5-2010

Online Layered File System (OLFS): A Layered and Versioned Filesystem and Performance Analysis

Joseph P. Kaylor

Konstantin Läufer
Loyola University Chicago, klaeufer@gmail.com

George K. Thiruvathukal
Loyola University Chicago, gkt@cs.luc.edu

Recommended Citation

Joe Kaylor, Konstantin Läufer, and George K. Thiruvathukal, Online Layered File System (OLFS): A layered and versioned filesystem and performance analysi, In Proceedings of Electro/Information Technology 2010 (EIT 2010).

Online Layered File System (OLFS): A Layered and Versioned Filesystem and Performance Analysis

Joe Kaylor, Konstantin Läufer, and George K. Thiruvathukal
Loyola University Chicago
Department of Computer Science
Chicago, IL 60640 USA

***Abstract*—We present a novel form of intra-volume directory layering with hierarchical, inheritance-like namespace unification. While each layer of an OLFS volume constitutes a subvolume that can be mounted separately in a fan-in configuration, the entire hierarchy is always accessible (online) and fully navigable through any mounted layer.**

OLFS uses a relational database to store its layering metadata and either a relational database or any (virtual) host file system as its backing store, along with metadata and block caching for improved performance. Because OLFS runs as a virtual file system in user-space, its capabilities are available to all existing software without modification or special privileges. We have developed a reference implementation of OLFS for FUSE based on MySQL and XFS, and conducted performance benchmarking against XFS by itself.

We explore several applications of OLFS, such as enhanced server synchronization, transactional file operations, and versioning.

I. Introduction

Virtually all software applications involve some form of data management. Unix [6] and other modern Unix-like operating systems (OS) support this requirement through the concept of the file system, which includes abstractions such as files, directories, symbolic links, block devices, character devices, pseudo devices, pipes, and sockets. These abstractions allow applications to interact with each other, with the OS, and with the underlying hardware through standard interfaces.

Virtual and Stackable File Systems: Originally, file systems were tightly integrated into the OS, making it difficult to add new functionality or modify existing functionality related to the file system. *Virtual* file systems [7] employ the *vnode* (virtual node) as a common layer of abstraction on top of specific file systems, which makes it possible to add new types of file systems to a system without modifying the OS core. *Stackable* file systems take further advantage of this abstraction by allowing vnodes to be stacked on top of each other in such a way that higher vnodes intercept file system operations on those one below them;[1] in this way, file system functionality can be modified or enhanced. Zanok et al. [20] discuss this approach and its ramifications in detail. In many operating systems, the vnode abstraction has been used to implement user mode file system frameworks such as FUSE [16].

Namespace Unification: Unix supports the ability to mount external file systems from external resources or local block devices into the main file system hierarchy. Some Unix-like operating systems even allow for mounting an external resource to multiple places, and *chroot* can be used to present a different file system namespace to an application. Conversely, it is often desirable to combine files from different physical or logical locations in such a way that they appear to reside in a single directory from a user's or application's perspective. This capability of combining multiple file-system namespaces into a single one is called *namespace unification*.

The 3-D File System (3DFS) [8] provides *viewpathing*, which involves a process-specific list of directories that appear in unified form in a designated directory. 3DFS is implemented non-transparently as a user-level library, intended primarily to support namespace management for build environments [4].

The Plan 9 distributed operating system [12] allows additional resources and protocols to be presented in the file system namespace. Plan 9 includes three file system calls: *mount*, *unmount*, and *bind* [13]. Mount allows a program to attach a namespace from a file server to the current name space, similarly to mounting a file system on a Unix system, while unmount detaches a namespace. Bind allows a program to define an ordered union of two folders in the namespace hierarchy. In this system, the creation of new folders or files is by default not permitted, but if the create permission (bit) is set when the bind or mount call is made, then the lookup order is used to determine which folder will accept the new folder or file. This allows for the creation of files and folders in a private directory without affecting other parts of the namespace.

In 4.4BSD-Lite, Union Mounts [11] constitute a kernel-level mechanism to arrange multiple file system mounts in a linear hierarchy. Changes or files in the lower part of the hierarchy (closer to the client) would override files in the higher part of the hierarchy. In this system, when a directory is created at the bottom of the hierarchy, a shadow directory is created in the highest part of the hierarchy and is then inherited down through the hierarchy; this directory remains a part of the topmost file system and not a part of any other file system. When files are deleted in the lower layers, they are copied up to the higher layers, or if they exist in the higher layers

[1] The term "layered" is also used for this architecture, but we use "stackable" to avoid confusion with "directory layering" as discussed in this paper.

already, they are marked in the higher layers so that they do not appear in the lower layers. This system did not ensure cache coherency among the layers: if a change occurred in a higher layer, the change was not necessarily visible in the lower layers of the hierarchy.

In Linux, UnionFS [18] provides a similar kernel-level namespace unification mechanism as Union Mounts, but with greater preservation of Unix file system semantics and support for more flexible fan-out configurations, where the strictly linear hierarchy gives way to the ability to access all branches present in any order through a single node.

