Versatility and Unix Semantics in a Fan-Out Unification File System

Charles P. Wright, Jay Dave, Puja Gupta, Harikesavan Krishnan,
Erez Zadok, and Mohammad Nayyer Zubair
Stony Brook University

Technical Report FSL-04-01b

Abstract

Administrators often prefer to keep related sets of files in different locations or media, as it is easier to maintain those files separately. Users, on the other hand, prefer to see all files in one location for convenience. One solution that accommodates both needs is virtual unification—that is providing a merged view of several directories, without physically merging them. For example, 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. Unification can also be used for snapshotting, by marking some data sources read-only and then utilizing copy-on-write for the read-only sources. It is difficult to virtually unify of a set of files while maintaining Unix semantics. Past efforts to provide such unification often compromised on the set of features provided or Unix compatibility.

We designed a unification file system called *Unionfs* which uses a promising fan-out stacking technique rarely used before. Unionfs maintains Unix semantics while offering advanced unification features such as dynamic insertion and removal of namespaces at any point in the unified view, support for any mix of read-only and read-write components, efficient in-kernel duplicate elimination, and more. Our flexible Unionfs implementation supports traditional unification, snapshotting, and sandboxing. We implemented a prototype of Unionfs on Linux. Our evaluation shows a 2–3% performance overhead for typical user-like workloads.

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 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 unification, which we explore 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 various 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 can not be used because mounting two file systems in the same place would hide the files that were mounted first. Automounters use symbolic links to create this illusion [15], but symbolic links still expose the physical directory structure to users and utilities. Furthermore, many applications (e.g., shells) interpret symbolic links. Automounters also precompute the entire union without keeping it up-to-date. A unification file system can simply unify the various mount points into a common directory. File servers may come online or go offline at any time. Therefore, it is necessary that a unification file system can dynamically add and remove branches.

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. On our group's FTP server, we only keep physical copies of the Fedora ISO images; we loopback-mount the ISO images, and then we unify their contents to provide direct access to the RPMs and SRPMs.

Snapshotting is a useful tool for system administrators, who need to know what changes are made to the system while installing new software [9, 13]. 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.

Along similar lines, 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. 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 only see the existing data, and changes made by other good processes. Likewise, the bad processes only see the existing data and changes made by bad procceses.

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 same 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 the aforementioned difficulties in maintaining Unix semantics in union file systems, it is not surprising that none of the past implementations solved all problems satisfactorily.

We have designed and built *Unionfs*, a unification file system that addresses all of the known complexities of maintaining Unix semantics without compromising versatility and features offered. We support two file deletion modes that address even partial failures. We allow an efficient and cache-coherent insertion or deletion of any arbitrary read-only or read-write directory into the union. Unionfs also includes efficient in-kernel handling of files with the same name; a careful design that minimizes data movement across branches; several modes for permission inheritance; and support for snapshots and sandboxing. We compare Unionfs's features with past alternatives and show that Unionfs provides new features and also useful features from past work. Our performance evaluation shows a small overhead of 2–3% under normal user workloads and acceptable overheads even under demanding workloads.

The rest of this paper is organized as follows. Section 2 describes our design and Section 3 elaborates on the design of each Unionfs operation. Section 4 surveys related work. Section 5 compares the features of Unionfs with those offered by previous systems. Section 6 analyzes Unionfs's performance. 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. 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 appears to still 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 metadata. 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 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 kept 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 can not be overridden by another branch.
- **Maintain Unix Semantics** One of our primary goals was to maintain Unix semantics. 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 attempt to leave files in the branch in which they already exist and avoid copying data across branches until required.

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

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 hardlinks 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.

Stacking VFS Operations Stackable file systems are a technique to layer new functionality on existing file systems [19]. 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.

