Unionfs: User- and Community-Oriented Development of a Unification File System

David Quigley, Josef Sipek, Charles P. Wright, and Erez Zadok
Stony Brook University
{dquigley,jsipek,cwright,ezk}@cs.sunysb.edu

Appears in the proceedings of the 2006 Ottawa Linux Symposium (OLS 2006)

Abstract

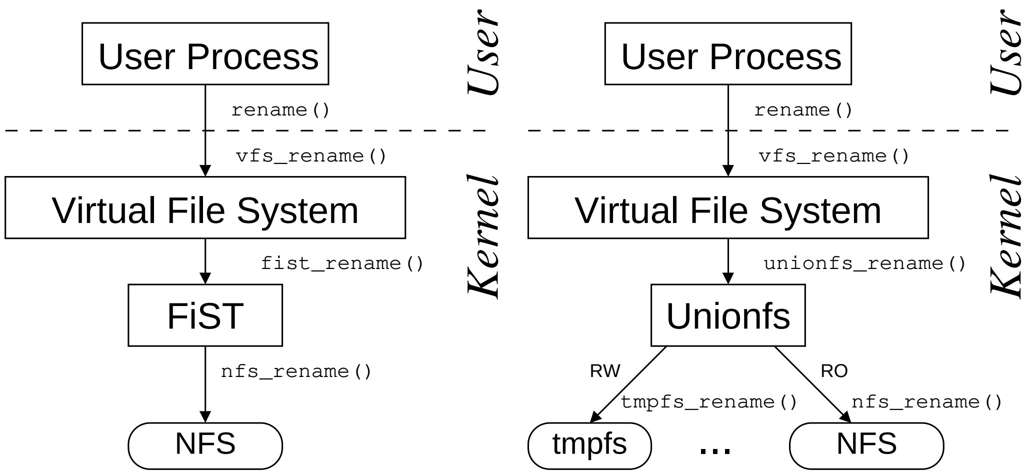
Unionfs is a stackable file system that virtually merges a set of directories (called branches) into a single logical view. Each branch is assigned a priority and may be either read-only or read-write. When the highest priority branch is writable, Unionfs provides copy-on-write semantics for read-only branches. These copy-on-write semantics have lead to widespread use of Unionfs by LiveCD projects including Knoppix and SLAX. In this paper we describe our experiences distributing and maintaining an out-of-kernel module since November 2004. As of March 2006 Unionfs has been downloaded by over 6,700 unique users and is used by over two dozen other projects. The total number of Unionfs users, by extension, is in the tens of thousands.

1 Introduction

Unionfs is a stackable file system that allows users to specify a series of directories (also known as branches) which are presented to users as one virtual directory even though the branches can come from different file systems. This is commonly referred to as namespace unification. Unionfs uses a simple priority system which gives each branch a unique priority. If a file exists in multiple branches, the user sees only the copy in the higher-priority branch. Unionfs allows some branches to be read-only, but as long as the highest-priority branch is read-write, Unionfs uses copy-on-write semantics to provide an illusion that all branches are writable. This feature allows Live-CD developers to give their users a writable system based on read-only media.

There are many uses for namespace unification. The two most common uses are Live-CDs and diskless/NFS-root clients. On Live-CDs, by definition, the data is stored on a read-only medium. However, it is very convenient for users to be able to modify the data. Unifying the read-only CD with a writable RAM disk gives the user the illusion of being able to modify the CD. Maintaining an identical system configuration across multiple diskless systems is another application of Unionfs. One simply needs to build a read-only system image, and create a union for each diskless node.

Unionfs is based on the FiST stackable file system templates, which provide support for layering over a single directory [12]. As shown in Figure 1(a), the kernel's VFS is responsible for dispatching file-system–related system



(a) Wrapfs layers over a single directory. (b) Unionfs layers over multiple directories.

Figure 1: User processes issue system calls, which the kernel's virtual file system (VFS) directs to stackable file systems. Stackable file systems in turn pass the calls down to lower-level file systems (e.g., tmpfs or NFS).

calls to the appropriate file system. To the VFS, a stackable file system appears as if it were a standard file system, but instead of storing or retrieving data, a stackable file system passes calls down to lower-level file systems that are responsible for data storage and retrieval. In this scenario, NFS is used as the lower-level file system, but any file system can be used to store the data (e.g., Ext2, Ext3, Reiserfs, SQUASHFS, isofs, and `tmpfs`). To the lower-level file systems, a stackable file system appears as if it were the VFS. This makes stackable file system development difficult, because the file system must adhere to the conventions both of file systems (for processing VFS calls) and of the VFS (for making VFS calls).

