Reducing Storage Management Costs via Informed User-Based Policies

Erez Zadok, Jeffrey Osborn, Ariye Shater, Charles Wright, and Kiran-Kumar Muniswamy-Reddy
Stony Brook University
Jason Nieh
Columbia University

Appears in the 12th NASA Goddard, 21st IEEE Conference on Mass Storage Systems and Technologies (MSST 2004)

Abstract

Storage consumption continues to grow rapidly, especially with the popularity of multimedia files. Storage hardware costs represent a small fraction of overall management costs, which include frequent maintenance and backups. Our key approach to reducing total storage management costs is to reduce actual storage consumption. We achieve this in two ways. First, we classify files into categories of importance. Based on these categories, files can be backed up with various frequencies, or even not at all. Second, the system may also reclaim space based on a file's importance (e.g., transparently compress old files). Our system provides a rich set of policies. We allow users to tailor their disk usage policies, offloading some of the management burdens from the system and its administrators. We have implemented the system and evaluated it. Performance overheads under normal use are negligible. We report space savings on modern systems ranging from 25% to 76%, which result in extending storage lifetimes by 72%.

1 Introduction

Despite seemingly endless increases in the amount of storage and decreasing hardware costs, managing storage is still expensive. Furthermore, backing up more data takes more time and uses more storage bandwidth—thus adversely affecting performance. Users continue to fill increasingly larger disks. In 1991, Baker reported that the size of large files had increased by ten times since the 1985 BSD study [1, 10]. In 2000, Roselli reported that large files were getting ten times larger than Baker reported [12]. Our recent studies show that just three years later, large files are ten times larger than Roselli reported.

Storage management costs have remained a significant component of total storage costs. In 1989, Gelb reported that in the '70s, storage management costs at IBM were several times more than hardware costs, and projected that they would reach ten times the cost of the hardware [6]. Today, management costs are indeed five to ten times the cost of underlying hardware and are actually increasing as a proportion of cost because each administrator can only manage a limited amount of storage [5, 9]. We believe that reducing the rate of consumption of storage is the best solution to this problem. Our studies and independent studies [13] indicate that significant savings are possible.

To improve storage management via efficient use of storage, we designed the *Elastic Quota System* (Equota). Equota allows users maximal freedom, with minimal administrator intervention. Elastic quotas enter users into a contract with the system: users can exceed their quota while space is available, under the condition that the system does not provide as rigid assurances about the file's safety. Users or applications may designate some files as *elastic*. Non-elastic (or persistent) files maintain existing semantics. Elastic quotas creates a hierarchy of data's importance: the most important data will be backed up frequently; some data may be compressed and other data can be compressed in a lossy manner; and some files may not be backed up at all. Finally, if the system is running short on space, the elastic files may even be removed. Users and system administrators can configure flexible policies to designate which files belong to which part of the hierarchy. Elastic quotas introduce little overhead for normal operation, and demonstrate that through this new disk usage model, significant space savings are possible.

2 Motivational Study

Storage needs are increasing—often as quickly as larger storage technologies are produced. Moreover, each upgrade is costly and carries with it high fixed costs [5]. We conducted a study to quantify this growth, with an eye toward reducing the rate of growth.

We identified four classes of files, three of which can reduce the growth rate and also the amount of data to backed up. First, there are files that cannot be considered for reducing growth. These files are important to users and should be backed up frequently, say daily. Second, studies indicate that 82–85% of storage is consumed by files that have not been accessed in more than a month [2]. Our studies confirm this trend: 89.1% of files or 90.4% of storage has not been accessed in the past month. These files can be compressed to recover space. They need not be backed up with the same frequency as the first class of files, because files that have not changed recently are less likely to change in the near future. Third, multimedia files such as JPEG or MP3 can be re-encoded with lower quality. This method carries some risk because not all of the original data is preserved, but the data is still available and useful. These files can be backed up less frequently than other files. Fourth, previous studies show that over 20% of all files—representing over half of the storage—are regenerable [13]. These files need not be backed up. Moreover, if the site policy chooses, then these files can even be removed when space runs short.