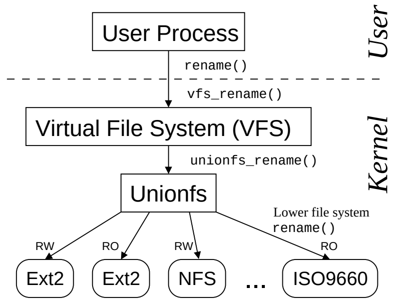

Figure 1: Unionfs: A Fan-out file system can access N different branches directly.

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 [7, 17]. 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. In this way Unix semantics are preserved even if the operation fails in some branches, because the user-level view remains unchanged. For all operations in Unionfs, we utilize 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 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 [14], 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 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.

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 can not exist in the same branch. Depending on a mount-time flag, Unionfs creates whiteouts in unlink-like operations as discussed in Sections 3.3 and 3.4. Whiteouts for files are created atomically by renaming F to `.wh.`F. For other types of objects, `.wh.`F is created, and then the original object is removed.

3 Design Details

In this section, we describe individual Unionfs operations. We describe lookup and open in Section 3.1, creating new objects in Section 3.2, deleting objects in Section 3.3, rename in Section 3.4, dynamic branch insertion and deletion in Section 3.5, sandboxing using split-view caches in Section 3.6, and readdir in Section 3.7.

3.1 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 lookup on each branch to the right of the leftmost file to populates the branches that were skipped.

Unionfs provides a unified and a merged view of directories in all the branches. Therefore if the leftmost entry is a directory, Unionfs looks up the directory in all the 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.

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 underlying branches, in preparation for readdir as described in Section 3.7. 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.

3.2 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.

3.3 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
$X[i]$	X's lower-level object in branch i

