Tolerating File-System Mistakes with EnvyFS

Lakshmi N. Bairavasundaram[†], Swaminathan Sundararaman,
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
[†]*NetApp, Inc.* *University of Wisconsin-Madison*

Abstract

We introduce *EnvyFS*, an N-version local file system designed to improve reliability in the face of file-system bugs. EnvyFS, implemented as a thin VFS-like layer near the top of the storage stack, replicates file-system metadata and data across existing and diverse commodity file systems (e.g., ext3, ReiserFS, JFS). It uses majority-consensus to operate correctly despite the sometimes faulty behavior of an underlying commodity *child* file system. Through experimentation, we show EnvyFS is robust to a wide range of failure scenarios, thus delivering on its promise of increased fault tolerance; however, performance and capacity overheads can be significant. To remedy this issue, we introduce *SubSIST*, a novel single-instance store designed to operate in an N-version environment. In the common case where all child file systems are working properly, SubSIST coalesces most blocks and thus greatly reduces time and space overheads. In the rare case where a child makes a mistake, SubSIST does not propagate the error to other children, and thus preserves the ability of EnvyFS to detect and recover from bugs that affect data reliability. Overall, EnvyFS and SubSIST combine to significantly improve reliability with only modest space and time overheads.

1 Introduction

File systems make mistakes. A modern file system consists of many tens of thousands of lines of complex code; such a system must handle memory-allocation failure, disk faults, and system crashes, and in all cases preserve the integrity of both user data and its own metadata. Thus, it is perhaps no surprise that many recent studies have uncovered hundreds of bugs in file systems [14, 18, 34, 49, 51].

Bugs manifest in numerous ways. In the best case, a system immediately crashes; recent research has shown how to cope with such "fail-stop" behavior by both isolating said file system from the rest of the kernel and transparently restarting it [16, 44]. However, in more insidious scenarios, file-system bugs have been shown to accidentally corrupt the on-disk state of one or more blocks [34, 49, 51]; such "fail-silent" behavior is much more challenging to detect and recover from, and thus can lead to both data loss (due to a corrupt directory) or bad data passed back to the user.

One method to improve file systems and reduce fail-silent mistakes is thorough testing and other bug-finding techniques. For example, recent research has introduced a number of increasingly sophisticated and promising bug-finding tools [18, 29, 49, 51]. However, until such approaches are able to identify *all* file-system bugs, problems are likely to persist. Hence, file-system mistakes are here to stay; the challenge is how to cope with them.

In this paper, we advocate an approach based on the classic idea of N-version programming [1]. Specifically, we present the design and implementation of EnvyFS, a software layer that multiplexes file-system operations across multiple *child* file systems. EnvyFS issues all user operations to each child, determines the majority result, and delivers it to the user. By design, we thus eliminate the reliance on a single complex file system, instead placing it on a much simpler and smaller software layer.

A significant challenge in N-version systems is to formulate the common specification and to create the different versions. EnvyFS overcomes this challenge by using the Virtual File System (VFS) layer as the common specification and by leveraging existing Linux file systems already written by different open-source development groups (e.g., ext3 [46], JFS [8], ReiserFS [36]). In this manner, we build on work that leverages existing software bases to build N-version services, including NFS servers [37] and transaction-processing systems [47].

An important design goal in building EnvyFS is to keep it simple, thereby reducing the likelihood of bugs that arise from the sheer complexity of file-system code. At the same time, EnvyFS should leverage the VFS layer and existing file systems to the extent possible. We find that EnvyFS is indeed simple, being only a fraction of the size as its child file systems, and can leverage much of the common specification. However, limitations do arise from the nature of the specification in combination with our goal of simplicity. For example, because child file systems issue different inode numbers for files, EnvyFS is tasked with issuing inode numbers as well; in the interest of simplicity, EnvyFS does not maintain these inode numbers persistently (i.e., the inode number for a file is the same within, but not across, mounts).

A second challenge for EnvyFS is to minimize the performance and disk-space overheads of storing and retrieving data from its underlying child file systems. Our solution is to develop a variant of a single-instance store (an SIS) [11, 17, 35]. By utilizing content hashes to detect duplicate data, an SIS can significantly reduce the space and performance overheads introduced by EnvyFS. However, using an SIS underneath EnvyFS mandates a different approach, as we wish to reduce overhead without sacrificing the ability to tolerate mistakes in a child file system. We achieve this by implementing a novel SIS (which we call SubSIST) that ensures that a mistake in one file system (e.g., filling a block with the wrong contents) does not propagate to other children, and thus preserves the ability of EnvyFS to detect faults in an underlying file system through voting. Thus, in the common case where all file systems work properly, SubSIST coalesces most blocks and can greatly reduce time and space overheads; in the rare case where a single child makes a mistake, SubSIST does not do so, enabling EnvyFS to detect and recover from the problem.

