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

Technical Report FSL-03-01

Abstract

Storage consumption continues to grow rapidly, especially with the popularity of multimedia files. Worse, current disk technologies are reaching physical media limitations. Storage hardware costs represent a small fraction of overall management costs, which include backups, quota maintenance, and constant interruptions due to upgrades to incrementally larger storage. HSM systems can extend storage lifetimes by migrating infrequently-used files to less expensive storage. Although HSMs can reduce overall management costs, they also add costs due to additional hardware.

Our key approach to reducing total storage management costs is to reduce actual storage consumption. We achieve this in two ways. First, whereas files often have persistent lifetimes, we classify files into categories of importance, and allow the system to reclaim some space based on a file's importance (e.g., transparently compress old files). Second, 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 20% to 75%, which result in extending storage lifetimes by up to 72%.

1 Introduction

Despite seemingly endless increases in the amount of storage and the ever decreasing costs of hardware, managing storage is still expensive. Additionally, users continue to fill increasingly larger disks, worsened by the proliferation of large multimedia files and high-speed broadband networks. Baker reported in 1991 that the size of large files had increased by ten times since the 1985 BSD study [1, 19]; Roselli [21] reported in 2000 that large files were getting ten times larger than Baker's reported. Our recent studies (Section 3) show that just three years later, large files are ten times larger than Roselli reported. Moreover, storage requirements are continuing to grow at a rate of 50% a year [6]. Finally, existing hard disk technology is reaching physical limitations, making it harder and costlier to meet growing user demands [12].

Storage management costs have remained a significant component of total storage costs. Gelb reported in 1989 that even 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 [8]. 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 [3, 7, 14–16]. Up to 47% of storage costs are associated with administrators manually manipulating files [27]. We believe that reducing the rate of consumption of storage, and not waiting for the next generation of larger storage products, is the best solution to this problem.

Thankfully, significant savings are possible: old data can be compressed and regenerable data can be removed. Previous studies show that over 20% of all files—representing over half of the storage—are regenerable [23]. Our own study, however, shows that 15.3% of all files are regenerable, but they account for only 17.6% of storage space; this is because in recent years, non-regenerable multimedia files have begun taking large amounts of space, suggesting the need to handle multimedia files differently. Other studies indicate that 82–85% of storage is consumed by files that have not been accessed in more than a month [2, 24]. Our studies confirm this trend, and show that of the 9 million files in our study, 89.1% of them have not been accessed in the past month, taking up 90.4% of the total storage. As overall storage sizes increase, more infrequently-used files are left on expensive disks, instead of being consolidated or migrated—thereby increasing the total cost of storage.

We conclude from these studies that storage management has been a problem in the past, continues to be a problem today, and is only getting worse—all despite growing disk sizes. Morris described the idea of Autonomic Computing, which includes "the system's ability to adjust to its configuration and resource allocation to achieve predetermined goals" [16]. Golding et. al. assert, "storage systems must be self-managing" [9]. Hierarchical Storage Management (HSM) systems have multiple tiers of storage, from high-end disks to slow and inexpensive tape drives; infrequently-accessed data is moved to slower tiers of the HSM system. HSM's, however, add management costs and are not flexible enough for users. Our *Elastic Quota System* (Equota) is designed to help the management problem via efficient use of storage while allowing users maximal freedom, all 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 will be able to automatically reclaim the storage when the need arises. Users or applications may designate some files as *elastic*. When space runs short, the Elastic Quota System may reclaim space from those files marked as elastic; non-elastic files maintain existing semantics and are accounted for in users' traditional quotas. Elastic quotas creates a hierarchy of data's importance: the most important data can never be reclaimed; some data may be compressed; other data can be compressed in a lossy manner; and regenerable data may be deleted. 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.

The rest of this paper is organized as follows. Section 2 discusses background work. Section 3 describes a study we conducted which motivated our design in Section 4. Section 5 discusses the various policies we support. Section 6 presents measurements and performance results of various policies. We conclude in Section 7 and discuss future directions.

2 Background

Elastic quotas are complementary to Hierarchical Storage Management (HSM) systems. HSMs provide ways to reclaim disk space by moving less-frequently accessed files to a slower disk or tape. HSMs often provide a way to access files stored on the slower media, ranging from file search software to replacing the original, migrated file, with a link to its new location. There are many applications for HSM systems, such as online reference data (e.g., CAD/CAM drawings, medical imaging, etc.), archival storage (e.g., email archiving), and backup and remote disaster recovery (e.g., to improve restore times). Such data is becoming a growing component of total storage; by 2004 over 50% of storage reportedly will be reference information [17].

Several HSM systems are in use today including UniTree [5], SGI DMF (Data Migration Facility) [26], the Smart Storage Infinet system [29], IBM Storage Management [11], Veritas NetBackup Storage Migrator [30], and parts of IBM OS/400 [20]. 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 [31]. 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 [10, 28, 34]. 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.

In the past, HSMs consisted of fast primary storage such as magnetic disks, and then magnetic tape or optical media as a slower yet inexpensive layer. NetApp NearStore and similar products bring a new layer to this hierarchy: less expensive large disk arrays [18]. To conserve storage space, the AS/400 allows some disks to be compressed automatically [20]. Although these methods decrease the cost of the storage medium, they add additional devices that must be managed by administrators. To reduce overall storage management costs, we claim that the runaway space consumption rates must be reduced.

3 Motivation

It has long been suggested that storage needs are increasing—as quickly as larger storage technologies are produced. Moreover, each upgrade is costly and carries with it high fixed costs [7]. To determine how best to reduce management costs, we ran a comprehensive study to quantify this growth, with an eye toward reducing the rate of growth through an intelligent set of policies.

We have identified three methods of reducing growth, each with an increasing risk level. First, data can be compressed through lossless means such as a transparent compression file system [20, 32]. This method carries very little risk since no data is destroyed. Second, 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. There are no known published studies that give specific guidelines for multimedia recompression, but through practical experience within the industry and our own personal observations we have determined acceptable ratios. Third, regenerable files (e.g., reproducible `.o` files) can be removed. This method carries more risk since the file must be regenerated before it can be used again.

To determine what savings are possible given the current usage of disk space, we conducted a study of five sites, for which we had complete access. These sites include a total of 3908 users, over 9 million files, and 746GB of data dating back 15 years:

- **A**: A small software development company with 20 programmers and 80 management, sales, marketing, and administrative users with data from 1992–2003.
- **B**: A large academic department with 3581 users, the majority of which are students conducting research and working on homework assignments. Our data includes shared file servers, whose data was collected over 15 years.
- **C**: A research group with 177 users and data from 2000–2003.
- **D**: An ISP and network integrator with 10 developers and system administrators with data from 1998–2003.
- **E**: 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; D has outgrown its disk capacity, but lacks the resources to upgrade; and E has recently installed quotas to rein in disk usage.

