Versatility and Unix Semantics in Namespace Unification

CHARLES P. WRIGHT, JAY DAVE, PUJA GUPTA, HARIKESAVAN KRISHNAN, DAVID P. QUIGLEY, EREZ ZADOK, and MOHAMMAD NAYYER ZUBAIR

Administrators often prefer to keep related sets of files in different locations or media, as it is easier to maintain them separately. Users, however, prefer to see all files in one location for convenience. One solution that accommodates both needs is virtual namespace unification—providing a merged view of several directories without physically merging them. For example, namespace unification can merge the contents of several CD-ROM images without unpacking them, merge binary directories from different packages, merge views from several file servers, and more. Namespace unification can also enable snapshotting, by marking some data sources read-only and then utilizing copy-on-write for the read-only sources. For example, an OS image may be contained on a read-only CD-ROM image—and user's configuration, data, and programs could be stored in a separate read-write directory. With copy-on-write unification, the user need not be concerned about the two disparate file systems.

It is difficult to maintain Unix semantics while offering a versatile namespace unification system. Past efforts to provide such unification often compromised on the set of features provided or Unix compatibility—resulting in an incomplete solution that users could not use.

We designed and implemented a versatile namespace-unification system called *Unionfs*. Unionfs maintains Unix semantics while offering advanced namespace-unification features: dynamic insertion and removal of namespaces at any point in the merged view, mixing read-only and read-write components, efficient in-kernel duplicate elimination, NFS interoperability, and more. Since releasing our Linux implementation, it has been used by thousands of users and over a dozen Linux distributions, which helped us discover and solve many practical problems.

Categories and Subject Descriptors: D.4.3 [**Operating Systems**]: File Systems Management—*File organization*; D.4.3 [**Operating Systems**]: File Systems Management—*Directory structures*; E.5 [**Data**]: Files—*Organization/structure*; H.3.2 [**Information Storage and Retrieval**]: Information Storage—*File organization*

General Terms: Design, Experimentation, Management

Additional Key Words and Phrases: Namespace Management, Unification, Directory Merging, Stackable File Systems, Snapshotting.

1. INTRODUCTION

For ease of management, different but related sets of files are often located in multiple places. Users, however, find it inconvenient to access such split files: users prefer to see everything in one place. One proposed solution is to virtually merge—or unify—the views of different directories (recursively) such that they appear to be one tree; this is done without physically merging the disparate directories. Such *namespace unification* has the benefit of allowing the files to remain physically separate, but appear as if they reside in one location. The collection of merged directories is called a *union*, and each physical directory is called a *branch*. When creating the union, each branch is assigned a precedence and access permissions (i.e., read-only or read-write). At any point in time new branches may be inserted, or existing branches may be removed from the union. There are many possible uses for namespace unification. We explore six of them next.

—Modern computing systems contain numerous files that are part of many software distributions. There are often several reasons to spread those files among different locations. For example, a wide variety of packages may be installed in separate sub-trees under `/opt`. Rather than requiring users to include large numbers of directories in their PATH environment variable, the administrator can simply unify the various components in `/opt`.

—Another example of unification is merging the contents of several file servers. In a large organization, a user may have files on a variety of servers (e.g., their personal files on one, and each project could have its own server). However, on workstations it should appear as if all the user's files are in a common location—regardless of which server the files are really on. A standard mount cannot be used because mounting two file systems in the same place would hide the files that were mounted first. A namespace-unification file system can simply unify the various mount points into a common directory. Lastly, file servers may come online or go offline at any time. Therefore, a namespace-unification file system must be able to add and remove branches dynamically.

—Large software collections are often distributed as split CD-ROM images because of the media's size limitations. However, users often want to download a single package from the distribution. To meet both needs, mirror sites usually have both the ISO images and the individual packages. This wastes the disk space and bandwidth because the same data is stored on disk and downloaded twice. For example, on our group's FTP server, we keep physical copies of the Fedora distribution only as ISO images; we loopback-mount the ISO images, and then we unify their contents to provide direct access to the RPMs and SRPMs.

—It is becoming more and more common for OS software to be distributed on *live CDs*. Live CDs are bootable CD-ROMs (or DVDs) that contain an entire operating system in compressed format. Live CDs require no installation process, which simplifies administration: upgrading a machine is as simple as replacing the old CD with a new CD. The major drawbacks of live CDs are that the file system is read-only, and configuration changes are lost after a reboot. Unionfs can help solve both of these problems. To support the illusion of a writable CD-ROM, a high-priority RAM disk and the low-priority CD image can be unified. When a user tries to write to a file on the CD-ROM, Unionfs transparently copies the file to the higher-priority RAM disk. If the RAM disk is replaced with a persistent read-write file system, such as a USB flash drive (or hard disk), then configuration changes can be preserved across reboots.

—Snapshotting is a useful tool for system administrators, who need to know what changes are made to the system while installing new software [McKusick and Ganger 1999; Hitz et al. 1994]. If the installation failed, the software does not work as advertised, or is not needed, then the administrator often wants to revert to a previous good system state. Unification can provide a file system snapshot that carries out the installation of new software in a separate directory. Snapshotting is accomplished by adding an empty high-priority branch, and then marking the existing data read-only. If any changes are made to the read-only data, Unionfs transparently makes the changes on the new high-priority branch. The system administrators can then examine the exact changes made to the system and then easily keep or remove them.

—Similarly, when an Intrusion Detection System (IDS) detects a possible intrusion, it should prevent further changes to the file system, while legitimate users should be able to perform their tasks. Furthermore, false alarms can be very common, so the system should take some steps to protect itself (by carefully tracking the changes made by that process), but not outright kill the suspicious process. If an intrusion is suspected, then the IDS can create snapshots that the system administrator can examine afterward. If the suspected intrusion turns out to be a false positive, the changes can be merged into the file system. In addition to file system snapshots, Unionfs also supports *sandboxing*. Sandboxes essentially create a namespace fork at the time a snapshot is taken. Processes are divided into two (or more) classes: *bad* processes, which the IDS suspects are intrusions; and all other processes are *good*. The good processes write to one snapshot, and the bad processes write to another. The good processes see only the existing data, and changes made by other good processes. Likewise, the bad processes see only the existing data and changes made by bad procceses. The result is that bad data is never exposed to good processes.

Although the concept of virtual namespace unification appears simple, it is difficult to design and implement it in a manner that fully complies with expected Unix semantics. The various problems include handling files with identical names in the merged directory, maintaining consistency while deleting files that may exist in multiple directories, handling a mix of read-only and read-write directories, and more. Given these difficulties, it is not surprising that none of the past implementations solved all problems satisfactorily.

We have designed and built *Unionfs*, a namespace-unification file system that addresses *all* of the known complexities of maintaining Unix semantics without compromising versatility and the features offered. We support two file deletion modes that address even partial failures. We allow efficient insertion and deletion of arbitrary read-only or read-write directories into the union. Unionfs includes efficient in-kernel handling of files with identical names; a careful design that minimizes data movement across branches; several modes for permission inheritance; and support for snapshots and sandboxing. We have publicly released Unionfs, and it has been downloaded by thousands of users and is in use by over a dozen other projects. This wide dissemination has helped us to discover important design details that previous implementations have not considered. For example, maintaining a persistent and unique inode mapping, and resuming directory reads over NFS are both crucial for unifying distributed namespaces. We have also developed efficient yet easy-to-use user-space management utilities, which are essential for a production-quality system.

The rest of this article is organized as follows. Section 2 describes our overall design and Section 3 elaborates on the details of each Unionfs operation. Section 4 surveys related work. Section 5 compares the features of Unionfs with those offered by previous systems. We show that Unionfs provides new features and also useful features from past work. Section 6 analyzes Unionfs's performance. We show a small overhead of 0.2–1.5% for normal workloads, and acceptable performance even under demanding workloads. We conclude in Section 7 and suggest future directions.

2. DESIGN

Although the concept of virtual namespace unification appears simple, it is difficult to design and implement it in a manner that fully complies with expected Unix semantics. There are four key problems when implementing a unification file system.

The first problem is that two or more unified directories can contain files with the same name. If such directories are unified, then duplicate names must not be returned to user-space or it could break many programs. The solution is to record all names seen in a directory and skip over duplicate names. However, that solution can consume memory and CPU resources for what is normally a simpler and stateless directory-reading operation. Moreover, just because two files may have the same name, does not mean they have the same data or attributes. Unix files have only one data stream, one set of permissions, and one owner; but in a unified view, two files with the same name could have different data, permissions, or even owners. Even with duplicate name elimination, the question still remains which attributes should be used. The solution to this problem often involves defining a priority ordering of the individual directories being unified. When several files have the same name, files from the directory with a higher priority take precedence.

The second problem relates to file deletion. Since files with the same name could appear in the directories being merged, it is not enough to delete only one instance of the file because that could expose the other files with the same name, resulting in confusion as a successfully deleted file still appears to exist. Two solutions to this problem are often proposed. (1) Try to delete all instances. However, this multi-deletion operation is difficult to achieve atomically. Moreover, some instances may not be deletable because they could reside in read-only directories. (2) Rather than deleting the files, insert a *whiteout*, a special high-priority entry that marks the file as deleted. File system code that sees a whiteout entry for file F behaves as if F does not exist.

The third problem involves mixing read-only and read-write directories in the union. When users want to modify a file that resides in a read-only directory, the file must be copied to a higher-priority directory and modified there, an act called a *copyup*. Copyups only solve part of the problem of mixing read-write and read-only directories in the union, because they address data and not meta-data. Past unification file systems enforced a simpler model: all directories except the highest-priority one are read-only. Forcing all but the highest-priority branch to be read-only tends to clutter the highest-priority directory with copied-up entries for all of the remaining directories. Over time, the highest-priority directory becomes a de-facto merged copy of the remaining directories' contents, defeating the physical separation goal of namespace unification.

The fourth problem involves name cache coherency. For a union file system to be useful, it should allow additions to and deletions from the set of unified directories. Such dynamic insertions and deletions in an active, in-use namespace can result in incoherency of the directory name-lookup cache. One solution to this problem is to simply restrict insertions into the namespace to a new highest-priority directory.

