The Design and Implementation of Elastic Quotas: A System for Flexible File System Management

Ozgur Can Leonard
Sun Microsystems

Jason Nieh
Columbia University

Erez Zadok, Jeffrey Osborn, Ariye Shater, and Charles Wright
Stony Brook University

Columbia University Technical Report CUCS-014-02, June 2002

Abstract

We introduce elastic quotas, a disk space management technique that makes disk space an elastic resource like CPU and memory. Elastic quotas allow all users to use unlimited amounts of available disk space while still providing system administrators the ability to control how the disk space is allocated among users. Elastic quotas maintain existing persistent file semantics while supporting user-controlled policies for removing files when the file system becomes too full. We have implemented an elastic quota system in Solaris and measured its performance. The system is simple to implement, requires no kernel modifications, and is compatible with existing disk space management methods. Our results show that elastic quotas are an effective, low-overhead solution for flexible file system management.

1 Introduction

The increasing ubiquity of Internet access and the increasing number of online information sources are fueling a growing demand for storage capacity. Vast amounts of multimedia data are ever more widely available to users who continue to download more of it. As the storage demands of users and the storage capacity of installations increases, the complexity of managing storage effectively now dominates the total cost of ownership, which is five to ten times more than the original purchase cost [7]. This is particularly true in large multi-user server computing environments, where allocation of storage capacity should ideally be done in a manner that effectively utilizes the relevant resource while multiplexing it among users to prevent monopolization by a single user. Although the cost per megabyte of disk space has been on the decline, any given installation will have a finite amount of disk space that needs to be partitioned and used effectively. While there is ongoing work in developing better mechanisms for partitioning resources such as processor cycles and network bandwidth [1, 5, 11, 17, 19, 30], file system resource management has not received as much attention.

Quotas are perhaps the most common form of file system resource management. However, there are two fundamental problems with this simple fixed-size limit disk-allocation scheme because it does not effectively account for the variability of disk usage among users and across time. The first problem is that in a large heterogeneous setting, some users will use very little of their quota, whereas experienced users, regardless of their quota, will find their quotas too constraining. Assuming a large environment where having quotas is required due to administrative costs, a potentially substantial portion of the disk is allocated to users who will not use their allocation, and is thus wasted. The second problem is that users' disk usage is often highly variable and much of this variability is caused by the creation of files that are short-lived; eighty percent of files have lifetimes of only a few seconds [8, 18, 29]. As a result, the vast majority of files created have no long term impact on available disk capacity, yet the current quota system would disallow such file operations when a quota limit has been reached even when there may be available disk capacity. The existence of a separate storage for temporary files is often ineffective because its separate file name-space requires complex reconfiguration of applications to use it, not to mention its tendency to require administrator intervention to avoid it being filled up. The result is often frantic negotiation with a system administrator at the least opportune time for additional space that wastes precious human resources, both from a user's and an administrator's perspective.

Traditional file systems operate on the assumption that all data written is of equal importance. Quota systems then place the burden of removing unwanted files on the user. However, users often have critical data and non-critical data storage needs. Unfortunately, it is not uncommon for a user to forget the reasons for storing a file, and in an effort to make space for an unimportant file, delete important ones. In a software development environment, an example of critical data would be source code, whereas non-critical data would be the various object files and other by-products that are generally less interesting to the developer. Deletion of the former would be devastating whereas the latter are generally only of interest during compilation. This burden of file management becomes only more complex for the user as improving storage capacity gives users the ability to store, and the need to manage, many more files than previously possible.

To provide more flexible file system resource management, we introduce *elastic quotas*. Elastic quotas provide a mechanism for managing temporary storage that improves disk utilization in multi-user environments, allowing some users to utilize otherwise unused space, with a mechanism for reclaiming space on demand. Elastic quotas are based on the assumption that users occasionally need large amounts of disk space to store non-critical data, whereas the amount of essential data a user keeps is relatively constant over time. As a result, we introduce the idea of an *elastic file*, a file for storing non-critical data whose space can be reclaimed on demand. Disk space can be reclaimed in a number of ways, including removing the file, compressing the file, or moving the file to slower, less expensive tertiary storage and replacing the file with a link to the new storage location [26]. In this paper, we focus on an elastic space reclamation model based on removing files.

Elastic quotas allow hard limits analogous to quotas to be placed on a user's persistent file storage whereas space for elastic files is limited only by the amount of free space on disk. Users often know in advance what files are non-critical and it is to the users' benefit to take advantage of such knowledge before it is forgotten. Elastic quotas allow files to be marked as elastic or persistent at creation time, later providing system-wide automatic space reclamation of elastic files as the disk fills up. Files can be re-classified as elastic or persistent after creation as well using common file operations. Elastic files do not need to be stored in a designated location, but instead users can make use of any locations in their normal file system directory structure for such files. A system-wide daemon reclaims disk space consumed by elastic files in a manner that is flexible enough to account for different cleaning policies for each user.

Elastic quotas are particularly applicable to any situation where a large amount of space is required for data that is known in advance to be temporary. Examples include Web browser caches, decoded MIME attachments, and other replaceable data. For instance, files stored in Web browser disk caches can be declared elastic so that such caches no longer need to be limited in size; instead, cached elastic data will be automatically removed if disk space becomes scarce. Users then benefit from being able to employ larger disk caches with potentially higher cache hit rates and reduced Web access latency without any concern about such cached data reducing their usable persistent storage space.

We designed elastic quotas to be simple to implement and install, requiring no modification of existing file systems or operating systems. The main component of the elastic quota system is the *Elastic Quota File System* (EQFS). EQFS is a thin stackable file system layer that can be stacked on top of any existing file system exporting the *Virtual File System* (VFS) interface [14] such as UFS [16] or EXT2FS [4]. EQFS stores elastic and persistent files separately in the underlying file system and presents a unified view of these files to the user. It makes novel use of the user ID space to provide efficient per user disk usage accounting of both persistent and elastic files using the existing quota framework in native file systems. A secondary component of the elastic quota system is the *rubberd* file system cleaner. Rubberd is a user-level program that cleans up elastic files when disk space becomes scarce. We have implemented a prototype elastic quota system in Sun's Solaris operating system and measured its performance on a variety of workloads. Our results on an untuned elastic quota system prototype show that our system provides its useful elastic functionality with low overhead compared to a commercial UFS implementation.

This paper describes the design and implementation of elastic quotas and is organized as follows. Section 2 describes the system model of how elastic quotas are used. Section 3 describes the design of the Elastic Quota File System. Section 4 describes the rubberd file system cleaner. Section 5 presents measurements and performance results comparing an elastic quota prototype we implemented in Solaris 9 to the Solaris 9 UFS file system. Section 6 discusses related work. Finally, in Section 7 we present some concluding remarks and directions for future work.

2 Elastic Quota Usage Model

To explain how elastic quotas are used, we first define some key elastic quota file concepts. A file can be either *persistent* or *elastic*. A persistent file is a file whose space will never be automatically reclaimed by the system. In a traditional file system, all files are considered persistent. An elastic file is a file whose space may be reclaimed on demand by the system. A file can change from being persistent to elastic and vice versa, but it can never be both at the same time. Each user is assigned an elastic quota, which is analogous to a traditional disk quota. Like traditional quotas, an elastic quota is a fixed-size limit that restricts the maximum amount of disk space that a user can use for storing persistent files. Unlike traditional quotas, an elastic quota does not limit the amount of disk space used for elastic files.

