Breaking Apart the VFS for Managing File Systems

Kuei Sun, Matthew Lakier, Angela Demke Brown and Ashvin Goel
University of Toronto

Abstract

File system management applications, such as data scrubbers, defragmentation tools, resizing tools, and partition editors, are essential for maintaining, optimizing, and administering storage systems. These applications require fine-grained control over file-system metadata and data, such as the ability to migrate a data block to another physical location. Such control is not available at the VFS API, and so these applications bypass the VFS and access and modify file-system metadata directly. As a result, these applications do not work across file systems, and must be developed from scratch for each file system, which involves significant engineering effort and impedes adoption of new file systems.

Our goal is to design an interface that allows these management applications to be written once and be usable for all file systems that support the interface. Our key insight is that these applications operate on common file system abstractions, such as file system objects (e.g., blocks, inodes, and directory entries), and the mappings from logical blocks of a file to their physical locations. We propose the Extended Virtual File System (eVFS) interface that provides fine-grained access to these abstractions, allowing the development of generic file system management applications. We demonstrate the benefits of our approach by building a file-system agnostic conversion tool that performs in-place conversion of a source file system to a completely different destination file system, showing that arbitrary modifications to the file system format can be handled by the interface.

1 Introduction

File system management applications help with maintaining, optimizing, and administering file systems. Examples of such applications include file-system upgrade tools, defragmentation tools, and file-system resizing tools. Unlike typical applications that are file-system agnostic because they use the virtual file-system interface (VFS) to access their data, the management applications perform low-level allocation, mapping, and placement of physical blocks in a file system. These operations are not exposed by the VFS API, and thus these applications must bypass the VFS, and access the file-system metadata directly.

As a result, file system developers and experts must write these applications from scratch for each file system, because they are tightly coupled with the format of the file system. For example, a defragmentation tool for Ext4 cannot be reused for Btrfs, not even in parts, because the two file systems use different formats for block allocation and free space management. The effort required for building these applications is significant, and thus newer file systems such as F2FS [7] and BetrFS [6] lack a rich set of management tools, which stymies their adoption and hinders innovation in file system technology.

The goal of our work is to simplify the development of file system management applications. The VFS interface has been highly successful because it abstracts the key objects (e.g., files and directories) and operations (e.g., create, delete, read, write) that are provided by any file system. Our approach is to provide a new abstraction, similar to VFS, that enables file system management applications to be written in a generic, file-system agnostic manner. Ideally, the applications are developed once using this interface and they work for file systems that implement this interface. This relieves file system developers from the onus of building these essential (but often neglected) applications, and instead they can focus their effort on improving the file system itself.

We introduce the **E**xtended **V**irtual **F**ile **S**ystem (eVFS) interface, which provides a fine-grained abstraction for manipulating the file system. The key insight of eVFS is that the management applications operate on common abstractions that are shared across file systems, such as the allocation of file system objects (e.g., blocks or extents, inodes, and directory entries) and the mappings from logical blocks of a file to physical blocks.

We argue that these operations should be exposed to the management applications through a common API, and that doing so will simplify their design. The functionality required for a management task, such as defragmentation, is more easily expressed in terms of operations such as remapping extents when we abstract away the implementation details of those operations on a specific file system. We further argue that providing an API to manipulate the file system metadata does not change the file system's trust model. Management applications are already trusted to operate directly on metadata without the VFS, and bugs in them may cause file system inconsistency or corruption [3, 5]. Hence, exposing these operations through eVFS may improve the robustness of management applications, since the file-system specific

implementation of the interface can be provided once by file system experts.

Ultimately, our aim is to expose these operations for online use without affecting existing file system applications that are unaware of management applications or the eVFS API. To do so, the eVFS API provides a transactional interface that ensures the atomicity of the eVFS operations. Currently, however, we have only explored offline use, where the transactional support provides crash consistency, which is often missing in management applications [4].