We designed Unionfs to address these problems while supporting n underlying *branches* or directories with the following three goals:

—**No artificial constraints on branches** To allow Unionfs to be used in as many applications as possible, we do not impose any unnecessary constraints on the order or attributes of branches. We allow a mix of multiple read-write and read-only branches. Any branch can be on any file system type. We support dynamic insertion and removal of branches in any order. The only restriction we impose was that in a read-write union, the highest-priority branch must be read-write. This restriction is required because a highest-priority read-only branch cannot be overridden by another branch.

—**Maintain Unix Semantics** One of our primary goals was to maintain Unix semantics, so that existing applications continue to work. A Unionfs operation can include operations across several branches, which should succeed or fail as a unit. Returning partial errors can confuse applications and also leave the system in an inconsistent state. Through a careful ordering of operations, a Unionfs operation succeeds or fails as a unit.

—**Scalability** We wanted Unionfs to have a minimal overhead even though it consists of multiple branches across different file systems. Therefore, we only look up a file in the highest priority branch unless we need to modify the file in other branches; once found, we use the OS caches to save the `lookup` results. We delay the creation of directories that are required for copyup. We leave files in the branch in which they already exist and avoid copying data across branches until required.

Next, we describe the following three general aspects of Unionfs's design: stacking VFS operations, error propagation, copyup and parent directory creation, and whiteouts. We provide operational details of Unionfs in Section 3.

Stacking VFS Operations. Stackable file systems are a technique to layer new functionality on existing file systems [Zadok and Nieh 2000]. A stackable file system is called by the VFS like other file systems, but in turn calls another file system instead of performing operations on a backing store such as a disk or an NFS server. Before calling the lower-level file system, stackable file systems can modify the operation, for example encrypting data before it is written to disk.

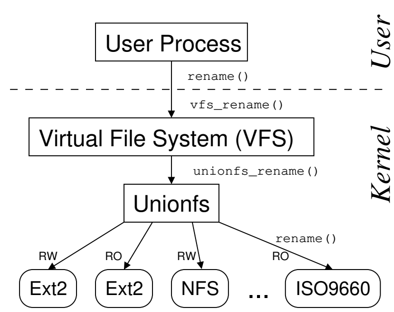

Fig. 1. Unionfs: A stackable fan-out file system can access N different branches directly. In this example, the branches are a read-write Ext2 file system, a read-only Ext2 file system, a read-write NFS file system, and a read-only ISO9660 file system.

Unionfs is a stackable file system that operates on multiple underlying file systems. It has an n-way fan-out architecture as shown in Figure 1 [Rosenthal 1990; Heidemann and Popek 1994]. The benefit of this approach is that Unionfs has direct access to all underlying directories or branches, in any order. A fan-out structure improves performance and also makes our code base more applicable to other fan-out file systems like replication, load balancing, etc.

Unionfs merges the contents of several underlying directories. In Unionfs, each branch is assigned a unique precedence so that the view of the union presented to the user is always unambiguous. An object to the *left* has a higher precedence than an object to the *right*. The *leftmost* object has the highest precedence.

For regular files, devices, and symlinks, Unionfs performs operations only on the leftmost object. This is because applications expect only a single stream of data when accessing a file. For directories, Unionfs combines the files from each directory and performs operations on each directory. Operations are ordered from left to right, which preserves the branch precedence. A delete operation in Unionfs may be performed on multiple branches. Unionfs starts delete operations in reverse order, from right to left, so that if any operation fails, then Unionfs does not modify the leftmost entry until all lower-priority operations have succeeded. This preserves Unix semantics even if the operation fails in some branches, because the user-level view remains unchanged. For all operations in Unionfs, we take advantage of existing VFS locking to ensure atomicity.

Error Propagation. Unionfs may operate on one or more branches, so the success or the failure of any operation depends on the successes and the failures in multiple branches. If part of an operation fails, then Unionfs gives the operation another chance to succeed. For example, if a user attempts to create a file and gets an error (e.g., a read-only file system error), then Unionfs attempts to create the file to the left.

Copyup and Parent Directory Creation. Unionfs attempts to leave a file on the branch where it initially existed. However, Unionfs transparently supports a mix of read-only and read-write branches. Instead of returning an error from a write operation on a read-only branch, Unionfs moves the failed operation to the left by copying the file to a higher priority branch, a *copyup*.

To copy up a file, Unionfs may have to create an entire directory structure (e.g., to create the file `a/b/c/d`, it creates `a`, `b`, and `c` first). Unlike BSD Union Mounts, which clutter the highest-priority branch by creating the directory structure on every lookup [Pendry and McKusick 1995], Unionfs creates directories only when they are required.

An important factor for security is the permissions of the copied-up files and the intermediate directories created. Unionfs provides three modes for choosing permissions: COPYUP_OWNER sets the mode and the owner to that of the original file; COPYUP_CONST sets the mode and the owner to fixed ones specified at mount time; and COPYUP_CURRENT sets the mode and the owner based on the current umask and owner of the process that initiated the copyup. These policies fulfill the requirements of different sites. COPYUP_OWNER provides the security of the original file and preserves Unix semantics, but charges the owner's quota. COPYUP_CONST allows administrators to control the new owner and mode of copied up files. COPYUP_CURRENT is useful when the current user should have full permissions on the copied up files, and affects the current user's quota.

The Unix permission model allows situations where a user can change a file, but cannot change the directory in which it resides. However, if said file exists on a read-only branch, then it must be copied up, which involves creating a new file (i.e., modifying the directory). To solve this problem, we change the current process's file-system user ID to the owner of the directory before creating the file, and restore it to the original afterward. We have previously used this technique to implement ad-hoc groups in our stackable encryption file system [Wright et al. 2003]. Because only the copy-up operation is performed with the elevated permissions and Unionfs already checked the permissions on the copied-up file, this method does not introduce any security holes.

Whiteouts. Whiteouts are used to hide files or directories in lower-priority branches. Unionfs creates whiteouts as zero length files, named `.wh.`F where F is the name of the file or directory to be hidden. This uses an inode, but no data blocks. The whiteouts are created in the current branch or in a higher priority branch of the current branch. One or more whiteouts of a file can exist in a lower priority branch, but a file and its whiteout cannot exist in the same branch. Depending on a mount-time option, Unionfs creates whiteouts in unlink-like operations as discussed in Sections 3.5 and 3.6. Whiteouts for files are created atomically by renaming F to `.wh.`F, and then truncating `.wh.`F. For other types of objects, `.wh.`F is created, and then the original object is removed.

3. OPERATIONAL DETAILS

In this section, we describe individual Unionfs operations. First, we describe Linux VFS objects in Section 3.1. We describe lookup and open in Section 3.2, directory reading in Section 3.3, creating new objects in Section 3.4, deleting objects in Section 3.5, rename in Section 3.6, dynamic branch insertion and deletion in Section 3.7, user-space utilities in Section 3.8, and sandboxing using split-view caches in Section 3.9

3.1 VFS Objects

We discuss Unionfs using Linux terminology. Unionfs defines operations for four VFS objects: the *superblock*, the *inode*, the *file*, and the *directory entry*. The superblock stores information about the entire file system, such as used space, free space, and the location of other objects (e.g., inode objects). The superblock operations include unmounting a file system and deleting an inode. The inode object is a physical instance of a file that stores the file data and attributes such as owner, permissions, and size. Operations that manipulate the file system namespace, like `create`, `unlink`, and `rename`, are inode operations. The file object represents an open instance of a file. Each user-space file descriptor maps to a file object. The file operations primarily deal with opening, reading, and writing a file. The directory entry, also called a *dentry*, represents a cached name for an inode in memory. On `lookup`, a dentry object is created for every component in the path. If hard links exist for a file, then an inode may have multiple names, and hence multiple dentries. The kernel maintains a *dentry cache* (dcache) which in turn controls the *inode cache*. The dentry operations include revalidating dentries, comparing names, and hashing names.

3.2 Lookup and Open

Lookup is one of the most important inode operations. It takes a directory inode and a dentry within that directory as arguments, and finds the inode for that dentry. If the name is not found, it returns a *negative* dentry—a dentry that does not have any associated inode. Only the leftmost file is used for read-only meta-data operations or operations that only modify data. Unionfs proceeds from left to right in the branches where the parent directory exists. If the leftmost entry that is found is a file, then Unionfs terminates the search, preventing unnecessary lookups in branches to the right. We call this early termination a *lazy lookup*. In operations that operate on all underlying files, such as `unlink`, Unionfs calls the lower-level `lookup` method on each branch to the right of the leftmost file to populate the branches that were skipped.

Unionfs provides a unified view of directories in all branches. Therefore if the leftmost entry is a directory, Unionfs looks it up in all branches. If there is no instance of the file or the directory that Unionfs is looking up, it returns a negative dentry that points to the leftmost parent dentry.

In each branch, Unionfs also looks up the whiteout entry with the name of the object it is looking for. If it finds a whiteout, it stops the `lookup` operation. If Unionfs found only negative dentries before the whiteout dentry, then `lookup` returns a negative dentry for the file or the directory. If Unionfs found any dentries with corresponding inodes (i.e., objects that exist), then it returns only those entries.

Inode Numbers. If an object is found during `lookup`, then the `read_inode` method is called to instantiate the corresponding in-memory inode object. A key question that `lookup` must answer is what inode number to assign to the newly instantiated Unionfs inode. There are two key requirements for inode numbers: (1) the inode number must be unique within the Unionfs file system, and (2) the inode number should be persistent. The first requirement, *uniqueness* stems from the VFS using the inode number as a key to read the on-disk inode and user-level applications using the inode to identify files uniquely. User-level applications retrieve files' inode numbers to detect hard links. If two files on the same file system have the same inode number, then applications assume that the files are the same. If a union contains files from two branches on separate file systems, then it is possible for the lower-level inode numbers to "trick" the application into losing one of the files. The second requirement, *persistence*, is motivated by support for NFS mounts and hard link detection. If inode numbers change, then the kernel NFS server returns a "Stale NFS file handle" to NFS clients. User-level applications like `diff` compare inode numbers to check if two files are the same, and if two files have different inode numbers, then `diff` must compare the whole file, even if there are no differences.