Users are only allowed to exceed their quotas by declaring files as being elastic. When a file is declared elastic, the system is effectively informed that the file may be removed from the system at a later time if disk space becomes scarce. The elastic quota system provides a contract between users and the system. The system agrees to allow users to use more than their quotas of disk space. In return, users agree that the system will be able to remove enough files to bring the users back within their quotas when disk space becomes scarce. The system guarantees that only elastic files are ever removed.

The elastic quota system provides a flexible interface consisting of two components: the *EQFS interface* and the *rubberd configuration interface*. The EQFS interface is a simple file system interface for declaring files elastic and managing persistent and elastic quotas. The rubberd configuration interface supports administrator-defined and user-defined policies that determine how and when disk space is reclaimed. We defer our discussion of the rubberd interface until Section 4 and focus first on the EQFS interface.

To allow users, user-level tools, and applications to declare files as elastic, EQFS provides a file system interface that can be used explicitly by users or as the basis for building tools for elastic file system management. To provide information about the elasticity of files without requiring any modification to existing file system utilities, the interface provides multiple views of the same directory hierarchy. These views enable users to declare files as elastic, and separate elastic and persistent files using common file operations. There are four views, which are most easily thought of as four directories. We refer to these views as `/home`, `/ehome`, `/persistent`, and `/elastic`.

Each view appears as a separate directory. `/home` and `/ehome` are two identical views of the file system. The key difference between the two is that files created in `/home` are persistent, whereas files created in `/ehome` are elastic. All other file operations are identical. `/persistent` and `/elastic` are read-only views of the file system. In `/persistent` only persistent files are visible; conversely in `/elastic` only elastic files are visible.

In all cases, users can use existing utilities for copying files, moving files, listing directories, editing files, etc. without requiring any changes to such tools to determine which files are elastic versus persistent. Furthermore, as discussed in Section 3, the interface can be supported in a way that requires no changes to existing file systems.

Consider the following example of how the EQFS file system interface provides its elastic functionality using common file operations. Suppose there is a student Mary who is using her computer account on a school system that provides elastic quotas. Mary's home directory is located at `/home/mary`. Mary often receives large MIME attachments, which she can not decode into her home directory without exceeding her quota. She simply extracts them into `/ehome` to use them. Since she still has the file within her mail inbox, there is no danger of losing the data.

The flexibility of the elastic quota usage model is that a file's effective location does not change when its status changes from persistent to elastic or vice versa. For instance, using this model for temporary storage is quite different from using a directory for just temporary files, such as `/tmp`. For example, developers may want to compile a large package, but do not have space for the temporary files associated with the build. Without elastic quotas it is necessary to edit `Makefiles` or move the entire tree to `/tmp`. Using the elastic quota usage model, developers would simply change directories from `/home` to the corresponding directory under `/ehome` and then can compile the program, without worrying about exceeding their quota.

The EQFS interface provides a useful foundation upon which developers and users can easily create tools that use normal file system interfaces to take advantage of elastic quota functionality. For example, higher-level functionality could be built on top of elastic quotas to allow users to specify that certain types of files should be considered elastic after some period of time. One policy may be that `*.o` files should be considered elastic if they were created more than a week ago. This could easily be implemented using a `cron` job to find and move files from `/home` to `/ehome`.

3 Elastic Quota File System

To support the elastic quota usage model, we created the *Elastic Quota File System* (EQFS). An important benefit of the elastic quota system is that it allows elastic files to be mixed together with persistent files and located in any directory in the file system. To provide this benefit, the system must efficiently find elastic files anywhere in the directory hierarchy. Furthermore, the system must account for the disk usage of persistent files separately from elastic files since only persistent files are counted against a user's quota. With substantial implementation effort, one could build a new file system from scratch with an elasticity attribute associated with each file and a quota system that accounts for elastic and persistent files separately. The problem with this approach is that it requires users to migrate to an entirely new file system to use elastic quotas.

EQFS addresses these design issues by first storing persistent and elastic files in separate underlying directories to efficiently account for and identify elastic files. EQFS then using file system stacking [10, 21, 25] to stack on top of both the persistent and elastic directories to present a unified view of the files to the user. Using file system stacking, a thin layer is inserted directly above an existing file system, thus allowing the layer to intercept and modify requests coming from upper layers or data being returned from lower layers. Although stackable file systems run in kernel space for best performance, they do not require kernel modifications and can extend file system functionality in a portable way [32].

Section 3.1 describes how EQFS stacks on top of both persistent and elastic directories, and how this supports the multiple views. Section 3.2 describes how EQFS utilizes the separation of persistent and elastic storage with traditional quota functionality to provide efficient disk usage accounting. Finally, Section 3.3 summarizes the implementation of individual EQFS file operations.

3.1 File System Stacking

One of the main features that EQFS must provide is a way to associate an attribute with each file that indicates whether it is elastic or persistent. Taking a file system stacking approach, one way to do this would be to store a separate attributes file for each file in the underlying file system that is manipulated by the upper layer file system. The approach provides a general way to extend file attributes, but would require accessing an entirely separate file for determining whether the respective file is elastic. However, it requires substantial additional overhead for an elasticity attribute that could potentially be stored as a single bit of information. Another design alternative would be for the stackable file system to manipulate special-purpose inode bits on the lower file system, to be able to flag a file as elastic. However, stackable file systems are designed to be modular and independent from the underlying file systems they mount on. Such access violates the principles of stacking as it makes a stackable file system dependent on the specific implementation details of the underlying file system.

To provide an efficient stacking approach, we designed EQFS to stack on top of two underlying directories in the native disk file system, one for storing all persistent files and the other for storing all elastic files. Because of the separate directories for persistent and elastic files, EQFS can infer whether a file is persistent or elastic from the location of the file. Although the system introduces a new file property — namely its persistence or lack thereof — EQFS does not need to store this property as part of the on-disk inode. In fact, EQFS does not maintain any state other than what it uses to stack on top of the underlying persistent and elastic directories. EQFS can be stacked on top of any existing file system exporting the *Virtual File System* (VFS) interface [14], such as UFS [16] or EXT2FS [4]. The VFS was designed as a system-independent interface to file systems and is now universally present in UNIX operating systems, including Solaris, FreeBSD, and Linux. By building on top of the VFS, EQFS serves as a higher-level file system abstraction that does not need to know about the specifics of the underlying file system.