As shown in Figure 1(b), Unionfs extends the FiST templates to layer over multiple directories, unify directory contents, and perform copy-on-write. In this example, Unionfs is layered over two branches: (1) a read-write `tmpfs` file system and (2) a read-only NFS file system. The contents of these file systems are virtually merged, and if operations on the NFS file system return the read-only file system error code (`EROFS`) then Unionfs transparently copies the files to the `tmpfs` branch.

We originally released Unionfs in November 2004, after roughly 18 months of development as a research project [10, 11]. We released Unionfs as a standalone kernel module because that was the most expedient way for users to begin using it and it required less initial effort on our part. Unionfs was quickly adopted by several LiveCDs such as SLAX [7] (December 2004) and Knoppix [5] (March 2005). As of March 2006, Unionfs has been downloaded by over 6,700 users from 81 countries and is distributed as part of other projects. Our mailing list currently has 336 subscribers with 53 of them subscribed to our CVS update list. Unionfs is an integral part of several LiveCDs, so the actual number of Unionfs users is much larger.

Maintaining Unionfs out side of the kernel has both benefits and complications. By maintaining the tree out side of the kernel, our user base is expanded: users can still use their vendor's kernel, and we are able to support several kernel versions. We were also able to release Unionfs more frequently than the kernel. However, this makes our code more complex since we must deal with changing interfaces between kernel versions. It also raises questions about the point at which support for a particular older kernel version should be dropped. At this point, Unionfs has become established enough that we are moving towards a release that is ready for mainline kernel submission.

Unionfs has complex allocation schemes (particularly for dentry and inode objects), and makes more use of `kmalloc` than other file systems. One hurdle we had to overcome was lack of useful memory-allocation debugging support. The memory-allocation debugging code in recent `-mm` kernels does not provide sufficient debugging information. In our approach, we log `kmalloc` and dentry allocations, and then post-process the log to locate memory leaks and other errors.

In our efforts to move toward kernel inclusion we have come across many aspects that conflict with maintaining an out-of-kernel module. One of the main issues is the ability to separate research code from practical code. Features such as persistent inodes and atomically performing certain operations increase code complexity, conflicting with the mantra "less code is better code." We also had to change the way we separate file system components to provide simpler and more easily maintainable code. In addition to this, we also have to keep up with changes in kernel interfaces such as the change of locking primitives introduced in Linux 2.6.16.

The rest of this paper is organized as follows. In Section 2 we describe Unionfs use cases. In Section 3 we describe the challenges of maintaining an out-of-tree module. In Section 4 we describe some limitations of Unionfs. In Section 5 we present a brief performance evaluation of Unionfs. Finally, we conclude in Section 6.

2 Use Cases

We have identified three primary use cases for Unionfs. All of these common use cases leverage Unionfs's copy-on-write semantics. The first and most prevalent use of Unionfs is in LiveCDs. The second is using Unionfs to provide a common base for several NFS-mounted machines. The third is to use Unionfs for snapshotting.

LiveCDs. LiveCDs allow users to boot Linux without modifying any data on a hard disk. This has several advantages:

- Users can try Linux without committing to it [5, 7].
- Special-purpose open-source software can be distributed to non-technical users (e.g., for music composition [4]).
- System administrators can rescue machines more easily [2].
- Many similar machines can be set up without installing software (e.g., in a cluster environment [9], or at events that require certain software).

The simplest use of Unionfs for LiveCDs unifies a standard read-only ISO9660 file system with a higher-priority read-write `tmpfs` file system. Current versions of Knoppix [5] use such a configuration, which allows users to install and reconfigure programs.

Knoppix begins its boot sequence by loading an initial RAM disk (`initrd`) image of an Ext2 file system and then executing a shell script called `/linuxrc`. The `linuxrc` script first mounts the `/proc` and `/sys` file systems. Next, Knoppix loads various device drivers (e.g., SCSI, IDE, USB, FireWire) and mounts a compressed ISO9660 image on /KNOPPIX. After the Knoppix image is mounted, the Unionfs module is loaded. Next a tmpfs file system is mounted on /ramdisk. Once Unionfs is mounted, this RAM disk becomes the destination for all of the changes to the CD-ROM. Next, the directory /UNIONFS is created and Unionfs is mounted on that directory with the following command:

```
mount -t unionfs \
  -o dirs=/ramdisk=rw:/KNOPPIX=ro \
  /UNIONFS /UNIONFS
```

The -t unionfs argument tells the mount program that the file system type is Unionfs. The -o dirs=/ramdisk=rw,/KNOPPIX=ro option specifies the directories that make up the union. Directories are listed in a order of priority, starting with the highest. In this case, the highest-priority directory is /ramdisk, which is read-write. The /ramdisk directory is unified with /KNOPPIX, which is read-only. The first /UNIONFS argument is a placeholder for the device name in /proc/mounts, and the second /UNIONFS argument is the location where Unionfs is mounted. Finally, linuxrc makes symbolic links from the root directory to /UNIONFS. For example, /home is a link to /UNIONFS/home. At this point the linuxrc script exits, and init is executed.