To determine what savings are possible given the current usage of disk space, we conducted a study of four sites, for which we had complete access. These sites include a total of 3898 users, over 9 million files, and 735.8GB of data dating back 15 years: (A) a small software development company with 100 programmers, management, sales, marketing, and administrative users with data from 1992–2003; (B) an academic department with 3581 users who are mostly students, using data from shared file servers, collected over 15 years; (C) a research group with 177 users and data from 2000–2003; and (D) a group of 40 cooperative users with personal Web sites and data from 2000–2003.

Each of these sites has experienced real costs associated with storage: A underwent several major storage upgrades in that period; B continuously upgrades several file servers every six months; the statistics for C were obtained from a file server that was recently upgraded; and D has recently installed quotas to rein in disk usage.

We considered a transparent compression policy on all uncompressed files that have not been accessed in 90 days. We do not include already compressed data (e.g., `.gz`), compressed media (e.g., MP3 or JPEG), or files that are only one block. In this situation, we save between 4.6% from group B to 51 % from group C. We yield large savings on group C: it has many `.c` files that compress to 27% of their original size. Group B contains a large number of active users, so the percentage of files that were used in the past 90 days is less than that in the other sites. The next hatched bar in Figure 1 is the savings from lossy compression of still images, videos, and sound files. The results varied from a savings of 2.5% for group A to a savings of 35% for group D. Groups B and D are more liberal sites, and therefore contain a large number of personal `.mp3` and `.avi` files. As me-

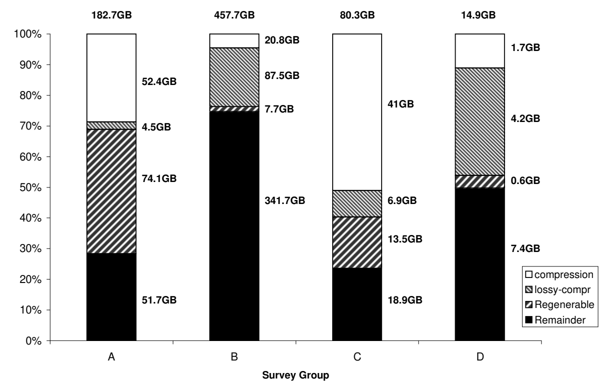

Figure 1: Space consumed by different classes. Actual amounts appear to the right of the bars, with the total size on top.

dia files grow in popularity and size, so will the savings from a lossy compression policy. The next bar down represents space consumed by regenerable files, such as `.o` files (with corresponding `.c`'s) and `~` files, respectively. This varied between 1.7% for group B to 40.5% for group A. This represents the amount of data that need not be backed up, or can be removed. Group A had large temporary backup tar files that were no longer needed (ironically, they were created just prior to a file server migration). The amount of storage that cannot be reduced through these policies is the dark bar at the bottom.

Overall, using the space reclamation methods, we can save between 25% to 76.5% of the total disk space.

To verify if applying the aforementioned three space reclamation methods would reduce the rate of disk space consumption, we correlated the average savings we obtained in the above environments with the SEER [8] and Roselli [12] traces. We require filename and path information, since our space reclamation methods depend on file types, which are highly correlated with names [3]. We evaluated several other traces, but only the combination of SEER and Roselli's traces provides us with the information we required. The SEER traces have pathname information but do not have file size information. Roselli's traces do not contain any file name information, but have the file size information. We used the size information obtained by Roselli to extrapolate the SEER growth rates. The Roselli traces were taken around the same time of the SEER traces, and therefore give us a good estimate of the average file size on a system at the time.