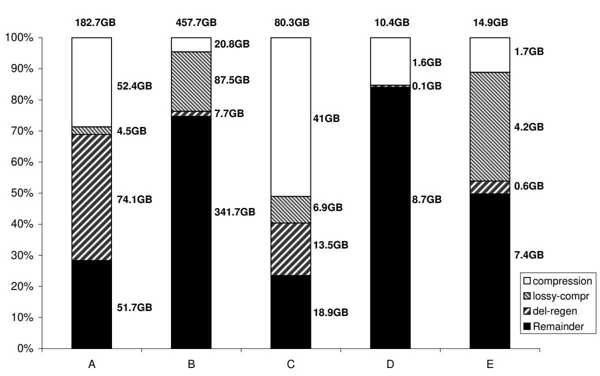

Figure 1: Possible Storage Savings. Actual amounts appear to the right of the bars, with the total size on top.

To simulate a space-reclamation policy on these five sites, we considered the three space reclamation methods discussed above. The savings from compression, lossy compression, and correlated removal can be seen in Figure 1. The top white bar is the amount saved by risk-free compression; the next hatched bar is the savings from lossy compression; and the next bar down represents savings from the removal of regenerable files. The amount of storage remaining after cleaning is the dark bar at the bottom.

First we considered a transparent compression policy. We considered all compressible files that have not been accessed or modified in 90 days as candidates for compression. In this situation, we save between 4.6% (20.8GB) from group B and 54.1% (41GB) from group C. We yield large savings on group C: it has 400,988 `.c` files that compress to 27% of their original size thereby saving 4GB storage. 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.

Next, we tested the ability to reclaim space using lossy compression. We did not consider media files in the transparent compression method because greater savings can be achieved through lossy compression. The lossy compression savings reflect the sum of our savings from compressing still images, videos, and sound files. The results varied from no savings on group D to a savings of 35% (4.2GB) for group E. The largest overall space savings came from group B, where 19% (87.5GB) was saved. Group D shows no savings from lossy compression because it is a commercial product development group where personal files are not allowed, whereas groups B and E are more liberal sites, and therefore 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.

Finally, we considered the removal of all regenerable or expendable files, such as `.o` files (with corresponding `.c`'s) and `~` files, respectively. We account for the fact that these files may have already been compressed, and our savings take into account their sizes after compression. We observed savings between 0.7% (70MB) for group D and 40.5% (74.1GB) for group A. The files on group D were predominantly executables and `.tar` files which cannot be regenerated, whereas group A had large temporary backup tar files that were no longer needed (ironically, they were created just prior to a file server migration).

Overall, using all three methods, we save between 16% (1.6GB) and 74.2% (135.6GB) of total disk space, averaging 48% savings across all five groups.

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 [13] and Roselli [21] traces. We also evaluated the usefulness of the Sprite [22] and BSD [19] traces. However, we chose to use the SEER traces as they were more recent and provided us with both the path and growth information needed for our study. We require filename and path information, since our space reclamation methods depend on file types determined by extension.

The SEER traces range from 1 to 6 months and record the system call activity of nine different users working on both connected and disconnected Linux-based laptop computers with 810MB IBM DVAA-2810 hard disks. (We obtained some of this information directly from Geoff Kuenning, the chief researcher on the SEER project, since it was not available in existing publications.) In spite of the filename and growth information available in the SEER traces, they lack file size information needed to analyze the usefulness of our space reclamation policies. To compute the disk space consumed during the trace period, we found the number of files that were created during the trace period and multiplied this with the average file size in the Roselli traces. We chose to use the Roselli traces here because they were taken in 1996, at around the same time the SEER traces were taken, and would therefore give us a better estimate of the average file size on a system at that time. Unfortunately, the Roselli traces were anonymized and contained no file name or path information, so we could not run sample policies on them, leading us to correlate the two traces together to come up with a realistic usage scenario. As the cleaning policies are based on the file extensions, we computed the space consumed by each extension in the SEER traces by multiplying the number of files of each extension by the average size of files of the corresponding extensions that we obtained in our study of the five groups.

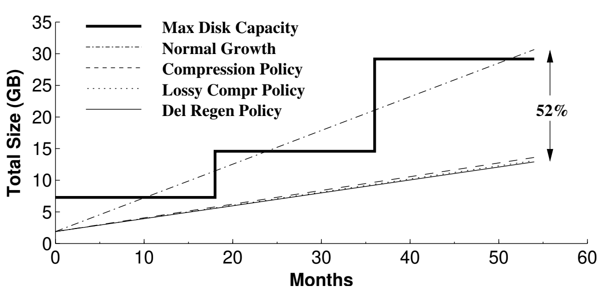

Figure 2: SEER File System Growth and Potential Savings from Equota Space Reclamation Policies

The results of our analysis can be seen in Figure 2. All of these lines begin at 1.8GB, since each of the 9 laptops in the SEER traces had Slackware Linux installed, which consumed about 200MB each in system files and swap space at that 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. These techniques stretch the disks' lifetimes beyond the 18 month doubling period of Moore's law. Although the savings from lossy and correlated 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 five group study. Also, lossy-compression–based policies are centered around media files, which have increased in popularity 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, 21], so will the savings from lossy compression. Based on these results, we have concluded that policies such as these three are very promising storage management cost-reduction techniques.

4 Design

Our two primary design goals were to allow for (1) versatile and (2) efficient elastic quota policy management techniques. An additional goal was to avoid changes to the existing OS to support elastic quotas. To achieve versatility we designed a flexible policy management configuration language for use by administrators and users; a number of user-level and kernel features exist to support this flexibility. To achieve efficiency we designed the system to run as a kernel file system with DB3 [25] databases accessible to the user-level tools. Finally, we used a stackable file system to ensure we do not have to modify existing file systems such as Ext3 [33].

Given the above goals, the design of an elastic quota file system has to answer one key question: how to identify a file as elastic vs. persistent. A related question is how to efficiently locate all elastic files on a given file system. We mark a file as elastic using a single inode bit; such spare bits are available in most modern disk-based file systems such as FFS, UFS, and Ext2/3. Changing elasticity here involves using a standard `ioctl` to turn the bit on or off. Our prototype uses the Ext3 *nodump* bit, which indicates that a file should not be backed up, so the semantics already make sense for elastic files. Most stackable file systems attempt to achieve complete independence from the underlying file system. Our implementation, however, takes advantage of specific Ext2 or Ext3 features without modifying them (only 9 lines of Ext2/Ext3 specific code are needed). Locating all elastic files requires recursive scanning of all files and checking if the inode bit is on or off. To improve performance and versatility in this design we also record elastic file information in DB3 databases: user IDs, filenames, and inode numbers. These DB3 databases improve performance, but if lost, they can be regenerated from information contained within the file system.