In the VFS, a *virtual node* (vnode) is a handle to a file maintained by a running kernel. This handle provides a common view for public data associated with a file, and is the vehicle for interaction between the kernel proper and the file system. Vnodes export an interface for the set of generic operations commonly applicable to files and directories, known as *vnode operations* (vops). A stackable file system is one that stacks its vnodes on top of those of another file system. Stacked file systems are thus able to modify the kernel's view of the underlying file system by intercepting data and requests flowing between underlying file systems and the kernel proper through their vnode-private data and stacked vops. Our design of EQFS provides four important benefits:

1. **Compatibility with existing file systems**: Because EQFS simply stacks on top of existing file systems, it is compatible with and does not require any changes to existing file systems. Furthermore, EQFS can be used with commodity file systems already deployed and in use. EQFS is ignorant of the underlying file systems and makes no assumptions about the underlying persistent and elastic directories. In particular, the underlying directories need not be on the same file system, or even of the same file system type.
2. **No modifications to commodity operating systems**: Since EQFS stacks on top of the widely used VFS interface, EQFS can be implemented as a kernel module that can be loaded and used without modifying the kernel or halting system operation. Users can therefore use elastic quotas in the large installed base of commodity operating systems without upgrading to an entirely new system.
3. **Leveraging existing development investments**: EQFS leverages existing functionality in file systems instead of replicating it. EQFS is a thin layer of functionality that extends existing disk-based file systems rather than replacing them. EQFS's ability to reuse existing file system functionality results in a much simpler implementation.
4. **Low performance overhead**: Since file system performance is often crucial to overall system performance, elastic quotas should impose as little performance overhead as possible. EQFS runs in kernel space to minimize performance overhead.

EQFS is a thin stackable layer that presents multiple views of the underlying persistent and elastic directories as `/home`, `/ehome`, `/persistent`, and `/elastic`, as described in Section 2. To provide a unified view of all files, EQFS creates `/home` and `/ehome` by merging the contents of the underlying persistent and elastic directories. For example, if files A and B are stored in one directory and C and D are stored in another, merging the two directories will result in a unified directory that contains A, B, C, and D. `/persistent` and `/elastic` are created by simply referring to the respective underlying persistent and elastic directories. Figure 1 illustrates the structure of the views and underlying directories in EQFS.

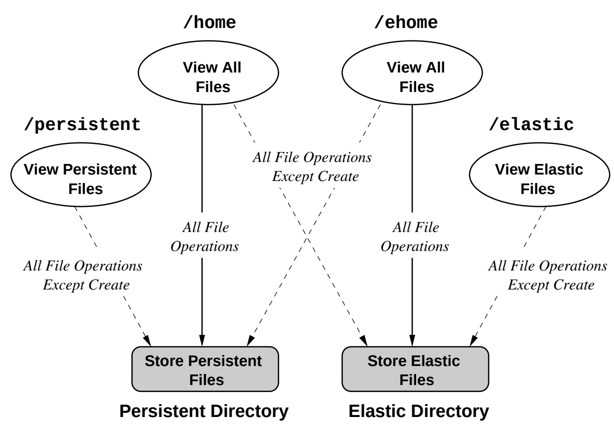

Figure 1: Views and directories in EQFS

EQFS makes merging of the underlying persistent and elastic directories possible by ensuring that both directories have the same structure and by avoiding file name conflicts. As discussed in Section 2, each of the four views exports the same directory structure to the user. Similarly, for each directory visible from these views, EQFS maintains a corresponding underlying directory for persistent files and a corresponding underlying directory for elastic files. If the directory structures were not the same, it would be ambiguous how to unify the two structures when some directories could be present in one but not the other. EQFS avoids file name conflicts by not exposing the underlying directories directly to users and only permitting file creation through `/home` and `/ehome`. `/persistent` and `/elastic` cannot be used for file creation. If the underlying directories were not protected, a user could create a file in the persistent directory and a file in the elastic directory, both with the same name. This would cause a file name conflict when the underlying directories are unified. File name conflicts are not possible using the views.

EQFS discriminates between the underlying directories unified in `/home` and `/ehome` in order to make file creations in `/home` persistent and file creations in `/ehome` elastic. EQFS unifies the two underlying directories by treating one of them as the primary directory and the other one as the secondary sister directory. The contents of the two directories are joined into a unified view, but any file creations are always made to the primary directory. EQFS populates `/home` by treating the underlying persistent directory as the primary directory and the underlying elastic directory as the sister directory. Conversely, `/ehome` has the elastic directory as a primary underlying directory and the persistent directory as a sister underlying directory. As a result, elastic files created in `/ehome` are elastic because the underlying primary directory is elastic.

3.2 Disk Usage Accounting

Quotas usually keep track of disk blocks or inodes allocated to each user or group. Traditional quota systems are implemented by specific file system code. EQFS utilizes this native quota functionality to simplify its implementation. However, as regular quotas do not have elastic files for which no usage limits exist, EQFS must build these extended semantics using existing primitives.

EQFS solves this disk usage accounting problem by defining a *shadow user ID* for each user. A shadow user ID is a second unique user ID that is internally assigned by EQFS to each user of elastic quotas. EQFS uses a mapping between normal user IDs and shadow user IDs that allows it to deduce one ID from the other in constant time. Persistent files are owned by and accounted for using normal user IDs, whereas elastic files are owned by and accounted for using shadow IDs. Shadow user IDs are made to infinite quotas, allowing disk space used by elastic files to not be limited by users' quotas. The shadow ID mapping used in our EQFS implementation defines the shadow ID for a given user ID as its twos-complement. Since the user ID is typically a 32-bit integer in modern systems and even the largest systems have far fewer than two billion users, at least half of the user ID space is unused. Our implementation takes advantage of the large underutilized ID space using the simple twos-complement mapping.

Rubberd takes advantage of the underlying quota system to obtain information on users' elastic space consumption in constant time. Even though there is no quota limit set on a user's shadow ID, the quota system still accounts for the elastic file usage.

3.3 File Operations

EQFS provides its own set of vnode operations (vops), most of which can be summarized as transforming the user ID to the shadow user ID if necessary, and then passing the operation on to the underlying vnode(s). The most notable exceptions to this generalization are the following vops which require additional functionality to maintain EQFS semantics: LOOKUP, READDIR, RENAME, MKDIR,CREATE, and LINK.

The LOOKUP vop returns the vnode for the given file name in the given directory. Since EQFS directory vnodes are associated with two underlying directories, LOOKUP must potentially search both directories for the file before returning the EQFS version of the underlying vnode. To enforce the invariant of always having two underlying directories for an EQFS directory, EQFS lazily creates missing directories in `/elastic` or `/persistent` if it cannot find them. This makes it easy to migrate existing file systems to EQFS; simply mount `/persistent` on a spare partition, and the first access to a directory will cause its sister to be created.

READDIR returns a subset of entries in a directory. Since directories in an EQFS mount are mirrored in both underlying sources, any given EQFS directory will contain duplicates for its subdirectories, which are eliminated before being returned by the kernel. Our READDIR implementation caches the merged result of both underlying directories for improved performance.

RENAME slightly departs from its traditional semantics to support changing the elasticity of a file; if the file names are the same and the target directory corresponds to the same logical directory on the mirror mount point (i.e., `/ehome/mary` and `/home/mary`), the file is moved to the mirror mount's primary directory, and its ownership updated accordingly. Thus, renaming a file from `/home/mary/foo` to `/ehome/mary/foo` will make it elastic, and the converse will make it persistent.

MKDIR creates a directory and returns the corresponding vnode. Under EQFS, this vop first checks both the primary and sister sources to make sure that there are no entries with the given name, passing the operation down to both file systems to create the named directory if successful. Directories can be created under either `/home` or `/ehome`, but they are mirrored persistently under both views. Note that directories are considered to be persistent, and like persistent files, are only removed if done so explicitly by the user.