Other LiveCDs (notably SLAX [7]) use Unionfs both for its copy-on-write semantics and as a package manager. A SLAX distribution consists of several *modules*, which are essentially SQUASHFS file system images [6]. On boot, the selected modules are unified to create a single file system view. Unifying the file systems makes it simple to add or remove packages from the LiveCD, without regenerating entire file system images. In addition to the SQUASHFS images, the highest-priority branch is a read-write tmpfs which provides the illusion that the CD is read-write.

SLAX uses the pivot_root system call so that the root file system is indeed Unionfs, whereas Knoppix creates symbolic links to provide the illusion of a Unionfs-rooted CD. SLAX also begins its boot sequence by loading an Ext2 initrd image and executing linuxrc, which mounts /proc and /sys. Next, SLAX mounts tmpfs on /memory. The next step is to mount Unionfs on /union with a single branch /memory/changes using the following command:

```
mount -t unionfs \
 -o dirs=/memory/changes=rw
 unionfs /union
```

Aside from the branch configuration, the major difference between this command and the one from Knoppix is that instead of using /UNIONFS as a placeholder, the text unionfs is used instead. We recommend this approach (or better yet, the string none), because it is less likely to be confused with an actual path or argument.

After mounting the mostly empty Unionfs, SLAX performs hardware detection. The next step is to load the SLAX modules, which are equivalent to packages. The first step in loading a module is to mount the SQUASHFS image on /memory/images. After the SQUASHFS image is mounted, SLAX calls our unionctl to insert the module into the Union. The following command is used to insert SLAX's kernel module:

```
unionctl /union --add \
  --after 0 --mode ro \
  /memory/images/01_kernel
```

```
rootfs / rootfs rw 0 0
/dev/root /mnt/live ext2 rw,nogrpid 0 0
/proc /mnt/live/proc proc rw 0 0
tmpfs /mnt/live/memory tmpfs rw 0 0
unionfs / unionfs rw,dirs=/mnt/live/memory/changes=rw:...:/mnt/live/
→memory/images/02_core.mo=ro:/mnt/live/memory/images/01_kernel.mo=ro
→ 0 0
/dev/hdb /mnt/live/mnt/hdb iso9660 ro 0 0
/dev/hdb /boot iso9660 ro 0 0
/dev/loop0 /mnt/live/memory/images/01_kernel.mo squashfs ro 0 0
/dev/loop2 /mnt/live/memory/images/02_core.mo squashfs ro 0 0
...
```

Figure 2: The `/proc/mounts` file on SLAX after `linuxrc` is executed. Note that the Unionfs line has been split (denoted by →). For brevity, we exclude seven additional SLAX packages.

The `--after 0` argument instructs Unionfs to insert the new directory, `/memory/images/01_kernel`, after the first branch, and the `--mode ro` argument instructs Unionfs to mark this branch read-only. This process is repeated for each module. SLAX then creates `/union/proc`, `/union/sys`, `/union/dev`, `/union/tmp`, and `/union/mnt/live`. SLAX then changes the present working directory to `/union` and unmounts `/sys`. Next, Unionfs is made the root file system using `pivot_root`:

```
pivot_root . mnt/live
```

This command makes Unionfs the root file system, and remounts the initial RAM disk on `/union/mnt/live`. Finally, SLAX starts `init` using Unionfs as the root file system:

```
/usr/bin/chroot . sbin/init
```

After this procedure, SLAX produces the `/proc/mounts` file seen in Figure 2.

NFS-mounted machines. Another use of Unionfs is to simplify the administration of diskless machines. A set of machines can share a single read-only NFS root file system. This enables administrators to maintain a common image for all of the machines. This root file system is then unified with a higher-priority read-write branch so that users can customize the machine or save data. If persistence is not required, then a `tmpfs` file system can be used as the highest priority branch. If persistence is required, then a read-write NFS mount or a local disk could be used for the user's files.

Figure 3 shows a sample NFS `/etc/exports` file for a diskless client configuration. To ensure that none of the clients can tamper with the shared binaries on the server, we export the `/bin` directory read-only. We then export the persistent storage folder for each client individually. This ensures that one client cannot tamper with the persistent folder of another.