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

```
1 create_whiteout(X, i)
2   while (i $\geq$ 1) {
3     if create $\bar{X}$ 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, 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` can detect and repair an error.

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.

3.4 Rename

Rename 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 in Unionfs may involve multiple operations like rename, unlink, create, copyup, and whiteout creation, it is important to provide 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. (1) If rename succeeds, then S is renamed to D and S does not exist. (2) 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 modes based on the unlink flag: 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 3.3, 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 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 the user sees does not change, because the leftmost source and destination are preserved. During the Unionfs rename operation, the source and destination directories are locked, so the user can not view an inconsistent state. However, if an unclean shutdown occurs, the file system may be in an inconsistent state. Our solution (not yet implemented) is to create a temporary state file before the rename operation and then remove it afterward. Our high-level fsck can then detect and repair any errors.

Rename in DELETE_ALL mode first renames each source file to the destination in its own branch, from right to left. The branch that contains the leftmost destination (L_D) is skipped in this first pass, because if a subsequent operation were to fail we could not undo the rename in L_D. The second phase is to remove the destination file 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). If 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. If this operation succeeds, then the operation as a whole succeeds, otherwise the operation fails. If the leftmost destination is not to the left of the leftmost source ($L_D >= L_S$), then the source is renamed to the destination in that branch. Again, if this operation succeeds, the operation as a whole succeeds, otherwise the operation fails. If the operation fails, we revert the renamed files to their original name.

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.

3.5 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 revalidate and d_revalidate 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 a dentry or inode is similar to the lookup procedure, but instead of creating new objects, it modifies existing objects. Refreshing an open file is similar to the open procedure, but preserves the existing upper-level file structure.

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 object or 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 handle this functionality so that the service is exposed to all file systems.

3.6 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 [18]. 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 Tracefs-like filter [1] 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 selection algorithm, though we plan to create an interface for modules to register their own selection algorithms.

3.7 Readdir

Readdir returns directory entries in an open directory. A directory in Unionfs can contain multiple directories from different branches, and therefore a readdir 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. 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. Unionfs does not return names that have already been recorded.

4 Related Work

We begin by describing the origins of fan-out file systems. Next, we briefly describe several other representative unification systems. We then 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 [17]. 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 [6, 7]. 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 is 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 [2]. 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 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 [12]. 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 [19]; 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 [8]. 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 [15] 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 [14], 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 the bottom of the view. Every lookup operation in a lower layer creates a corresponding directory tree called a *shadow directory* in the upper layer. 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 [9, 13]. 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 [4]. 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 can also be created per file or 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 [10]. Another form of sandboxing is to monitor system calls, and if they deviate from a policy, prevent them from being executed [5].

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 fifteen features and metrics of these systems, and we summarized them in Table 1:

	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	Multiple writable branches	✔				✔
6	Dynamic insertion & removal of any branch					✔
7	Dynamic insertion & removal of the top branch	✔	✔		✔	✔
8	No file system type restrictions	✔	✔	✔	[b]	✔