Versioning and Snapshotting: Versioning is the capability to keep track of changes to objects in a file system for a variety of applications including backups and disaster recovery. This capability can stem from version control applications [9], [2] or backup tools, both of which have to be invoked explicitly. Instead, this capability can also be built into the file system, and, given the ease of accidental file deletion in Unix, the lack of built-in support for versioning in Unix has long been considered a significant omission.

Hewlett-Packard's OpenVMS [1] operating system includes a versioning capability in its file system known as On-Disk Structure (ODS). In ODS, every file has a version attribute expressed as part of the file name. Each time a file is saved, the original file is preserved, and the new data is written to a new file with an incremented version number in its file name. File names for different versions of the same file are generated as `file.txt;1`, `file.txt;2`, and so on.

There have been various attempts to provide versioning for Unix-like operating systems at the block level [17] or at the file system level [5], [15], [14]. Most of these, however, are disk-based rather than stackable and, thus, cannot be combined with other existing file systems. Several recent systems [10], [3], on the other hand, are stackable and can provide versioning for any existing file system.

Snapshotting refers to the occasional capture of the current state in the history of a file system. Although it is less fine-grained than full versioning, it is useful in that the resulting snapshots are usually available online. It is possible to achieve snapshotting through namespace unification by marking the main resources as read-only and using copy-on-write to record the changes to a designated writable directory that is part of the union [18].

A. OLFS and its Goals

In this paper, we present the Online Layered File System (OLFS), which provides a novel form of intra-volume directory layering with hierarchical, inheritance-like namespace unification. While each layer of an OLFS volume constitutes a subvolume that can be mounted separately in a fan-in configuration, the entire hierarchy is always accessible (online) and fully navigable through any mounted layer. OLFS's builds upon existing work with versioning, layering, and file system namespaces to make two contributions over existing work. The first contribution is the way OLFS uses the folder structure of the file system to control the layering structure and the use of special files to expose metadata about those layers. Because OLFS uses folders and files, no new system calls are required and no software needs to be rewritten to take advantage of OLFS. Other systems require the use of mount time options[18], [11] to control the structure of the layering which can prevent changes to the layering structure while the file system is in use. Other systems require new system calls in the operating system or special libraries to take advantage of layering features in the file system [13]; this requires existing software to be modified to take advantage of these features. The second contribution OLFS makes is its performance. OLFS is able to introduce layering into the operating system with only a small amount of overhead. This is particularly significant because of the challenges in achieving high performance with user mode file systems due to the additional context switches required for each operating system call into the file system.

OLFS has been constructed with several design goals in mind.

No new API or tool set : The layering capability should be available to all existing software without modification. Even if a new system is better than an older system, there is a replacement cost. Rewriting or enhancing every piece of software to work with improved file system namespace semantics if done through a new system call interface or tool set would be prohibitively expensive. The interaction with the file system and with the layering mechanism is done entirely through the creation and deletion of files and folders.

Accessible and modifiable to all users : Other than mounting the file system, no special privileges should be needed. If a user wants to create their own private layer, then they should be able to do so as long as parent layer permissions permit. No privileged system calls are needed.

Inheritance : Changes made in higher layers should be available to lower layers. If a new folder is created in a higher layer or a file is renamed, if file contents are updated, or if some folder or file is deleted these changes should be visible to the lower layers.

Overriding : Lower layers should be able to override the content of higher layers. If a lower layer alters a file from the parent layer then a copy should be made in the lower layer that is separate from the parent layer. Merging changes from higher to lower layers is costly because the correct way to merge any change depends greatly upon the programs manipulating and reading a given file.

Fan-in branching: Multiple layers should be able to be children of a given parent layer. It should be possible for instance to have Java byte code in a parent layer and in two child sibling layers to have different native libraries for different machine architectures.

Online operation: All layers are accessible at all times through the file system namespace for reading and writing. If a file is overridden in a lower layer, the file should be able to be read from and written to in both the higher and lower layers. While the file system is in use, the layer structure should be modifiable.

User mode : To allow for the file system to be more readily ported to several operating systems, improve system reliability and recoverability. The user mode interface provided

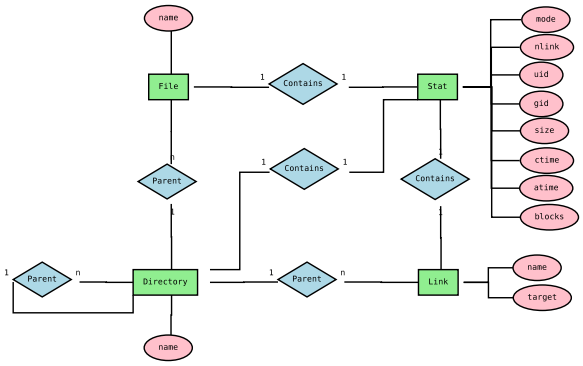

Figure 1. An E/R diagram for a typical tree-structured file system, e. g. Unix

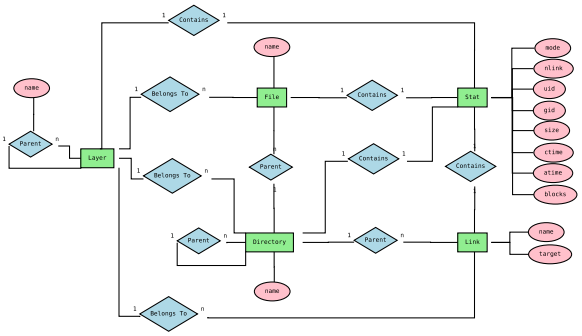

Figure 2. An E/R diagram for a layered file system like OLFS

by the FUSE framework [16][16] allows for a file system to be ported to several operating systems that provide a user-space file system library. Currently, this is limited to Unix and Unix-like operating systems. User-mode operating system services by design provide for greater system stability than do kernel mode services. If the file system were to crash, it could be quickly restarted without affecting the entire operating system and other users.

