost and by the number of files that are inaccessible as a result of this loss. Correspondingly, we measure the importance of a chunk either by the number of files that depend on it (the reference count), or by the amount of data (the byte count) that depends on it. This approach defines the weight of a chunk as either the reference count or as the byte count that depend on it, and determines the number of replicas for each chunk using a logarithmic function of the chunk's weight.

The following calculation justifies our intuition to use a log-based function to calculate the number of replicas for a chunk based on its weight. Assume that we have n files that all depend on one common chunk, called the central chunk. Each file also depends on another peripheral chunk, as shown in Figure 4, that is particular to that file alone. Assume that we keep k replicas of the single central chunk and l replicas of the remaining peripheral chunks. We assume that all chunks have the same size. The total storage used is

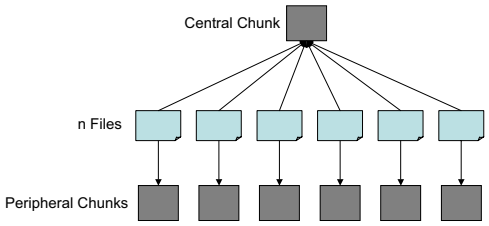

Figure 4. Central and peripheral chunks

then proportional to

$$S = k + l \cdot n$$

Assume that a single storage device fails with probability p. The central chunk is lost with probability p^k and the peripheral chunks with probability p^l each. We lose all files if we lose the central chunk; otherwise the expected number of lost files L is np^l, so that

$$L = np^k + (1 - p^k)np^l \approx np^k + np^l$$

Taking the derivative of the approximation for the loss, we obtain the following relation for an optimal k:

$$n \log(p)p^k - \log(p)p^l = 0$$

Solving for k gives

$$k = \frac{S}{n+1} + \frac{n}{n+1} \cdot \frac{\log(n)}{\log(1/p)}$$

The first addend converges to zero and the second is proportional to the $\log(n)$. If the central chunk is much larger than the smaller chunks and n is fixed, then replicating the peripheral chunks at a higher rate than the central chunk leads to lower expected loss.

Even if a chunk has only a single dependency, it must be protected. Therefore, our heuristic keeps at least 2 copies of every chunk. We choose a function of type

$$k = f(w) = \min(\max(2, a + b \log(w)), k_{max})$$

to calculate the number of replicas k depending on a chunk's weight w. Here, a and b are constants that will yield different storage space utilization and robustness levels; a and b need to be determined experimentally depending on the data set. A base level of replication, a, is added as an additional tuning parameter that is independent of w to offset the effect of $b \log(w)$. As b increases, the number of replicas, based on the weight w of a chunk, increases. For some chunks with a large weight, w, the number of replicas suggested by our logarithmic function can be very large. As k increases the gain in reliability obtained due to each additional replica diminishes. For this reason, the maximum number of copies of a chunk is capped at k_{max}.

4.2. Chunk Distribution

In addition to the replication level for various chunks, the placement of the replicas also affects the reliability of our storage scheme. If a device is lost and almost all chunks on the device belong to the same set of files that reside on the lost device, then the effect of this failure has limited repercussions for the rest of the system. Conversely, if a file depends on chunks distributed over a large set of devices, then it is more vulnerable since it is easier to lose this file through the failure of any of those devices. Consequentially, we want to reduce *inter-device* dependencies. Of course, we should store copies of the same chunk on different devices. Other than that, we try to store chunks belonging to the same file on the same device.

Since our system stores archival data, we assume that files enter the system in batches. As a file enters, the chunks are extracted and stored, filling up the disks as data arrives, on one disk at a time. When the current disk is full, a new disk is used. If a chunk is new, it is stored on the current disk, but not yet replicated in anticipation of another file in the same batch using the same chunk. This lazy replication scheme reduces inter-device dependencies. If a chunk is already in the system, the system determines whether, after updating its weight, another replica must be stored. If this is the case, the replica is stored on the current disk. Otherwise, the system does nothing—there are sufficient replicas for the chunk already. After the batch of files has finished processing, the weights of all the chunks are checked to see if any of them need to be replicated. In such cases, replicas for the latest chunks are stored on the most recently used disk. While our scheme does not completely eliminate inter-device dependencies, it greatly reduces them.