Figure 4 shows the commands used to create a union from a shared binary directory and to provide a persistent backing store for that directory on a second NFS mount. The first command mounts `/bin` for our client. The next command mounts the persistent data store for

```
/bin client1(ro) client2(ro)
/store/client1 client1(rw)
/store/client2 client2(rw)
```

Figure 3: The contents of `/etc/exports` on the server which contains the clients' binaries.

```
mount -t nfs server:/bin /mnt/nfsbins
mount -t nfs server:/store/`hostname -s` /mnt/persist
mount -t unionfs none /bin -o dirs=/mnt/persist:/mnt/nfsbins=nfsro
```

Figure 4: Creating a union with two NFS-based shares for binaries and persistent data.

our client based on its hostname. Finally, we create a union containing the exported `/bin` and `/store/`hostname -s`` directories and mount it at `/bin` on our local client. To have a full system that is exported via NFS, one simply exports `/` instead of just `/bin`. This permits a full system to be exported to the diskless clients. However, such a set requires the additional steps present in LiveCDs which allows you to use `/proc` and `/dev`.

Snapshotting. The previous usage scenarios all assumed that one or more components of the union were read-only by necessity (either enforced by hardware limitations or the NFS server). Unionfs can also provide copy-on-write semantics by logically marking a physically read-write branch as read-only. This enables Unionfs to be used for file system snapshots. To create a snapshot, the `unionctl` tool is used to invoke branch management `ioctls` that dynamically modify the union without unmounting and remounting Unionfs. First, `unionctl` is used to add a new high-priority branch. For example, the following command adds `/snaps/1` as the highest priority branch to a union mounted on `/union`:

```
unionctl /union --add /snaps/1
```

Next, `unionctl` is called for each existing branch to mark them as read-only. The following command will mark the branch `/snaps/0` read-only:

```
unionctl /union --mode /snaps/0 ro
```

Any changes made to the file system take place only in the read-write branch. Because the read-write branch has a higher priority than all the other branches, users see the updated contents.

3 Challenges

While developing Unionfs we encountered several issues that we feel developers should address before they decide whether or not to aim for kernel inclusion. Backward compatibility, changes in kernel interfaces, and experimental code are three such issues. In section 3.1, we consider the advantages and disadvantages of maintaining a module outside of the mainline kernel. In section 3.2, we discuss the implications of developing a module that is aiming for inclusion in the mainline Linux kernel.

3.1 Developing an out of kernel module

When first releasing Unionfs, we wanted to ensure that as many people as possible could use it. To accommodate this, we attempted to provide backward compatibility with past kernel versions. Initially, when Unionfs supported Linux 2.4 it was easy to keep up with changing kernels, since most of the changes between kernel versions were bug fixes.

In December of 2004, Unionfs was ported to Linux 2.6 which introduced additional complications. VFS changes between 2.4 and 2.6 (e.g., file pointer update semantics and locking mechanisms) required `#ifdef`ed sections of code to provide backward compatibility with Linux 2.4. In addition, since we were supporting Linux 2.6, we had to be conscious of the fact that the 2.6 kernel interfaces could change between versions.

The benefit of supporting multiple kernel versions was that we could enable the use of Unionfs on many different platforms. Although LiveCD creators mostly preferred Linux 2.6 kernels, we found that some of them were still working with 2.4. In addition, several people were using Unionfs for embedded devices, which at the time tended to use 2.4 kernels. However, providing backward compatibility came with a few disadvantages and raised the question of how far back we would go. Because there is no standard kernel for LiveCD developers, there were bug reports and compatibility issues across many different kernel versions.

Although Unionfs supported multiple kernel versions, we had to choose which versions to focus on. We increased the minimum kernel version Unionfs required if: (1) it would make us `#ifdef` code that was already `#ifdef`ed for backward compatibility, or (2) if it made the code overly complex. After Unionfs was ported to Linux 2.6, we found ourselves repeatedly raising the minimum kernel version due to the large number of interface-breaking changes. For example, 2.6.11 introduced the `unlocked_ioctl` operation. The most invasive change has been 2.6.16's new mutex system. Even though we have stopped supporting backward compatibility, users often submit backward-compatibility patches which we apply but do not support.

Along with backward compatibility came increased code complexity. Although backward compatibility does not generally add much code, the readability of the code decreased since we kept many sections of `#ifdef`ed code. Moreover, it made debugging more difficult as Unionfs could run in more environments. In February of 2005, we decided to drop support for Linux 2.4 to reduce the size and complexity of the code. By placing the restriction that Unionfs will only support Linux 2.6, we were able to cut our code base by roughly 5%. Although this is not a large percentage, this increased maintainability greatly since it lowered the number of environments that we had to maintain and test against. By removing Linux 2.4 from our list of supported kernels, we eliminated eleven different kernel versions that we were supporting. This also allowed us to remove a number of bugs that were related to issues with backward compatibility and which applied to Linux 2.4 only. Before dropping support for a specific kernel version, we release a final version of Unionfs that supports that kernel version.