Preservation of Unix semantics: As part of their detailed comparison of namespace unification file systems, Wright et al. [19, p. 92] present a list of 18 features including several aspects of Unix semantics. OLFS supports most of these features and is generally compliant with Unix semantics except for whiteout deletion, a mechanism for marking a file as deleted in a child layer in such a way that it can continue to exist in a parent layer. We plan to support this capability in the near future.

II. File System Organization

In this section, we describe OLFS's concepts and architecture. There are six fundamental objects in OLFS: files, directories, links, layers, stats, and blocks. The relationship of the objects in OLFS is similar to conventional file systems except for the organization of the layers.

A normal file system is constructed as a hierarchy of directories with files and links as leafs in the hierarchy. In a conventional file system directories, files, and links contain metadata describing their size, security, modification, access, and creation information. The following entity relationship diagram describes such a file system:

In OLFS, the normal file system hierarchy is augmented with layers. Files, links, and directories belong to a separate hierarchy of layers in addition to the conventional directory hierarchy. The following entity relationship diagram describes the OLFS layering architecture:

The combined layer and directory hierarchy is presented to the system as a hierarchy of directories. The root of the file system is the current layer root's directory hierarchy. In the root of the file system is a specially handled `/layers` folder that contains the absolute root of the layer hierarchy and presents as a directory hierarchy the layer structure. By adding, removing, moving, and renaming folders under the `/layers` folder, the system can alter the layering hierarchy. Each folder added to the layers special directory represents a new layer. The following is a high-level description of the layering semantics:

1) When a layer folder (named `layerA` for example) is created three items (or objects) are simultaneously created:
   - A folder named `layerA` which represents the actual layer object. Folders under the `layerA` folder are child layers and the folder that contains `layerA` is the parent layer.
   - A folder named `layerA.data` contains the directory hierarchy that belongs directly to `layerA` or which is inherited by any parent of `layerA`. Because of the use of the trailing `.data` characters in the name, layers that end with names of `.data` are not permitted.
   - A read only file named `layerA.xml` which contains XML that describes the layering graph and which file system objects belong to which layer. This is meant to be consumed by enhanced file browsers to present to the user which files, directories, and links belong to which layer.
2) No layer folder can be removed unless all of its child layers are deleted.
3) All directories, files, and links that exist in a parent layer are inherited in child layers.
4) File modifications made to files from a parent layer in a child layer follow copy-on-write semantics for the entire file. If a change is made to a file in a child layer, it is considered to be overridden meaning that changes made to the file in the child layer are not visible in the parent layer and further changes to the file in the parent layer are not reflected in the child layer.
5) Folders `cannot` be removed unless all files, links, and directories in the current and all child layers are removed.
6) The layer graph is directional and acyclic.

The following is an example of an OLFS file system diagram:

This file system contains two layers: the root layer and a child layer called `layerA`. The root layer contains two files (`fileA` and `fileB`) and a folder (`folderA`); the folder contains a single file (`fileC`). The child layer of the root layer (`layerA`) inherits `fileA` and `folderA` from the root layer. The contents of `fileB` have been modified in `layerA` so `layerA` has an independent copy of `fileB`. In `folderA`, `fileC` is