5. Experimental Setup

Our data set consists of two sets of files obtained from the Internet Archive [29] and the other from the *Santa Cruz Sentinel* [30]. As described in Section 3, the data set from the Internet Archive contains web sites as they develop over time. *The Santa Cruz Sentinel*, our local newspaper, maintains an archive, as do many newspapers. This set consists of HTML, PDF, image (TIFF and JPG) and Microsoft Word files with quite a bit of repetitive information such as templates for web pages. Table 1 gives statistics for both data sets, showing that both data sets are well-suited for chunk-based compression. The use of chunk-based compression results in substantial savings in storage space when compared to the storage space required when using LZ-compression to compress each file individually.

We used our prototype program `chc` [34] to chunk files. The files that form the target data set were input to `chc`, producing an output composed of chunks derived from the

Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '06)
0-7695-2573-3/06 $20.00 © 2006 IEEE

Table 1. Statistics of the Experimental Data

	Internet Archive	Santa Cruz Sentinel
Number of Files	196664	158900
Minimum File Size	1 B	2 B
Maximum File Size	21 MB	263.78 MB
Average File Size	52.50 kB	301.46 kB
Total File Space	9.84 GB	40.22 GB
LZ-compressed File Space	5.62 GB	31.14 GB
Unique Chunks	6240360	28806477
Minimum Chunk Size	9 B	9 B
Maximum Chunk Size	12.61 kB	12.61 kB
Average Chunk Size	299.90 B	243.11 B
Total Chunk Space	1.83 GB	7.5 GB

original files. These chunks were further compressed individually using the *zlib* [10] compression library. `chc` captures a list of chunk identifiers for each file, as well as the identifier and size for each chunk. Extended size blocks—*megablocks* [34]—were used to store both chunks and LZ-compressed files to minimize the storage overhead from unused portions of blocks. Since the metadata for file identifiers and size of every file needs to be stored for LZ-compressed files as well, the storage overhead due to this metadata has been omitted for both chunk-based compression and LZ-compressed files. However, the overhead of a 128-bit content address for every chunk, whether original or replica, and for all chunk identifiers per file has been accounted for when calculating the total storage space required when using chunk-based compression.

To evaluate the success of our heuristic-based replication strategy, we measured the ratio of availability to the utilized storage space. Evaluation of the latter is easy, while the former is difficult because availability depends on too many factors such as data placement, speed of recovery and device failure rates. Further, availability calculations make simplifying assumptions that are not always justified, such as independent failures of devices and constant device failure rates. In addition, an archival storage system tries to protect data over a period of time that is longer than the lifespan of the individual devices and, in such a system, common causes of failures such as batch and vintage failures become important. Instead of trying to make a number of reasonable assumptions and ending up with a large number of possible storage systems, we decided to measure availability in the form of *robustness*, defined as the fraction of data available given a certain percentage of unavailable storage devices, rather than in the usual metric of mean time to data loss or percentage of data loss per year.

In this assessment, we assume a simple model based on replication—the only way we introduce redundancy is by storing more replicas. Though we decided to use replication instead of more involved mechanisms to generate redundancy, there are still many potential parameters to choose in a storage system, such as replica placement, failure detection and repair. Since our target applications are so large that they store data on hundreds, if not thousands of disk drives, we use artificially small devices to store the data so that our sample workloads are stored over many devices. By not modeling repairs of failed devices, we are being conservative. This is important because, in any real system, repairs would occur after a failure, so there would be a much smaller chance of data loss.