9	Creating shadow directories		✔[c]	✔	✔	✔[c]
10	Copyup-on-write		✔	✔	✔	✔[d]
11	Whiteout support		✔[e]	✔	✔	✔
12	Snapshot support					✔
13	Sandbox support					✔
14	Implementation technique	VFS (stack)	User Library	User NFS Server + Kernel helper	Kernel FS (stack)	Kernel FS (fan-out)
15	Operating systems supported	Plan 9	Many[f]	SunOS 4.1	4.4BSD	Linux[g]
16	Total LoC	6,247[h]	16,078	16,613	3,997	9,784
17	Customized functionality					✔

Table 1: Feature Comparison. A check mark indicates that the feature is supported, otherwise it is not.
[a] *Through a mount time flag, 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, 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.*

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.
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. **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.
6. **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.
7. **Dynamic insertion and removal of any branch**: Only Unionfs can dynamically insert or remove a branch anywhere in the union.
8. **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.
9. **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 errors, like "read-only file system" (EROFS).

10. **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.
11. **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.
12. **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.
13. **Sandbox support**: Only Unionfs supports sandboxing processes.
14. **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 [19].
15. **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.
16. **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 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 611 lines of user-space management utilities.
17. **Customized functionality**: Unionfs has a flexible design that provides several modes of operation using mount-time flags. 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 3.3.

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 Pentium-IV 1.7GHz with 1GB of RAM. The machine ran Red Hat Linux 9 and a vanilla 2.4.26 kernel. All experiments were located on a dedicated 200GB Maxtor IDE disk. In order 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 [3]. We chose Ext2 as the base file system since it is widely used and well-tested. To ensure a cold cache, we unmounted the underlying file system once after loading the test data. We also ran a *chill* program that we wrote, which allocates and accesses as much memory as possible, thereby forcing the kernel to evict data structures and buffers. 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 were less than 5% of the mean.

6.1 Confi gurations

We begin this section by describing the configurations that we use for the tests. We use 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 1, 2, 3, 4, 8, and 16 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.

A test run is uniquely described by the mount-time options, the data distribution, and the number of branches.

6.2 General Purpose Benchmarks

We chose two representative general-purpose workloads: (1) Postmark, an I/O intensive benchmark [11], and (2) a CPU-intensive compile benchmark, building the Am-utils package [15]. 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 lookups, 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 [11]. We used 200 subdirectories to prevent linear directory lookups from dominating the results.

The Am-utils build (version 6.1b4) contains over 60,000 lines of code. It performs several hundred small configuration tests, and then it builds a shared library, ten binaries, four scripts, and documentation. This benchmark contains a fair mix of file system operations, representing the typical performance impact for users.

Figure 2 shows the elapsed, system, and user time for Postmark in the DWHT and DALL modes. The results stayed relatively constant as the number of branches increased, demonstrating Unionfs's scalability. The elapsed time overheads for DALL is in the range of 12.7–14.3% above that of Ext2. The elapsed time overheads for DWHT, however, are higher than DALL: varying from 23.5–27.5%. This higher overhead is due to two factors.

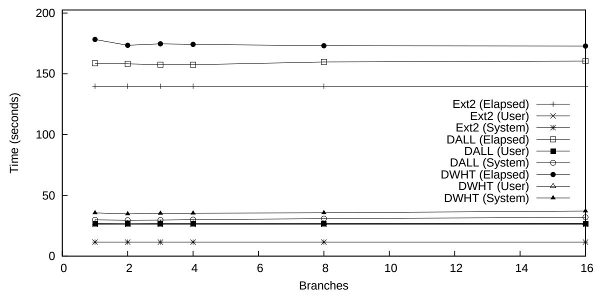

Figure 2: Postmark: 20,000 files and 200,000 transactions.

First, whiteout creation requires two steps—truncating the file and then renaming it. Second, all directory operations consume more time, because lookups take longer as the number of directory entries increases.

The results for all of our Am-utils compile benchmarks have overheads ranging from 1.5–2.5% for elapsed time and 6.0–9.1% for system time. Although Postmark shows that Unionfs may be 2.4 times slower under high I/O loads, the Am-utils results demonstrate that most users are unlikely to notice a performance degradation using Unionfs.

Snapshots We also ran Am-Utils and Postmark while taking snapshots every 60, 30, and 15 seconds. Am-Utils had an elapsed time overhead of 19.0–20.1% over Ext2 for all intervals. Four, eight, and sixteen snapshots were taken for intervals of 60, 30, and 15 seconds, respectively. This demonstrates that Unionfs efficiently performs snapshots for user-like workloads. The Postmark results are shown in Table 2. 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, delete, and lookup take more time. This indicates that periodically merging snapshots would be beneficial.