Even though we removed 2.4 support from Unionfs, it did not end the problems of backward compatibility. With Linux 2.6, a new development process was introduced where code and interface changes that would previously have been introduced in a development kernel are placed into the stable branch. Linux 2.6.16 introduced a new set of locking mechanisms where semaphores were replaced with mutexes. Although this is one of the larger changes we have seen, there are many such changes that force us to deal with backward compatibility within the 2.6 branch itself. This led us to decide in February of 2006 to drop backward compatibility completely and only work with the latest release candidate so that we can closely follow the kernel on our path to inclusion. Since we make a release of Unionfs before every major change we still have working copies of Unionfs for Linux 2.4 and earlier versions of Linux 2.6.

3.2 Kernel Inclusion

In our efforts to prepare Unionfs to be submitted to the kernel mailing list for discussion, we had to address three major issues. First, due to the incremental nature of Unionfs's development, the code base needed large amounts of reorganization to conform to kernel coding conventions. Second, Unionfs user-space utilities use older methods for interfacing with the kernel that needed to be replaced by newer more desired methods, such as the use of `configfs` and `sysfs`. Finally, features that were placed in Unionfs for research interests needed to be removed to make the code base more practical.

Since the Linux kernel is a massive project with people contributing code to every component, there are very strict guidelines for the way code should be organized and formatted. While reviewing the code base for Unionfs, we realized that some of the functions were unnecessarily long. Even now, due to the complex fan-out nature of Unionfs, many of the functions are longer than we would like due to loops and conditionals.

When looking into the methods available for a user-mode process to communicate with our file system, we noticed one trend. Every time a person introduces an `ioctl`, there is an objection and a suggestion to find a better way of handling what is needed. Because Unionfs uses several `ioctls` for adding branches, marking branches read-only, and identifying which branch a file is on, we decided that other methods should be explored. The preferred methods for modifying and viewing kernel object states are `configfs` and `sysfs`. Although both are good options, they both have shortcomings that prevented us from using them.

In the case of `configfs`, the major concern was that the module is optional. This issue could be addressed by marking `configfs` to be selected by Unionfs, but that ignores a larger issue. Many of the users of Unionfs are using it in embedded devices and on LiveCDs. If we use `configfs` to control Unionfs's configuration, we are forcing those users to use a larger kernel image that exceeds their memory and storage constraints. With `sysfs` we came across the issue of not having any file-system–related kernel objects defined by `sysfs`. To use `sysfs`, we would have to design and implement a complete set of VFS kernel objects for `sysfs` and submit them for kernel inclusion in addition to Unionfs.

To solve our problem of using `ioctls` for branch manipulation, we decided to use the remount functionality that already exists in the kernel. Remount allows one to change the configuration of a mount while leaving its files open so processes can continue to use the files after the remount operation is complete. This lets us provide the ability to change branch configurations easily without the need for `ioctls`, by parsing the new options that are passed in and applying the differences between the new and old options. However, this still requires us to maintain two `ioctls` for querying files and another for debugging.

As of this writing, we are addressing a problem associated with crossing mount points within a union. The most common occurrence of this problem is when a LiveCD performs a pivot_root or a chroot to a Unionfs mounted path. Currently LiveCD developers mount Unionfs and then they proceed to move the mount points for /proc and /sys to /unionfs/proc and /unionfs/sys, respectively. After this they pivot_root to the union so that proc and sys are visible. The reason that this problem exists is that currently Unionfs stacks on top of the superblock for each branch. This presents a problem because it does not give us access to the data structures that permit us to cross mount points. Our solution to this problem is to redo how Unionfs stacks on top of the branches by stacking on a dentry and a vfsmount structure. This will give us the additional information that is needed to build the structures necessary to cross mount points. Even with the ability to cross mount points, it is not advised to stack on pseudo file systems such as sysfs and procfs. Since sysfs and procfs are not only accessed through Unionfs, but rather are also manipulated directly by the kernel, inconsistencies can arise between the caches used by Unionfs and these file systems.

Because Unionfs started as a research project, it had many theoretically interesting features from a design perspective, which users did not need in practice. Unionfs contains functionality for copyup, this occurs when a file that exists on a read-only branch is modified. When the file is modified Unionfs attempts to copy the file up to the nearest read-write branch. Some of the early features of Unionfs included several copyup modes, which allowed copyup to take the permissions of the current user, the original permissions of the file, or a set of permissions specified at mount time.

In addition, there were several delete modes which performed one of three actions:

- delete=whiteout (default) locates the first instance of the file and unlinks only that instance. This mode differs from delete=first in that it will create a whiteout for that file in the branch it removed the file from.
- delete=all finds every instance of the file across all branches and unlink them.
- delete=first located the first instance of the file and unlinked only that instance without creating a whiteout.

In the case of the delete mount option we found that no one was using the delete=first and delete=all options and that the delete=whiteout option was strongly preferred. Because our user base is predominantly composed of LiveCD developers, delete=first was removed and delete=all is only present if Unionfs is compiled with UNIONFS_DELETE_ALL defined.