As proof of concept, we have built an offline file system conversion tool using the eVFS interface. This tool performs crash-consistent, in-place conversion of one file system to another entirely different file system. It can thus also be used to modify the file-system specific options of the file system, such as the file system size, or upgrade a file system (Ext3 to Ext4). The application is generic, and thus supporting additional file systems should require no modifications to the application. This experience suggests that the eVFS API will allow building a variety of generic, file system management applications.

2 Approach

The goal of designing the eVFS interface is to enable file-system agnostic management applications. As such, the interface must be generic while providing fine-grained control over the allocation of file system objects, and mappings from one object to another (e.g. directory entries to files, files to blocks). Therefore, we must first define the various file system objects that are generic across file systems.

At a high level, file systems manage four types of objects: files or directories, blocks or extents, directory entries, and file-system wide settings (such as the block size, file system size, or label). Thus, we provide an interface for managing each of the objects, and any mappings between them. In this section, we motivate the eVFS design by describing use cases for accessing and manipulating these objects, including the need for transactional support.

Inodes In the eVFS interface, similar to VFS, every file system object, such as a file or directory, is uniquely identified by an inode number and structure. File system management applications frequently need to read, create or update inode structures and their mappings to physical blocks. For example, a defragmentation tool needs to scatter-gather fragmented blocks of a file into a new contiguous extent, which involves updating the logical to physical block mappings of an inode. The eVFS interface thus provides support for allocating and updating an inode. The inode allocation interface is finer-grained than VFS file creation, since it does not allocate a directory entry or file blocks. The eVFS interface also allows mapping and unmapping logical offsets of a file to specific physical blocks, providing precise control over these mappings.

Blocks and Extents Many file system management applications require fine-grained control over the physical layout of a file on disk. For example, an in-place file system conversion tool needs to recreate files on the destination file system that directly map to existing data blocks belonging to the same files on the source file system, while avoiding copying blocks as much as possible (see Section 3.1 for more detail). Thus, the interface allows allocation of blocks and extents at specific physical addresses.

To allocate blocks and extents, management applications need to know the locations and sizes of free spaces. Maintenance applications also require knowledge of the remaining free space in the file system to determine whether to start garbage collection. This information must be obtained by processing block allocation metadata and is thus a file-system specific operation. However, with eVFS, we abstract away the file-system specific details and provide a function that enables applications to iterate through all free space extents in a file system, or find the nearest available free extent, without needing to know the format of the file system. Similarly, the API allows applications to check whether an extent is currently in use.

For file systems that support copy-on-write semantics and snapshots, management applications can make informed decisions based on whether an extent is private to a file or shared by multiple files. For example, it is easier to relocate private extents during garbage collection. To enable such logic, the interface supports retrieving a reverse mapping that lists the inodes and their logical offsets that map to a particular extent. With this information, a defragmentation tool can move an extent by remapping all inodes that reference the extent to its new location.

Directory Entries File systems use directory entries to support mapping name(s) to an inode. Consider our in-place file system conversion tool that needs to recreate directories. It must iterate through the entries in the source file system while making copies to the destination file system. Therefore, the interface supports adding, updating, or removing individual directory entries, as well as iterating through the entries of a directory inode.

File-System Wide Settings A file system stores various parameters and options that describe the file system format and the features that are supported. Some of these parameters are common across different file systems, such as the total size of the file system, block size,