4.1 Architecture

Figure 3 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:

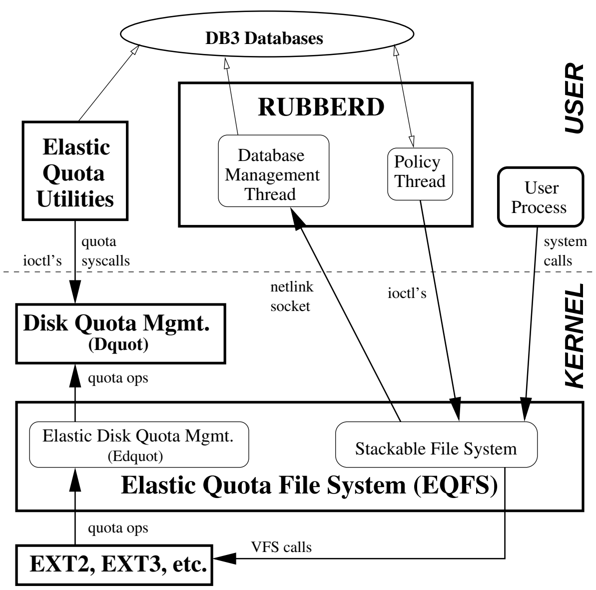

Figure 3: Elastic Quota Architecture

1. **EQFS**: The elastic quota file system is a stackable file system that is mounted on top of another disk-based file system such as Ext3. EQFS includes a component (Edquot) that indirectly manages the kernel's native quota accounting. EQFS may also be configured to send messages to a user space component, *Rubberd*.
2. **DB3 databases**: These databases record information about elastic files. We have two types of databases. First, for each user we maintain a database that maps inode numbers of elastic files to their names, which we denote in this paper as I→N. Having separate databases for each user allows us to locate all of the elastic files of a user easily, as well as enumerate all elastic files by going over each per-user database. The second type of database we maintain (denoted U→A) records an *abuse factor* for each user denoting how "good" or "bad" has a given user been with respect to historical utilization of disk space. We describe abuse factors in detail in Section 5.3.
3. **Rubberd**: This user-level daemon contains two threads. The database management thread is responsible for updating the various DB3 databases. The policy thread executes cleaning policies at given times, which often involves querying the DB3 databases.
4. **Elastic Quota Utilities**: These utilities include enhanced versions of the `quota-utils` package, used to query, set, and control user and group quotas. We enhanced these utilities to support elastic quotas. We also created new utilities that can build or query the DB3 databases; this is useful to build the DB3 databases from an existing file system or to quickly list all elastic files owned by a user.

4.2 System Operation

Before we describe our system's operation, we describe how quotas are accounted on a system without elastic quotas. In traditional operating systems, quota accounting is often integrated with the native disk-based file system. Since Linux supports a number of file systems with quotas, quota accounting is an independent VFS component called *dquot*. Usually, system calls invoke VFS operations which in turn call file-system–specific operations. However, unlike other VFS code, Dquot code does *not* call the file system. Instead, the native file system calls the Dquot operations directly. This Dquot operations vector is initialized when quotas are turned on for that file system by the system administrator or at boot time. The reason for this reverse calling sequence is that only the native disk-based file system knows when a user operation has resulted in a change in the consumption of inodes or disk blocks.

Our stackable file system EQFS intercepts file system operations, performs related elastic quota operations, and then passes the operation to the lower file system (Ext2, Ext3, etc.). EQFS also intercepts quota management operations and inserts its own set of operations in a component called *edquot*. Figure 3 shows that the calling convention for regular file system operations is from a user process issuing a system call, to the stackable file system, and then down to the lower file system. However, the calling convention for elastic quota operations is reversed: from the lower file system, through our elastic quota management (Edquot), and to the VFS's own quota management (Dquot). We devised this novel interception form—or *reverse stacking*—to avoid changing either the VFS (i.e., Dquot) or native file systems.

Each user on our system has two UIDs: one that accounts for persistent quotas and another that accounts for elastic quotas. The latter, called the *shadow UID*, is simply the ones-complement of the former. The shadow

UID does not modify existing ownership or permissions semantics; it is only used for quota accounting. Users execute system calls which the VFS translates into file system operations. We pass those operations to Ext3. When Ext3 calls EQFS's Edquot operations, Edquot determines if the operation was for an elastic or a persistent file, by inspecting the file's elastic inode bit. If the accounting operation was for an elastic file, Edquot tells Dquot to account for the changed resource (inode or disk block) in the shadow UID. In this manner it is easy to account for elastic and persistent quotas separately; both kernel and user-level utilities can easily find out how much persistent or elastic space a user is using, which eases Rubberd's policy management tasks.

We have two methods for keeping track of elastic files. The first, called *full mode*, is to track each relevant file operation. The second, called *null mode*, is to periodically (e.g., nightly) generate a list of elastic files from the file system. The advantage of full mode is that the list of elastic files will always be current; the advantage of the null mode is that overhead is minimized during normal system operation. The mode can be selected using the "netlink" option in `rubberd.conf` (see Table 1). We evaluated both modes in Section 6.

If running in full mode, whenever EQFS performs certain operations that affect an elastic file, it informs Rubberd of that event. Rubberd records the status of that elastic file in the DB3 databases. EQFS informs Rubberd about creation, deletion, renames, hard links, or ownership changes of elastic files. Additionally, whenever a persistent file is made elastic or an elastic file is made persistent, EQFS treats this as a creation or deletion event, respectively. EQFS communicates this information with Rubberd's database management thread over a Linux kernel-to-user socket called *netlink*.

When operating in full mode, Rubberd's database management thread listens for netlink messages from EQFS. When it receives a message, Rubberd decodes it and applies the proper operation on the per-user I→N database. For example, users can make a file elastic using the `chattr` (change file attributes) utility on Linux. When they turn on the elastic bit on a file, EQFS sends a "create elastic file" netlink message to Rubberd along with the UID, inode number, and the name of the file. Rubberd then performs a DB3 "put" method to insert a new entry in that user's I→N database, using the inode number as key and the file's name as the entry's value.

Rubberd's policy thread executes a given policy as defined by the system administrators. Suppose Rubberd's policy thread is executing a removal policy to reclaim disk space by deleting regenerable elastic files. Rubberd invokes `unlink` operations through EQFS, which in turn are passed to Ext3 and Dquot. When using full mode, EQFS sends a netlink message to Rubberd's database management thread—in this case a "delete elastic file" netlink message. Rubberd is multi-threaded because it has to concurrently invoke EQFS system calls and receive and process netlink messages from EQFS.