CREATE creates the named file if it does not exist, otherwise it truncates the file's length to zero. Like the MKDIR vop, it must first check both sources to make sure that it does not create duplicates. If the file does not exist, the the file is always created in the primary directory, as outlined earlier. Note that to prevent namespace collisions when `/persistent` and `/elastic` are merged into `/home` and `/ehome`, the system must disallow direct namespace additions such as new files, directories or links to these underlying directories without passing through the EQFS layer. To ensure this, `/persistent` and `/elastic` are covered by a thin loopback-like file system layer which maintains no state and passes all operations on to the underlying file system, sans namespace additions.

LINK creates hard links. Hard links are created as inheriting the elastic properties of the file that is being linked, regardless of whether the operation is done under `/home` or `/ehome`. When a hard link is created to a persistent file, the hard link is considered persistent; a hard link that is created to an elastic file is considered elastic. Hard links to files across `/home` and `/ehome` are disallowed to avoid conflicting semantics in which a file is linked as both persistent and elastic.

4 Rubberd

To provide a mechanism for reclaiming disk space used by elastic files when the disk becomes too full, we developed *rubberd*, a user-level space reclamation agent that leverages the separate elastic file storage and quota functionality provided by EQFS. Rubberd is a highly configurable cleaning agent, and its behavior can be set by the administrator, and also in part by the users of the system.

System administrators can specify the location of the elastic file system root, and parameters for describing the disk utilization characteristics required to start and stop cleaning operations and determine the rate of cleaning operations. The *start cleaning* threshold is the percentage of total disk space above which rubberd will start cleaning elastic files. The *stop cleaning* threshold is the percentage of total disk space below which rubberd will stop cleaning elastic files. The *disk usage sampling interval* is the amount of time that rubberd waits before it checks if the total disk usage is above the start cleaning threshold. System administrators can also choose whether or not to allow users to specify their own policies for determining the ordering in which elastic files are removed.

When reclaiming disk space, rubberd works in conjunction with the EQFS quota system to identify users who are over their quota limits. By default, rubberd removes elastic files from users who are over their quota in proportion to how much each user is over quota.

Note that rubberd only removes elastic files from users whose total disk space consumption, including both persistent and elastic usage, is over quota. If a user is consuming a large amount of elastic space but is below quota, none of that user's elastic files will be removed. In the absence of a user-specified removal policy, rubberd will remove elastic files from a given user in order of least recent file access time first.

Section 4.1 describes the rubberd cleaning mechanisms that enable rubberd to efficiently support a wide-range of cleaning policies. Section 4.2 describes the mechanism by which users can select the order of removal for elastic files to be cleaned. Section 4.3 describes the default rubberd proportional cleaning algorithm we use.

4.1 Cleaning Mechanisms

The key goal in the design of rubberd was to efficiently support a wide-range of removal policies provided by the system administrator or users without adversely impacting normal file operations. For example, when rubberd wakes up periodically, it must be able to quickly determine if the file system is over the start cleaning threshold. If the system is over the threshold, 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, such as a file's owner, size, last access time, or name.

To meet this goal, rubberd was designed as a two-part system that separates obtaining file attributes from the actual cleaning process. Rubberd scans the file system nightly for all elastic files under `/elastic` to build a lookup log of information about the elastic files and their attributes. This log serves as a cache that rubberd then uses to lookup file attributes to determine what files to clean when the file system is over the start cleaning threshold. The log can be stored in a database, or a file or set of files. The prototype rubberd that we built used per-user linear log files that are created in parallel for faster completion time. We chose this approach over using popular databases such as NDBM or DB3 primarily for the sake of simplicity.

Rubberd's nightly scanning is analogous to other system processes such as backup daemons or GNU `updatedb` [15] that are already widely used and do nightly scans of the entire file system. Because rubberd does a full scan of elastic files, it can obtain all the information it may need about file attributes. Rubberd does not require any additional state to be stored on normal file operations, which would impact the performance of these operations. Since the vast majority of files created typically have short lifetimes of a few seconds[8, 18, 29], rubberd also avoids wasting time keeping track of file attributes for files that will no longer exist when the file system cleaning process actually takes place. Although the cleaning log will not have information on files just recently created on a given day, such recent files typically consume a small percentage of disk space [8]. Furthermore, we expect that most removal policies, including the default removal policy, will remove older files that have not been accessed recently.

When rubberd cleans, it uses the files in the log when applying a cleaning policy and searches the log for files that match the desired parameters (owner, creation time, name, size, etc.). By default, rubberd uses a disk usage sample interval of one hour so that it may reuse the log 23 times before another file system scan occurs. Since our lookup log is updated nightly at a time of least system activity, rubberd also initiates a cleanup check right after the log is updated. Because the log is by default updated only once a day, it is possible that rubberd could run out of elastic files in the log to clean while the disk utilization is still above the stop cleaning threshold. In this case, rubberd will initiate a more expensive full scan of the `/elastic` directory to update the elastic files log and restart the cleaning phase using this updated log. In this way, rubberd is able to optimize the common case cleaning using the log while limiting the need to do recursive scans of `/elastic` only as a last resort.

Normally, the rubberd cleaner simply runs as a low priority process to minimize its impact on other activities in the system; a progress-based regulation approach could also be used [6]. However, if the system is sufficiently busy that the rubberd cleaner does not complete before its next scheduled cleanup check, the priority of rubberd is raised to that of a system-level process to ensure that the cleaning process is given enough time to run. Rubberd cleaning can also be initiated by an administrator by sending a signal to the cleaning process, presumably because the administrators determined that cleaning is needed right away. In this case, the cleaner is also run at a higher system-level priority.

4.2 User Policy Files

If the system administrator has allowed users to determine their own removal policies, users are then allowed to use whatever policy they desire for determining the order in which files are removed. A user-defined removal policy is simply a file stored in `/var/spool/rubberd/`*`username`*. The file is a newline-delimited list of file and directory names or simple patterns thereof, designed to be both simple and flexible 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 could be specified. Table 1 shows a few examples and explains them.

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 directory

Table 1: Example user removal policy file entries

Management of this removal policy file is done similarly to how `crontab` manages per-user `cron` jobs. A separate user tool allows a user to add, delete, or edit their policy file — as well as to install a new policy from another source file. The tool verifies that any updated policy conforms to the proper syntax. This tool also includes options to allow users to initialize their default policy file to the list of all their elastic files, optionally sorted by name, size, modification time, access time, or creation time.

4.3 Default Cleaning Algorithm

Rubberd's default removal policy proportionally distributes the amount of data to be cleaned based on the amount by which users exceed their quota limits. Rubberd is flexible enough that many other cleaning algorithms and policies could also be used, but due to space constraints, a detailed discussion of different cleaning algorithms and policies is beyond the scope of this paper. Rubberd's default proportional share cleaning behavior is provided by a simple algorithm that is easy to implement. When rubberd wakes up every sample interval, it begins by checking the current disk usage on the system. If the usage is over the start cleaning threshold T_{start}, rubberd calculates the total amount of disk space to clean (C_{total}) as follows:

$$C_{total} = S \times \frac{(T_{start} - T_{stop})}{100} \tag{1}$$

where T_{stop} is the stop cleaning threshold and S is the total size of the disk.