To test the robustness of the system, we began by selecting a percentage of devices independently and at random and failing them, starting with 1 device and continuing until 7% of the total devices have failed. By showing availability at relatively low levels of device failure, we simulated the effects of temporary device loss. The devices would be replaced later, but the data on them is lost due to failure. The same is true for mirrored LZ-compressed files. We used the chunk distribution strategy of Section 4 to store chunks extracted from both the data sets onto a set of devices. The same distribution strategy was used with LZ-compressed files. The mirrored LZ-compressed files of the Internet Archive were stored evenly on 188 devices of 64 MB each while those of the *Santa Cruz Sentinel* were stored evenly on 132 devices of 512 MB each. However, every device was filled to capacity. Hence, measuring the *percentage* of failed devices was equivalent to measuring the percentage of data lost. The capacity of every device was increased for *Santa Cruz Sentinel* data to avoid fragmentation of a file across several devices. We had to take care not to fragment files when using LZ-compression because this would introduce the same type of multiple-device dependencies for files that arise when using the chunking method, the effects of which we were measuring. Chunks for both data sets were stored on smaller devices than those used when storing the same data that was LZ-compressed to make sure that we distributed chunks onto the same number of devices as those used by the mirrored LZ-compressed files thereby facilitating a fair comparison between the two. We ended up using an additional 5 disks on average when storing chunks. We could not ensure using *exactly* 188/132 devices since it was not possible to know apriori the number of redundant chunks that would be added with different redundancy schemes. Once we randomly chose the failed devices, we then calculated how many files and how much data we could reconstruct using the remaining devices. The performance of chunk-based compression was compared with that of LZ-compressed files on the basis of the robustness and storage space consumed.

Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation
of Computer and Telecommunication Systems (MASCOTS '06)
0-7695-2573-3/06 $20.00 © 2006 IEEE

6. Results

We calculate the weight, w, of a chunk using two heuristics: the number of files and the size of data depending on a chunk. The weight of a chunk in terms of the number of files, F, depending on a chunk, is calculated as $w = F$. The weight of a chunk in terms of the size of the dependent data is calculated as $w = D/d$, where D is the sum of the sizes of all the files that depend on this chunk and d is the average size of a chunk. The number of replicas, k, calculated using w in the log based function of Section 4.1, is rounded off to the nearest integer. For each experiment we have measured the storage space used as a percentage of the storage space used by the original *uncompressed* data.

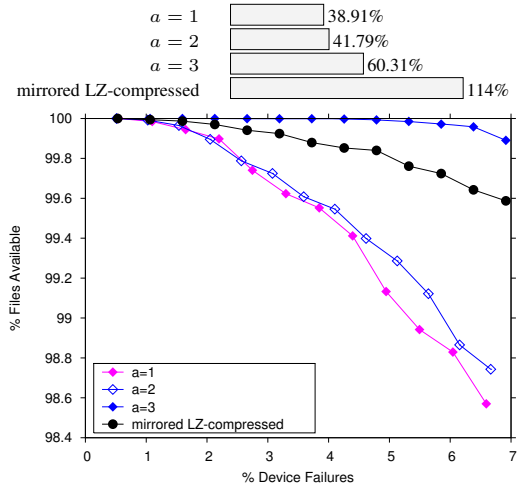

Figure 5. Effect of a on robustness using heuristic $w = F$ with $b = 1$, $k_{max} = 4$

The first set of experiments demonstrates the use of the two heuristics. We wanted to study how the robustness is affected by varying the base level of replication, a. By increasing the base replication level, the number of replicas for *all* the chunks increases, resulting in better robustness. The results of these experiments, conducted using Internet Archive data and with $k_{max} = 4$, are shown in Figures 5 and 6.

In Figure 5 we show robustness using the number of files depending on a chunk as a heuristic, *i. e.* $w = F$. Hence, we measured availability in the number of files available, not amount of data available. Here, with $b = 1$, we vary a and see that the robustness increases with increasing values of a. The system is not very robust when $a = 1$ because when using $a = 1$, 90% of the chunks were replicated just once. We showed in Section 3 that when all the chunks are uniformly replicated just once, the robustness suffers. We see the same effects when $a = 2$, where around 80% of the total chunks were replicated just once. At $a = 3$, our system is *more* robust than mirrored LZ-compressed files and uses only 52.75% of the storage space required by mirrored LZ-compressed files.