When using null mode, the DB3 databases will not be up-to-date with respect to the file system. Nevertheless, this mode is useful for a time-based policy such as cleaning oldest files first, since older files are likely to remain in the DB3 database. Since Rubberd obtains information about all files at cleaning time, even if the file was updated after the nightly generation of the database, Rubberd will still use up-to-date file attributes. If Rubberd is not able to reclaim enough space using the previously-generated databases, it will initiate a more expensive recursive scan of the file system to generate up-to-date databases. System administrators must weigh the added benefit of up-to-date accounting with the extra performance overhead introduced by EQFS's full mode.

Rubberd is configured to wakeup periodically and record historical abuse factors for each user, denoting the user's average elastic space utilization over a period of time. Rubberd gets the list of all users, their elastic and persistent disk usage, and their elastic and persistent quotas (if any). With these numbers, Rubberd computes an updated abuse factor, and stores this value in the U→A database. We describe abuse factors in more detail in Section 5.3.

4.3 Elasticity Modes

EQFS supports five methods of determining when a file becomes elastic. This allows us to leverage user and application-specific knowledge when determining how to reclaim space.

First, users can toggle the file's elasticity by using the standard Linux `chattr` tool. This allows users to control elasticity on a per file basis. Once a file is made elastic or persistent, moving it to other directories on the system does not change its elasticity.

Second, users can use `chattr` to toggle the elastic bit on a directory inode. EQFS inherits the elastic bit to any newly-created file or sub-directory, similarly to how a new file's group is inherited in a setgid directory. This elasticity mode is useful for whole elastic hierarchies, such as `/tmp` or a user's Web browser cache directory.

Third, users can tell EQFS (via an `ioctl`) whether files that are newly created should be elastic or not. This mode works similarly to how the `newgrp` command sets the default group that all newly-created files or directories should use. One use for this mode is when users unpack an important source distribution; before beginning to build the package, users can set this elasticity mode for all future files. That way, all newly created files during the build, regardless of their location, will be elastic: objects, libraries, executables, headerdependency files, etc.

Fourth, users can tell EQFS (again, via an `ioctl`) which newly-created files should be elastic by their name. Specifically, users can specify a small number of file extension strings that are matched by `eqfs_create` from a newly-created file name. This mode is particularly useful because users often think of the importance of files by their type—or extension (e.g., `.c` are more important than `.o` files because the latter can be easily regenerated from the former).

Finally, application developers may know best which files are are best marked elastic. Since many temporary files are not created by users, but rather by programs, we added a new flag to the `open` and `creat` EQFS file system methods: `O_ELASTIC`. This flag tells EQFS to create the new file as elastic. For example, Emacs can automatically create its ˜ backup files elastically.

5 Elastic Quota Policies

The core of the elastic quota system is its handling of space reclamation policies. EQFS is the file system that provides elasticity support and works in conjunction with Rubberd, the user-space daemon that implements cleaning policies, as seen in Figure 3.

We start (Section 5.1) with a general discussion of the design issues involved in policies; as we see, there are often conflicting concerns that must be carefully balanced to provide a convenient, fair, and versatile system. In Section 5.2 we discuss the design of Rubberd's policy engine from the perspectives of users and administrators. In Section 5.3 we discuss how Rubberd determines fairly how much disk space to reclaim and from which users; that culminates in Section 5.4 where we detail the actual methods and algorithms we use to reclaim disk space; and finally in Section 5.5 we describe how elastic quotas may be used in various situations.

5.1 Policy Design Considerations

File system management involves two parties: the running system and the people involved (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 administrator-defined high watermark. If so, Rubberd must be able to locate all elastic files quickly because those files are candidates for removal. Moreover, depending on the policy, Rubberd will also need to find out certain attributes of elastic files: owner, size, last modification time, etc.

To the people involved, file system reclamation policies must consider three factors: convenience, fairness, and gaming. These three factors are important especially in light of efficiency, because some policies could be executed more efficiently than others. We describe these three factors next. However, the overall design goals of this work were to provide as much flexibility to both administrators and users to decide on the suitable set of policies that meet their site's needs.

Convenience The system 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 reclaimed 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 an algorithm is also more fair, because it accounts for historical usage, it might be more difficult to understand by users.

The reclamation method also heavily influences convenience. Having a file removed is less convenient than a transparent compression policy. Equally important, the user should be aware of what non-transparent actions have been performed on their files by the system.

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. It is important to provide a number of policies that can be tailored to a site's own needs. For example, some users might consider a largest-file-first removal policy unfair because recently-created files may be reclaimed after a short period of time. Other users might feel that an oldest-creation-time policy is unfair because it does not account for recency or frequency of use.

For these reasons, the policies that are more fair are based on individual users' disk space usage. In particular, 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 first 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 reclaimed first. This not just allows users to override system-wide policies, but also to define new policies based on filenames and other attributes (e.g., remove `*.o` and `*˜` files first).

Gaming Gaming is defined as the ability of individual users to circumvent the system and prevent their files from being reclaimed first. Good policies should be resistant to gaming. For example, a global LRU policy that removes older files could be circumvented by files' owners simply by reading or `touch`ing those files. Some policies are more difficult to game, for example a policy that removes the largest files first. Users could split their large files into smaller chunks, but then have to assemble the parts back before the large file could be used again. Policies that are difficult to game include a per-user worst-offender policy. Regardless of the file's times or sizes, a user still owns the same total amount of data. Such policies work well on multi-user systems where it is expected that users will try to game the system.

There are certain situations where gaming may not be an important factor in choosing policies. Certain global policies (e.g., by time or size) may still be useful in situations such as with a small group of cooperative users who do no have an incentive to circumvent the system; such gaming could hurt their colleagues' ability to work. Another useful scenario where gaming is not an issue is a single-user workstation: to such a user, elastic quotas can be a useful method of ensuring that temporary files get automatically cleaned periodically.

5.2 Rubberd Configuration Files

Administrators Administrators typically control two configuration files in `/etc`: (1) an elastic quotas configuration file (`policy.conf`) and (2) a Rubberd configuration file that defines startup options (`rubberd.conf`).

The policy configuration file (`policy.conf`) uses a simple syntax as follows. The configuration file may define multiple policies, one per line. When Rubberd has to reclaim space, it first determines how much space it should reclaim—the *goal*. Rubberd then executes each policy in order until the goal is reached or no more policies can be executed. Each line in this file has the following space-delimited format:

$$type\ method\ sort\ [filter\ ...] \tag{1}$$

The first parameter, *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. In this way administrators can permit users to define policies on their files. The second parameter, *method*, defines how space should be reclaimed. Our prototype defines four methods currently: `gzip` for a policy that transparently compresses files (on access the file is automatically decompressed with no user intervention), `lossy` for a policy that re-encodes multimedia files using lower bitrates, `rm` for a policy that deletes files, and `custom` which allows a customized command to be run. In this way, administrators can define a system policy that first compresses files and then removes them: such a policy has the benefit that enough space may be reclaimed by compressing files and users can still get access to their elastic files through transparent decompression. A custom policy using `mv` and `tar` could be used together as an HSM system, archiving and migrating files to slower media at cleaning time. The third parameter, *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`. The remaining entries on the policy line are optional and define filename filters to apply the policy to. If not specified, the policy applies to all files.