Function Prototype	Description
`struct evfs * fs_open(struct evfs_mount * mnt)`	open the file system with parameters specified by *mnt*
`long super_make(struct evfs_super * sup)`	make a new file system with parameters specified by *sup*
`long super_set(struct evfs_super * sup)`	update an existing file system's settings
`long extent_alloc(long addr, long len)`	allocate the extent defined by {*addr*, *len*}
`long extent_free(long addr, long len)`	free the extent defined by {*addr*, *len*}
`long extent_reverse(long addr, long len, struct evfs_reverse * rv)`	fills *rv* with the inode number and logical offset of all inodes that map to the extent defined by {*addr*, *len*}
`int extent_active(long addr, long len)`	return 1 if extent defined by {*addr*, *len*} is active, else 0
`long extent_iterate(long ino_nr, void * priv, long (* cb)(long log_blk_nr, long phy_blk_nr, long len, void * priv))`	iterate through all extents mapped to inode *ino_nr* in the form of {*log_blk_nr*, *len*} → {*phy_blk_nr*, *len*} and process them via callback function *cb*
`long freesp_iterate(void * priv, long (* cb)( long addr, long len, void * priv))`	iterate through all free space extents in the file system and process them via callback function *cb*
`long inode_alloc(long ino_nr, struct evfs_inode * i)`	allocate the inode *ino_nr* with the inode structure *i*
`long inode_free(long ino_nr)`	free the inode *ino_nr*
`long inode_map(long ino_nr, long log_blk_nr, long phy_blk_nr, long len)`	map physical extent {*phy_blk_nr*, *len*} to the logical extent {*log_blk_nr*, *len*} for inode *ino_nr*
`long inode_unmap(long ino_nr, long addr, long len)`	unmap logical extent {*addr*, *len*} for inode *ino_nr*
`long inode_iterate(void * priv, long (* cb)( long ino_nr, struct evfs_inode * i, void * priv))`	iterate through all active inodes in the file system and process them via callback function *cb*
`long dirent_add(long dir_nr, struct evfs_dirent * d)`	add a new entry *d* to directory inode *dir_nr*
`long dirent_iterate(long dir_nr, void * priv, long (* cb)(struct evfs_dirent * d, void * priv))`	iterate through all directory entries for inode *dir_nr* and process them via callback function *cb*
`struct evfs_txn * txn_begin(struct evfs * evfs)`	start a new transaction and returns the associated handle
`long txn_commit(struct evfs_txn * txn)`	commit the transaction *txn*
`long txn_abort(struct evfs_txn * txn)`	abort the transaction *txn*

Table 1: eVFS API. Parameter `struct evfs_txn` is omitted for all functions except for `fs_open` and the last 3 transaction-related functions.

etc. Therefore, they can be exposed to support management applications that modify the layout or format of the file system (e.g., updating file system to a newer version, changing the block size of an existing file system).

The eVFS interface provides generic support for managing file-system wide settings in two ways. First, it allows updating simple settings such as labels or file system feature flags that do not require restructuring the file system, similar to the functionality provided by `tune2fs` [17] for the Ext3/Ext4 file systems. Second, to support generic restructuring, the interface provides support for creating an empty file system, that performs the same task as `mkfs`. As described later in more detail, this interface allows our in-place file system conversion tool to reformat the device to the destination file system, and then the new file system metadata can be recreated, while keeping the file contents of the existing file system intact. Similarly, a file system can be resized using this approach, although less efficiently than a custom file-system specific resizing tool.

Transactions Since many of the operations supported by the interface make the file system temporarily inconsistent, the interface also provides transactional support to ensure atomicity so that other applications do not see partial updates made by management applications. Transactional support is also necessary for providing crash consistency, which is often missing in management applications [4]. Thus, eVFS also enables building robust management applications that are resilient to power failures.

3 Implementation

In this section, we present our prototype of the eVFS API and discuss our implementation of the API. Next, we describe the in-place file system conversion tool that we have built using the eVFS interface.

Table 1 shows a partial set of functions in the eVFS API. These functions provide fine-grained control by allowing extents, inodes, and directory entries, to be individually manipulated. We chose to use extent-based representation for storage space since it generally requires less metadata than the corresponding block-based representation, and is thus preferred by modern file systems. An application is expected to start a transaction before issuing most eVFS operations.

We have implemented a subset of the eVFS API for two file systems, the extent-based Ext4 file system, and the log-structured F2FS file system, which enables con-

verting an Ext4 file system to an F2FS file system. Our current implementation only works for offline use, i.e., the application has exclusive access to the unmounted file system. The file-system specific implementation of the API uses the Spiffy framework [14] that provides robust parsing and serialization libraries, helping avoid bugs while handling file system metadata.