At the rate of growth exhibited in the traces, the hard drives in the machines would need to be upgraded after 11.14 months. Adding a compression policy extends the disks' lifetime to 18.5 months. Adding a lossy compression policy extends the disks' lifetime to 18.7

months. Finally, the savings from removing regenerable files extended the disks' lifetime to 19.2 months. Although the savings from lossy files are small here, we believe this is a result of the data available in the traces. Although the SEER traces did provide filenames, only certain filenames remained unanonymized, leaving us to estimate growth based on averages we computed across all 9 million files in the four group study. Also, lossy-compression–based policies are centered around media files, which have increased in popularity only in recent years. Nevertheless, our conservative study shows that we can still reduce growth rates by 52%. Furthermore, as the number and footprint of large-sized media files increase and large files get even larger [1, 12], so will the savings from lossy compression. Based on these results, we have concluded that our policies are promising storage management cost-reduction techniques.

3 Design

Our two primary design goals were to allow for versatile and efficient elastic quota policy management. To achieve versatility we designed a flexible policy configuration language for use by administrators and users. To achieve efficiency we designed the system to run as a kernel file system with a database, which associates user IDs, file names, and inode numbers. Our present implementation marks a file as elastic using a single inode bit. A more complex hierarchy could be created using extended attributes.

Architecture Figure 2 shows the overall architecture of our system. We describe each component in the figure and then the interactions between each component. There are four components in our system: (1) **EQFS** is a stackable file system that is mounted on top of another file system such as Ext3 [17]. EQFS includes a component (Edquot) that indirectly manages the kernel's native quota accounting. EQFS also sends messages to a user space component, *Rubberd*. (2) **Berkeley DB** (BDB) databases records information about elastic files [14]. We have two types of databases. First, for each user we maintain a database that maps inode numbers of elastic files to their names. Having separate databases for each user allows us to easily locate and enumerate each user's elastic files. The second type of database records an *abuse factor* for each user denoting how "good" or "bad" a given user has been with respect to historical utilization of disk space. We describe abuse factors in detail in Section 4. (3) **Rubberd** is a user-level daemon that contains two threads. The database management thread is responsible for updating the BDB databases. The policy thread periodically executes cleaning policies. (4) **Elastic Quota Utilities** are enhanced quota utilities that maintain the BDB databases and control both persistent and elastic quotas.

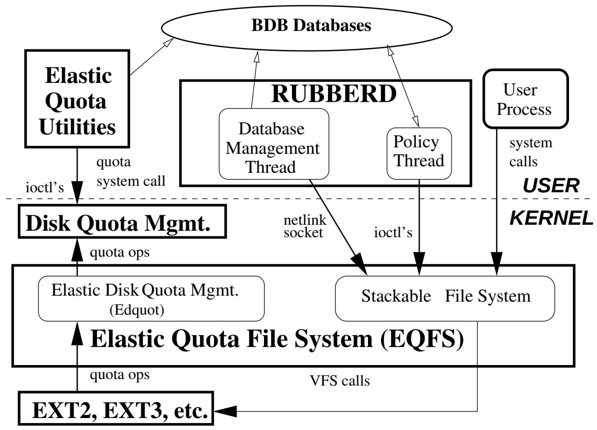

Figure 2: Elastic Quota Architecture

System Operation EQFS intercepts file system operations, performs related elastic quota operations, and then passes the operation to the lower file system (e.g., Ext2). EQFS also intercepts the quota management system call and inserts its own set of quota management operations, *edquot*. Quota operations are intercepted in reverse (e.g., from Ext2 to the VFS), because only the native disk-based file system knows when an operation has resulted in a change in the consumption of inodes or disk blocks.

Each user on our system has two UIDs: one that accounts for persistent usage and another that accounts for elastic usage. The latter, called the *shadow UID*, is simply the ones-complement of the former. The shadow UID is only used for quota accounting and does not modify permissions. When an Edquot operation is called, Edquot determines if it was for an elastic or a persistent file, and informs dquot to account for the changed resource (inode or disk block) for either the UID or shadow UID. This allows us to use the existing quota infrastructure and utilities to account for elastic usage.

EQFS informs Rubberd of events that create or change the association of an inode to file name or owner. EQFS informs Rubberd about creation, deletion, renames, hard links, and ownership changes of elastic files. EQFS communicates this information to Rubberd's database management thread over a Linux kernel-to-user socket called *netlink*. Rubberd records this information in the BDB databases. For example, when a file is made elastic, EQFS sends a "create elastic file" message to Rubberd along with the UID, inode number, and the name of the file. Rubberd then inserts a new entry in that user's database, using the inode number as the key and the file name as the value.