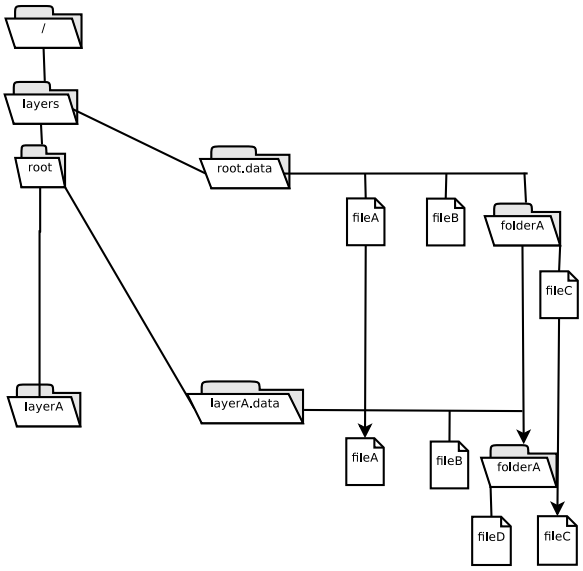

Figure 3. An OLFS instance showing the key abstractions

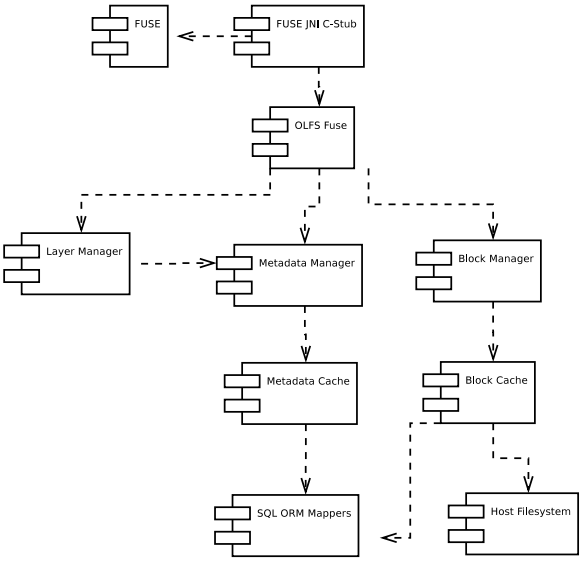

Figure 4. OLFS Architecture

inherited from the root layer and a new file (`fileD`) belongs to `layerA`. If the file system is mounted with `/layers/root` as the mounted layer, then the files and folders available under the mount point would be the same as the files and folders available under the `/layers/root.data` folder. Similarly, if the file system is mounted with `/layers/root/layerA` as the mounted layer, then the files and folders available under the `/layers/root/layerA.data` folder.

A. File System Design and Architecture

OLFS is composed of ten major components: FUSE, the FUSE JNI stub, OLFS FUSE, the layer manager, metadata manger, block manager, metadata cache, block cache, SQL mappers, and the host file system.

The FUSE library [16] is responsible for passing calls to and from the Linux VFS into the OLFS FUSE JNI Stub. The JNI stub marshals the calls into the main Java interface exposed by OLFS. The main interface is composed of the regular FUSE functions such as chmod, chown, open, read, write and others.

The layer manager is responsible for translating all paths passed into OLFS into actual file system objects. The layer manager handles the translation of a path while taking into account the inheritance and override semantics, and the presentation of the special layer and data folders. The layer manager relies directly upon the metadata manager to obtain information about layers, files, and folders. The main lookup algorithm is discussed in the next section.

The metadata and block managers with their respective caches coordinate the reading and writing of file system metadata and block data. The block manager is responsible for translating read and write calls to reads and writes to underlying blocks in the file system. The metadata manager handles the lookup of file, folder, and layer objects from the cache. Both of the cache components rely upon a backing store to obtain their data. The metadata cache uses a SQL server as its backing store and the block cache can either use a SQL server or a host file system as its backing store.

OLFS is implemented almost entirely in Java. The only non-Java code in OLFS is for the JNI stub for the FUSE library. The Java platform was chosen because of the rich set of development tools available for the platform, for portability, and support for the architecture choices made for OLFS . With the Java platform, it is possible to run OLFS on any platform that supports FUSE and has a compatible Java runtime. Java was also chosen because OLFS is architected as an object oriented domain model with a service layer and a mapping layer. Java's support for object oriented development is very appropriate for the OLFS domain model. In addition, Java's JDBC support allows the OLFS mapping layer to be portable across database vendors.

The choice of building OLFS on the Java platform with a domain model made it much simpler to practice test driven development throughout the development life cycle. The first stable version of OLFS contained just over 25,000 lines of Java code and over 400 unit tests. The use of TDD during the development of OLFS made development faster and refactoring much simpler.

In addition to the build of OLFS for the Linux platform, we were able to port OLFS to Windows using the Dokan user mode file system framework.

B. Layer Lookup Algorithm

The second possibility is to build a shell namespace extension into the Windows shell. Shell extensions are currently used to augment the Windows file system namespace by adding the special My Computer and Network Places folders in the Windows Explorer. Because of compatibility issues with different versions of Windows, existing managed languages, and documentation, we did not use a shell extension to port OLFS to Windows.

The algorithm for resolving a UNIX path to a file, link, or directory object in the file system is demonstrated by the following pseudo code in Algorithm 1.

Algorithm 1 Main OLFS path lookup algorithm

```
    OLFS Tuples:

    File:   { layer, parent, name, stat }
    Folder: { layer, parent, name, stat, children }
    Layer:  { parent, name, stat }
    Stat:   { mode, uid, gid, size, blocks,
              atime, mtime, ctime }

    lookup(path, mounted_layer, root_layer,
       root_folder) {
       let N = null
       let R = root_folder
       let C = root_layer
       let M = mounted_layer

       for each (path_seg P in path)
            N = lookup_child_in_layer(R, C, P)
            while(N IS NULL AND C IS NOT M)
                C = child_layer(C, M)
                N = lookup_child_in_layer(
                    R, C, P)

            if (N IS NULL)
                raise File Not Found exception

            let N2 = N
            while (N IS NOT NULL AND C IS NOT M)
                let C2 = child_layer(C, M)
                N2 = lookup_child_in_layer(
                     R, C2, P)
                if (N2 IS NOT NULL)
                    C = C2
                    N = N2

            if (N IS A Folder)
                R = N
            else if (P IS NOT P.last_segment)
                raise Malformed Path exception

       return N
    }
```