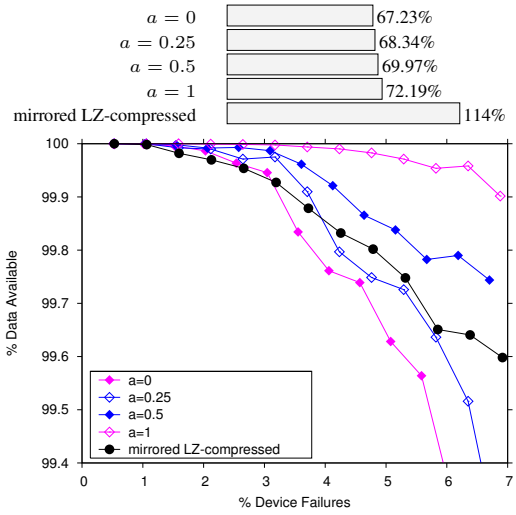

Figure 6. Effect of a on robustness using heuristic $w = D/d$ with $b = 0.4$, $k_{max} = 4$

In Figure 6, we show a similar effect of a on the robustness, but, using dependent data as a heuristic, *i. e.*, $w = D/d$, with $b = 0.4$. Again, we see that by increasing a the system's robustness improves. At $a = 0.5$, our system is more robust than mirrored LZ-compressed files and uses only 61.20% of the storage space used by mirrored LZ-compressed files. Further increase in a increases the robustness even more, albeit at the expense of additional storage space.

The results of the above experiments show that the robustness of our system exhibits the same trends when we use either heuristic, $w = F$ or $w = D/d$. The rest of the results presented here use dependent data as the heuristic, *i. e.*, $w = D/d$; however, the same trends are found with the number of references being used as a heuristic.

If we do not restrict the number of replicas of a chunk, k, to a predefined maximum, k_{max}, some chunks end up having a very large number of replicas, especially for higher values of b. However, as the number of replicas increases, the gain in robustness that every replica rewards us with diminishes in value. To study the effect of varying k_{max}, we measured the robustness of the Internet Archive data with $b = 0.55$ and $a = 0$, as shown in Figure 7 for $k_{max} = 4$ and $k_{max} = 5$ compared with that obtained with no limit on k. It is clear that limiting the number of replicas with k_{max}

Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation
of Computer and Telecommunication Systems (MASCOTS '06)
0-7695-2573-3/06 $20.00 © 2006 IEEE

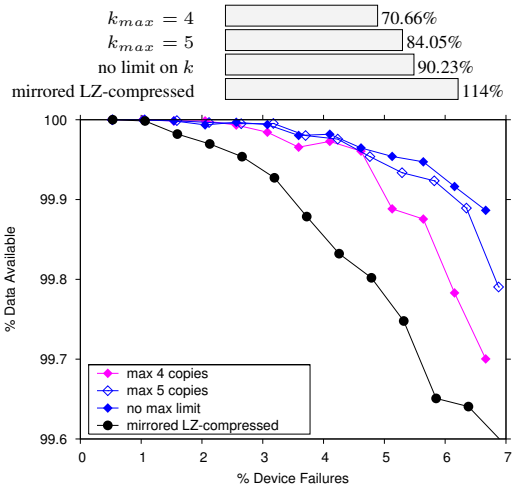

Figure 7. Effect of limiting k on robustness using heuristic $w = D/d$, $b = 0.55$, $a = 0$