3.1 File System Conversion Tool

Converting an existing file system to a different file system is a tedious and time-consuming process that normally involves copying the full content of a file system to another disk, reformatting the disk, and then copying everything back to the new file system. In contrast, an in-place file system conversion only updates file system metadata, and does not move any regular file data unless its location must be used by statically allocated metadata of the destination file system. This technique can speed up the conversion dramatically. While some such conversion tools exist,[1] they are hard to implement correctly and not generally available.

We have designed and implemented a crash-consistent, in-place file system conversion tool using the eVFS interface. To do so, the conversion tool uses user-level, block-based redo journaling for ensuring crash consistency. Unlike typical journals that have a fixed size (e.g., the Ext4 journal), the journal is dynamically allocated from blocks that are currently free in both the source and the destination file systems, which ensures that both abort and redo recovery are possible since these blocks are not in use by either file system. The free space information is obtained by using the eVFS API. The dynamic allocation of blocks also allows converting heavily fragmented file systems, and maximizing utilization of the free space for the journal.

As an optimization, when a destination file system block is written to free space in the source file system, the block is written directly without being journaled. By journaling the rest of the blocks that will overwrite the source file system, we ensure crash consistency.

A complication occurs when a block that is currently in use by the journal is allocated to the destination file system. Allocating this block would cause the journal to be overwritten during checkpointing. In this case, the conflicting journal block is remapped to a different free block, and then this freed block can be updated directly.

If the journaling layer runs out of free space, the conversion process is aborted. We guarantee that this error occurs before the transaction commits. Therefore, no data loss is possible on a conversion failure.

The conversion tool starts a new transaction and then creates an empty destination file system on the device storing the source file system. Next it iterates through the inodes of the source file system, and creates the corresponding inodes in the destination file system. For a regular file, it iterates through each extent in the source inode, allocating the corresponding extent in the destination file system, and then copying over the mappings to the destination inode. For a source file system extent, we also check whether it overlaps with block(s) that are allocated in the destination file system. If so, we relocate the extent to free space in the destination file system, and update the source inode that maps to this extent. For directories, we iterate through the entries to recreate them in the destination file system. Then we commit the transaction and allow checkpointing to create the destination file system. The commit information needs to be located in a well-known location that is not in use by either file system. Currently, we use the boot record to store this information.

[1] The `convert` utility converts FAT32 to NTFS [15], and updating to iOS 10.3 upgrades the file system from HFS+ to APFS [16]

Spiffy Converter		**eVFS Converter**	
Application		Application	
Generic	504	Generic	224
Ext4	218	Ext4	-
F2FS	1780	F2FS	-
Libraries		Libraries	
Generic	2250	Generic	2625
Journaling	-	Journaling	1350
Ext4	-	Ext4	276
F2FS	-	F2FS	2152

Table 2: Lines of code for the Spiffy and the eVFS file-system conversion tools.

4 Evaluation

In this section, we evaluate the programming effort needed to build the in-place file system conversion tool. The file system conversion tool is based on our previous work [14], and redesigning the application to use the eVFS interface did not result in statistically significant changes to its performance, and so we omit the discussion of our performance results.

Table 2 shows the programming effort for building the file-system conversion tool using the Spiffy framework [14] (Spiffy converter) and the eVFS interface (eVFS converter). Both the converters use the same logic, but the Spiffy converter's application code uses 2502 lines, which includes almost 2000 lines of file-system specific code, and this converter can only convert from Ext4 to F2FS. The eVFS converter uses 224 lines of generic file-system conversion code, less than 10% of the Spiffy application), and could be used to convert between any pair of file systems that implement the eVFS API. The libraries used by both applications provide generic code (e.g., bitmaps, hash tables, etc.) for supporting

management applications. The file-system specific code used by the eVFS converter is part of the eVFS library and can be used by other management applications. Unlike the Spiffy converter, the eVFS converter is crash-consistent, requiring 1350 lines of journaling code.