Next, rubberd finds the amount of elastic disk usage over quota for each user (E_u). This value is retrieved from the quota system, by querying it for the user's current quota usage based on the user's UID for persistent disk usage, the corresponding shadow UID for elastic disk usage, and comparing the sum of both usage values to the user's actual fixed quota. Rubberd sums all E_u values for all over-the-quota users into E_{total}. Then, rubberd computes the portion of disk space that should be cleaned from each user (C_u) as follows:

$$C_u = C_{total} \times \frac{E_u}{E_{total}} \tag{2}$$

Before rubberd can begin to remove users' files, it decides the order in which the files will be removed. Rubberd removes files as long as the total sum of removed files is less than C_u. First, rubberd removes files in the order they are listed in the user's custom policy file. If the policy file does not exist, or all files corresponding to the policy file have been removed, rubberd will then use the system default removal policy for removing more elastic files if more files need to be removed. The system default policy is to remove files by earliest access time first, which is based on the assumption that these files are generally the least likely files to be used again in the future.

5 Evaluation

To evaluate elastic quotas in a real world operating system environment, we implemented a prototype of our elastic quota system in Solaris 9, the latest operating system release[1] from Sun Microsystems. We chose Solaris because it is widely used in large production environments such as the file servers on which elastic quotas would operate. We present some experimental results using our prototype EQFS and rubberd implementations. We compared EQFS against Solaris 9 UFS [2], the most popular file system used on Solaris servers. We also measured the impact of rubberd on a running system.

We conducted all experiments on a Sun-Fire 480R multiprocessor system with four 750 MHz UltraSPARC-III CPUs and 4 GB of RAM, running Solaris 9. We believe this is a moderate size machine for the type of large file servers that elastic quotas will be useful on. Although such installations will probably include RAID arrays or SAN products, we focused on the native disks that were in the machine; this helped us to analyze the results without worrying about interactions with other storage systems. For all our experiments, we used a local UFS file system installed on a Seagate Cheetah 36LP disk with 36 GB capacity and 10000 rpm. UFS includes optional logging features used in some installations that enable a form of journaling that logs meta-data updates to provide higher reliability guarantees. We considered both UFS and UFS logging (LUFS) in our experiments. For each experiment, we only read, wrote, or compiled the test files in the file system being tested. All other user utilities, compilers, headers, and libraries resided outside the tested file system. Unless otherwise noted, all tests were run with a cold cache by unmounting all file systems that participated in the given test after the test completed and mounted the file systems again before running the next iteration of the test.

We report experimental results using both file system benchmarks and real applications. Sections 5.1 and 5.2 describe the file system workloads we used for measuring EQFS and rubberd performance, respectively. Sections 5.3 shows results for three file system workloads comparing EQFS to UFS to quantify the performance overhead of using EQFS. Section 5.4 shows results quantifying the impact of rubberd's actions on a running system: reclaiming storage, building its database, etc.

5.1 EQFS Benchmarks

To measure EQFS performance, we stacked EQFS on top of UFS and compared its performance with native UFS. We measured the performance of four file system configurations on a variety of file system workloads: UFS without logging (UFS), UFS with logging (LUFS), EQFS on top of UFS (EQFS/UFS), and EQFS on top of LUFS (EQFS/LUFS). We used three file system workloads for our experiments: PostMark, a recursive find, and a compilation of a large software package, the Solaris 9 kernel.

[1]FCS due summer 2002.

PostMark The first workload we used was PostMark [13], a well-known file system benchmark that creates a large pool of continually changing files to simulate a large electronic mail server workload. PostMark creates an initial pool of text files of various sizes, then performs *transactions* by reading from, appending to, or creating and deleting files. The workload provides a useful measure of file system performance for users performing daily tasks such as reading mail, editing files, and browsing their directories. This workload exercises some of the more complex EQFS file operations and provides a conservative measure of EQFS overhead. We only report PostMark measurements for EQFS using `/home` since EQFS performs identically when using either `/home` or `/ehome` in this experiment.

Because the default PostMark workload is too small, we configured PostMark to perform 5000 transactions starting with an initial pool of 2500 files with sizes between 8 KB and 64 KB, matching file size distributions reported in file system studies [29]. Previous results obtained using PostMark show that a single PostMark run may not be indicative of system performance under load because the load is single-threaded whereas practical systems perform multiple concurrent actions [28]. Therefore, we measured the four file systems running 1, 2, 4, and 8 PostMark runs in parallel. This not only allows us to conservatively measure EQFS's performance overhead, but also evaluate EQFS's scalability as the amount of concurrent work done increases. The latter is even more important than the former, since raw speed can be improved by moving to a larger machine, whereas poorly-scaling systems cannot be easily helped by using larger machines.

Recursive Find The second workload we used was a recursive scan of the full Solaris source base — which is a collection of 32416 Java, C, and assembly files in 7715 subdirectories — using `find . -print`. Since EQFS is implemented as a stackable union file system, some EQFS file operations must be performed on both `/elastic` and `/persistent`. For example READDIR must merge two directory contents; and LOOKUP must find a file in either of these two directories. Since LOOKUP operations are common [20], and merging two directory contents can be costly, this `find` test, when run with a cold cache, is intended to show the worst-case performance overhead of EQFS when using these file system operations. To measure EQFS performance with this workload, all files were stored persistently and we performed the recursive scan using both `/home` and `/ehome`.

Solaris Compile The third workload we used was a build of the Solaris 9 kernel, which provides a more realistic measure of overall file system performance. The kernel build is inherently parallel, and as such the elapsed time masks overheads due to disk latency. As in all such measurements, the increase in system time is of interest, as it indicates the extra processing done by EQFS. This build processes 5275 C and assembly source files in 1946 directories to produce 4020 object files and more than 10,000 other temporary files. We used Sun's Workshop 5.0 compilers and set the maximum concurrency to 16 jobs to keep the CPU busy and to ensure that the overhead is not underrepresented due to time spent performing I/O. Overall this benchmark contains a large number of reads, writes, and file lookups, as well as a fair mix of most other file system operations such as unlink, mkdir, and rename. To measure EQFS performance with this workload, all source files were stored persistently and we performed the build in both `/home` and `/ehome`. When using `/ehome`, all object files are created elastic.

5.2 Rubberd Benchmarks

To evaluate rubberd, we measured how long it took to build its nightly elastic files log and use it for cleaning elastic files. The rubberd log we used contains the names of elastic files and `lstat(2)` output. 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. The working set includes the actual files of 121 users, many of whom are software developers. The file set includes 1,194,133 inodes and totals over 26 GB in size; more than 99% of the file set are regular files. 24% of the users use less than 1 MB of storage; 27% of users use between 1–100 MB; 38% of users use between 100 MB–1 GB of storage; and 11% of users consume more than 1 GB of storage each. Average file size in this set is 21.8 KB, matching results reported elsewhere [20]. We treated this entire working set as being elastic. Previous studies [23] show that roughly half of all data on disk and 16% of files are regeneratable. Hence by treating all files as elastic, we are effectively modeling the cost of using rubberd on a disk consuming a total of 52 GB in 7.5 million files. Using EQFS mounted on LUFS, we ran three experiments with the working set for measuring rubberd performance: building the elastic files log, cleaning elastic files using the log, and cleaning elastic files while running a file system workload.