To solve these two problems, Unionfs maintains a persistent map of unique inode numbers. When a file is first seen by Unionfs, it is assigned a unique Unionfs inode number and entered into two maps: a forward map and a reverse map. The forward map translates the Unionfs inode number to the lower-level inode number and the corresponding branch. The Unionfs inode number is also stored in a reverse map for each lower-level file system. The reverse map is used by the `lookup` and `readdir` operations. When a file or directory is subsequently accessed, Unionfs uses the lower-level inode as a key in the reverse map to retrieve the correct Unionfs inode number. This prevents utilities like `tar` from mistakenly identifying two distinct files as the same file, or utilities like `diff` from identifying two different files as the same file. NFS file handles on Linux identify objects using inode numbers. Therefore, when Unionfs is presented with an inode number, it must be able to identify which object it belongs to. The forward map can be used to retrieve the corresponding lower-level object for NFS requests. We have not yet implemented this feature for NFS file handles, but we describe a design for it in Section 7.1.

The forward and reverse maps have a low-space overhead. The forward map has a constant overhead of 4,377 bytes, and an overhead of 9 bytes per file or directory in the Union. If we assume the average file size is 52,982 bytes (this is the average size on our groups file server's `/home` directory, which has over five million files belonging to 82 different users), then the space overhead is 0.017% of the used disk space. The reverse maps each have a fixed overhead of 64 bytes, and an additional 8 bytes per inode allocated on the lower-level file system. Most FFS-like file systems allocate a fixed number of inodes per-block. For example, Ext2 allocates one inode per 4096-byte block by default. This means that the reverse maps have a space overhead of just under 0.2% of the file system size. Overall, the space overhead for the forward and reverse map is negligible at around 0.212%.

Open. When opening a file, Unionfs opens the lower-level non-negative dentries that are returned by the lookup operation. Unionfs gives precedence to the leftmost file, so it opens only the leftmost file. However, for directories, Unionfs opens all directories in the underlying branches, in preparation for `readdir` as described in Section 3.3. If the file is in a read-only branch and is being opened for writing, then Unionfs copies up the file and opens the newly copied-up file.

3.3 Readdir

The `readdir` operation returns directory entries in an open directory. A directory in Unionfs can contain multiple directories from different branches, and therefore a `readdir` operation in Unionfs is composed of multiple `readdir` operations.

Priority is given to the leftmost file or directory. Therefore, Unionfs's `readdir` starts from the leftmost branch. Unionfs eliminates duplicate instances of files or directories with the same name in the kernel. Any whiteout entry to the left hides the file or the directory to the right. To eliminate duplicates, Unionfs records the names of files, directories, and whiteouts that have already been returned in a hash table that is bound to the open instance of the directory (i.e., in the kernel `file` structure). Unionfs does not return names that have already been recorded.

Previous unification file systems either did not perform duplicate elimination, or performed it in user level (e.g., in the C library). Duplicate elimination is important because most user-space applications cannot properly handle duplicate names (e.g., a utility like `cp` might copy the data twice). User-level duplicate elimination is not sufficient because Unionfs could not be exported over NFS. Moreover, on Linux there are many C libraries available, and changing all of them is not practical.

It is essential to find and add entries in the hash table efficiently. We use fixed-sized hash elements that are allocated from a dedicated kernel-cache. To reduce collisions, each directory-reading operation computes an appropriate hash-table size. The first time a directory is read, the size of the hash table is determined heuristically based on the size in bytes of the lower-level directories. As entries are added, a counter is incremented. When the directory-reading operation completes, this value is stored for the next operation. On subsequent directory-reading operations, the hash table is automatically sized based on this value.

On Unix, most processes that read directories open the directory, read each entry sequentially, and then close the directory. In a traditional Unix directory, processes can use the `seek` system call to obtain the current offset in a directory, then close the directory. After re-opening the directory, the process can call `seek` to resume reading from previously returned offset. Unionfs directories are different from most directories in that it is only possible to seek to the current offset (programs such as `ls` require this functionality) or the beginning of the directory.

Offsets have the further requirement that they are unique, and should also be increasing. The underlying file systems may use all 32-bits of the offset and there are several underlying directories, so it is not possible to create a simple one-to-one function between

Unionfs directory offset and the directory offset on a given branch. Unionfs uses an artificial directory offset that consists of a 12-bit unique identifier, and a 20-bit count of returned entries.

Special care must be taken to support directory-reading operations over NFS. The NFS (v2 and v3) protocol is stateless, so there is no concept of an open or closed file. For each individual directory-reading operation, the in-kernel NFS server re-opens the directory, seeks to the last offset, reads a number of entries, records the offset, and closes the directory. To support reading directories over NFS, Unionfs maintains a cache of partially-read directories' state indexed by the directory offset and a unique identifier.

When a directory is closed, if there are unread entries, then the current hash table of names is stored in the in-memory inode structure indexed by the current offset. When a seek operation is performed on the directory, the 12-bit identifier is used to find the appropriate structure. The 20-bit offset is also compared and an error is returned if they do not match. If the partially-read directory state is not used for more than five seconds, then it is expunged. This prevents state from partially-read directories from consuming too much kernel memory, but directory-reading operations cannot be resumed after a five second pause.

3.4 Creating New Objects

A file system creates objects with create, mkdir, symlink, mknod, and link. Although these operations instantiate different object types, their behavior is fundamentally similar.

Unionfs creates a new object using the negative dentry returned by the `lookup` operation. However, a negative dentry may exist because a whiteout is hiding lower-priority files. If there is no whiteout, then Unionfs instantiates the new object. A file and its whiteout cannot exist in the same branch. If Unionfs is creating a file and finds a whiteout, it renames the whiteout to the new file. The rename of the whiteout to the file ensures the atomicity of the operation and avoids any partial failures that could occur during `unlink` and `create` operations.

For mkdir, mknod, and symlink, Unionfs instantiates the new object and then removes the whiteout. To ensure atomicity, the inode of the directory is locked during this procedure. However, if mkdir succeeds, the newly-created directory merges with any directories to the right, which were hidden by the removed whiteout. This would break Unix semantics as a newly created directory is not empty. When a new directory is created after removing a whiteout, Unionfs creates whiteouts in the newly-created directory for all the files and subdirectories to the right. To ensure that the on-disk state is consistent in case of a power or hardware failure, before mounting Unionfs, a *high-level `fsck`* can be run. Any objects that exist along with their whiteout are detected, and can optionally be corrected—just like when a standard `fsck` detects inconsistencies.

3.5 Deleting Objects

Unionfs supports two deletion modes: DELETE_ALL and DELETE_WHITEOUT. We describe each mode with pseudo-code. We use the following notations:

L_X Index of the leftmost branch where X exists
R_X Index of the rightmost branch where X exists
$\bar{X}$ Whiteout entry for X
$X[i]$ lower-level object of X in branch i

Additionally, we omit most error handling to conserve space yet provide the essence of each function.

To create a whiteout, we use the function described by the following pseudo-code:

```
1 create_whiteout(X, i)
2   while (i ≥ 1) {
3     if create X̄ succeeds then return
4     i--
5   }
```

As shown in lines 2–4, Unionfs attempts to create a whiteout starting in branch i. If the creation of $\bar{X}$ fails on i, then Unionfs attempts to create $\bar{X}$ to the left of branch i on branch $i-1$. If the operation fails, then Unionfs continues to attempt the creation of the whiteout, until it succeeds in a branch to the left of branch i.

The following pseudo-code describes `unlink`:

```
1 unionfs_unlink(X)
2   if mode is DELETE_ALL {
3     for i = R_X downto L_X
4       if X[i] exists then unlink(X[i])
5   }
6   if an error occurred or mode is DELETE_WHITEOUT
7     create_whiteout(X, L_X)
```

In the `unlink` operation for DELETE_WHITEOUT mode, Unionfs creates a whiteout $\bar{X}$ using the `create_whiteout` operation.

For the `unlink` operation in DELETE_ALL mode, Unionfs scans from right to left, attempting to `unlink` the file in each branch as shown in the lines 2–5. This behavior is the most direct translation of a delete operation from a single branch file system. The delete operation moves in reverse precedence order, from right to left. This ensures that if any delete operation fails, the user-visible file system does not change. If any error occurred during the deletions, a whiteout is created by calling the `create_whiteout` procedure.

Whiteouts are essential when Unionfs fails to `unlink` a file. Failure to delete even one of the files or directories in the DELETE_ALL mode results in exposing the file name even after a deletion operation. This would contradict Unix semantics, so a whiteout needs to be created in a branch with a higher priority to mask the files that were not successfully deleted.

Deleting directories in Unionfs is similar to unlinking files. Unionfs first checks to see if the directory is empty. If any file exists without a corresponding whiteout, Unionfs returns a "directory not empty" error (ENOTEMPTY). A helper function, called *isempty*, returns true if a directory, D, is empty (i.e., a user would not see any entries except `.` and `..`).

In the DELETE_WHITEOUT mode, Unionfs first checks if the directory is empty. If the directory is empty, then Unionfs creates a whiteout in the leftmost branch where the source exists to hide the directory. Next, Unionfs removes all whiteouts within the leftmost directory and the leftmost directory itself. If the operation fails midway, our `fsck` will detect and repair any errors.

The deletion operation for directories in DELETE_ALL mode is similar to the `unlink` operation in this mode. Unionfs first verifies if the directory is empty. A whiteout entry is created to hide the directory and as a flag for `fsck` in case the machine crashes. Next, Unionfs scans the branches from right to left and attempts to delete the lower-level directory and any whiteouts within it. If all deletions succeed, then the whiteout is removed.

3.6 Rename

Renaming files is one of the most complex operations in any file system. It becomes even more complex in Unionfs, which involves renaming multiple source files to multiple destination files—while still maintaining Unix semantics. Even though a `rename` operation in Unionfs may involve multiple operations like `rename`, `unlink`, `create`, copyup, and whiteout creation, Unionfs provides atomicity and consistency on the whole.

For `rename`, the source S can exist in one or more branches and the destination D can exist in zero or more branches. To maintain Unix semantics, $rename(S, D)$ must have the following two key properties. First, if `rename` succeeds, then S is renamed to D and S does not exist. Second, if `rename` fails, then S remains unchanged; and if D existed before, then D remains unchanged.