Consider the following `policy.conf` file:

```
global        rm    size   ~ .bak core .tmp
user          gzip  mtime
user_profile  rm    atime  .o
user_profile  lossy atime  .jpg .mpg .mp3 .avi
```

The first line starts a simple global policy that will delete obviously unnecessary elastic files such as backup files used by editors. When Rubberd tries to reclaim space, it will try to bring the system down to the goal level by this first policy line. If that is insufficient, Rubberd will proceed and apply the second policy line, which defines a per-user policy that will compress all elastic files. Next, Rubberd will perform a user policy that removes compiler object files that have not been read in a while, but allows users to override the system defaults. Finally, if still not enough space has been reclaimed, Rubberd will apply a lossy compression policy on multimedia files, again allowing users to override the default selection based on atime.

The Rubberd configuration file (`rubberd.conf`) is simple and defines the parameters described in Table 1.

Parameter	Meaning
hi_watermark N	% disk usage to begin cleaning
lo_watermark N	% disk usage to stop cleaning
statfs_interval S	disk usage check interval (sec)
mount_point M	name of EQFS mount to monitor
netlink on|off	process netlink messages?
abuse_cur M	mode to compute current usage
abuse_avg I N	linear historical abuse factors
abuse_exp I D	exponential historical abuse factors

Table 1: Rubberd configuration file parameters. We describe abuse factors in Section 5.3.

Users If the system administrator has allowed users to determine their own reclamation policies, users can then use whatever policy they desire for determining the order in which their files are reclaimed first. The user policy file can only instruct Rubberd to *prefer* those files for reclamation first; if not enough space can be reclaimed, Rubberd will continue to reclaim space as defined in the system-wide policy file, `policy.conf`.

Entry	Meaning
class/foo.tgz	a relative pathname to a file
~/misc	a non-recursive directory
~/tmp//	a recursive directory
src/eqfs/*.o	all object files in a specific directory
src//*.o	all object files recursively under `src`
~//*.mp3	all MP3 files anywhere in home dir.

Table 2: Example user policy file entries

A user-defined policy is simply a newline-delimited list of file and directory names or simple patterns thereof, designed to be both flexible and easy to use. Each line can list a relative or absolute name of a file or directory. A double-slash (`//`) syntax at the end of a directory name signifies that the directory should be scanned recursively. In addition, simple file extension patterns can be specified. Table 2 shows a few examples and explains them.

5.3 Abuse Factors

To reclaim some disk space, Rubberd must fairly distribute the amount of reclaimed space among all users that consume any elastic space. To decide how much disk space to reclaim from each user, Rubberd computes an *abuse factor* (AF) for all users. Then Rubberd distributes the amount of space to reclaim from each user proportionally to their AF. For example, suppose Rubberd needs to clean 6MB of disk space from two users; user A's AF is 10 and user B's AF is 20; then Rubberd will clean 2MB from user A and 4MB from user B.

Deciding how to compute an AF, however, can vary depending on what is perceived as fair by users and administrators for a given site. Therefore, we provide a variety of methods for administrators to tailor the computation of AFs to the site's needs. First, we define two types of AF calculations: one that considers the current usage and a second that considers historical usage. Current usage is better at tracking users' existing elastic usage; historical usage takes into account users' behavior patterns over longer periods of time.

As an example, consider two users: user A has never used elastic space and just in the past day began consuming 100MB; user B has used exactly 50MB of elastic space each day for the past five days. Based on the current usage policy alone, user A's AF will be double that of user B. During cleaning, twice as much disk space will be reclaimed from user A than from user B. This policy can be considered fair to the system—and all users on the system—because it will clean space based on how much is currently being used. However, such a policy may unfairly punish user A who, on average, has not used as much as user B: user A's usage over the five days, averaged per day, is just 20MB. Therefore, a historical usage policy may be considered more fair because it takes into account long-term behavior. The converse could also be true: a past disk space abuser could have a high average usage, but currently is not using much disk space; a history-based AF could result in many of this user's elastic files being compressed or deleted. Interestingly, historical abuse factors may promote more responsible disk usage over time, and reward those with lower average usage by allowing them to consume more disk space during a shorter period of time.

Table 1 shows the three Rubberd configuration parameters used to compute abuse factors. Rubberd always computes the current usage per user (U_c) at configurable intervals. If the administrator configured the use of historical factors, then Rubberd also computes a running composite AF and stores it in a DB3 file.

Current Usage The Rubberd configuration file (`rubberd.conf`) parameter `abuse_cur` takes a single parameter that defines the mode in which total current usage (U_c) is computed currently:

- U_e In this mode we only consider the total elastic usage (U_e, in disk blocks) that the user consumes. This mode considers elastic usage separately from persistent quotas or persistent usage; it is most useful in environments with small persistent quotas.
- $U_e - A_p$ Users who use elastic files and also have a persistent quota may not have consumed all of their persistent quota. Such users could argue that U_e alone is not a fair assessment of their usage because they have persistent quota available (A_p) and they could simply convert some of their elastic files to persistent ones. Therefore, this method computes a user's current usage as the amount of elastic space consumed minus the available persistent quota the user has (truncated to zero).
- $U_e + U_p$ Similarly to the previous mode, this mode considers the current usage as the total amount of disk space a user consumes—the sum of both elastic and persistent usage. This mode could be useful in environments where certain users could have very different persistent quotas. In such an environment, users with large persistent quotas could be viewed as "hogging" disk space as compared to users with smaller persistent quotas.

Our system supports several more modes to compute current usage, based on percentages of usage vs. some total; we omit discussion of those for brevity.

Historical Usage The Rubberd configuration parameter `abuse_avg` computes a linear average of usage over a period of time. This option takes two parameters: I defines the interval in seconds between samplings of current usage; N defines the number of samples to include in the running average. This mode gives equal importance to each sample interval, but quickly "forgets" usage prior to the oldest sample. The smaller I is, the more closely this mode tracks elastic usage.

The configuration parameter `abuse_exp` computes an exponentially decaying average. This option takes two parameters: I is the sampling interval; D is the decay factor. For example, with $D = 2$, the computation half-life decays every I seconds. The benefit of this mode is that it never forgets entirely a user's past usage, but considers more recent usage progressively more important than older usage.