We also had several modes to describe permissions with which a whiteout was to be created. When a file is deleted Unionfs will create a .wh.name file where name is the name of the file. This tells Unionfs that it should remove this file from the view presented to the user. These options were removed since we found that copyup=currentuser and copyup=mounter went completely unused by our users:

- copyup=preserve (default) creates the new file with the same permissions that existed on the file which was unlinked.
- copyup=currentuser creates the new file with the UID, GID, and umask taken from the current user.
- copyup=mounter creates the new file with UID, GID, and permissions specified in the options string of the Unionfs mount.

Although the extra options were interesting research concepts, they were not practical for what our users were using Unionfs for and only served to increase code complexity.

Another instance of where ideas that are good for research purposes fail in practice is in the creation of whiteouts. Initially, when a whiteout was created while removing a file, the whiteout was created atomically via `rename` and was then truncated. This was done so that if the process failed half-way through, there would not be any ambiguity about whether the file existed. This added additional complexity to the code without sufficient gains in either performance or functionality. Since then, we have removed atomic whiteout creation due to the inherent difficulty of maintaining the semantics of open, yet deleted, files.

4 Limitations

During the development of Unionfs, we had to make certain design decisions to help the overall implementation. Such decisions often impose limitations. We have identified three such limitations in Unionfs: modification of lower-level branches, `mmap` copyup with dynamic branch management, and scalability. We discuss each in detail below.

Modification of lower-level branches. The current design of Unionfs and other stackable file systems on Linux results in double caching of data and meta-data. This is an unfortunate side-effect of the way the Linux VFS is implemented—there is no easy coordination between objects cached at different levels [1]. This forces us to maintain a list of lower VFS objects for each upper object. For example, a Unionfs inode contains an array of pointers to all the corresponding inodes on the underlying branches. Unionfs has to copy certain information from the underlying inode for a file to the Unionfs inode: metadata information such as file size, access permissions, group and owner, and so on.

Since Unionfs expects the underlying inode (and therefore the file) to have certain properties about the file (e.g., have a size consistent with that saved in the Unionfs inode) it is possible for inconsistencies to appear if a process modifies the lower inode directly without going through Unionfs. We encourage our users to avoid modifying the lower branches directly. This works well in scenarios where many of the branches are stored on read-only media (e.g., LiveCDs). However, there are some people who want to use Unionfs to provide a unified view of several frequently changing directories. Moreover, if users delete or rename files or directories, then Unionfs points to the older object, again yielding an inconsistent view.

`mmap` copyup with dynamic branch management. When Unionfs was first implemented in early 2004, only a bare-bone functionality existed: the full set of system calls was not implemented. Some of these system calls, in particular `mmap`, are required for certain programs to function properly. The `mmap` system call allows programs to map portions of files into a process's address space. Once a file is `mmap`ped, a process can modify it by simply writing to the correct location in memory. Currently, Unionfs does not natively implement `mmap` operations, but rather passes them down unchanged to the lower-level file system. This has the advantage of preventing double caching of data pages and its associated performance and consistency pitfalls. However, this comes with the drawback that Unionfs does not receive notification of `readpage` or `writepage` calls, so it cannot perform copyup during a `commit_write`. The problem occurs when a process tries to modify a page backed by a file on a read-only medium. Just like in the regular open-for-write case, we must copyup the file to a writable branch and then perform the correct address space operations.

In March 2006, Shaya Potter, a Unionfs user and contributor, released a partial implementation of `mmap`. The major problem with it is the lack of copyup functionality while using `mmap`. Additionally, one has to be careful with the implementation since certain file systems (e.g., OCFS2, GFS) must take additional steps before calling `prepare_write` and `commit_write`. We have made this `mmap` functionality a compile-time option which is off by default.

Scalability. Although we do not consider it as serious as the two previous issues, the last issue is scalability. Even though most Unionfs users want two to six branches, there are some that want more. In its current state, the maximum number of branches that Unionfs supports is 1,024 due to the use of `FD_SET` and related macros. However, the overhead of using Unionfs becomes high with just 200 branches, even for simple operations such as `readdir` and `lookup` (see our evaluation in Section 5). The problem with these operations is that Unionfs needs to iterate through all the branches; for each branch it needs to determine whether or not it is a duplicate, whiteout, and so on. Currently, we are storing stacking information in a simple linear array. This structure, while easy to access and use, has a search complexity of $O(n)$.

Of course, there are other operations, such as `llseek` operating on directories, which should be examined and possibly optimized. For other operations, Unionfs is a bit more efficient because it can use the dentry cache objects that have been populated by lookup.