In general, `rename` is a combination of a `link` of the source file to the destination file and an `unlink` of the source file. So `rename` has two different behaviors based on the `unlink` mode: DELETE_WHITEOUT and DELETE_ALL (the latter is the default mode).

In the DELETE_WHITEOUT mode, Unionfs only renames the leftmost occurrence of the source and then hides any occurrences to the right with a whiteout. Using the notation of Section 3.5, the procedure is as follows:

```
1 unionfs_rename(S,D) { /* DELETE_WHITEOUT */
2   create whiteout for S
3   rename(S[L_S], D[L_S])
4   for i = L_S − 1 downto L_D
5     unlink(D[i])
6   }
```

In line 2, Unionfs creates a whiteout for the source. This makes it appear as if the source does not exist. In line 3, Unionfs then renames the leftmost source file in its own branch. Next, Unionfs traverses from right to left, starting in the branch that contains the leftmost source and ending in the leftmost branch where the destination exists. If the destination file exists in a branch, then it is removed.

To maintain the two aforementioned key properties of `rename`, we make the assumption that any lower-level `rename` operation performed can be undone, though the overwritten file is lost. If any error occurs, we revert the files that we have renamed. This means that the view that users see does not change, because the leftmost source and destination are preserved. During the Unionfs `rename` operation, the source and destination directories are locked, so users cannot view an inconsistent state. However, if an unclean shutdown occurs, the file system may be in an inconsistent state. Our solution is to create a temporary state file before the `rename` operation and then remove it afterward. Our high-level `fsck` could then detect and repair any errors.

Unionfs also supports `rename` in DELETE_ALL mode. Using the notation of Section 3.5, the procedure for `rename` in DELETE_ALL mode is as follows:

```
union_rename(S,D) {
  for i = R_S downto L_S {
    if (i != L_D && S[i] exists)
```

```
4        rename(S[i], D[i])
5    }
6    for i = L_S − 1 downto L_D + 1
7        unlink(D[i])
8    if (S[L_D] exists)
9      rename(S[L_D], D[L_D]);
10   else if (L_D < L_S)
11     unlink(D[L_D])
12 }
```

Lines 2–5 rename each source file to the destination, moving from right to left. The if statement on line 3 skips the branch that contains the leftmost destination (L_D), because if a subsequent lower-level operation were to fail, it is not possible undo the `rename` in L_D. Lines 6–7 are the second phase: the destination file is removed in branches to the left of the leftmost source file (L_S). This prevents higher-priority destination entries from hiding the new data. Finally, the branch that contains the leftmost destination file is handled (L_D) in lines 8–11. If the source file exists in the same branch as the leftmost destination (L_D), then the lower-level source is renamed to the destination in that branch. If this last operation succeeds, then the Unionfs `rename` operation as a whole succeeds, otherwise the Unionfs `rename` operation fails. If the source did not exist in the branch as the leftmost destination (L_D), and the leftmost destination is to the left of the leftmost source ($L_D < L_S$), then the file is removed to prevent it from hiding the new data. Again, if this last operation succeeds, then the Unionfs operation as a whole succeeds, otherwise the Unionfs operation fails. If the Unionfs operation returns an error, we revert the renamed files to their original name on the lower-level. This preserves the property that `rename` should not change the users' view if it returns an error.

Unionfs handles read-only file system errors differently than other errors. If a read-write operation is attempted in a read-only branch, then Unionfs copies up the source file and attempts to rename it to the destination. To conserve space and provide the essence of our algorithms without unnecessary complication, we elided these checks from the previous examples.

3.7 Dynamic Branch Insertion/Deletion

Unionfs supports dynamic insertion and deletion of branches in any order or in any position. Unionfs's inodes, dentries, superblock, and open files all have generation numbers. Whenever a new branch is added or removed, the superblock's generation number is incremented. To check the freshness of objects, the VFS calls the `revalidate` and `d_revalidate` methods on inodes and dentries, respectively. If an object's generation number does not match the super-block, then the data structures are refreshed from the lower-level file systems and the generation number is updated. Refreshing objects is similar to instantiating a new object, but instead updates the existing Unionfs structures, because other kernel code has references to them. To refresh a dentry or inode we use a special code path in `lookup`, and to refresh an open file object we have code that is similar to open.

In most cases, Unionfs does not permit the removal of an in-use branch (opening a file increments the branch's reference count, and closing the file decrements the count). However, when a process changes its working directory, the VFS does not inform the file system. If a branch is removed, but a process is still using it as its working directory, then a new inode is created with an operations vector filled with functions that return a "stale file handle" error. This is similar to NFS semantics.

The VFS provides methods for ensuring that both cached dentry and inode objects are valid before it uses them. However, file objects have no such revalidation method. This shortcoming is especially acute for stackable file systems because the upper-level file object is very much like a cache of the lower-level file objects. In Unionfs this becomes important when a snapshot is taken. If the file is not revalidated, then writes can continue to affect read-only branches. With file revalidation, Unionfs detects that its branch configuration has changed and updates the file object.

Our current prototype of file-level revalidation is implemented at the entry point of each Unionfs file method to allow Unionfs to operate with an unmodified kernel. However, some simple system calls such as `fstat` read the file structure without first validating its contents. Ideally, the VFS should revalidate file objects so that this functionality is uniformly exposed to all file systems.

3.8 User-Space Utilities

Unionfs provides great flexibility to the user: branches can be added and removed, or their permissions can be changed. It is essential for the system administrator to be able to manage branches easily. For efficiency, all branch management `ioctls` use branch numbers for management (e.g., remove branch n). Users, however, find this type of interface cumbersome. Additionally, the branch configuration of Unionfs can change continuously. For example, snapshots may be taken (thereby adding branches) or merged (thereby removing branches).

Unionfs provides a user-space utility, `unionctl` that queries and manipulates branches. Unionfs exposes its current branch configuration through the `/proc/mounts` file. When `unionctl` is invoked, it reads `/proc/mounts` to identify the union and its branches. Paths specified on the command line are converted into branch numbers using this information. These branch numbers are then passed to the corresponding `ioctls`.

A simple shell script, `snapmerge`, merges several Unionfs snapshots. Merging snapshots is advantageous as it improves performance by limiting the number of active branches and saves disk space because only the most recent version of a file is kept. Finally, Unionfs includes a debugging utility to increase or decrease logging levels, and to force the revalidation of all Unionfs objects.

3.9 Split-View Caches

Normally, the OS maintains a single view of the namespace for all users. This limits new file system functionality that can be made available. For example, in file cloaking, users only see the files that they have permission to access [Spadavecchia and Zadok 2002]. This improves privacy and prevents users from learning information about files they are not entitled to access. To implement this functionality in a UID/GID range-mapping NFS server, caches had to be bypassed. Unionfs can divert any process to an alternative view of the file system. This functionality can be integrated with an IDS to create a sandboxing file system. Using a filter provided by an IDS, Unionfs can direct good processes to one view of the union, and bad processes to another view.

In Linux, each mount point has an associated *vfsmount* structure. This structure points to the superblock that is mounted and its root dentry. It is possible for multiple vfsmounts to point to a single super-block, but each vfsmount points to only one superblock and root. When the VFS is performing a `lookup` operation and comes across a mount point, there is an associated vfsmount structure. The VFS simply dereferences the root dentry pointer, and follows it into the mounted file system.

To implement split-view caches, we modified the generic `super_operations` operations vector to include a new method, *select_super*. Now, when the VFS comes across a mount point, it invokes `select_super` (if it is defined), which returns the appropriate root entry to use for this operation. This simple yet powerful new interface was accomplished with minimal VFS changes: only eight new lines of core kernel code were added.

Internally, Unionfs has to support multiple root dentries at once. To do this, we create a parallel Unionfs view that is almost a completely independent file system. The new view has its own super-block, dentries, and inodes. This creates a parallel cache for each of the views. However, Unionfs uses the lower-level file systems' data cache, so the actual data pages are not duplicated. This improves performance and eliminates data cache coherency problems. The two views are connected through the super-blocks so that when the original Unionfs view is unmounted, so are the new views.

Our current prototype uses a hard-coded algorithm for `select_super`, though we plan to create an interface for modules to register their own `select_super` algorithms.

4. RELATED WORK

We begin by describing the origins of fan-out file systems. Then, we briefly describe four other representative unification systems. In Section 5 we compare the features offered by each of these systems with Unionfs. We finally describe snapshotting and sandboxing systems.

Fan-out File Systems. Rosenthal defined the concept of a fan-out file system, and suggested possible applications such as caching or fail-over [Rosenthal 1990]. However, Rosenthal only suggested these file systems as possible uses of a versatile fan-out vnode interface, but did not build any fan-out file systems. Additionally, Rosenthal's stacking infrastructure required an overhaul of the VFS. The Ficus Replicated File System is a multi-layer stackable fan-out file system that supports replication [Guy et al. 1990; Heidemann and Popek 1994]. Ficus has two layers, a physical layer that manages a single replica and a logical layer that manages several Ficus physical layer replicas. Ficus uses the existing vnode interface, but overloads certain operations (e.g., looking up a special name is used to signal a file open). Ficus was developed as a stackable layer, but it does not make full use of the naming routines providing by existing file systems. Ficus stores its own directory information within normal files, which adds complexity to Ficus itself.

In Unionfs, we have implemented an n-way fan-out file system for merging the contents of directories using existing VFS interfaces.

Plan 9. Plan 9, developed by Bell Labs, is a general-purpose distributed computing environment that can connect different machines, file servers, and networks [AT&T Bell Laboratories 1995]. Resources in the network are treated as files, each of which belongs to a particular *namespace*. A namespace is a mapping associated with every directory or file name. Plan 9 offers a *binding* service that enables multiple directories to be grouped under a common namespace. This is called a *union directory*. A directory can either be added at the top or at the bottom of the union directory, or it can replace all the existing members in the structure. In case of duplicate file instances, the occurrence closest to the top is chosen for modification from the list of member directories.