Algorithm 2 OLFS path lookup auxiliary functions

```
child_layer(current_layer, mounted_layer) {
    let C = mounted_layer
    while (C.parent IS NOT current_layer)
        C = C.parent
    return C
}

lookup_child_in_layer(parent_folder,
    current_layer, name) {
    let L = current_layer
    let P = parent_folder
    let S = name
    return P.children[S,L]
}
```

The complexity of the `lookup()` algorithm is $O(nm)$ where n is the number of folders in the path and m is the number of edges from the root vertex to the layer vertex of the layer hierarchy used in the lookup. The complexity of `child_layer()` is $O(m)$ where m is the number of edges from the current vertex to the end vertex in the layer hierarchy. In general, OLFS can be expected to perform better when the layer hierarchy is not very deep ($m \ll n$) although caching strategies for recently used layers can also allow the $O(m)$ factor to be eliminated in practice.

The complexity of `lookup_child_in_layer()` depends upon the data structure used to store the data. In the research prototype the folder, layer and file structures are held in hash maps for fast lookups. With a hash map data structure the complexity of the lookup is $O(1)$ with a cost of $O(n)$ to build or precompute the hash map where n is the number of files, folders and layers.

C. Performance - Architecture

User mode file systems such as OLFS have many advantages such as stability, error recovery, and portability. Performance, however, can be a challenge in user mode file systems. In kernel mode file systems, calls to open, read, write, and other system calls are marshaled directly to file system code. In Linux, system calls to files are passed to the VFS subsystem and then are answered by the file system cache or by an implementation of file system code. In user mode file systems implemented through FUSE in Linux, system calls are sent to the kernel, marshal led to the VFS subsystem and then through FUSE VFS code and finally through a device to the user program implementing the file system. With a call to a FUSE file system more context switches must be made which can affect latency and throughput.

OLFS answers the performance challenges of a user mode file system in two important ways: a metadata cache, and a block cache. Since there are more context switches from a user program to a user mode file system than to a kernel mode file system it is important to minimize the number of context switches caused by the user mode file system itself. The block store for OLFS can be either a SQL server or another file system, the metadata store for OLFS resides in a SQL server. When a call is made into OLFS by FUSE, if the call is directed immediately to the block store or metadata store, another context switch must be made either to read a file off an underlying file system or to make a networking call to a SQL server. Direct calls to these stores increase the number of context switches made by OLFS. The use of a metadata and block cache by OLFS reduces the number of these context switches by allowing OLFS to answer frequently made calls with an in memory cache. The block cache in OLFS is implemented as an LRU block cache. The metadata cache is an LRU cache on the file level.

D. Performance Comparison: XFS and OLFS

The performance comparison was performed on a dual core AMD machine with 4 gigabytes of RAM running Linux kernel

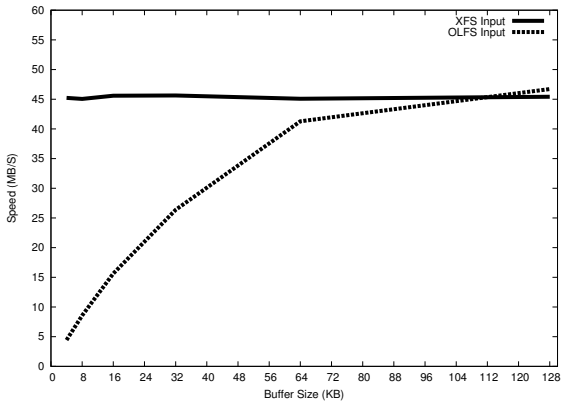

Figure 5. Read performance comparison of XFS and OLFS for different buffer sizes

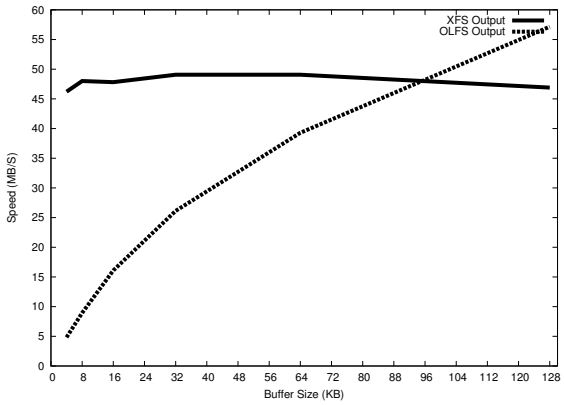

Figure 6. Write performance comparison of XFS and OLFS for different buffer sizes

version 2.6.24 compiled for 64 bit. OLFS was tested on the same machine using MySQL Server 5.0 as the backing store for the file system metadata and the same XFS local file system as the backing store for blocks. OLFS was run using the Sun Java 1.6.0_07 64 bit virtual machine.