5 Related Work

We had previously built a file system conversion tool using the Spiffy framework [14]. Spiffy uses an annotation language to enable complete specification of the file system format and then generates a robust library for parsing and serializing the file system data structures. However, Spiffy only helps identify the types of these structures, and not their semantics, and thus still requires significant file-system specific code. In contrast, the eVFS interface is generic across file systems. In this work, we use Spiffy for the file-system specific implementation of the eVFS API, thus ensuring a robust implementation.

There are several libraries for accessing and manipulating file systems, such as `libext2fs` [17] and `libfsntfs` that comes with `ntfsprogs` [8]. While most of the functions provided by these libraries are file-system specific, some are generic across file systems, such as iterating through all inodes in the file system, which we have adapted for the eVFS interface.

Many parallel file systems use a dedicated metadata service backed by an object storage service that provides persistence for both file data and file system metadata [10]. With namespace management decoupled from the data path, clients can stream data to the storage servers, thus maximizing I/O bandwidth [18]. While these object stores conceptually split the VFS API across metadata and data operations, the eVFS API splits the operations themselves.

Many existing works extended storage interfaces to simplify writing file systems, while improving their reliability [12], security, functionality [11], or performance [19, 1]. Our work is similar in spirit to these works in that we have extended the file system interface to reduce the effort of building file system management applications. However, instead of pushing the decision making down to the lower layer, we instead expose previously internal operations in the file system, thus enabling applications that require more fine-grained control over the file system.

6 Limitations and Future Work

While our approach helps with building generic management applications, these applications will need to specifically handle file systems that support subsets of the eVFS API. For example, in-place update file systems, such as Ext4, generally do not track the reverse mapping from extents to inodes, and so cannot implement this API call efficiently. As a result, certain applications will either not be able to support these file systems, or will require different logic for such file systems.

We are currently working on supporting several deployed VFS-based file systems, such as Btrfs and XFS. Since our API is generic, we believe it should be possible to extend it for non-VFS file systems as well.

Our API is designed to provide control over extents, but these extents may be mapped to a non-linear physical address space. For example, modern file systems such as Btrfs and ZFS incorporate volume management and RAID-style redundancy within the file system, and thus the extents may map to physically discontiguous chunks of physical storage. Since some management applications may need control over these physical chunks as well, we plan to explore the feasibility of generically exposing this address space.

Our eVFS API is extent-based, and so we believe that it should be possible for new non-volatile memory (NVM) file systems to implement the API, allowing them to benefit from management applications originally designed for traditional block-based storage.

Our current eVFS implementation is designed for offline applications, and provides crash consistency support. For online applications, we plan to provide transactional support for eVFS operations. While a transactional file system would simplify this implementation, such file systems are not in common use because supporting transactions adds significant complexity to the entire kernel [13]. Instead, our plan is to use an optimistic concurrency control scheme for committing eVFS operations in existing file systems. This approach will ensure that existing VFS applications are minimally affected by eVFS operations, even when eVFS applications may run long transactions. In turn, the eVFS applications must be designed to handle transaction aborts. Our implementation will take advantage of recent file system designs that use multiple operation logs, track dependencies between operations, and merge the operations [9, 2].

7 Conclusion

The eVFS interface exposes a new, low-level file system abstraction that enables control over allocation and modification of extents, inodes, and directory entries, and the mapping between extents and inodes. These operations are necessary for building generic file system management applications that perform fine-grained updates to file system metadata. We showed the feasiblity of our approach by building a file system conversion tool. The application requires no changes to support a file system that implements the eVFS interface. We believe eVFS will enable many exciting applications and reduce the programming effort for building these applications.

References

[1] ANAND, A., SEN, S., KRIOUKOV, A., POPOVICI, F., AKELLA, A., ARPACI-DUSSEAU, A., ARPACI-DUSSEAU, R., AND BANERJEE, S. Avoiding file system micromanagement with range writes. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation* (2008), USENIX Association, pp. 161–176.