Elastic File Log Creation The first rubberd benchmark we used measured the time it took to build an elastic file log by scanning the entire `/elastic` directory through EQFS. The scan is recursive and builds per-user log files in parallel with a separate child process for each user, storing `lstat(2)` information on each file in the 26 GB data set described above. Thus, the completion time to create the log is determined by the users with the most elastic files. Building such a disk scan may take a while and can disrupt user activity, particularly when run on larger file systems. As a result, the log is intended to be built at night or when few users are active. Nevertheless, once the log is created, we expect that scanning it to find elastic files suitable for removal can be executed much faster than scanning the file system directly, especially if the set of files to be removed is significantly smaller than the set of elastic files on the system.

Elastic File Cleaning The second rubberd benchmark we used measured the time it took to use the elastic file log to clean a portion of the disk on an otherwise idle system using our default cleaning policy. Rubberd operates by retrieving the list of files for each user, ordering them based on the default cleaning algorithm as described in Section 4.3, and then removing files in order from this list. To provide a conservative measure of cleaning overhead, we set the rubberd parameters such that 5 GB of disk space, roughly 1/4 of the space used by elastic files, would need to be removed to achieve the desired state. While we do not propose using such a high hysteresis value for normal file systems, we chose a large value to avoid under-representing the cost of rubberd operation.

Rubberd Cleaning with Solaris Compile The third rubberd benchmark we used measured the time it took to run the second rubberd benchmark in conjunction with the Solaris Compile described in Section 5.1. This experiment measures the more practical impact of rubberd cleaning on a system operating under load. Here, we ran the previous elastic file cleaning benchmark on the same file set, but at the same time we ran the parallel Solaris compilation, simulating high CPU and I/O load. In this experiment, the kernel build was performed under `/ehome`, although we did not need to worry about rubberd causing the build to fail as the database contained enough files from which to satisfy the cleaning request. Note that both the kernel build and rubberd cleaning were executed on the same physical disk.

5.3 EQFS Results

PostMark The following two figures show the results for running PostMark on each of the four file systems. Figure 2 shows the total throughput of the system and Figure 3 shows the total time it takes to complete all of the runs. The results for LUFS show that EQFS incurs less than 10% overhead over LUFS, with the EQFS/LUFS throughput rate and completion time being within 10% of LUFS. The results for UFS are even better, showing that EQFS incurs hardly any overhead, with the EQFS/UFS throughput rate and completion time being within 1% of UFS. These results show that EQFS's overhead is relatively modest even for a file system workload that stresses some of the more costly EQFS file operations.

EQFS exhibits higher overhead when stacked on LUFS versus UFS in part because LUFS performs better and is less I/O bound than UFS, so that any EQFS processing overhead becomes more significant. LUFS logs transactions in memory, clustering meta-data updates and flushing them out in larger chunks than regular UFS, resulting in higher throughput and lower completion time than regular UFS for

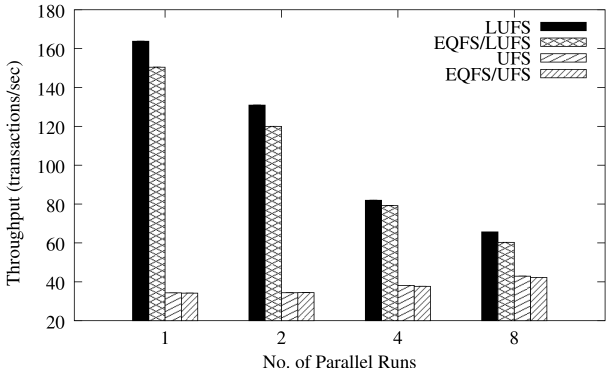

Figure 2: PostMark transactions per second results

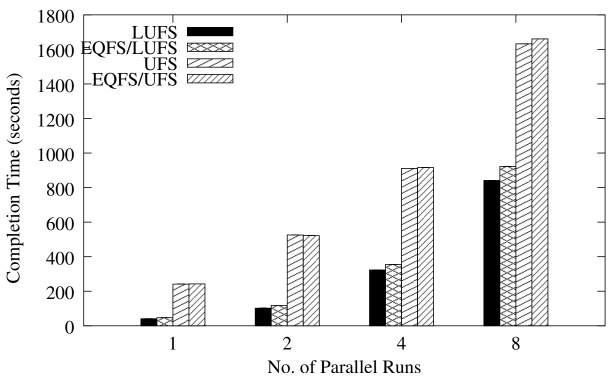

Figure 3: PostMark completion time results

PostMark. However, UFS scales better than LUFS, as evident by the fact that the total throughput rate for UFS increases slightly with more parallel PostMark runs whereas the throughput rate for LUFS decreases significantly. More importantly, the results show that EQFS scales with the performance of the underlying file system and in no way impacts performance adversely as the amount of concurrent work done increases.

Recursive Find Figure 4 shows the results for running the recursive `find` benchmark on each of the file systems. We show results for running the benchmark with both cold cache and warm cache. The cold cache results show that EQFS incurs roughly 80% overhead in terms of completion time when stacked on top of UFS or LUFS, taking about 80% longer to do the recursive scan than the native file systems. The high EQFS overhead is largely due to the frequent READDIR operations that are done by the recursive scan. Using a cold cache with the recursive scan, each READDIR operation requires going to disk to read the respective directory block. Because EQFS must merge both persistent and elastic directories, READDIR requires two directory operations on the underlying file system. This causes twice as much disk I/O as using the native file system to read directories, resulting in a significantly higher completion time. This is compounded by the fact that most FFS-like file systems such as UFS make an attempt to cluster meta-data and data together on disk; UFS does not necessarily place the two sister directories close to each other on disk, hence reading the two directories not only causes multiple I/O requests, but also causes the disk to seek more, which slow overall performance. Overall the recursive `find` benchmark is not representative of realistic file workloads, but provides a measure of the worst-case overhead of EQFS as READDIR is the most expensive EQFS operation.

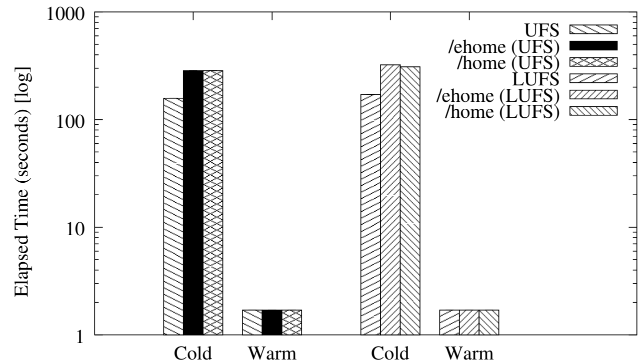

Figure 4: Elapsed times (seconds, log-scale) of a recursive `find`, using cold and warm caches

In this test all files found were located under `/persistent`. This meant that looking up files via `/home` found the files in the primary directory, whereas when looking them up via `/ehome`, the files were logically located in the sister directory and EQFS had to perform two LOOKUP operations to find those files. Nevertheless, Figure 4 shows that the overhead of looking up those files with an extra LOOKUP was small: 4.2% when mounted on LUFS and only 0.1% when mounted on top of UFS.

When using a warm cache, Figure 4 shows that EQFS incurs essentially no overhead versus the native file system when stacked on top of either UFS or LUFS. For all file systems, the recursive find took less than two seconds to complete, roughly two orders of magnitude faster than when using a cold cache. Like other Solaris file systems, our EQFS implementation utilizes the Solaris Directory Name Lookup Cache (DNLC). The warm cache results illustrate the full benefits of caching. Since the directory contents are already merged and cached, EQFS does not spend additional time merging directories, resulting in negligible performance overhead. There is also no difference in EQFS performance when using `/home` versus `/ehome` since LOOKUP requests are satisfied from the cache and EQFS does not call the underlying file system.