The first test performed on both file systems was to write an 8 gigabyte file to the file system using buffer sizes of 4kb, 8kb, 16kb, 32kb, 64kb and 128kb. The write speed was determined by dividing the size of the file written by the number of seconds each test took. The test was repeated ten times for each file system. For XFS, the write performance for each of these buffer sizes was fairly consistent. For OLFS, performance was poor for small buffer sizes, but much better for larger buffer sizes of 64kb and 128kb. With a buffer size of 128kb, OLFS was able to perform better than XFS; for all other buffer sizes, the performance was worse. The standard deviation for the XFS tests was in the range of 2.5 MB/s to 5.86 MB/s; the standard deviation for OLFS was in the range of 1.41 MB/s to 6.35 MB/s.

The second test was the same as the first, except that it read the 8 gigabyte file from the file system with the same buffer size selection. The read speed was determined by dividing the size of the file read by the number of seconds each test took. The test was repeated ten times for each file system. As with the read test, the performance for XFS was fairly consistent

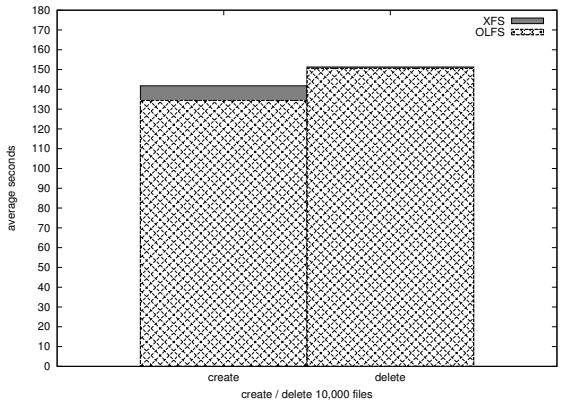

Figure 7. Metadata performance comparison for XFS and OLFS

across buffer sizes. OLFS had poor performance for smaller buffer sizes and similar performance to XFS for buffer sizes of 64kb. The standard deviation for the XFS tests was in the range of 3.00 MB/s to 3.87 MB/s. The standard deviation for the XFS tests was in the range of 1.24 MB/s to 10.78 MB/s.

Metadata performance was measured by recording the time to create ten thousand files and to delete ten thousand files. XFS was able to create ten thousand files in an average of 141.8 seconds and delete ten thousand files in an average of 151.3 seconds. OLFS performed better for both tests. OLFS was able to create ten thousand files in an average of 134.4 seconds and delete ten thousand files in 150.6 seconds.

III. Applications

OLFS and the architecture that it presents has several possible uses in existing software environments. Since OLFS presents its layering feature to the system through the manipulation of file system objects, major changes to existing software are not needed to take advantage of the architecture. Several immediate possibilities exist for applications of OLFS.

A. Server Synchronization

With OLFS it is possible to make existing backup and mirroring systems such as rsync generational, reliable, and have shorter down times. The purpose of this demonstration is to show real world performance statistics for OLFS and to demonstrate how an existing tool can be enhanced by OLFS.

rsync is a UNIX application that is used to synchronize files and directories for backup and mirroring systems. As it is currently implemented, rsync does not allow the user to create generational backups. If a user invokes rsync to backup their current files, the backup destination is overwritten with the new backup and the old copies of files are lost unless they are archived to a different location or medium before the rsync backup is invoked. Also, if a backup operation by rsync is interrupted by some software or hardware failure (disk failure, power failure, software crash), then the destination would have a partial backup. In the case of a failure mid-backup, the complete old backup may be more important than an old backup that is partially overwritten with a new backup. Rsync also has the disadvantage of not taking a backup as a

snapshot which can be required if the user needs to restore files to a point in time; a backup that takes several hours to complete may contain several files that were subsequently modified and some that were not. Also, by not taking the backup as a snapshot rsync can interfere with normal file operations. If an application is currently creating or writing to a file, rsync will not be able to read from that file and will fail to back it up. To get a complete snapshot of a set of files, those files must not be in use during the backup and measures need to be taken to ensure that the backup does not fail (redundant hardware, power supplies, etc..).

OLFS can augment the feature set that rsync provides with the layering architecture. This can be accomplished without altering rsync and by only performing a few simple steps before a backup operation starts. To take advantage of OLFS, the user creates a child layer for the currently mounted layer by creating a new directory in the `/layers` folder and then re-mounts the file system with the new child layer as the mounted layer. Next, the user begins the normal rsync operation except he or she uses the `/layers` special folder as the base for the backup operation instead of a path from the root of the file system. For example, if the user wanted to backup `/home/user1` that exists in `/layers/root/layerA`, she should create a new layer `/layers/root/layerA/layerB` and remount it as their current layer. Then the user would begin their backup operation from `/layers/root/layerA.data/home/user1` instead of `/home/user1`.

While the backup operation is occurring in the parent layer, the user can continue to read, write, and create new files and folders in the file system. The backup that will be taken will be a snapshot of the files at the time the user created the new layer and re-mounted the file system because any changes that are made in the mounted child layer will remain in that child layer. If a hardware or software failure occurs during the operation of rsync, the operation can be resumed reliably because a point in time copy is preserved.

The backup can be made to be generational by using OLFS at the backup destination. Before a new backup is started, a new layer folder can be created and rsync can copy the files into the layer's data folder without transmitting the entire file because the parent layer's files are inherited in the new layer. If a user needs to access a backup of a certain generation, they need only access a different folder in the layer structure. Also, if a failure happens during an rsync backup at either end of the operation, the older copy is still intact and accessible.

