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
{*ezk, jrosborn, ashater, cwright, kiran*}*@fsl.cs.sunysb.edu*

Jason Nieh
Columbia University
nieh@cs.columbia.edu

Abstract

Storage management costs continue to increase despite the decrease in hardware costs. We propose a system to reduce storage maintenance costs by reducing the amount of data backed up and reclaiming disk space using various methods (e.g., transparently compress old files). Our system also provides a rich set of policies. This allows administrators and users to select the appropriate methods for reclaiming space. Our performance evaluation shows that the 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, 8]. In 2000, Roselli reported that large files were getting ten times larger than Baker reported [9]. Our recent studies show that just merely by 2003, large files are ten times larger than Roselli reported.

Today, management costs are 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 [4, 7]. We believe that reducing the rate of consumption of storage is the best solution to this problem. Independent studies [10] as well as ours indicate that significant savings are possible.

To improve storage management via efficient use of storage, we designed the *Elastic Quota System* (Equota). 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 create 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 administrators can configure flexible policies to designate which files belong to which part of the hierarchy. Elastic quotas introduce little overhead for normal operations 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 [4]. We conducted a study to quantify this growth, with an eye toward reducing this rate of growth.

We identified four classes of files, three of which can reduce the growth rate and also the amount of data to be backed up. Similar classifications have been used previously [6] to reduce the amount of data to be 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 as least-

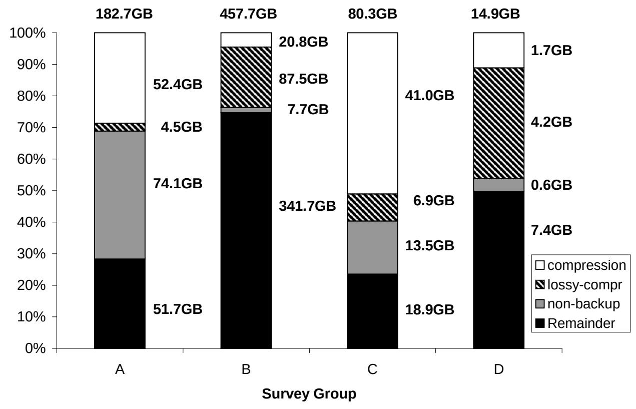

Figure 1. Space consumed by different classes. Actual amounts appear to the right of the bars, with the total size on top.

recently-used files are unlikely 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 [10]. These files need not be backed up. Moreover, these files can 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, to which we had complete access. These sites include a total of 3,898 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 3,581 users, 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.

Figure 1 summarizes our study, starting with the top bar. 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 long. 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 well. 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 bar down (top hatched) 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 contain a large number of personal `.mp3` and `.avi` files. As media 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. The amount of storage that cannot be reduced through these policies is the dark bar at the bottom. Overall, using the three space reclamation methods, we can save between 25% to 76.5% of the total disk space.

To verify if applying the aforementioned 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 [5] and Roselli [9] 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. We observed that our policies extended the disks' lifetime to 19.2 months. The disk space growth rates were reduced by 52%. Based on these results, we have concluded that our policies offer 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

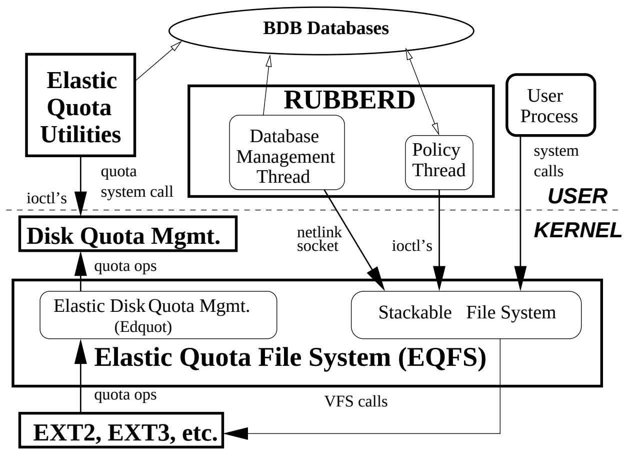

Figure 2. Elastic Quota Architecture

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. 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 [13]. 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 record information about elastic files [11]. We have two types of databases. First, for each user we maintain a database that maps inode numbers of elastic files to their names, allowing 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. (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.

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. 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 communicates information about creation, deletion, renames, hard links, and ownership changes of elastic files to Rubberd's database management thread over a *netlink* socket. Rubberd records this information in the BDB databases. Rubberd also records historical abuse factors for each user periodically, denoting the user's elastic space utilization over a period of time.

Elasticity Modes EQFS can determine a file's elasticity in five ways. (1) Users can explicitly toggle the file's elasticity, allowing them 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. (5) Developers can mark files as elastic using two new flags we added to the `open` and `creat` system calls. These flags 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. 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 as some policies can be executed more efficiently than others. We describe these three factors next.

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 or removal policy unfair because recently-created files may not remain on the system long enough to be used. For these reasons, we also provide policies that are based on individual users' disk space usage: users that consume more disk space over longer periods of time are considered the *worst offenders*. Once the worst offenders are determined and the amount of disk space to clean from the users is calculated, the system must decide 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.

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 and fair algorithm could count the elastic space usage over time as a weighted average, although 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 circumvent 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 defines 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. (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 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. HSM systems operate similarly, however, at a system-wide level [6].

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. 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 as 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 user's abuse factor as the linear average over time, whereas the exponential method calculates the user's abuse with an exponentially decaying average.

4.3. Cleaning operation

To reclaim elastic space, Rubberd periodically wakes up and performs a `statfs` to determine if the 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. 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 by name as required by `stat`). Rubberd then sorts the candidates based on the policy (e.g., largest or oldest files first). In the application phase, we reclaim disk space (e.g., compress the file) from the sorted candidates. Cleaning terminates once enough space has been reclaimed.

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, SGI DMF (Data Migration Facility), the SmartStor Infinet system, IBM Storage Management, Veritas NetBackup Storage Migrator, and parts of IBM OS/400. HP AutoRaid migrates data blocks using policies based on access frequency [12]. 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 [13]. 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. 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.

We evaluated our Linux prototype 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] Gartner, Inc. Server Storage and RAID Worldwide. Technical report, Gartner Group/Dataquest, 1999. `www.gartner.com`.

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

[6] Julie Lugar. Hierarchical storage management (HSM) solutions today. `http://www.serverworldmagazine.com/webpapers/2000/10_camino.shtml`, October 2000.

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

[8] 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.

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

[10] 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.

[11] 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`.

[12] 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.

[13] 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.