3-D File System (3DFS). 3DFS was developed by AT&T Bell Labs, primarily for source code management [Korn and Krell 1990]. It maintains a per-process table that contains directories and a location in the file system that the directories overlay. This technique is called *viewpathing*, and it presents a view of directories stacked over one another. In addition to current directory and parent directory navigation, 3DFS introduces a special file name "`...`" that denotes a third dimension of the file system and allows navigation across the directory stack. 3DFS is implemented as user-level libraries, which often results in poor performance [Zadok and Nieh 2000]; atomicity guarantees also become difficult as directory locking is not possible.

TFS. The Translucent File System (TFS) was released in SunOS 4.1 in 1989 [Hendricks 1990]. It provides a viewpathing solution like 3DFS. However, TFS is an improvement over 3DFS as it better adheres to Unix semantics when deleting a file. TFS transparently creates a whiteout when deleting a file. All directories except the topmost are read-only. During mount time, TFS creates a file called `.tfs_info` in each mounted directory, which keeps sequence information about the next mounted directory and a list of whiteouts in that directory. Whenever the user attempts to modify files in the read-only directories, the file and its parent directories are copied to the topmost directory. TFS is implemented as a user-level NFS server that services all directory operations like `lookup`, `create`, and `unlink`. TFS has a kernel-level component that handles data operations like read and write on individual files. TFS was dropped from later releases of SunOS. Today, the Berkeley Automounter Amd [Pendry et al. 2003] supports a TFS-like mode that unifies directories using a symbolic-link shadow tree (symlinks point to the first occurrence of a duplicate file).

4.4BSD Union Mounts. Union Mounts, implemented on 4.4BSD-Lite [Pendry and McKusick 1995], merge directories and their trees to provide a unified view. This structure, called the *union stack*, permits directories to be dynamically added either to the top or to the bottom of the view. Every `lookup` operation in a lower layer creates a corresponding directory tree in the upper layer called a *shadow directory*. This clutters the upper-layer directory and converts the read-only `lookup` into a read-write operation. A request to modify a file in the lower layers results in copying the file into its corresponding shadow directory. The copied file inherits the permissions of the original file, except that the owner of the file is the user who mounted the file system. A delete operation creates a whiteout to mask all the occurrences of the file in the lower layers. To avoid consumption of inodes, Union Mounts make a special directory entry for a whiteout without allocating an inode. Whiteouts are not allocated inodes in order to save resources, but (ironically) shadow directories are created on every `lookup` operation, consuming inodes unnecessarily.

Snapshotting. There are several commercially and freely available snapshotting systems, such as FFS with SoftUpdates and WAFL [McKusick and Ganger 1999; Hitz et al. 1994; Peterson and Burns 2005]. These systems perform copy-on-write when blocks change. Most of these systems require modifications to existing file systems and the block layer. Clotho is a departure from most snapshotting systems in that it requires only block layer modifications [Flouris and Bilas 2004]. Snapshotting with Unionfs is more flexible and portable than previous systems because it can stack on any existing file system (e.g., Ext2 or NFS). Because Unionfs is stackable, snapshots could also be created per file or per file type.

Sandboxing. Sandboxing is a collection of techniques to isolate one process from the others on a machine. The `chroot` system call restricts the namespace operations of some processes to a subset of the namespace. Jails extend `chroot` to allow partitioning of networking and process control subsystems [Kamp and Watson 2000]. Another form of sandboxing is to monitor system calls, and if they deviate from a policy, prevent them from being executed [Fraser et al. 1999].

5. FEATURE COMPARISON

In this section we present a comparison of our Unionfs with the four most representative comparable systems: Plan 9 union directories, 3DFS, TFS, and BSD Union Mounts. We identified the following eighteen features and metrics of these systems, and we summarized them in Table I:

(1) **Unix semantics: Recursive unification**: 3DFS, TFS, BSD Union Mounts, and Unionfs present a merged view of directories at every level. Plan 9 merges only the top level directories and not their subdirectories.

(2) **Unix semantics: Duplicate elimination level**: 3DFS, TFS, and BSD Union Mounts eliminate duplicate names at the user level, whereas Unionfs eliminates duplicates at the kernel level. Plan 9 union directories do not eliminate duplicate names. Elimination at the kernel level means that Unionfs can be used with multiple C libraries, statically linked applications, and exported to any type of NFS client.

(3) **Unix semantics: Deleting objects**: TFS, BSD Union Mounts, and Unionfs adhere to Unix semantics by ensuring that a successful deletion does not expose objects in lower layers. However, Plan 9 and 3DFS delete the object only in the highest-priority layer, possibly exposing duplicate objects.

(4) **Unix semantics: Permission preservation on copyup**: All file systems except Unionfs do not fully adhere to Unix semantics. BSD Union Mounts make the user who mounted the Union the owner of the copied-up file, whereas in other systems a copied-up file is owned by the current user. Unionfs, by default, preserves the owner on a copyup. Unionfs supports other modes that change ownership on a copyup as described in Section 2.

(5) **Unix semantics: Unique and persistent inode numbers**: Only Unionfs supports unique and persistent inodes during `lookup` and directory reading. This allows applications to identify a file reliably using its device and inode numbers. Persistent inode numbers also make it possible for NFS file handles to survive reboots.

(6) **Multiple writable branches**: Unionfs allows files to be directly modified in any branch. Unionfs attempts to avoid frequent copyups that occur in other systems and avoids shadow directory creation that clutters the highest-priority branch. This improves performance. Plan 9 union directories can have multiple writable components, but Plan 9 does not perform recursive unification, so only the top-level directory supports this feature. Other systems only allow the leftmost branch to be writable.

(7) **Dynamic insertion and removal of the highest priority branch**: All systems except TFS support removal of the highest-priority branch. BSD Union Mounts can only remove branches in the reverse order that they were mounted.

	Feature	Plan 9	3DFS	TFS	4.4BSD	Unionfs
1	Unix semantics: Recursive unification		✔	✔	✔	✔
2	Unix semantics: Duplicate elimination level		User Library	User NFS Server	C Library	Kernel
3	Unix semantics: Deleting objects			✔	✔	✔
4	Unix semantics: Permission preservation on copyup					✔[a]
5	Unix semantics: Unique and persistent inode numbers					✔
6	Multiple writable branches	✔				✔
7	Dynamic insertion & removal of the highest priority branch	✔	✔		✔	✔
8	Dynamic insertion & removal of any branch					✔
9	No file system type restrictions	✔	✔	✔	[b]	✔
10	Creating shadow directories		✔[c]	✔	✔	✔[c]
11	Copyup-on-write		✔	✔	✔	✔[d]
12	Whiteout support		✔[e]	✔	✔	✔
13	Snapshot support					✔
14	Sandbox support					✔
15	Implementation technique	VFS (stack)	User Library	User NFS Server + Kernel helper	Kernel FS (stack)	Kernel FS (fan-out)
16	Operating systems supported	Plan 9	Many[f]	SunOS 4.1	4.4BSD	Linux[g]
17	Total LoC	6,247[h]	16,078	16,613	3,997	11,853
18	Customized functionality					✔

Table I. Feature Comparison. A check mark indicates that the feature is supported, otherwise it is not.
[a] Through a mount-time option, a copied-up file's mode can be that of the original owner, current user, or the file system mounter.
[b] BSD Union Mounts allow only an FFS derivative to be the topmost layer.
[c] Lazy creation of shadow directories.
[d] Unionfs performs copyup only in case of a read-only branch.
[e] 3DFS uses whiteouts only if explicitly specified.
[f] 3DFS supports many architectures: BSD, HP, IBM, Linux, SGI, Solaris, Cygwin, etc.
[g] Unionfs runs on Linux 2.4 and 2.6, but it is based on stackable templates, which are available on three systems: Linux, BSD, and Solaris.
[h] Since Plan 9's union directories are integrated into the VFS, the LoC metric is based on an estimate of all related code in the VFS.

(8) **Dynamic insertion and removal of any branch**: Only Unionfs can dynamically insert or remove a branch anywhere in the union.

(9) **No file system type restrictions**: BSD Union Mounts require the topmost layer to be an FFS derivative which supports on-disk whiteout directory entries. Other systems including Unionfs have no such restriction.

(10) **Creating shadow directories**: 3DFS and TFS create shadow directories on write operations in read-only branches. BSD Union Mounts create shadow directories in the leftmost branch even on `lookup`, to prepare for a possible copyup operation; this, however, clutters the highest-priority branch with unnecessary directories, and turns a read-only operation into a read-write operation. Unionfs creates shadow directories only on write operations and on error conditions such as 'read-only file system" (EROFS).

(11) **Copyup-on-write**: Plan 9 union directories do not support copyup. 3DFS, TFS, BSD Union Mounts, and Unionfs can copy a file from a read-only branch to a higher-priority writable branch.

(12) **Whiteout support**: Plan 9 does not support whiteouts. 3DFS creates whiteouts only if manually specified by the user. BSD Union Mounts, TFS, and Unionfs create whiteouts transparently.

(13) **Snapshot support**: Only Unionfs is suitable for snapshotting, because it supports file-object revalidation, unifies recursively, adheres to Unix deletion semantics, allows dynamic insertion of branches, lazily creates shadow directories, and preserves attributes on copy-up.

(14) **Sandbox support**: Only Unionfs supports sandboxing processes.

(15) **Implementation technique**: Plan 9 union directories are built into the VFS layer. 3DFS is implemented as a user-level library; whereas it requires no kernel changes, applications must be linked with the library to work. Such user-level implementations often suffer from poor performance. TFS is a user-space localhost NFS server that works with standard NFS clients. Running in user-space increases portability, but decreases performance. TFS has a kernel level component for performance, but that reduces its portability. BSD Union Mounts is a kernel-level stackable file system with a linear stack, whereas Unionfs is a kernel-level stackable file system with an n-way fan-out. Stackable file systems have better performance than user-space file systems and are easier to develop than disk-based or network-based file systems [Zadok and Nieh 2000].

(16) **Operating systems supported**: 3DFS comes with a customized C library for several systems: BSD, HPUX, AIX, Linux, IRIX, Solaris, and Cygwin. Plan 9 is an operating system by itself. TFS was supported on SunOS 4.1. BSD Union Mounts are implemented on 4.4BSD and current derivatives (e.g., FreeBSD). Unionfs runs on Linux, but since it is based on stacking templates, it can easily be ported to Solaris and BSD.