Interval(s)	Snapshots	Overhead
15	35.4	275%
30	9.9	104%
60	4	54%

Table 2: 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 the lookup operation by running stat on each file and directory.
- **READDIR** evaluates the overhead of 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 the what to operate on.

For our data set we used 25 copies of Am-utils distribution for a total of 650 directories with 10,750 files that take 200.9MB of disk space. For the distributed data set, the number of files remains constant, but there are duplicated directories. For the duplicated set, each branch has a copy of all files and directories.

STAT Figure 3 shows the benchmark results for STAT. For a single branch, Unionfs has an overhead of 6.4% over Ext2. A Unionfs lookup 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 2.0 for two branches over a single branch, 2.9 for three branches, 3.6 for four branches, 6.2 for eight branches, and 9.4 for sixteen branches.

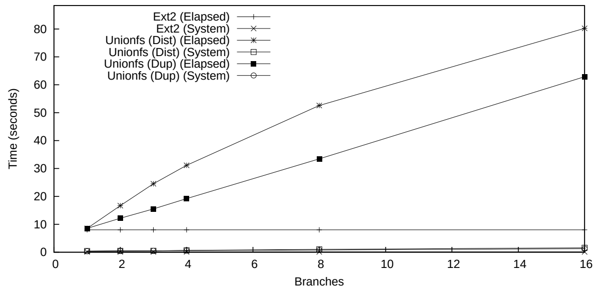

Figure 3: STAT *results.*

On the other hand, for a DUP data distribution, Unionfs can stop after the first branch for files, but must access all of the directories. For a single branch, the overhead is again 6.4%. The overhead for two branches over a single branch is 43%, for three branches the overhead increases to 82%. For sixteen branches, the overhead is a factor of 6.4.

Most of this overhead is I/O. With a cold cache, the benchmark took from 8.0–80 seconds. With a warm cache, the benchmark takes 1.2 seconds for Ext2, and from 1.2–1.8 seconds for Unionfs. We believe that the warm-cache results are closer to most user workloads, because most files are accessed multiple times [16].

READDIR Figure 4 shows the benchmark results for READDIR. For a single branch, Unionfs has an overhead of 2.0% over Ext2. A Unionfs readdir with a DIST distribution scans all the branches from left to right listing the contents of 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 still must 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 2.2 for two branches over a single branch, 3.3 for three branches, 4.2 for four branches, 6.3 for eight branches, and 10.1 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 again 2.0%. The overhead for two branches over a single branch is a factor of 2.4, 3.7 for three branches, 5.1 for four branches, 11.7 for eight branches, and 25.5 for sixteen branches.

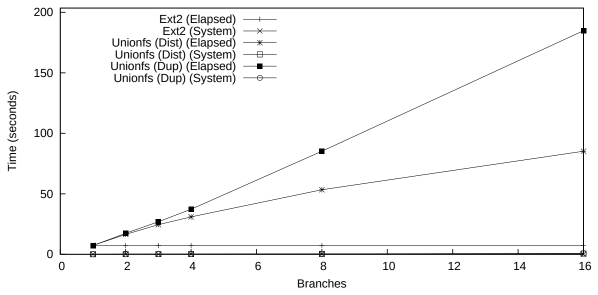

Figure 4: READDIR *results.*

Most of this overhead is I/O. Indeed, to perform the test on 16 duplicate copies of the data set on Ext2 took 19.5 times as long as to read a single copy. For a 16-branch Unionfs configuration with duplicated data, this translates into an overhead of 31.6% over Ext2 with 16 duplicated data sets. With a cold cache, the benchmark took from 7.2–184 seconds. With a warm cache, the benchmark completes in 0.05 seconds for Ext2, and from 0.07–0.2 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 5 shows the benchmark results for our UNLINK microbenchmark. Before running the benchmark we ran `find` over the Unionfs file system. We did this to warm the cache so that lookup costs would not interfere with the unlink results. The lookup operation on files only access the first file, so significant I/O is still be required for Unionfs.

With a DALL configuration, the mean elapsed time increases from 0.20 seconds for Ext2 to 0.57 seconds for a single branch Unionfs (a factor of 2.9). This increase is due to a 3.4 times increase in system time. When additional branches are added, the elapsed time increases for three reasons. First, lookups 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 logically empty. Third, directories themselves must be deleted in many branches. A union with two branches had a 17% elapsed time overhead when compared to a single branch; three branches had an overhead of 35%, four branches had an overhead of 52%, 8 branches had an overhead of 116%, and 16 branches were slower by a factor of 3.5.

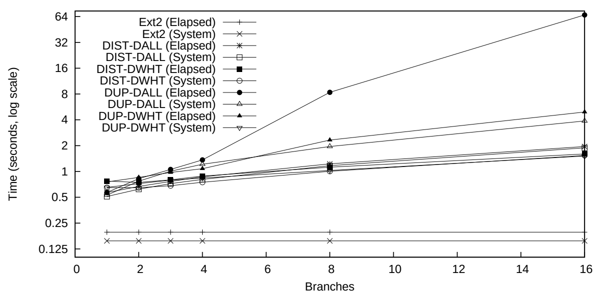

Figure 5: UNLINK *results.*

With a DWHT configuration, the mean elapsed time increases from 0.20 seconds to 0.77 seconds (a factor of 3.9); again primarily because of an increase in system time. The reason that DWHT takes 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 3.4% for two and three branches, 14% for four branches, 45% for eight branches, and 107% for sixteen branches. The overhead increases as the number of branches increases because readdir is called to verify that the directory is logically empty.