5.4 Cleaning Operation

To reclaim elastic space, Rubberd periodically checks (via `statfs`) to see if the high watermark was reached. If so, Rubberd spawns a new thread to perform the actual cleaning. The thread reads the global policy file and applies each policy sequentially until the low watermark is met or all policy entries are enforced.

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 according to the abuse factor. For global policies we skip this step 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 we first retrieve from the DB3 databases all possible candidate inode numbers. Then Rubberd gets the status information (size and times) for each file using `bistat`, a custom bulk-inode stat ioctl we wrote which bypasses name lookups and retrieves a number of `stat` structures at once. Rubberd then sorts the candidates based on the policy (say, by size or age). For global policies we iterate through each user database and store all candidates in an array. For user policies we simply fetch all entries from the appropriate database. When a file pattern is specified in `policy.conf`, we retrieve each file name from the database and compare it against the pattern. We discard that name since most files will not have any cleaning operations performed on them. The last phase of the candidate selection is to sort the entire set of candidates as defined in `policy.conf`.

In the application phase, we start at the first element of the candidate array and retrieve its name (or names if a hard link exists) from the DB3 database. Then we reclaim disk space using the administrator supplied method. For example we, compress the file if the "gzip" policy was configured. As we perform the application phase, we tally the number of blocks reclaimed based on the previously-obtained `stat` information; this avoids having to call `statfs` after each file removal to check if the low watermark was reached.

Each time Rubberd completes an application phase, it runs `statfs` and computes the number of blocks that still need to be cleaned. If this number is not positive then cleaning terminates. This gives smaller abusers a slight advantage. Since we can only reclaim space on a per-file basis, this means that the goal for each user is really a minimum goal. For example, suppose Rubberd computes that it needs to reclaim 2MB from a given user, and then compresses the oldest file which happens to save 3MB in size: Rubberd winds up reclaiming more space than the minimum computed for that user. This excess space reclaimed from the largest abusers ends up benefiting the smallest abusers, because Rubberd will reclaim less space from these users.

5.5 Usage Scenarios

The Equota system is flexible and can be configured to work well in many situations. Here we describe two possible scenarios in which Equota might be used.

Large Group File Server The first scenario is that of a large university-wide server. Users on such a large server usually are anonymous to each other, and will try to get as much out of the system as possible. Gaming would be a major concern, as there would be little to no cooperation between users. In such a situation, both persistent and elastic quotas would have to be set. Although the purpose of elastic systems is to allow an almost infinite amount of space to users, it would be necessary on such a large system to set elastic quotas. Users would not be allowed to use over a certain amount of elastic space, thus avoiding denial-of-service attacks and other gaming of the system. We expect Rubberd to monitor disk usage more closely at intervals as short as an hour, and reclaim a large percentage of disk space when the system goes over the high watermark. In such a hostile environment, Rubberd will use a long historical abuse factor, so as to account for longer-trends of disk abuse.

Small Developer Community Server The second scenario is that of a cooperative group of software developers. In such a group, both elastic and persistent quotas may be unlimited: all of the disk space will be available to elastic or persistent files. Equota's automatic cleaning mechanisms may be attractive to such a group that would rather spend time programming than managing files. Since the group is cooperative, they are working toward a common goal, and the chance for gaming is small. Such a group would use Equota to mark certain patterns of files for deletion, such as all regenerable files (compiler-generated ones). These advanced users might modify some of their tools to use the O_ELASTIC flag to designate certain application-generated files elastic by default. Such a user community will also make extensive use of per-user policy files, for example to mark personal MP3 files elastic.

6 Performance Evaluation

To evaluate elastic quotas in a real world operating system environment, we implemented a prototype of our elastic quota system on Linux 2.4.18. We present some experimental results using our prototype EQFS and Rubberd implementations. We compared EQFS to Linux's Ext3 journaling file system. We then measured the impact of Rubberd on a running system.

All of our experiments were conducted on a 1.7Ghz Intel Pentium 4 machine, with 128MB of RAM, running Red Hat Linux 7.3, using a vanilla Linux kernel version 2.4.18. We believe this machine represents a small group file server that could benefit from running the Equota system. For the experimental file system, we used a 30GB 7200 RPM Western Digital Caviar IDE disk. All other libraries, executables, user utilities, headers, and system data resided on the root file system located on a 20GB 7200 RPM Western Digital Caviar IDE disk. All tests were performed with a cold cache, achieved by unmounting and remounting the file systems between test iterations. We repeated all experiments several times to ensure stability and observed low standard deviations for most of our tests. We report any significant standard deviations that arose in our tests.

6.1 Steady State System Benchmarks

To measure the performance of EQFS, we stacked it above Ext3 and compared its performance to Ext3. We tested the performance of EQFS in four different configurations: basic Ext3 (EXT3), EQFS stacking alone with netlink messages turned off (NULL), EQFS with netlink messages turned on and Rubberd processing messages but not writing them to DB3s (NET), and EQFS with netlink messages and Rubberd updating databases (FULL). This set of configurations isolates the overhead of each individual system component. For non-elastic files the performance overhead is the same as that of NULL regardless of the configuration, because the only overhead incurred is that of stacking. We used two workloads for our experiments: (1) unpacking, configuration, build and deletion of the Gcc 3.1 source tree, and (2) an *inoder* program we wrote to create a large file set and then remove it.

Gcc compile The first workload we used was to configure and build the Gcc source. Gcc contains about 15,000 files and provides us with a fair mix of reads, writes, and lookups. We unpacked the distribution, ran a `configure` and `make`, and then deleted the build tree.

inoder-rm For the second workload, we wrote a program called `inoder`. It creates 1000 4KB files within 100 directories, for a total of 100,000 files. By making a uniform dataset, we can measure the performance more precisely. Since EQFS manipulates meta-data operations (e.g., creation, deletion, etc.), this benchmark demonstrates the worst-case overhead of our system.

6.2 Cleaning Benchmarks

To evaluate Rubberd, we measured its file system cleaning performance. To provide realistic results on common file server data sets, we used a working set of files collected over a period of 18 months from our own production file server. Figure 4 shows the frequency and size distribution of our data set. The working set includes the actual files of 121 users, many of whom are software developers or students. The file set includes 1,194,133 inodes and totals over 26GB in size; more than 99% of the inodes are regular files. 24% of the users use less than 1MB of storage; 27% of users use between 1–100MB; 38% of users use between 100MB–1GB of storage; and 11% of users consume more than 1GB of storage each. Average file size in this set is 21.8KB, matching results reported elsewhere [21]. The most popular file size was 4KB, while the total size distribution was bi-modal, peaking at 32KB and 1GB. Even though there are only a handful of files in the 1GB range, they represent a large portion of the total space consumed by the file set. We treated this entire working set as being elastic, a worst case scenario for our system. Using EQFS mounted in full mode on Ext3, we ran experiments with the working set for measuring Rubberd's performance by cleaning elastic files using the DB3 databases.