Solaris Compile Figure 5 shows the results for running the Solaris compile on each of the file systems. Results are reported in terms of elapsed time and system time. Although we do not report user time, we note that the sum of user and system time is higher than elapsed time, due to the parallel nature of the build and the multiprocessor machine used. The results show that EQFS incurs almost no overhead in completion time when stacked on top of UFS or LUFS, taking less than 1% longer to complete the compilation. EQFS incurs less than 5% overhead versus UFS or LUFS in terms of system time. These results show that EQFS imposes very little performance overhead, and does not limit file system scalability for realistic application workloads such as a large parallel compilation.

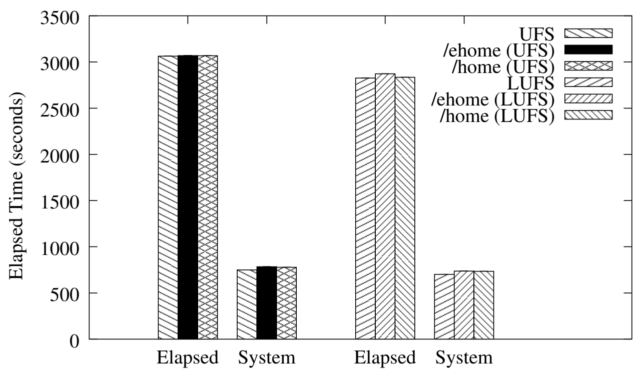

Figure 5: Elapsed and system times (seconds) of a large compile benchmark

The performance of EQFS when doing the compile from `/ehome` is slightly worse than when doing the compile from `/home` because the source files are located in the underlying persistent directory. As a result, LOOKUP operations for uncached entries from `/ehome` will cause a lookup in both underlying directories. We analyzed the cost and frequency of various file operations for the compilation and found that while LOOKUP operations are the most frequent, accounting for almost half of all file operations, the total time spent doing LOOKUP operations was small. Since the same file is typically referenced multiple times during the build, requests are satisfied from the cache, resulting in little performance difference between compiling in `/home` versus `/ehome`.

For comparison purposes, we also measured the overhead of a null stacking layer and found that it incurred about 0.5% overhead when stacked on top of UFS or LUFS. This means that EQFS only imposes roughly 0.5% more overhead beyond the basic stacking costs, even though EQFS provides significant additional functionality. EQFS's low overhead is due in part to its effective use of the DNLC for vnode caching. Previously published results for similar compilation benchmarks on trivial stacking systems [31] that simply copy data between layers show a 14.4% increase in system time, significantly higher than what we measure for EQFS.

5.4 Rubberd Results

Elastic File Log Creation Table 2 shows the results for building the elastic file log. The results show that the entire log was created in only about 10 minutes using a cold cache. This indicates that the cost of building the elastic file log is small and should have little if any effect on system operation if run during off-peak hours. Table 2 also shows that the entire log was created in less than three minutes when using a warm cache. In practice, we expect actual numbers to be closer to those of a cold cache.

Time	Cold	Warm	Speedup
Elapsed	638.5	175.4	364%
User	7.3	7.2	1.3%
System	76.1	72.0	5.7%

Table 2: Times (seconds) to build the elastic file log

Elastic File Cleaning Table 3 shows the results of running the elastic file cleaning benchmark to clean 5 GB of disk space. The entire cleaning process took less than two minutes. Compared to the time it took to scan the disk and build the elastic file log, the overhead of cleaning is more than five times less, which shows the benefit of using the log for cleaning. In the absence of the elastic file log, removing the same set of data would have involved scanning the entire disk in order to find candidate files, which would have taken significantly longer. As expected, the figures indicate that the job is primarily I/O bound, with user and system times amounting to a mere fraction of the completion time.

The cleaning cost is low enough that rubberd may be run multiple times during the course of a day without much overhead. For instance, if rubberd were run once an hour, the rubberd would only need three percent of the time to clean 120 GB of disk space a day. It is unlikely that this much storage space would need to be reclaimed daily for most installations, so that rubberd cleaning overhead in practice would typically be even lower.

Elapsed	User	System
111.8	16.5	10.1

Table 3: Times (seconds) to clean 5GB

Rubberd Cleaning with Solaris Compile Table 4 shows the completion time for executing our large Solaris compile benchmark while rubberd is running. These results measure the impact of running rubberd cleaning on the Solaris compilation by comparing the compilation completion times when rubberd is not running, when rubberd is running at low priority, and when rubberd is running at normal priority.

Comparing with the Solaris compilation results without rubberd running, we observe a 3.5% degradation in completion time when rubberd is running at low priority, and a 4% degradation when running at regular priority. Running rubberd as a lower priority job does not make a large difference, primarily since both jobs are I/O bound, hence CPU scheduling priority has a very small impact on completion time. Furthermore, we observe that there are numerous lull times dur-

Rubberd Status	Elapsed Time
Not running	2872.1
Low Priority	2974.5
Normal Priority	2991.5

Table 4: Elapsed time (seconds) to build kernel in `/ehome` *in three ways: alone (rubberd not running), with rubberd running at a low priority, and with rubberd running at a normal priority.*

ing a regular system's operation in which it would be possible to schedule rubberd to run with an even lower impact on system operation [6].

Overall, however, we observe that the impact of rubberd running even once an hour with a conservatively large amount of data to remove does not significantly hamper normal system operation. It is also important to note that as these files are temporary they would be removed anyhow; rubberd provides the added convenience of automatically doing so when disk space becomes low and before the disk fills up and hampers user productivity.

6 Related Work

Elastic quotas are complementary to Hierarchical Storage Management (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 some way to access files stored on the slower media, ranging from file search software to leaving behind a link to the new file storage location in the original file location. Examples of HSM systems include the Network Appliance Snapshot system [3], the Smart Storage Infinet system [26], IBM Storage Management [12], and UniTree [27]. The UniTree HSM system uses a combination of file size and the age of a file in hours to compute the eligibility of a file to be moved to another medium. Rubberd can be similarly configured to clean files based on size and time; however, it also uses more complex algorithms to compute disk space usage over time. Elastic quotas can be used with HSM systems as a mechanism for determining which files are moved to slower storage. Given an HSM system, rubberd could then reclaim disk space when it becomes scarce by moving elastic files to the slower layers of the HSM storage hierarchy.

The design of EQFS builds on previous work in stackable file systems. Rosenthal first implemented file system stacking on top of the VFS interface in SunOS 4.1 [21] more than a decade ago. Skinner and Wong developed further prototypes for extending file systems in SunOS [25]. Guy and Heidemann developed slightly more generalized stacking in the Ficus layered file system [9, 10]. Stacking in 4.4 BSD is derived from Heidemann's work. More recently, Zadok and Nieh have developed a system for stackable file system code generation that simplifies stackable file system development and improves file system portability [32]. EQFS uses the idea of file system stacking but does not require much of the functionality in more generalized stacking infrastructures, such as being able to manipulate file data. As a result, the performance overhead of using EQFS is lower than the performance overheads that have been reported for using these other systems.