Rubberd periodically records historical abuse factors for each user, denoting the user's elastic space utilization over a period of time as described in Section 4.2.

Elasticity Modes EQFS can determine file's elasticity in five ways. (1) Users explicitly can toggle the file's elasticity. This allows users to control elasticity on a per file basis. (2) Users can toggle the elastic bit on a directory inode. Newly created files or sub-directories inherit the elastic bit. (3) Users can tell EQFS to create all new files elastically (or not). (4) Users can tell EQFS which newly-created files should be elastic by their extension. This mode is particularly useful because users often think of the importance of files by their type (e.g., `.c` are more important than `.o` files). (5) Finally, application developers may know best which files are temporary and can be marked elastic. This can be facilitated through two new flags we added to the `open` and `creat` system calls that tell EQFS to create the new file as elastic or persistent.

4 Elastic Quota Policies

The core of the elastic quota system is its handling of space reclamation policies. File system management involves two parties: the running system and the people (administrators and users). To the system, file system reclamation must be efficient so as not to disturb normal operations. For example, when Rubberd wakes up periodically, it must be able to quickly determine if the file system is over the high watermark. If so, Rubberd must be able to locate all elastic files quickly because those files are candidates for reclamation. To the people involved, file system reclamation policies must consider three factors: fairness, convenience, and gaming. These three factors are important especially in light of efficiency, because some policies can be executed more efficiently than others. We describe these three factors next. However, our overall design goal in this work was to provide flexibility to allow both administrators and users to use a suitable set of policies.

Fairness Fairness is hard to quantify precisely. It is often perceived by the individual users as how they personally feel that the system and the administrators treat them. Nevertheless, it is important to provide a number of policies that could be tailored to the site's own needs. For example, some users might consider a largest-file-first compression/removal policy unfair because recently-created files may not remain on the system long enough to be used. For these reasons, the policies that are more fair are based on individual users' disk space usage: users that consume more disk space over longer periods of time should be considered the *worst offenders*. Overall, it is more fair if the amount of disk space being cleaned is proportional to the level of offense of each user who is using elastic space. Once the worst offender is determined and the amount of disk space to clean from that user is calculated, however, the system must define which specific files should be reclaimed from that user. Basic policies allow for time-based or size-based policies for each user. For the utmost in flexibility, users are allowed to define their own ordered list of files to be processed first. This not just allows users to override system-wide policies, but also to define new policies based on file names and other attributes (e.g., lossy compress `*.jpg` files first).

Convenience For a system to be successful, it should be easy to use and simple to understand. Users should be able to find out how much disk space they are consuming in persistent and elastic files and which of their elastic files will be removed first. Administrators should be able to configure new policies easily. The algorithms used to define a worst offender should be simple and easy to understand. For example considering the current total elastic usage is simple and easy to understand. A more complex algorithm could count the elastic space usage over time as a weighted average. Although such algorithm is also more fair because it accounts for historical usage, it might be more difficult for users to understand.

Gaming Gaming is defined as the ability of individual users to circumvent the system and prevent their files from being processed first. Good policies should be resistant to gaming. For example, a global LRU policy that compresses older files could be circumvented simply by reading those files. Policies that are difficult to game include a per-user worst-offender policy. Regardless of the file's attributes, a user still owns the same total amount of data. Such policies work well on systems where it is expected that users will try to exploit the system.