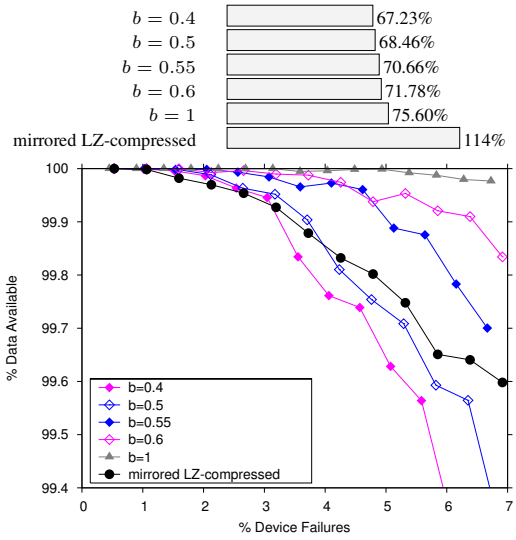

Figure 8. Effect of b on robustness using heuristic $w = D/d$ with $a = 0$, $k_{max} = 4$

does not result in a noticeable loss in robustness, but *does* result in significant savings in storage space.

In our next experiment, we studied the effects of varying b, which will improve robustness by increasing the number of replicas for the more important (higher weight) chunks. This comparison is shown in Figure 8 using Internet Archive data. As b increases, for a given w we begin to get higher values for k, resulting in an increase in the storage space required and the robustness of the system as can be seen in Figure 8. At $b = 0.55$, our system is more robust than mirrored LZ-compressed files, but uses only 61.98% of the storage space required by LZ-compressed files.

Figure 9 depicts the robustness of the second data set, from the Santa Cruz Sentinel, when using different values for b. Here, too, our approach is more robust than when using mirrored LZ-compressed files. With $b = 1$, we use only 48.41% of storage space of the base LZ-compression approach.

As we increase the redundancies the storage space required by metadata also increases. For the Internet Archive data the storage space used by the metadata constituted 5% of the total storage space. For the *Santa Cruz Sentinel* the metadata required 5.6% of the total storage space.

We have used both, the number of files and the amount of the dependent data as heuristics for determining the weight of a chunk. The choice of heuristic depends on the corpus. If the sizes of files in a corpus are indicative of their importance, then the dependent data heuristic should be chosen. However, if the importance of a file in a corpus is independent of its size, or all the files in the corpus are equally important, then the number of files should be chosen as a heuristic. The same metric used in the heuristic must then be used for measuring the robustness of the system; *i. e.*, when using the number of files in the heuristic we use the number of available files as the measure of robustness, whereas when using dependent data as heuristic we use the amount of available data. In other words, if all the files are equally important, then one should measure the system robustness in the number or percentage of files available. We have investigated the effects of the parameters a, b and k_{max} on the robustness and the storage costs of an archival system using chunk-based compression. By choosing an appropriate combination of these parameters we can achieve both a higher robustness and lower storage space utilization compared to traditional LZ-compression techniques.

7. Related Work

Several systems that exploit data redundancy at different levels of granularity have been developed in order to improve storage space efficiency. One class of systems detects redundant chunks of data at granularities that range from entire file, as in EMC's Centera [11], down to individual fixed-size disk blocks, as in Venti [23] and variable-size data chunks as in LBFS [17].

RAID [6] is a device driven method for introducing redundancy and thus ensuring the reliability for storage systems. OceanStore [14] aims to provide continuous access to persistent data on a global scale and uses automatic replication strategies to boost reliability of the system in the face

Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation
of Computer and Telecommunication Systems (MASCOTS '06)
0-7695-2573-3/06 $20.00 © 2006 IEEE

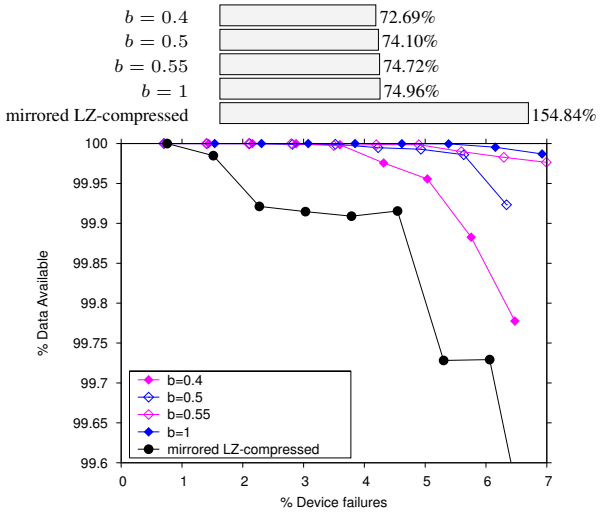

Figure 9. Effect of b on robustness using heuristic $w = D/d$ with $a = 0$, $k_{max} = 5$, Santa Cruz Sentinel data