The idea of unification used in EQFS is similar to the union mounts in 4.4 BSD [17] and Plan 9 [19]. However, EQFS differs from these systems in four ways. First, EQFS provides true fan-out stacking on two underlying directories as opposed to linear stacking. EQFS does not need to use linear stacking in part because it only provides unification of two underlying directories as opposed to unification of an arbitrary number of underlying file systems. Second, EQFS does not require complex mechanisms to resolve differences in directory structure or file name conflicts in the underlying file systems. Third, EQFS provides not just one unified view of the underlying directories, but two unified views with different semantics for file creation. Fourth and most importantly, EQFS does not treat the underlying directories as read only, eliminating the need for the potentially expensive copy-up operation required on UnionFS. These differences are part of the reason for the much lower performance overhead of EQFS versus more generalized union file systems.

The use of a disk cleaner to reclaim disk space consumed by elastic files has some similarities to mechanisms for supporting versioned files in file systems such as Cedar [24] and Elephant [22, 23]. Versioning file systems keep track of multiple versions of data automatically. As disk space fills up, versioning file systems reclaim disk space by discarding file versions according to some policy, such as discarding the oldest file versions first. The overall problem of supporting versioned files is different from the problem addressed by the elastic quota system. EQFS can complement versioning systems by differentiating between files that should be versioned (i.e., `/persistent`) and the temporary files for which versioning is not necessary, while rubberd removes non-versioned temporary files from `/elastic`.

Much previous work [1, 5, 11, 17, 19, 30] has been done to develop mechanisms for sharing resource such as processor cycles and network bandwidth such that resources can be utilized fully yet fairly. These resources are elastic in the sense that they can be allocated to a user in such a way that the allocation can be increased or decreased over time based on availability. For example, a processor scheduler enables a group of users to share processor cycles fairly, but allows a user to monopolize the resource when no one else is using it. Elastic quotas can be thought of as a way to make disk space an elastic resource as well. The ability to use disk space elastically opens up new opportunities for applying elastic resource management ideas such as proportional sharing [5, 11, 17, 30] to disk space, a previously unexplored area.

7 Conclusions and Future Work

We have introduced elastic quotas, a novel disk space management technique that brings elasticity to file systems. Elastic quotas provide traditional persistent file semantics for critical data while providing a new elastic model that is easy to use and matches well with the temporary nature of the vast majority of files. Elastic quotas simplify file system management while providing more efficient utilization of storage space in multi-user computing environments. We have demonstrated the viability of elastic quotas by creating an elastic quota system consisting of the Elastic Quota File System (EQFS) and the rubberd file cleaner. EQFS operates as a thin stackable layer that provides elastic quotas with existing file systems without any modifications to those systems. Rubberd reclaims disk space as needed in a manner that is consistent with user preferences.

We have implemented an elastic quota system in Solaris and measured its performance, finding it to be a low-overhead solution to temporary storage management. Our results show that using EQFS adds very little overhead to existing file systems, with performance on a large parallel compilation within one percent of native file system performance. Our results also show that rubberd storage reclamation on a production file system workload is fast, taking just a couple minutes to clean several gigabytes of elastic files. More importantly, rubberd provides the convenience of automatic file cleaning, relieving users of the burden of this task.

Our experimental results have encouraged us to deploy elastic quotas on a production system to further explore the ways in which users will take advantage of reclaimable space and flexible storage limits. Although we have currently focused on file removal policies for storage reclamation of elastic files, we plan to continue to investigate the benefits of combining file compression and hierarchical storage management techniques with elastic quotas. We hope that elastic quotas will provide a useful foundation for future work in exploring elastic resource management in file systems.

References

[1] G. Banga, P. Druschel, and J. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 45–58, Berkeley, CA, February 1999. USENIX.

[2] J. L. Bertoni. Understanding Solaris Filesystems and Paging. Technical Report TR-98-55, Sun Microsystems Research, November 1998. http://research.sun.com/research/techrep/1998/abstract-55.html.

[3] K. Brown, J. Katcher, R. Walters, and A. Watson. SnapMirror and SnapRestore: Advances in Snapshot Technology. Technical report, Network Appliances, Inc. http://www.netapp.com/tech_library/3043.html.

[4] R. Card, T. Ts'o, and S. Tweedie. Design and implementation of the second extended filesystem. In *Proceedings to the First Dutch International Symposium on Linux*, Seattle, WA, December 1994.

[5] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queuing Algorithm. In *Proceedings of ACM SIGCOMM '89*, pages 1–12, Austin, TX, September 1989.

[6] J. R. Douceur and W. J. Bolosky. Progress-based regulation of low-importance processes. In *Proceedings of 17th ACM Symposium on Operating Systems Principles*, pages 247–60, Kiawah Island Resort, SC, December 1999. ACM Press.

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

[8] T. Gibson. *Long-term Unix File System Activity and the Efficacy of Automatic File Migration*. PhD thesis, Department of Computer Science, University of Maryland Baltimore County, May 1998.

[9] R. G. Guy, J. S. Heidemann, W. Mak, T. W. Page Jr., G. J. Popek, and D. Rothmeier. Implementation of the Ficus replicated file system. In *Proceedings of the Summer USENIX Technical Conference*, pages 63–71, Summer 1990.

[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] G. Henry. The Fair Share Scheduler. *AT&T Bell Laboratories Technical Journal*, 63(8):1845–1857, October 1984.

[12] IBM Tivoli. Achieving cost savings through a true storage management architecture. http://www.tivoli.com/products/documents/white papers/sto_man_whpt.pdf, 2002.

[13] J. Katcher. PostMark: a New Filesystem Benchmark. Technical Report TR3022, Network Appliance. http://www.netapp.com/tech_library/3022.html.

[14] S. R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proceedings of the Summer USENIX Technical Conference*, pages 238–47, Summer 1986.

[15] K. Lindsat. Secure Locate. http://www.geekreview.org/slocate, 2002.

[16] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–97, August 1984.

[17] J. Nieh and M. Lam. The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications. In *Proceedings of the 16th Symposium on Operating Systems Principles*, volume 31(5), pages 184–197, New York, October 1997. ACM Press.

[18] 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.

[19] A. Parekh and R. Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single-Node Case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, June 1993.

[20] D. Roselli, J. R. Lorch, and T. E. Anderson. A comparison of file system workloads. In *Proceedings of the Annual USENIX Technical Conference*, June 2000.

[21] D. S. H. Rosenthal. Evolving the Vnode interface. In *Proceedings of the Summer USENIX Technical Conference*, pages 107–18, Summer 1990.

[22] D. J. Santry, M. J. Feeley, N. C. Hutchinson, and A. C. Veitch. Elephant: The File System that Never Forgets. In *Proceedings of the IEEE Workshop on Hot Topics in Operating Systems (HOTOS)*, March 1999.

[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. D. Schroeder, D. K. Gifford, and R. M. Needham. A Caching File System for a Programmer's Workstation. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 25–34, December 1985.

[25] 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.

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

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

[28] VERITAS Software. Veritas file server edition performance brief: A postmark 1.11 benchmark comparison. Technical report. http://eval.veritas.com/webfiles/docs/fsedition-postmark.pdf.

[29] W. Vogels. File System Usage in Windows NT 4.0. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 93–109, December 1999.

[30] C. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, September 1995.

[31] 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.

[32] 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.