4.1 Rubberd Configuration Files

When Rubberd has to reclaim space, it first determines how much space it should reclaim—the *goal*. The configuration file may define multiple policies, one per line. Rubberd then applies each policy in order until the goal is reached or no more policies can be applied. Each policy in this file has four parameters. (1) *type* defines what kind of policy to use and can have one of three values: `global` for a global policy, `user` for a per-user policy, and `user_profile` for a per-user policy that first considers the user's own personal policy file. This way administrators can permit users to define personalized policies. (2) *method*, defines how space should be reclaimed. Our prototype currently defines two policies: `gzip` compresses files and `rm` removes them. This allows administrators to define a system policy that first compresses files and then removes them if necessary. A policy using `mv` and `tar` could be used together as an HSM system, archiving and migrating files to slower media at cleaning time. (3) *sort*, defines the order of files being reclaimed. We define several keys: `size` (in disk blocks) for sorting by largest file first, `mtime` for sorting by oldest modification time first, and similarly for `ctime` and `atime`. (4) *filter* is an optional list of file name filters to apply the policy to. If not specified, the policy applies to all files.

If users can define their own policy files and Rubberd cannot reclaim enough space, then Rubberd continues to reclaim space as defined in the system-wide policy file.

4.2 Abuse Factors

When Rubberd reclaims disk space, it must provide a fair mechanism to distribute the amount of reclaimed space among users. To decide how much disk space to reclaim from each user, Rubberd computes an *abuse factor* (AF) for all users. Rubberd then distributes the amount of space to reclaim from each user proportionally to their AF. Deciding how to compute AF, however, can vary depending on what is perceived as fair by users and administrators for a given site. We define two types of AF calculations: current usage and historical usage.

Current usage can be calculated in three ways. First, Equota can consider the total elastic usage (in disk blocks) the user consumes. Second, it can consider the total elastic usage minus the user's available persistent space. Third, Equota can consider the total amount of space consumed by the user (elastic and persistent). These three modes give a system administrator enough flexibility to calculate the abuse fairly given any group of users (we also have modes based on a percentage of quota). Historical usage can be calculated either as a linear or an exponential average of a user's disk consumption over a period of time (using the same metrics as current usage). The linear method calculates a linear average over time to represent a user's abuse factor, while the exponential method calculates the user's abuse with an exponentially decaying average. These two historical methods provide further flexibility to the system administrator in the determination of abuse factors.

4.3 Cleaning Operation

To reclaim elastic space, Rubberd periodically wakes up and performs a `statfs` to determine if high watermark has been reached. If so, Rubberd spawns a new thread to perform the reclamation. The thread reads the global policy file and applies each policy sequentially, until the low watermark is met or all policy entries are applied.

The application of each policy proceeds in three phases: abuse calculation, candidate selection, and application. For user policies, Rubberd retrieves the abuse factor of each user and then determines the number of blocks to clean from each user proportionally to the abuse factor. For global policies this step is skipped since all files are considered without regard to the owner's abuse factor. Rubberd performs the candidate selection and application phases only once for global policies. For user policies these two phases are performed once for each user. In the candidate selection phase all candidate inode numbers are first retrieved from the BDB databases. Rubberd then gets the attributes (size and times) for each file (EQFS allows Rubberd to get these attributes more efficiently by inode number rather than pathname which is normally required for `stat`). Rubberd then sorts the candidates based on the policy (e.g., largest or oldest files first). In the application phase, we start at the first element of the candidate array and retrieve its name from the BDB database. We then reclaim disk space (e.g., compress the file). As we perform the application phase, we tally the number of blocks reclaimed based on the previously-obtained `stat` information; this avoids repeatedly calling `statfs` to check if the low watermark was reached. Once enough space is reclaimed, cleaning terminates.

5 Related Work

Elastic quotas are complementary to HSM systems. HSM systems provide disk backup as well as ways to reclaim disk space by moving less-frequently accessed files to a slower disk or tape. These systems then provide a way to access files stored on the slower media, ranging from file search software to replacing the migrated file with a link to its new location. Several HSM systems are in use today including UniTree [4], SGI DMF (Data Migration Facility), the SmartStor Infinet system, IBM Storage Management [7], Veritas NetBackup Storage Migrator [15], and parts of IBM OS/400 [11]. Most HSM systems use a combination of file size and last access times to determine the file's eligibility for migration. HP AutoRaid migrates data blocks using policies based on access frequency [16]. Wilkes et. al. implemented this at the block level, and suggested that per-file policies in the file system might allow for more powerful policies; however, they claim that it is difficult to provide an HSM at the file system level because there are too many different file system implementations deployed. We believe that using stackable file systems can mitigate this concern, as they are relatively portable [17]. In addition, HSMs typically do not take disk space usage per user over time into consideration, and users are not given enough flexibility in choosing storage control policies. We believe that integrating user- and application-specific knowledge into an HSM system would reduce overall storage management costs significantly.