of disasters. FARSITE [1] is a distributed file system that achieves reliability through replication of file system metadata, such as directories, and file data. FARSITE chooses replication instead of erasure coding schemes to avoid the additional overhead of latter when reconstructing a piece of information. Other file systems such as PASIS [12] and Glacier [13] also make use of aggressive replication to guard against data loss. The LOCKSS project [15] uses a peer-to-peer audit and repair protocol to preserve the integrity and long-term access to collections of documents. Baker *et al.* [5] suggest that long term reliability additionally requires auditing the integrity of data above the level of the storage devices. The surplus storage space we save by using interfile compression can be used to implement proactive policies for ensuring reliability [31], verifying the data integrity [27], and developing recovery strategies [32] for large scale storage systems.

8. Future Work

In addition to chunk-based compression, Deep Store also uses delta compression to archive data. We will study the characteristics of delta compression and develop heuristics for reliability as we have done here for chunk-based compression.

In this work, we have used only one method of introducing redundancies; replication. We will experiment with other mechanisms such as RAID-5 parity, erasure correcting codes, and Reed-Solomon block codes [3, 22, 26].

We will also address the problem of data placement or chunk storage in conjunction with the hardware and its failure statistics such as, mean time to failure of disks. While increasing the redundancy of a high risk chunk we will use such statistical data to formulate strategies regarding whether the redundant chunk must be stored on another sector of the disk, another disk or another device altogether.

9. Conclusions

The chunk-based inter-file compression used in the Deep Store archival system gives very good compression ratios by removing inter-file redundancies. However, these reduced redundancies can be detrimental to the robustness of the data. We have presented a simple strategy to increase the robustness of data with chunk-based compression without compromising the storage space savings obtained by the compression. Our strategy allows us to control the balance between storage space savings and reliability by a choice of heuristics and parameter variation. This strategy gives both a higher robustness and significant storage space savings compared with traditional LZ-based compression.

We have shown that choosing the right number of replicas for each data chunk can achieve a much higher robustness while using about half of the storage space required by mirrored LZ-compression. Furthermore, by controlling the parameters in our replication strategy, we can achieve an even higher robustness (close to 100%) for a small percentage of device failures. This higher robustness together with the savings in storage space is useful for future inclusion of repair models for the Deep Store archival system. Our performance can only improve when we use other redundancy strategies such as RAID and erasure codes. By adjusting the number of replicas of individual data chunks based on our heuristics, Deep Store and other long-term archives can reduce storage space requirements and thus costs while simultaneously increasing robustness, making the long-term storage of data both more affordable and more reliable.

10. Acknowledgments

The authors would like to thank Bruce Baumgart of Internet Archive, Mike Blaesser and Bob Smith of the Santa Cruz Sentinel for giving them access to their data for this work, Mary Baker of Hewlett-Packard Laboratories for her comments, Lawrence You for his help with the use of the Deep Store prototype, and Kevin Greenan for his excellent systems support.

This research was supported in part by a grant from Hewlett-Packard Laboratories, Microsoft Research, and by National Science Foundation Grant CCR-0310888. We would also like to thank the industrial sponsors of the

SSRC, including IBM Research, Intel, Microsoft Research, Network Appliance, Rocksoft, Symantec, and Yahoo! for their generous support.

References

[1] A. Adya, W. J. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, Dec. 2002. USENIX.

[2] M. Ajtai, R. Burns, R. Fagin, D. D. E. Long, and L. Stockmeyer. Compactly encoding unstructured inputs with differential compression. *Journal of the Association for Computing Machinery*, 49(3):318–367, May 2002.