[2] BHAT, S. S., EQBAL, R., CLEMENTS, A. T., KAASHOEK, M. F., AND ZELDOVICH, N. Scaling a file system to many cores using an operation log. In *Proc. of the Symposium on Operating Systems Principles (SOSP)* (2017), pp. 69–86.

[3] CARREIRA, J. A. C. M., RODRIGUES, R., CANDEA, G., AND MAJUMDAR, R. Scalable testing of file system checkers. In *Proceedings of the 7th ACM European Conference on Computer Systems* (New York, NY, USA, 2012), EuroSys '12, ACM, pp. 239–252.

[4] GATLA, O. R., HAMEED, M., ZHENG, M., DUBEYKO, V., MANZANARES, A., BLAGOJEVIĆ, F., GUYOT, C., AND MATEESCU, R. Towards robust file system checkers. In *Proc. of the USENIX Conference on File and Storage Technologies (FAST)* (2018), pp. 105–122.

[5] GUNAWI, H. S., RAJIMWALE, A., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. SQCK: A declarative file system checker. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Dec. 2008).

[6] JANNEN, W., YUAN, J., ZHAN, Y., AKSHINTALA, A., ESMET, J., JIAO, Y., MITTAL, A., PANDEY, P., REDDY, P., WALSH, L., BENDER, M. A., FARACH-COLTON, M., JOHNSON, R., KUSZMAUL, B. C., AND PORTER, D. E. Betrfs: Write-optimization in a kernel file system. *ACM Trans. Storage 11*, 4 (Nov. 2015), 18:1–18:29.

[7] LEE, C., SIM, D., HWANG, J., AND CHO, S. F2fs: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (2015), pp. 273–286.

[8] MATHES, S. The interoperability power of linux-ntfs tools. *Linux Journal 154* (2007), 1.

[9] PILLAI, T. S., ALAGAPPAN, R., LU, L., CHIDAMBARAM, V., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Application crash consistency and performance with ccfs. *ACM Transactions on Storage 13*, 3 (Sept. 2017), 19:1–19:29.

[10] SCHWAN, P., ET AL. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux symposium* (2003), vol. 2003, pp. 380–386.

[11] SHIN, J.-Y., BALAKRISHNAN, M., MARIAN, T., AND WEATHERSPOON, H. Isotope: Transactional isolation for block storage. In *Proc. of the USENIX Conference on File and Storage Technologies (FAST)* (2016), p. 23.

[12] SIVATHANU, G., SUNDARARAMAN, S., AND ZADOK, E. Type-safe disks. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2006), pp. 15–28.

[13] SPILLANE, R. P., GAIKWAD, S., CHINNI, M., ZADOK, E., AND WRIGHT, C. P. Enabling transactional file access via lightweight kernel extensions. In *Proc. of the USENIX Conference on File and Storage Technologies (FAST)* (2009), pp. 29–42.

[14] SUN, K., FRYER, D., CHU, J., LAKIER, M., BROWN, A. D., AND GOEL, A. Spiffy: enabling file-system aware storage applications. In *Proc. of the USENIX Conference on File and Storage Technologies (FAST)* (2018), pp. 91–103.

[15] TECHNET, M. How to convert fat disks to ntfs. `https://technet.microsoft.com/en-us/library/bb456984.aspx`.

[16] TOM WARREN. Apple is upgrading millions of iOS devices to a new modern file system today. `https://www.theverge.com/2017/3/27/15076244/apple-file-system-apfs-ios-10-3-features`. Accessed: 2017-03-27.

[17] TS'O, T. E2fsprogs: Ext2/3/4 filesystem utilities. `http://e2fsprogs.sourceforge.net/`, 2017.

[18] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D., AND MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation* (2006), USENIX Association, pp. 307–320.

[19] ZHANG, Y., ARULRAJ, L. P., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. De-indirection for flash-based ssds with nameless writes. In *Proc. of the USENIX Conference on File and Storage Technologies (FAST)* (2012).