6 Conclusions

The main contribution of this paper is in the exploration and evaluation of various elastic quota policies. These policies allow administrators to reduce the overall amount of storage consumed; and to control what files are backed up when, thereby reducing overall backup and storage costs. Our system includes many features that allow both site administrators and users to tailor their elastic quota policies to their needs. For example, we provide several different ways to decide when a file becomes elastic: from the directory's mode, from the file's name, from the user's login session, and even by the application itself. Through the concept of an abuse factor we have introduced historical use into quota systems. Finally, our work provides an extensible framework for new or custom policies to be added.

Our Linux prototype has a total of 14315 lines of code composed of 8309 lines of kernel code and 6006 lines of user code. We evaluated our system extensively. Performance overheads are small and acceptable for day-to-day use. We observed an overhead of 1.5% when compiling `gcc`. For a worst-case benchmark, creation and deletion of empty files, our overhead is 5.3% without database operations (a mode that is useful when recursive scans may already be performed by backup software) and as much as 89.9% with optional database operations. A full version of this paper, including a more detailed design and a performance evaluation, is available at `www.fsl.cs.sunysb.edu/docs/equota-policy/policy.pdf`.

References

[1] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a Distributed File System. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 198–212. Association for Computing Machinery SIGOPS, 1991.

[2] J. M. Bennett, M. A. Bauer, and D. Kinchlea. Characteristics of files in NFS environments. *ACM SIGSMALL/PC Notes*, 18(3-4):18–25, 1992.

[3] D. Ellard, J. Ledlie, and M. Seltzer. The Utility of File Names. Technical Report TR-05-03, Computer Science Group, Harvard University, March 2003.

[4] F. Kim. UniTree: A Closer Look At Solving The Data Storage Problem. `www.networkbuyersguide.com/search/319002.htm`, 1998.

[5] Gartner, Inc. Server Storage and RAID Worldwide. Technical report, Gartner Group/Dataquest, 1999. `www.gartner.com`.

[6] J. P. Gelb. System managed storage. *IBM Systems Journal*, 28(1):77–103, 1989.

[7] IBM Tivoli. Achieving cost savings through a true storage management architecture. `www.tivoli.com/products/documents/whitepapers/sto_man_whpt.pdf`, 2002.

[8] G. H. Kuenning. *Seer: Predictive File Hoarding for Disconnected Mobile Operation*. PhD thesis, University of California, Los Angeles, May 1997.

[9] J. Moad. The Real Cost of Storage. *eWeek*, October 2001. `www.eweek.com/article2/0,4149,1249622,00.asp`.

[10] J. Ousterhout, H. Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proceedings of the 10th ACM Symposium on Operating System Principles*, pages 15–24, Orcas Island, WA, December 1985. ACM.

[11] L. Rink. Hierarchical Storage Management for iSeries and AS/400. `www.ibm.com/servers/eserver/iseries/whpapr/hsm.html`.

[12] D. Roselli, J. R. Lorch, and T. E. Anderson. A Comparison of File System Workloads. In *Proceedings of the Annual USENIX Technical Conference*, pages 41–54, June 2000.

[13] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R.W. Carton, and J. Ofir. Deciding When to Forget in the Elephant File System. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 110–123, December 1999.

[14] M. Seltzer and O. Yigit. A new hashing package for UNIX. In *Proceedings of the Winter USENIX Technical Conference*, pages 173–84, January 1991. `www.sleepycat.com`.

[15] VERITAS. VERITAS NetBackup Storage Migrator. A White Paper, February 2002.

[16] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID Hierarchical Storage System. In *ACM Transactions on Computer Systems*, volume 14, pages 108–136, February 1996.

[17] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 55–70, June 2000.