(17) **Total LoC**: The number of Lines of Code (LoC) in the file system is a good measure of maintainability, complexity, and the amount of initial effort required to write the system. Plan 9 union directories are built into the VFS; therefore its LoC metric is an approximate estimate based on the most related code in the VFS. 3DFS has a relatively high LoC count because it comes with its own set of C library functions. TFS's LoC metric accounts for both its user-level NFS server and kernel component. The LoC metric for Unionfs (snapshot 061205-0016) and BSD Union Mounts, both implemented in the kernel, is considerably less than the user-level implementations. Unionfs has a larger LoC than BSD Union Mounts because it supports more features. The Unionfs LoC includes 804 lines of user-space management utilities.

(18) **Customized functionality**: Unionfs has a flexible design that provides several modes of operation using mount-time options. For example, Unionfs allows the users to choose the mode and the permissions of the copied-up files, with COPYUP_OWNER, COPYUP_CONST, and COPYUP_CURRENT as described in Section 2. Unionfs also provides two modes for deleting objects: DELETE_ALL and DELETE_WHITEOUT as described in Section 3.5.

6. PERFORMANCE EVALUATION

We evaluate the performance of our system by executing various general-purpose benchmarks and micro-benchmarks. Previous unification file systems are either considerably older or run on different operating systems. Therefore, we do not compare Unionfs's performance with other systems.

We conducted all tests on a 1.7GHz Pentium-IV with 1152MB of RAM. The machine ran Fedora Core 3 with all updates as of May 17, 2005. We used Unionfs snapshot 061205-0016. All experiments were located on a dedicated 250GB Maxtor IDE disk. To overcome the ZCAV effect, the test partition was located on the outer cylinders of the disk and was just large enough to accommodate the test data [Ellard and Seltzer 2003]. We chose Ext2 as the base file system because it is widely used and well-tested. To ensure a cold cache, we unmounted the underlying file system between each iteration of a benchmark. For all tests, we computed the 95% confidence intervals for the mean elapsed, system, and user time using the Student-t distribution. In each case, the half-widths of the intervals for elapsed and system time were less than 5% of the mean. For some experiments, user time was a small component of the overall experiment, in these cases the half-width is either below 5% of the mean, or less than 10ms (the accuracy at which user time is measured).

6.1 Configurations

We used the following two operating modes for our tests:

. **DALL** uses the DELETE_ALL mount-time option that deletes each occurrence of a file or a directory.

. **DWHT** uses the DELETE_WHITEOUT mount time option that creates whiteouts on a call to `rename`, `unlink`, or `rmdir`.

We used the following two data distribution methods:

. **DIST** distributes files and directories evenly across branches with no duplicate files. If two files in the same directory are distributed to different branches, then their parent directory is duplicated.

. **DUP** replicates each file and directory to every branch.

We conducted tests for all combinations of the aforementioned parameters for one, two, four, eight, and sixteen branches. We selected these branch numbers in order to study the performance of the system under different load conditions; one-branch tests were conducted to ensure that Unionfs did not have a high performance overhead compared with similar tests on Ext2; sixteen-branch tests, on the other hand, test the scalability of the system under high workloads; intermediate configurations help examine Unionfs performance on moderate workloads.

6.2 General Purpose Benchmarks

We chose two representative general-purpose workloads: (1) Postmark, an I/O-intensive benchmark [Katcher 1997], and (2) a CPU-intensive compile benchmark, building the OpenSSH package [OpenBSD 2005]. To provide comparable results, we selected the number of Ext2 directories based on the number of underlying Unionfs branches.

Postmark focuses on stressing the file system by performing a series of file system operations such as directory look ups, creations, and deletions on small files. A large number of small files is common in electronic mail and news servers where multiple users are randomly modifying small files. We configured Postmark to create 20,000 files and perform 200,000 transactions; these are commonly recommended parameters [Katcher 1997]. We used 200 subdirectories to prevent linear directory look ups from dominating the results.

The OpenSSH build (version 4.0p1) contains 74,259 lines of code. It performs several hundred small configuration tests, and then it builds 155 object files, one library, eleven binaries, and four scripts. This benchmark contains a fair mix of file system operations, representing a typical performance impact for users.

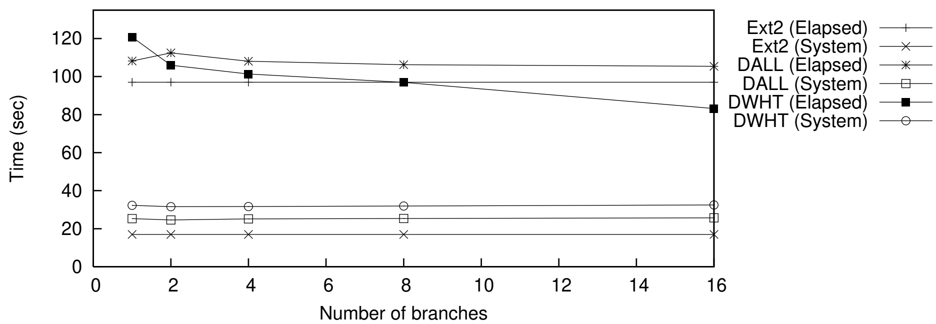

Fig. 2. Postmark: 20,000 files and 200,000 transactions.

Figure 2 shows the elapsed and system time for Postmark in the DWHT and DALL modes. The results for DALL stayed relatively constant as the number of branches increased, demonstrating Unionfs's scalability. The elapsed time overheads for DALL are in the range of 8.5–15.9% above that of Ext2. DWHT, however, has a higher elapsed time overhead of 24.4% for a single branch. This overhead is higher than DALL for two reasons. First, whiteout creation requires two steps: renaming the file and then truncating it. Second, all directory operations consume more time, because look ups take longer as the number of directory entries increases. Interestingly, as the number of branches increases, the overhead of DWHT actually decreases. For two branches, the overhead is 9.2%, and for sixteen branches, the results are actually faster than Ext2 by 14%. This result was unexpected, so we investigated it further. With DWHT the time taken to write data to disk decreases significantly. With sixteen branches, the number of sectors written decreased by 46.9%, the number of individual write operations decreased by 45.5%, and the total amount of time spent writing decreased by 52.9%. We verified that our code's performance characteristics were not anomalous by modifying Postmark to simulate the whiteout deletion mode directly on Ext2.

To better understand this behavior, we profiled Postmark using a tool that we developed called *FSprof* [Joukov et al. 2004]. FSprof instruments file systems to count the number of calls to each operation and their total latency. The most striking differences between the profiles was that the `delete_inode` operation took 72% less time on DWHT than Ext2; and the `writepages` also decreased by 53%. The `delete_inode` operation was significantly faster because with DWHT no inodes are deallocated during the transactions phase of Postmark (they are converted to whiteouts instead). After the transactions phase, Postmark deletes all remaining files and directories. While deleting these directories, DWHT removes any whiteout entries in them. Because all of the whiteouts are deleted at once, the disk blocks containing the inodes only need to be written once because of improved temporal locality.

The behavior of `writepages` is a bit more complicated. The number of calls to `writepages` increased, but the time spent decreased. This means that each individual `writepage` call must have spent less time. The bulk of time in a `writepage` operation is spent positioning the disk head, so this would indicate that the files within DWHT have better locality than in Ext2. To verify this theory we modified Postmark to record the Ext2 inode numbers of files that were created by Ext2 and Unionfs. Using the Ext2 inode number, we can determine the cylinder group that the file was allocated in. We then computed the number of cylinder groups that n consecutive operations spanned. If fewer cylinder groups are used, then better locality is exhibited. Figure 3 compares the number of consecutive operations, with the mean number of cylinder groups accessed for DWHT and Ext2. For 80 or more consecutive operations, DWHT used fewer cylinder groups. For 1,000 operations, Ext2 used 50 cylinder groups, whereas DWHT used only 31. Ext2 attempts to create new files in the same cylinder group as their parent directory. Because Postmark creates many files, they do not all fit within a cylinder group and are spread across the disk. After a file is deleted, however, its place can be taken by a new file. In DWHT, on the other hand, files are not deleted, so the full cylinder group cannot be reused. This forces Ext2 to allocate all of the new files in the same cylinder group, rather than going back to a previously full cylinder group, thereby improving locality.

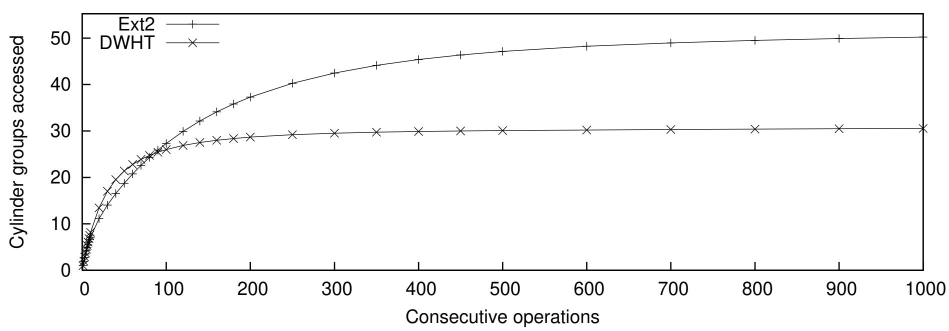

Fig. 3. Cylinder groups accessed by n consecutive operations.

The results for all of our OpenSSH compile benchmarks have overheads ranging from 0.2–1.5% for elapsed time and 1.1–6.2% for system time. Postmark and the OpenSSH compile both show that most users are unlikely to notice a performance degradation when using Unionfs.

Snapshots. We also ran OpenSSH and Postmark while taking snapshots every 60, 30, and 15 seconds. OpenSSH had an elapsed time overhead of 2.4–3.3% over Ext2 for all intervals. On average, 2.0, 4.0, and 7.1 snapshots were taken for intervals of 60, 30, and 15 seconds, respectively. This demonstrates that Unionfs efficiently performs snapshots for user-like workloads. Table II shows the Postmark results. Elapsed time overheads ranged from 34.4–79.8%. Postmark is a more I/O-intensive workload, and therefore each snapshot causes more data to be copied to the highest-priority branch. Additionally, because each snapshot increases the total number of files, directory operations such as `create`, unlink, and lookup take more time. This suggests that merging snapshots periodically would be beneficial.