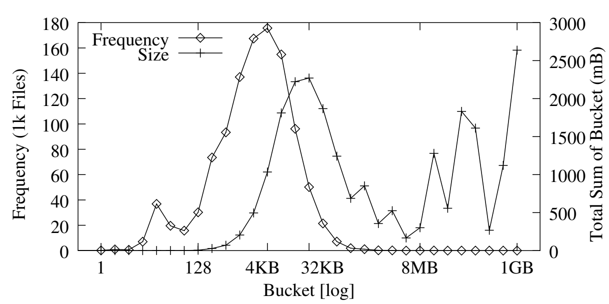

Figure 4: File Set Distribution

Cleaning Policy The Rubberd benchmark we used measured the time it took to clean a portion of the disk on an otherwise idle system using several cleaning policies. We chose to use the cleaning methods of `gzip` and `rm`. We do not report results for lossy compression because they were similar to the lossless benchmark results. We ran several types of cleaning tests to appropriately measure Equota's performance. We ran incremental and full cleaning tests for global `gzip` policies sorted by time and size. We ran the same tests using a user `rm` policy. Incremental cleaning is where the cleaning thread cleaned the system from 100% to 90%, from 90% to 80%, 80% to 70%, etc. until the file system was at 0% capacity. A full cleaning test is where the file system was cleaned from 100% to 90%, then 100% to 80%, 100% to 70%, etc. until 100% to 0%. While we do not expect Rubberd to clean more than 5% to 10% of the disk in practice, we chose such large values to avoid under-representing the cost of Rubberd's operation.

Between cleaning tests we needed to start from our original working set before each test. We recreated our file system from an identical image disk using `dd`. The source disk had our dataset with pre-built quota files and DB3s. This ensured that our file system was laid out in exactly the same way for each test.

6.3 Steady State System Results

Gcc Compile For the NULL mode benchmark, we recorded a 0.7% increase in elapsed time; 0.6% increase in user time; and a 5.5% increase in system time. For FULL mode, user time increased 0.4% from that of EXT3, elapsed time rose 1.5% and system time rose by 5.9% for FULL mode. Rubberd consumed 1.3 CPU seconds in NULL mode and 5.1 CPU seconds in FULL mode (out of 1950 seconds total elapsed time for the benchmark). This demonstrates that under normal conditions EQFS does not have a noticeable performance overhead.

Inoder-rm results The inoder test demonstrates the overhead for meta-data operations of various equota components. The results for inoder can be seen in Figure 5. The standard deviations for this test were between 1.7% and 13.6%. We have determined that the bursty journaling behavior of Ext3 caused some tests to manifest a higher standard deviation [4].

For our NULL, NET, and FULL configurations, file creation shows elapsed time overheads over EXT3 of 5.3%, 13.9%, and 89.9%, respectively; overheads for user time of 2.8%, 33.0%, and 96.9%; and overheads for system time of 14.2%, 24.2%, and 35.7%. For our NULL, NET, and FULL configurations, file deletion shows elapsed overheads over EXT3 of 40.0%, 53.7%, and 206.4%, respectively; overheads for user time of 20.0%, 82.8%, and 77.8%; and overheads for system time of 51.4%, 71.3%, and 81.7%. The results show that the largest

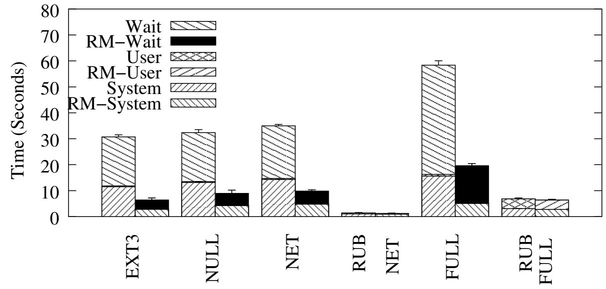

Figure 5: Creation and Deletion of 100,000 4KB Files

overhead in FULL mode is DB3 access. For NET, Rubberd system and user times were 1.93 and 0.71 seconds, respectively. For FULL, Rubberd system and user times grew to 5.84 and 7.37 seconds, respectively. Though the overhead for some operations is high in this intense test, we believe that the compile benchmarks more accurately represent actual user activity. The `inoder` results indicate that reducing DB3 operations is beneficial. To this end, administrators may elect to use the EQFS NULL mode (see Section 4.2).

6.4 Cleaning results

Compression policies At the beginning of our test, our file system was filled to 94% of its capacity. Our disk was cleaned from 94% to 41% (Figure 6), an effective compression ratio of 1.78:1. We could not clean to below 41% because not all objects can be compressed (e.g, already compressed files, directories, and the non-elastic quota file and rubberd DB3 databases). As expected, the CPU time of this compression policy dominates the test (89% of elapsed time). We also ran a full cleaning policy, and as expected the time for a given watermark was roughly the sum of the previous incremental watermarks.

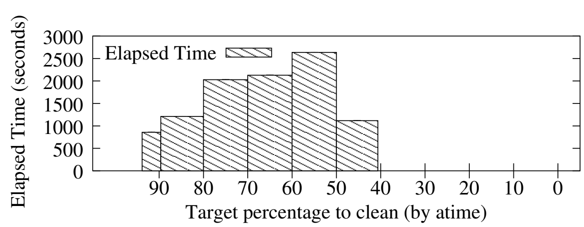

Figure 6: Elapsed time for an incremental compression policy sorted by atime using 10% goal increments. The width of the bar indicates the actual amount of space cleaned.

We also ran compression policies sorted by mtime, ctime, and size. The sorting attribute had little to do with the amount of time that a policy run took. This is because the time a compression policy takes is primarily a function of the amount of data reclaimed, which remains constant (only the order of files changes). To conserve space, we do not report these results.

Removal sorted by time The incremental removal test sorted by atime showed an almost constant elapsed, system, and user cleaning times (Figure 7). The system and user times are small (2% of elapsed) and follow the same trend as elapsed time (this was the case for all other tests as well).

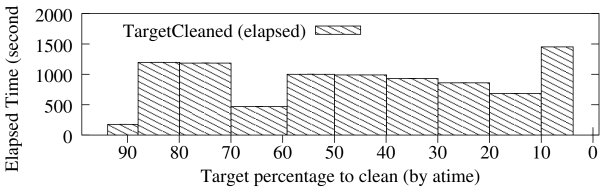

Figure 7: Elapsed time for incremental removal sorted by atime using 10% goal increments. The width of the bar indicates the actual amount of space cleaned.