5 Evaluation

We conducted our benchmarks on a 1.7GHz Pentium 4 machine with 1.25GB of RAM. Its system disk was a 30GB 7,200 RPM Western Digital Caviar IDE formatted with Ext3. In addition, the machine had one Maxtor 7,200 RPM 40GB IDE disk formatted with Ext2, which we used for the tests. We remounted the lower file systems before every benchmark run to purge file system caches. We used the Student-t distribution to compute the 95% confidence intervals for the mean elapsed, system, user, and wait times. Wait time is the elapsed time less CPU time used and consists mostly of I/O, but process scheduling can also affect it. In each case, the half-widths of the confidence intervals were less than 5% of the mean. The test machine was running a Fedora Core 4 Linux distribution with a vanilla 2.6.16-rc6 kernel.

In all the tests, the highest-priority branch was read-write, while all the other branches, if any, were read-only. More detailed evaluation can be found in our journal article [10].

5.1 Evaluation Workloads

We chose to perform two benchmarks to test the extreme cases—on one end of the spectrum there are CPU-intensive workloads, while on the other end there are I/O-intensive workloads.

OpenSSH build. Building OpenSSH [8] is a CPU-intensive benchmark. We used OpenSSH 4.0p1, which contains 74,259 lines of code. It performs several hundred small configuration tests, and then it builds 155 object files, one library, and four scripts. This benchmark contains a fair mix of file system operations, representing a workload characteristic of users. The highest-priority branch was read-write, while all the other branches, if any, were read-only

Postmark. Postmark v1.5 simulates the operation of electronic mail servers [3]. It performs a series of file system operations such as appends, file reads, creations, and deletions. This benchmark uses little CPU but is I/O intensive. We configured Postmark to create 20,000 files, between 512–10,240K bytes in size, and perform 200,000 transactions. We used 200 subdirectories to prevent linear directory look ups from dominating the results. All of the branches were read-write, to distribute the load evenly across branches. This is because Postmark does not have an initial working set, therefore using read-only branches does not make sense for this benchmark.

5.2 Results

On average, Unionfs incurred only 10.7% maximum overhead over Ext2 on the OpenSSH compile, and 71.7% overhead over Ext2 on Postmark. These results are somewhat worse compared to our previous benchmarks [10]. However, the difference in the OpenSSH compile benchmark appears mainly in I/O wait time, which could be contributed to copyup taking place. We did not use copyup in our previous benchmark.

OpenSSH build. We performed the OpenSSH compile with two different layouts of the data. The first distributed all the files from the source code tarball over all the branches using a simple round robin algorithm. The other layout consists of a copy of the entire source tree on each branch. For both layouts, we have measured and plotted the elapsed, system, and user times.

When the OpenSSH source code is uniformly distributed across all the branches, the overhead is a mere 0.99% (Figure 5). This is due

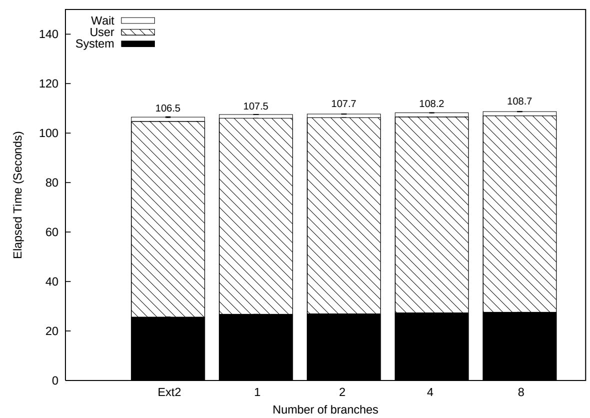


Figure 5: OpenSSH compile: Source code uniformly distributed across all branches.

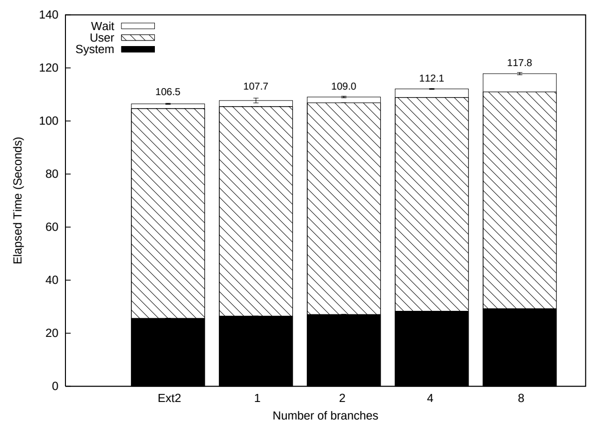


Figure 6: OpenSSH compile: Source code duplicated on all branches.

to the simple fact that we must perform several additional function calls before we hand of control to the lower file system (Ext2). With more branches, the overhead slightly increases to 2.1% with 8 branches. This shows that Unionfs scales roughly linearly for this benchmark.