For a DUP distribution with DALL, the elapsed time again increased to 0.57 seconds for a single branch from 0.20 seconds for Ext2, but 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 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 45%, and for three branches 85%. For four or more branches, the I/O time becomes a significant portion of the overall elapsed time and this is reflected in the overheads. For four branches the overhead is a factor of 2.3, for eight branches the overhead is a factor of 14.7, and for sixteen branches, the overhead is a factor of 117. The reason for such a high overhead for sixteen branches is that Unionfs must actually perform many operations, including I/O bound operations. We constructed a similar benchmark for Ext2, that will remove 4, 8, or 16 copies of our data set. We warmed the cache so that the cache data was the same as Unionfs for a corresponding number of branches. Only the first copy of the data set has a fully warm cache. All other copies have only the directories warmed. The overhead of Unionfs compared to this Ext2 test was 47% for four branches, 56% for eight branches, and only 79% for sixteen branches.

With a DWHT configuration, the mean elapsed time increases from 0.20 seconds to 0.77 seconds (a factor of 3.9). The overhead over a single branch is 12.9% for two branches, 29.1% for three branches, 42% for four branches, 207% for eight branches, and 553% 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.

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 bench like Postmark. Our microbenchmarks 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 and implemented Unionfs, a 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 unification systems have not previously been applied.

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 have used careful ordering of operations to atomically return success or failure to the user, and leave the file system in a consistent state. To detect inconsistencies that may arise from unclean shutdowns, we have created a high-level `fsck` that examines the logical structure of a union. Like a standard disk-based `fsck`, our file-system checker flags errors and allows the user to optionally correct them.

Future Work Unionfs is the first fully-functional stackable fan-out file system made available. We believe that fan-out stackable file systems have a potential for usefulness in other areas: replication, striping, fail-over, caching, and more. Unionfs has taught us much about the complexity of trying to balance versatility (providing many useful features) and maintaining Unix semantics. We are investigating more fan-out file systems with an eye toward useful general OS infrastructure.

One way to address partial failures is to allow applications to be aware of such conditions. Most system calls return either success or one error code from a set of predefined errors. A small number of system calls already return partial failures that application programmers must address (e.g., a *short read* or EAGAIN). We plan to explore ways of dynamically creating and querying new error codes that could return more information to applications. For example, if a write on a replication file system failed on some branches, the replication file system could return information about each branch.

References

[1] A. Aranya, C. P. Wright, and E. Zadok. Tracefs: A File System to Trace Them All. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 129–143, San Francisco, CA, March/April 2004.

[2] AT&T Bell Laboratories. *Plan 9 – Programmer's Manual*, March 1995.

[3] D. Ellard and M. Seltzer. NFS Tricks and Benchmarking Traps. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 101–114, San Antonio, TX, June 2003.

[4] M. D. Flouris and A. Bilas. 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)*, pages 315–328, College Park, Maryland, April 2004.

[5] T. Fraser, L. Badger, and M. Feldman. Hardening COTS Software with Generic Software Wrappers. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 2–16, May 1999.

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

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

[8] D. Hendricks. A Filesystem For Software Development. In *Proceedings of the USENIX Summer Conference*, pages 333–340, Anaheim, CA, June 1990.

[9] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference*, pages 235–245, San Francisco, CA, January 1994.

[10] P. H. Kamp and R. N. M. Watson. Jails: Confi ning the omnipotent root. In *Proceedings of the Second International System Administration and Networking Conference (SANE2000)*, Maastricht, The Netherlands, May 2000.

[11] J. Katcher. PostMark: A New Filesystem Benchmark. Technical Report TR3022, Network Appliance, 1997. www.netapp.com/tech˙library/3022.html.

[12] D. G. Korn and E. Krell. A New Dimension for the Unix File System. *Software-Practice and Experience*, pages 19–34, June 1990.

[13] M. K. McKusick and G. R. Ganger. Soft Updates: A technique for eliminating most synchronous writes in the fast fi lesystem. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 1–18, Monterey, CA, JUNE 1999.

[14] J. S. Pendry and M. K. McKusick. Union mounts in 4.4BSD-Lite. In *Proceedings of the USENIX Technical Conference on UNIX and Advanced Computing Systems*, pages 25–33, December 1995.

[15] J. S. Pendry, N. Williams, and E. Zadok. *Am-utils User Manual*, 6.1b3 edition, July 2003. www.am-utils.org.

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

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

[18] J. Spadavecchia and E. Zadok. Enhancing NFS Cross-Administrative Domain Access. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 181–194, Monterey, CA, June 2002.

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

Software and documentation are available from www.fsl.cs.sunysb.edu/project-unionfs.html.