Since files are chosen by atime, and size is not taken into consideration when choosing the file to be cleaned, the results take on a constant complexity. Our results show that cleaning took place between 94% and 5%. The disk can not be cleaned to zero because not all directories will be removed, and the quota file and DB3 databases, which take up less than 1% of the total disk space, are not elastic. Variations in these results can be seen in the low cleaning times for low watermarks 90% and 60%. The fileset we used had two large files that fell within the 94–90% and 70–60% intervals when sorted by atime. The higher numbers for the 10–5% interval were the result of many small files in that range of atimes. This shows us that removal is a function of meta-data operations. In a case where you want constant cleaning penalties, such as a system with a small low watermark, it is best to use an atime sort algorithm. Full removal policies are similar to full compression policies, in that a given watermark takes roughly the sum of previous incremental watermarks. We do not report results for sorting by mtime and ctime, because they are comparable to atime.

Removal sorted by size Incremental cleaning policies sorted by size showed different results than those sorted by atime. When cleaning by size there is a linear relationship between the low watermark and the time it takes to clean. The elapsed, system, and user times increase linearly with each iteration of this incremental policy (Figure 8). These results were expected because the removal of a file is a meta-data operation and is mostly independent of the size of the file being removed.

Full cleaning for a given watermark is again roughly the sum of all previous incremental watermarks. However, this yields the interesting property that as watermarks decrease the progression of time is $O(N^2)$ for full cleaning by size.

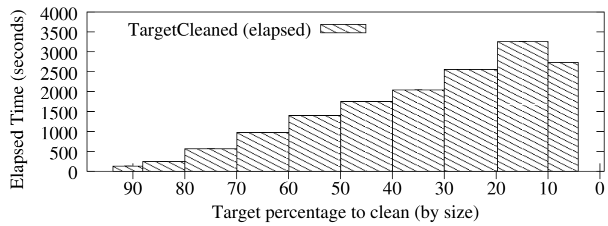

Figure 8: Elapsed time for incremental clean sorted by size using 10% goal increments. The width of the bar indicates the actual amount of space cleaned.

For both the full and incremental policies (by size), Rubberd cleaned to its first few goals with the deletion of the first few large files. As more files are deleted, smaller files are selected for removal, and thus it takes more of them to meet a goal. Since the goals are size based, as time goes on, it takes longer to meet lower watermarks for the file system. In normal operation, it will not be necessary to clean many small files when using a size-based policy since the larger files will hopefully bring the system quickly below the low watermark.

7 Conclusions and Future Work

The main contribution of this work is that we developed a system that reduces storage management costs, by extending the lifetime of disks up to 72%, through intelligent space reclamation policies. Three additional contributions include the following. First, we utilized user and application-specific knowledge to increase the usefulness of storage management policies. 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. Second, we use transparent compression as another layer in the HSM system, without the need for administrators to manage another device. Third, through the concept of an abuse factor we have introduced historical use into quota systems.

We conducted a comprehensive study which demonstrates that storage consumption and associated management costs continue to grow. Our study also shows that significant space savings are possible, which we believe will directly translate into management cost savings.

Our Linux prototype includes many features that allow both site administrators and users to customize elastic quota policies. Our policy engine is flexible, allowing a variety of methods for elastic space reclamation. Our evaluation shows that the performance overheads are small and acceptable for day-to-day use. Additionally, our work provides an extensible framework for new or custom policies to be added. Through the use of stacking, we can extend these benefits to any file system.

We plan to expand the definition of persistent and elastic files to include file lifetimes and priorities. A file lifetime would include a minimum lifetime and a maximum lifetime. A persistent file has an infinite minimum lifetime and an elastic file has a minimum lifetime of zero. The minimum lifetime would be useful for data that may not be relevant for longer than some predefined time period. A maximum lifetime would enable the automatic migration or deletion of files; this could be valuable because it would ensure a company's records retention policy is enforced or personal information is not available after a certain point. File priorities could determine when and how files should be backed up or migrated. We believe that file lifetimes and priorities should be first-class attributes that all file systems support, and plan to modify the OS and VFS accordingly.

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] A. Brown and D. Patterson. Towards Availability Benchmarks: A Case Study of Software RAID Systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 263–276, June 2000.

[4] R. Bryant, R. Forester, and J. Hawkes. Filesystem Performance and Scalability in Linux 2.4.17. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 259–274, June 2002.

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

[6] Forrester. Slaying The Storage Beast. `www.forrester.com`, March 2001.

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

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

[9] R. Golding, E. Shriver, T. Sullivan, and J. Wilkes. Attribute-managed storage. In *Workshop on Modeling and Specification of I/O*, 1995.

[10] J. S. Heidemann and G. J. Popek. File system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994.

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

[12] M. Kryder. Future Magnetic Recording Technologies. USENIX FAST '02 Keynote, January 2002. `www.usenix.org/events/fast02/kryder.pdf`.

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

[14] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-7)*, pages 84–92, Cambridge, MA, 1996.

[15] J. Moad. The Real Cost of Storage. *eWeek*, October 2001. `www.eweek.com/article2/0,3959,48653,00.asp`.

[16] R. Morris. Storage: From Atoms to People. USENIX FAST 2002 Keynote, January 2002. `www.usenix.org/events/fast02/morris.pdf`.

[17] Network Appliance. A New Approach for Storing Reference Information — NearLine Storage. Technical Report TR3193, September 2002.

[18] Network Appliance. NearStore. `www.netapp.com/products/nearstore`, 2003.

[19] J. Ousterhout, H. Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson. A trace-driven analysis of the usenix 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.

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

[21] 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.

[22] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 1–15, Asilomar Conference Center, Pacific Grove, CA, October 1991. Association for Computing Machinery SIGOPS.

[23] 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.

[24] M. Satyanarayanan. A Study of File sizes and Functional lifetimes. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, pages 15–24, 1981.

[25] 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`.

[26] SGI. Storage software. `www.sgi.com/products/storage/software.html`, 2003.

[27] D. Simpson. Corral your storage management costs. *Datamation*, 43(4):88–98, 1997.

[28] G. C. Skinner and T. K. Wong. "Stacking" Vnodes: A progress report. In *Proceedings of the Summer USENIX Technical Conference*, pages 161–74, June 1993.

[29] Smart Storage. SmartStor InfiNet: Virtual Storage for Today's E-Economy. A White Paper, September 2000.

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

[31] 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.

[32] E. Zadok, J. M. Anderson, I. Bădulescu, and J. Nieh. Fast Indexing: Support for size-changing algorithms in stackable file systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 289–304, June 2001.

[33] E. Zadok, I. Bădulescu, and A. Shender. Extending file systems using stackable templates. In *Proceedings of the Annual USENIX Technical Conference*, pages 57–70, June 1999.

[34] 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.