OLFS can also improve the availability time of mirrors during rsync operations. Before beginning a mirroring operation, the user can create a new layer as a child layer of the currently mounted layer and run the rsync operation into the new child layer. While the mirroring operation is being performed into the new child layer, the mirror can still serve files out of its current layer for the full duration of the backup operation. Without OLFS, the mirror may have been taken offline to perform the mirroring operation and likely have only occurred during periods of low traffic. With OLFS, as soon as the mirroring operation is finished in the child layer, the file system can be re-mounted with the child layer as the mounted layer and the new files can be served by the mirror.

Other file systems may offer live snapshots and backup systems (such as Volume Shadow Copy in Microsoft's NTFS), but those systems require the applications that take advantage of those architectures to be integrated with the API or tool set that comes with each implementation. With other implementations, current software must be rewritten or augmented. With OLFS, software does not need to be rewritten because the layering architecture is exposed directly in the file system.

B. Server Synchronization Performance

The performance comparison for server synchronization was performed using two dual core machines with 4 GB of RAM running Linux kernel version 2.6.28.1 and using rsync version 2.6.9. The test was to use rsync to perform a full tree copy of the Linux kernel source code at version 2.6.0 and incrementally update it with rsync through versions 2.6.5, 2.6.10, 2.6.15, 2.6.20, and 2.6.26. The source code tree for 2.6.0 contains 15,007 files is 167 MB in size and has an average file size of 11.4 KB. The test was repeated 25 times for XFS and for OLFS

The full tree copy of the version 2.6.0 source took 101.16 seconds on average for XFS and 199.8 seconds on average for OLFS. The first incremental copy to version 2.6.5 took 244.52 seconds for XFS and 351 seconds on average for OLFS. The second incremental copy to version 2.6.10 took 287.72 seconds on average for XFS and 351 seconds on average for OLFS. The third incremental copy to version 2.6.15 took 287.72 seconds on average for XFS and 393.44 seconds on average for OLFS. The fourth incremental copy to version 2.6.20 took 333.76 seconds on average for XFS and 456 seconds on average for OLFS. The final incremental copy to version 2.6.26 took 373.52 seconds for XFS and 505.12 seconds for OLFS. The full tree copy performance for OLFS was 50.63% as fast as XFS and the incremental copies averaged between 73.04% and 75.26% performance.

Overall these numbers are very good for OLFS because of the small average file size transferred and because of the even smaller average buffer size that rsync uses since it only sends over differences. The smaller buffer sizes cause a higher ratio of context switches per unit of data for OLFS than for XFS. Even in this scenario, OLFS was able to provide better than 70% of the performance that XFS provided in terms of transfer speed. Even though the transfer speeds for OLFS are longer, the downtime for the server will not be since OLFS can still provide the entire previous version during the incremental copy. Only a short period of time is needed to make the switch to the new version.

C. Transactional file operations

In traditional file systems, the system call is guaranteed to be atomic. Calls to methods on the file system such as write(), read(), and open() complete their operation completely or return an error code. If a program needs to make a set of read, write, or create operations on several files and folders, but crashes or is otherwise interrupted in its operation before completion the files and folders may be left in an inconsistent

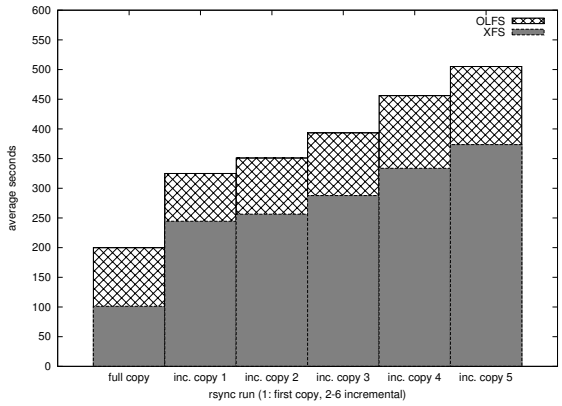

Figure 8. Rsync Performance Comparison

or even an unusable state. There is no mechanism in traditional file systems to make a series of file system operations atomic.

Work has been done to introduce ACID semantics to file systems in Linux such as was done with Amino [19]. Amino allowed for ACID transactions by intercepting the file system calls and using Berkley DB to provide the ACID semantics. With layering support, OLFS can provide transactional file operations to existing applications. For example, before a complex operation is performed, a new layer can be created on the file system and the operations can be performed on the files in the child layer either by remounting the file system to the child layer, using a chroot like mechanism, or by directing the operation to the new layer's path. If the program or programs performing the operation fail, the layer can be destroyed and the original data in the parent layer would still be in its original, consistent state. If the changes to the files are acceptable, then the files in the child layer could then be copied into the parent layer. The process of copying the files from the child layer to the parent layer could be interrupted by a system failure or software crash which can lead to an inconsistent state, but the copying operation can be resumed after the system recovers.

This transactional support can be used by existing applications without modification to those applications, but some external coordination in creating the layers and copying the results from a child layer to a parent layer would have to take place. Other applications could possibly read from and write to another application's layer, but if each application using the layers for transactional support are isolated with a chroot call, they can be effectively isolated.