Interval(s)	Snapshots	Overhead
15	12.4	79.8%
30	5.1	42.9%
60	3	34.4%

Table II. Postmark with snapshots on Unionfs. Elapsed time overhead is compared to Ext2.

6.3 Micro-Benchmarks

Unionfs modifies basic file system operations like lookup, readdir, unlink, and rmdir. We conducted the following three micro-benchmarks on Unionfs to evaluate the overhead of these operations:

—**STAT** evaluates the overhead of lookup by running stat on each file and directory.

—**READDIR** reads all the directories using readdir.

—**UNLINK** evaluates the overhead of the unlink and rmdir operations by unlinking each file in the system (if unlink returns EISDIR, then we use rmdir).

For all of the micro-benchmarks, we used a pre-computed list of files and directories. This avoids using readdir and stat to determine what files or directories to operate on.

We used 100 copies of the OpenSSH 4.0p1 distribution as our data set. The data set had a total of 1,300 directories with 40,000 files that took 472MB of disk space. For the distributed data set, the number of files remained constant, but there were duplicated directories. For the duplicated set, each branch had a copy of all files and directories.

STAT. Figure 4 shows the benchmark results for STAT. For a single branch, Unionfs has an overhead of 33.7% over Ext2. This is because Unionfs must look up both the file and its whiteout. A Unionfs lookup operation with a DIST distribution scans all the branches from left to right until it finds the file. So, there is an expected linear increase in the elapsed time by a factor of 2.3 for two branches over a single branch, 4.0 for four branches, 6.8 for eight branches, and 10.8 for sixteen branches. On the other hand, for a DUP data distribution, the overhead for two branches over a single branch is a factor of 2.1, for four branches the overhead is a factor of 7.5, for eight branches the factor of 17.3 times, and for sixteen branches a factor of 32.7. This overhead is mainly caused by look ups on directories. Indeed, if a single directory of 10,000 files is used instead of the OpenSSH data set, then the overhead is 1.2% for two branches and it increases to only 23.9% for sixteen branches, because the lookup procedure terminates after the first branch for files.

Most of this overhead is I/O. With a cold cache, the benchmark took from 5.9–213 seconds. With a warm cache, the benchmark took 0.2 seconds for Ext2, and 0.3–1.6 seconds for Unionfs. We believe that the warm-cache results are closer to most user workloads, because most files are accessed multiple times [Roselli et al. 2000].

READDIR. Figure 5 shows the benchmark results for READDIR. For a single branch, Unionfs has an overhead of 38.7% over Ext2. This overhead is due to the additional state that must be maintained, and additional layers of function calls. A Unionfs readdir with a DIST distribution scans all the branches from left to right, listing the contents of

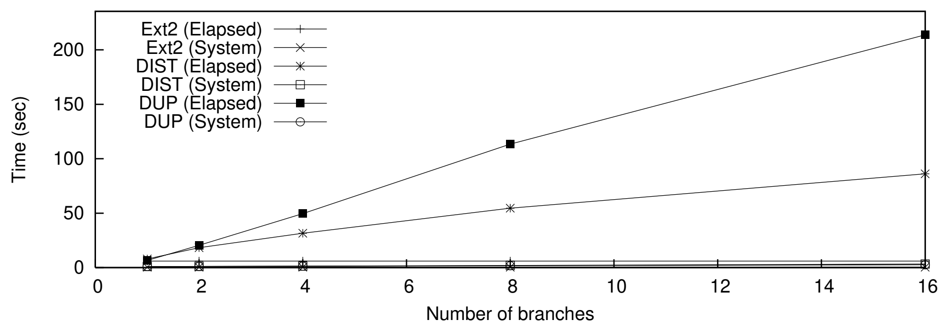

Fig. 4. STAT results.

the directories. Because each underlying directory contains fewer files, it examines the same number of entries as Ext2 (if you ignore duplicated directories), but the disk head must still seek to read each of the n small directories. So, again there is an expected linear increase in the elapsed time by a factor of 2.5 for two branches over a single branch, 4.3 for four branches, 6.5 for eight branches, and 10.4 for sixteen branches. Similary, for a DUP data distribution, Unionfs must physically read n directories from the disk for an n branch configuration. Additionally, the directories will be as large as before and duplicate elimination must be performed. For a single branch, the overhead is 20.8%. The overhead for two branches over a single branch is a factor of 3.7, 7.4 for four branches, 16.1 for eight branches, and 29.5 for sixteen branches.

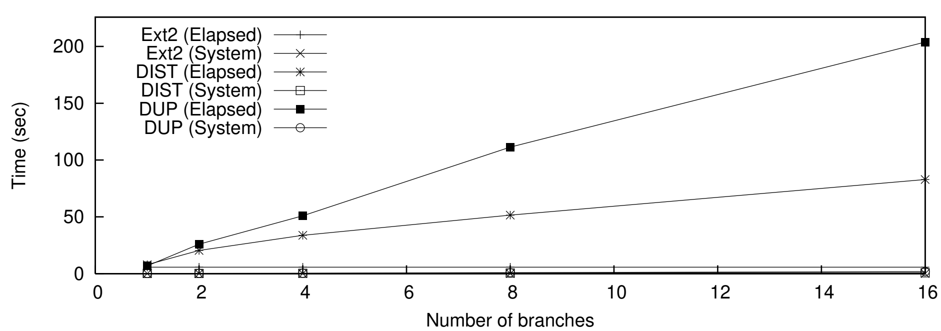

Fig. 5. READDIR results.

Most of this overhead is I/O. Indeed, to perform the test on sixteen duplicate copies of the data set on Ext2 took 14.0 times as long as to read a single copy. For a sixteen-branch Unionfs configuration with duplicated data, this translates into an overhead of 124.6% over Ext2 with sixteen duplicated data sets. With a cold cache, the benchmark took 5.7–203 seconds. With a warm cache, the benchmark completed in 0.06 seconds for Ext2, and 0.10–1.45 seconds for Unionfs. We believe that the warm cache performance is closer to user workloads, because directories are usually accessed multiple times.

UNLINK *Benchmark.* Figure 6 shows the benchmark results for our UNLINK micro-benchmark. With a DALL configuration on a distributed data set, the mean elapsed time increases by 5.0%, from 12.51 seconds for Ext2 to 13.15 seconds for a single branch Unionfs. This increase is due to a system time increase from 0.06 seconds to 3.93 seconds. When additional branches are added, the elapsed time increases for three reasons. First, `lookup` operations must be performed in branches to the right of the file's first occurrence (which requires reading those entire directories). Second, before removing a directory, Unionfs must read the directory in each branch to ensure that it is empty. Third, directories must be deleted in multiple branches. A union with two branches had a 74% elapsed time overhead when compared to a single branch; four branches were slower by a factor of 2.5; eight branches were slower by a factor of 4.8, and sixteen branches were slower by a factor of 7.3.

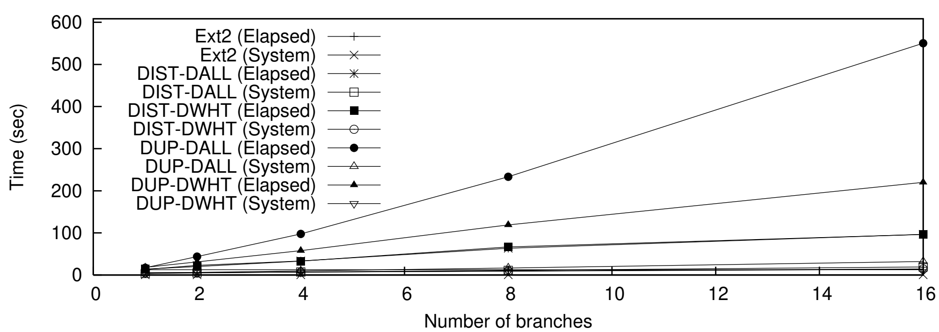

Fig. 6. UNLINK results.

With a DWHT configuration, the mean elapsed time increases by 12.32%, from 12.51 seconds to 14.06 seconds; again primarily because of an increase in system time. The reason that DWHT took more time than DALL for this configuration is that it is implemented as a `rename`, which consists of two distinct directory operations (adding an entry for the new name and removing the old name). Because DALL only performs one directory operation, it uses less time. For additional branches, the elapsed time increases at a slower rate than for DALL. This is because directories do not have to be deleted in all branches, and `lookup` does not need to be called in lower-priority branches. Overheads over a single branch are 43% for two branches, a factor of 2.3 for four branches, a factor of 4.8 for eight branches, and a factor of 6.9 for sixteen branches. The overhead increases as the number of branches increases because `readdir` is called to verify that the directory is logically empty (i.e., every lower-level entry has a corresponding higher-priority whiteout).

For a DUP distribution with DALL, the overhead for additional branches is greater than for the DIST data distribution. The increase is caused by three factors. First, to determine if a directory is empty, more entries need to be read. Second, before removing a file, Unionfs must perform `lookup` operations in all of branches except the leftmost. Third, to delete a single file, `unlink` must be performed in each branch. For two branches, the elapsed time increases by a factor of 2.5. Most of this overhead, 91%, was caused by additional I/O. For four branches the overhead is a factor of 5.7, for eight branches the overhead is a factor of 13.5, and for sixteen branches, the overhead is a factor of 31.9. The reason for such a high overhead for sixteen branches is that Unionfs must actually perform many operations, including more I/O-bound operations. We constructed a similar benchmark for

Ext2 that removes two, four, eight, and sixteen copies of our data set. The overhead of Unionfs compared to this Ext2 test was 47–83% for 2–16 branches.

With a DWHT configuration, the mean elapsed time increases by 45%, from 12.51 seconds to 18.21 seconds. The overhead over a single branch is 70.9% for two branches, a factor of 3.2 for four branches, a factor of 6.5 for eight branches, and a factor of 12.1 for sixteen branches. The overhead is less than for DALL because `lookup` and directory operations are not required in lower-priority branches. As the number of branches increases, overhead increases because more directory-reading operations need to be performed to verify that directories are logically empty. When compared to removing the files on Ext2, the overhead is 22.7% for two branches. For four or more branches, DWHT is faster than deleting the files on Ext2, because fewer namespace operations are required. The improvement is 12.6% for four branches, 15.2% for eight branches, and 26.8% for sixteen branches.