[3] G. A. Alvarez, W. A. Burkhard, and F. Cristian. Tolerating multiple failures in RAID architectures with optimal storage and uniform declustering. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 62–72, Denver, CO, June 1997. ACM.

[4] Avamar Technologies Inc. http://www.avamar.com.

[5] M. Baker, M. Shah, D. S. H. Rosenthal, M. Roussopoulos, P. Maniatis, T. Giuli, and P. Bungale. A fresh look at the reliability of long-term digital storage. In *Proceedings of EuroSys 2006*, pages 221–234, Apr. 2006.

[6] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[7] Data Domain. http://www.datadomain.com.

[8] Diligent Technologies. http://www.diligent.com.

[9] F. Douglis and A. Iyengar. Application-specific delta-encoding via resemblance detection. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 113–126. USENIX, June 2003.

[10] *zlib* Compression Library. http://www.zlib.net.

[11] EMC Corporation. EMC Centera: Content Addressed Storage System, Data Sheet, Apr. 2002.

[12] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient Byzantine-tolerant erasure-coded storage. In *Proceedings of the 2004 International Conference on Dependable Systems and Networking (DSN 2004)*, June 2004.

[13] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, May 2005. USENIX.

[14] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Cambridge, MA, Nov. 2000. ACM.

[15] P. Maniatis, M. Roussopoulos, T. J. Giuli, D. S. H. Rosenthal, and M. Baker. The LOCKSS peer-to-peer digital preservation system. *ACM Transactions on Computer Systems*, 23(1):2–50, 2005.

[16] J. McKnight, T. Asaro, and B. Babineau. Digital Archiving: End-User Survey and Market Forecast 2006–2010. *The Enterprise Strategy Group*, Jan. 2006.

[17] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Oct. 2001.

[18] Nexsan Technologies. http://www.nexsan.com.

[19] NIST. Secure hash standard. FIPS 180-1, Apr. 1995.

[20] NIST. Secure hash standard. FIPS 180-2, Aug. 2002.

[21] Permabit Inc. http://www.permabit.com.

[22] J. S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software—Practice and Experience (SPE)*, 27(9):995–1012, Sept. 1997. Correction in James S. Plank and Ying Ding, Technical Report UT-CS-03-504, U Tennessee, 2003.

[23] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *Proceedings of the First Conference on File and Storage Technologies (FAST)*, pages 89–101, Monterey, California, USA, 2002. USENIX.

[24] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[25] R. Rivest. The MD5 message-digest algorithm. Request For Comments (RFC) 1321, IETF, Apr. 1992.

[26] T. J. Schwarz. Generalized Reed Solomon codes for erasure correction in SDDS. In *Workshop on Distributed Data and Structures (WDAS 2002)*, Paris, Mar. 2002.

[27] T. J. E. Schwarz, Q. Xin, E. L. Miller, D. D. E. Long, A. Hospodor, and S. Ng. Disk scrubbing in large archival storage systems. In *Proceedings of the 12th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '04)*, pages 409–418. IEEE, Oct. 2004.

[28] Storage Technology Corp. http://www.storagetek.com.

[29] The Internet Archive. http://www.archive.org.

[30] The Santa Cruz Sentinel. http://www.santacruzsentinel.com.

[31] Q. Xin, E. L. Miller, T. J. Schwarz, D. D. E. Long, S. A. Brandt, and W. Litwin. Reliability mechanisms for very large storage systems. In *Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 146–156, Apr. 2003.

[32] Q. Xin, E. L. Miller, and T. J. E. Schwarz. Evaluation of distributed recovery in large-scale storage systems. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pages 172–181, Honolulu, HI, June 2004.

[33] L. L. You and C. Karamanolis. Evaluation of efficient archival storage techniques. In *Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies*, College Park, MD, Apr. 2004.

[34] L. L. You, K. T. Pollack, and D. D. E. Long. Deep Store: An archival storage system architecture. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)*, Tokyo, Japan, Apr. 2005. IEEE.

[35] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.