With the OpenSSH source code duplicated on all branches (Figure 6), the overheads were slightly higher. A single branch configuration incurred 1.2% overhead. The slight increase in time is a logical consequence of Unionfs having to check all the branches, and on each branch dealing with the full source code tree which slows down linear directory lookups.

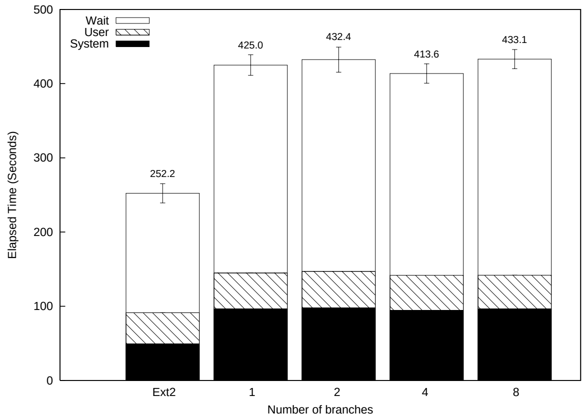


Figure 7: Postmark: 20,000 files and 200,000 transactions.

The 8-branch configuration increased runtime by 10.7%. As with the previous data layout, Unionfs scales roughly linearly.

Postmark. Figure 7 shows the elapsed, system, and user time for Postmark. The elapsed time overheads for Unionfs are in the range of 64.0–71.7% above that of Ext2. Since Postmark is designed to simulate I/O intensive workloads, and all the VFS operations have to pass through Unionfs, it is not surprising that the overhead of Unionfs becomes apparent. Fortunately, typical user workloads are not I/O bound and therefore one will not notice much performance degradation as shown by the OpenSSH compile benchmarks.

6 Conclusion

It is often easy to create a piece of software whose functionality is enough for the authors. However, that functionality is usually a subset of that required by real users. Since the first release in early 2004, user feedback has helped us make Unionfs more complete and stable than it would have been had a small team of developers worked on it without any community feedback. Our users have used Unionfs for applications that were not even considered back when Unionfs was originally designed, and located bugs that would otherwise have gone unnoticed.

For quite some time, Linux users wanted a namespace unifying file system; Unionfs gives them exactly that. While there are still several known issues to deal with, Unionfs is steadily becoming a polished software package. With the increasing use and popularity of Unionfs we felt that the next logical step was to clean up Unionfs and submit it for kernel inclusion.

7 Acknowledgments

Unionfs would not be in nearly as good a state if it was not for our user community, which has submitted bug reports, patches, and provided us with interesting use cases. There are far too many contributors to list individually (there are over 37 listed in our `AUTHORS` file, which only includes those who have submitted patches), and we extend thanks to all contributors and users, named and unnamed. Early adopters and bug reporters like Tomas Matejicek and Fabian Franz helped immeasurably. Recently, Junjiro Okajima has fixed many bugs and can be counted on for high-quality patches. The experimental `mmap` code currently in Unionfs was contributed by Shaya Potter. Jay Dave, Puja Gupta, Harikesavan Krishnan, and Mohammad Nayyer Zubair worked on Unionfs in its early stages.

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

Unionfs is released under the GPL. Sources and documentation can be downloaded from `http://unionfs.filesystems.org`.

References

[1] J. S. Heidemann and G. J. Popek. Performance of cache coherence in stackable filing. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 3–6, Copper Mountain Resort, CO, December 1995. ACM SIGOPS.

[2] Inside Security IT Consulting GmbH. Inside Security Rescue Toolkit. `http://insert.cd`, 2006.

[3] J. Katcher. PostMark: A New Filesystem Benchmark. Technical Report TR3022, Network Appliance, 1997. `www.netapp.com/tech_library/3022.html`.

[4] P. Kerr. m-dist: live linux midi distribution. `http://plus24.com/m-dist/`, 2005.

[5] K. Knopper. Knoppix Linux. `www.knoppix.net`, 2006.

[6] P. Lougher. SQUASHFS - A squashed read-only filesystem for Linux. `http://squashfs.sourceforge.net`, 2006.

[7] T. Matejicek. SLAX – your pocket OS. `http://slax.linux-live.org`, 2006.

[8] OpenBSD. OpenSSH. `www.openssh.org`, May 2005.

[9] J. Silverman. Clusterix: Bringing the power of computing together. `http://clusterix.net`, 2004.

[10] C. P. Wright, J. Dave, P. Gupta, H. Krishnan, D. P. Quigley, E. Zadok, and M. N. Zubair. Versatility and unix semantics in namespace unification. *ACM Transactions on Storage (TOS)*, 2(1):1–32, February 2006.

[11] C. P. Wright and E. Zadok. Unionfs: Bringing File Systems Together. *Linux Journal*, (128):24–29, December 2004.

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