The aforementioned benchmarks helped us evaluate the performance of all features that Unionfs provides. Our general-benchmarks show that Unionfs has small user-visible overheads, even for an I/O-intensive benchmark like Postmark. Our micro-benchmarks bring out the worst case Unionfs operations. We show that Unionfs has acceptable overheads, and for particularly expensive operations illustrate that performing the same underlying operations on multiple copies of the data with a plain Ext2 file system is also expensive.

7. CONCLUSIONS

We have designed, implemented, and released Unionfs, a namespace unification file system that is both versatile and adheres to Unix semantics. Our performance evaluation shows that Unionfs has a small overhead for typical user-like workloads, and our micro-benchmarks show that Unionfs has acceptable worst-case performance.

Unionfs is the first implementation of an n-way stackable fan-out unification file system. All underlying branches are directly accessed by Unionfs which allows it to be more intelligent and efficient. Unionfs supports a mix of read-only and read-write branches, features not previously supported on any unification file system. Unionfs also supports the dynamic addition and deletion of any branch of any precedence, whereas previous systems only allowed the highest or lowest precedence branch to be added or removed. Unionfs's flexibility and VFS enhancements allow it to be used for new applications, such as snaphsotting and sandboxing, where namespace unification systems have not previously been applied.

Unionfs has been directly downloaded by thousands users in the last six months, and is widely used in eighteen different Linux distributions, and because of this we have been able to identify and solve previously unidentified problems. For example, Unionfs has efficient in-kernel duplicate elimination with support for NFS. Unionfs also has support for unique and persistent inode numbers with a space overhead less than one quarter of a percent.

Even though operations may fail on any one of the underlying branches, Unionfs maintains Unix semantics. For deletion operations, Unionfs operates from low precedence to high precedence branches in order to leave the user-level view unmodified until the operation is guaranteed to succeed. We carefully ordered operations to return success or failure to the user atomically, and leave the file system in a consistent state.

7.1 Future Work

For operations that manipulate the namespace, Unionfs currently uses $O(\Sigma_{i=1}^{n} t_i)$ algorithms, where t_i is time it takes to complete the operation on branch i and n is the number of branches, because operations are performed serially on each branch. With only a handful of branches, this performs reasonably well (as demonstrated in Section 6), but as the number of branches increases performance begins to suffer. This is especially acute when branches are added automatically (e.g., in a snapshotting system). We plan to create a parallelized Unionfs, in which a pool of worker threads will be available to perform operations. Unionfs will begin execution of all operations, and then process the results after they complete. This scheme will allow the individual I/O operations to be interleaved, resulting in performance in $O(max(t_0, \ldots, t_n))$. This could be particularly useful if the underlying branches use physically different disks. For read-only operations, like `lookup`, all of the worker threads can be started at once. However, for some operations like `unlink`, there will be a small number of synchronization points at which the next operation depends on the result of the previous operation (e.g, a whiteout should be created only if there are read-only–file-system errors).

Unionfs maintains handles to lower-level file system objects in the private data of its in-kernel structures. Currently, accessing the underlying file systems causes cache incoherences with Unionfs because objects may cease to exist or be inserted without Unionfs's knowledge. Our revalidation method refreshes Unionfs's caches when branches are added or removed. We plan to extend the revalidation method to verify that an object (and its parent's), are kept up-to-date. If a lower-level object (or its parent) changes, then we will refresh the Unionfs cache.

For operations on file data, Unionfs is completely bypassed and the lower-level file system is consulted. This method of accessing data yields performance equal to that of the to the lower-level file system for memory-mapped operations, and performance quite close to the lower-level file system for standard `read` and `write` operations. Additionally, because data is only cached at the lower-level there are fewer cache coherency issues. Unfortunately, Unionfs is not notified of memory-mapped writes in this architecture, therefore writable shared mappings can be changed after a snapshot. We will improve Unionfs to use its own set of memory-mapping operations, so that it is notified of all memory-mapped writes. To ensure that performance is not hurt, we will use a *page flipping* technique so that caching is only performed on the Unionfs level. There are also two implementation details that motivate us to move to using our own memory-mapped operations. First, the `/proc` file system on Linux uses the memory-mapped region of the current executable to return its pathname, so the lower-level path name is returned when reading from `/proc/self/exe`. Second, the `sendfile` system call requires a matching file structure and address space structure. As Unionfs presently has no address space structure, we cannot properly implement `sendfile`, which is required for loop device mounts and improves performance for Web and NFS servers.

Unionfs's persistent inode maps provide the basis for NFS file handles that will survive remounts and reboots. To support persistent NFS file handles, Unionfs must be modified to read the lower-level inode in our own `read_inode` operation, and to find the parent of a given directory. Reading the lower-level inode is a relatively straightforward change. Determining the parent of a directory is more difficult, because each Unionfs directory maps to several lower-level directories. Unionfs can obtain a properly initialized and connected dentry from an inode using the following method. The basic idea is to walk from a given inode up to the Unionfs root, and then walk back down to the proper dentry. First, using the Unionfs inode number and the forward map, Unionfs can get the lower-level inode. Given this inode, Unionfs can look up `..` to obtain the lower-level parent. We can then use the reverse map to obtain the Unionfs inode number. This process is repeated until we find the root Unionfs inode. After we obtain the root Unionfs inode, we can walk back down the path to populate each directory entry by reading the parent directory to find the child entries (this type of search is similar to how the generic Linux export operations).

Finally, there are two features that we have described in our design, but have not yet implemented: (1) a `fsck` program for Unionfs, (2) creating a state file during `rename` for `fsck`. We plan to implement these features in the future.

ACKNOWLEDGMENTS

This work was partially made possible by an NSF CAREER award EIA-0133589, NSF Trusted Computing Award CCR-0310493, and HP/Intel gifts numbers 87128 and 88415.1.

Software and documentation are available from http://unionfs.filesystems.org. The source code for all benchmarks and the scripts we used in the evaluation section are available at www.fsl.cs.sunysb.edu/˜cwright/unionfs-tos/.

REFERENCES

AT&T Bell Laboratories 1995. *Plan 9 – Programmer's Manual*. AT&T Bell Laboratories.

ELLARD, D. AND SELTZER, M. 2003. NFS Tricks and Benchmarking Traps. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*. San Antonio, TX, 101–114.

FLOURIS, M. D. AND BILAS, A. 2004. Clotho: Transparent Data Versioning at the Block I/O Level. In *Proceedings of the 12th NASA Goddard, 21st IEEE Conference on Mass Storage Systems and Technologies (MSST 2004)*. College Park, Maryland, 315–328.

FRASER, T., BADGER, L., AND FELDMAN, M. 1999. Hardening COTS Software with Generic Software Wrappers. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*. 2–16.

GUY, R. G., HEIDEMANN, J. S., MAK, W., PAGE JR., T. W., POPEK, G. J., AND ROTHMEIER, D. 1990. Implementation of the Ficus replicated file system. In *Proceedings of the Summer USENIX Technical Conference*. 63–71.

HEIDEMANN, J. S. AND POPEK, G. J. 1994. File system development with stackable layers. *ACM Transactions on Computer Systems 12,* 1 (February), 58–89.

HENDRICKS, D. 1990. A Filesystem For Software Development. In *Proceedings of the USENIX Summer Conference*. Anaheim, CA, 333–340.

HITZ, D., LAU, J., AND MALCOLM, M. 1994. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference*. San Francisco, CA, 235–245.

JOUKOV, N., WRIGHT, C. P., AND ZADOK, E. 2004. FSprof: An In-Kernel File System Operations Profiler. Tech. Rep. FSL-04-06, Computer Science Department, Stony Brook University. November. www.fsl.cs.sunysb.edu/docs/aggregate˙stats-tr/aggregate˙stats.pdf.

KAMP, P. H. AND WATSON, R. N. M. 2000. Jails: Confining the omnipotent root. In *Proceedings of the Second International System Administration and Networking Conference (SANE2000)*. Maastricht, The Netherlands.

KATCHER, J. 1997. PostMark: A New Filesystem Benchmark. Tech. Rep. TR3022, Network Appliance. www.netapp.com/tech˙library/3022.html.

KORN, D. G. AND KRELL, E. 1990. A New Dimension for the Unix File System. *Software-Practice and Experience*, 19–34.

MCKUSICK, M. K. AND GANGER, G. R. 1999. Soft Updates: A technique for eliminating most synchronous writes in the fast filesystem. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*. Monterey, CA, 1–18.

OPENBSD. 2005. OpenSSH. www.openssh.org.

PENDRY, J. S. AND MCKUSICK, M. K. 1995. Union mounts in 4.4BSD-Lite. In *Proceedings of the USENIX Technical Conference on UNIX and Advanced Computing Systems*. 25–33.

PENDRY, J. S., WILLIAMS, N., AND ZADOK, E. 2003. *Am-utils User Manual*, 6.1b3 ed. www.am-utils.org.

PETERSON, Z. AND BURNS, R. 2005. Ext3cow: a time-shifting file system for regulatory compliance. *Trans. Storage 1,* 2, 190–212.

ROSELLI, D., LORCH, J. R., AND ANDERSON, T. E. 2000. A Comparison of File System Workloads. In *Proc. of the Annual USENIX Technical Conference*. San Diego, CA, 41–54.

ROSENTHAL, D. S. H. 1990. Evolving the Vnode interface. In *Proceedings of the Summer USENIX Technical Conference*. 107–18.

SPADAVECCHIA, J. AND ZADOK, E. 2002. Enhancing NFS Cross-Administrative Domain Access. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*. Monterey, CA, 181–194.

WRIGHT, C. P., MARTINO, M., AND ZADOK, E. 2003. NCryptfs: A Secure and Convenient Cryptographic File System. In *Proceedings of the Annual USENIX Technical Conference*. San Antonio, TX, 197–210.

ZADOK, E. AND NIEH, J. 2000. FiST: A Language for Stackable File Systems. In *Proceedings of the Annual USENIX Technical Conference*. San Diego, CA, 55–70.

Received June 2005; accepted June 2005