IV. Future Work

OLFS is a fully functional and robust prototype at this stage. We know, however, that there are a number of areas for improvement, both in terms of making a production-quality file system (that anyone can use) and future research. The following are aspects that we will continue to explore:

Parent layers: A /parent special folder would be greatly helpful for increased usability in traversing the layer hierarchy, similar to the role that a parent directory presents in a file system more generally.

Whiteout deletion: Deletion is messy business. Deletes that cascade into other layers certainly has the potential to confuse or irritate casual users. There needs to be an option for whiteout style deletion. The first delete, for example, would delete the override. The second delete would perform a whiteout operation.

Extended attributes: We currently expose layer metadata as a special file (i.e. `/layers/root/layerA.xml`). While useful, especially for developing and maintaining OLFS, the cold reality is that most of the file system utilities are not designed to take advantage of it.

Atomic layers: For many tasks, the capability to make contents of a layer all read only or locked would be a key enabler. We know from tools like rsync and similar mirroring/backup utilities that the contents of the target can but should not be modified during the entire process. The ability to support atomic layering fully would allow an owing PID (or tree of PIDs) to have exclusive access to the layer. There are significant issues pertaining to usability here, however, including the development of appropriate mount options.

V. Conclusion

The Online Layered File System (OLFS) represents a new approach to intra-volume file system layering through the use of the file system namespace. With OLFS layers, file system objects can be inherited from parent layers or overridden in child layers. Because the layering system is exposed through the file system namespace, all existing software can take advantage of the layering system without any modification. The layering system allows for several interesting applications such as server synchronization, file versioning, and transactional file operations.

The OLFS file system was built as a user-space file system using the FUSE library and written in the Java programming language. These architecture choices allow OLFS to run on many operating systems without modification. As a user-space file system, OLFS is able to provide its feature set with I/O performance close to a native file system for buffer sizes above 64KB and with 70% of native I/O speed. These performance numbers validate the decision to build the file system with FUSE and the Java platform, which we know is not along the critical path for performance in this research.

With the performance numbers for OLFS and its compatibility with existing software and operating systems, we believe that OLFS can be used in and improve existing systems and applications.

References

[1] L. L. Baldwin, Jr. *OpenVMS system management guide*. Digital Press, Newton, MA, USA, 1995.

[2] B. Collins-Sussman. The Subversion project: buiding a better CVS. *Linux J.*, 2002(94):3, 2002.

[3] B. Cornell, P. A. Dinda, and F. E. Bustamante. Wayback: a user-level versioning file system for Linux. In *ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 27–27, Berkeley, CA, USA, 2004. USENIX Association.

[4] G. Fowler. A case for make. *Softw. Pract. Exper.*, 20(S1):30–46, 1990.

[5] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *WTEC'94: Proc. USENIX Winter 1994 Technical Conf.*, pages 19–19, Berkeley, CA, USA, 1994. USENIX Association.

[6] B. W. Kernighan and R. Pike. *The UNIX Programming Environment*. Prentice Hall Professional Technical Reference, 1983.

[7] S. R. Kleiman. Vnodes: An architecture for multiple file system types. In *Proc. Summer USENIX Technical Conf.*, pages 238–247, 1986.

[8] D. G. Korn and E. Krell. A new dimension for the UNIX file system. *Softw. Pract. Exper.*, 20(S1):19–34, 1990.

[9] T. Morse. CVS. *Linux J.*, 1996(21es):3, 1996.

[10] K.-K. Muniswamy-Reddy, C. P. Wright, A. Himmer, and E. Zadok. A versatile and user-oriented versioning file system. In *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 115–128, Berkeley, CA, USA, 2004. USENIX Association.

[11] J.-S. Pendry and M. K. McKusick. Union mounts in 4.4BSD-lite. In *TCON'95: Proc. of the USENIX 1995 Technical Conf.*, pages 3–3, Berkeley, CA, USA, 1995. USENIX Association.

[12] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[13] R. Pike, D. Presotto, K. Thompson, H. Trickey, and P. Winterbottom. The use of name spaces in Plan 9. *SIGOPS Oper. Syst. Rev.*, 27(2):72–76, 1993.

[14] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the Elephant file system. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 110–123, New York, NY, USA, 1999. ACM.

[15] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger. Metadata efficiency in versioning file systems. In *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 43–58, Berkeley, CA, USA, 2003. USENIX Association.

[16] M. Szeredi. Filesystem in userspace. http://fuse.sourceforge.net, Feb. 2005.

[17] J. Wires and M. J. Feeley. Secure file system versioning at the block level. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 203–215, New York, NY, USA, 2007. ACM.

[18] 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. *Trans. Storage*, 2(1):74–105, 2006.

[19] C. P. Wright, R. Spillane, G. Sivathanu, and E. Zadok. Extending ACID semantics to the file system. *Trans. Storage*, 3(2):4, 2007.

[20] E. Zadok, R. Iyer, N. Joukov, G. Sivathanu, and C. P. Wright. On incremental file system development. *Trans. Storage*, 2(2):161–196, 2006.