We have implemented EnvyFS and SubSIST for Linux; currently, EnvyFS employs any combination of ext3, JFS, and ReiserFS as child file systems. Through fault injection, we have analyzed the reliability of EnvyFS and have found that it can recover from a range of faults in nearly all scenarios; many of these faults cause irreparable data loss or unmountable file systems in the affected child. We have also analyzed the performance and space overheads of EnvyFS both with and without SubSIST. We have found across a range of workloads that, in tandem, they usually incur modest performance overheads. However, since our current implementation of SubSIST does not persist its data structures, the performance improvements achieved through SubSIST represent the best case. We find that SubSIST also reduces the space overheads of EnvyFS significantly by coalescing all data blocks. Finally, we have discovered that EnvyFS may also be a useful diagnostic tool for file-system developers; in particular, it helped us to readily identify and fix a bug in a child file system.

The rest of the paper is organized as follows. In Section 2, we present extended motivation. We present the design and implementation of EnvyFS and SubSIST in Sections 3 and 4 respectively. We evaluate our system for reliability in Section 5 and performance in Section 6. We then discuss related work in Section 7 and conclude in Section 8.

2 Do File Systems Make Mistakes?

Before describing EnvyFS, we first briefly explain why we believe file systems do indeed make mistakes, and why those mistakes lead file systems to deliver corrupt data to users or corrupt metadata to themselves. Such failures are silent, and thus challenging to detect.

Recent work in analyzing file systems has uncovered numerous file system bugs, many of which lead to silent data corruption. For example, Prabhakaran et al. found that a single transient disk error could cause a file system to return corrupt data to the calling application [33, 34]. Further, a single transient write failure could corrupt an arbitrary block of the file system, due to weaknesses in the failure-handling machinery of the journaling layer [34]. Similar bugs have been discovered by others [50, 51].

Another piece of evidence that file systems corrupt their own data structures is the continued presence of file system check-and-repair tools such as fsck [30]. Despite the fact that modern file systems either use journaling [21] or copy-on-write [12, 19, 25, 38] to ensure consistent update of on-disk structures, virtually all modern file systems ship with a tool to find and correct inconsistencies in on-disk data structures [20]. One might think inconsistencies arise solely from faulty disks [6, 7]; however, even systems that contain sophisticated machinery to detect and recover from disk faults ship with repair tools [24]. Thus, even if one engineers a reliable storage system, on-disk structures can still become corrupt.

In addition to bugs, file systems may accidentally corrupt their on-disk structures due to faulty memory chips [31, 39]. For example, if a bit is flipped while a block is waiting to be written out, either metadata or data will become silently corrupted when the block is finally written to disk.

Thus, both due to poor implementations as well as bad memory, file systems can corrupt their on-disk state. The type of protection an N-version system provides is thus complementary to the machinery of checksums and parity and mirroring that could be provided in the storage system [28, 41], because these problems occur *before* such protection can be enacted. These problems cannot be handled via file-system backups either; backups potentially provide a way to recover data, but they do not help detect that currently-available data is corrupt. To detect (and perhaps recover) from these problems, something more is required.

3 EnvyFS: An N-Version File System

N-version programming [1, 2, 4, 5, 13, 15, 48] is used to build reliable systems that can tolerate software bugs. A system based on N-version programming uses N different versions of the same software and determines a majority result. The different versions of the software are created by N different developers or development teams for the same software specification. It is assumed (and encouraged using the specification) that different developers will design and implement the specification differently, lowering the chances that the versions will contain the same bugs or will fail in a similar fashion.

Developing N-version systems has three important steps (a) producing the specification for the software, (b) implementing the N different versions of the software, and (c) creating the environment that executes the different versions and determines a consensus result [1].

We believe the use of N-version programming is particularly attractive for building reliable file systems since the design and development effort required for the first two steps (i.e., specification and version development) can be much lower than for the typical case.

First, many existing commodity file systems adhere to a common interface. All Linux file systems adhere to the POSIX interface, which internally translates to the Virtual File System (VFS) interface. Thus, if an N-version file system is able to leverage the POSIX/VFS interface, then no additional effort will be needed to develop a new common specification. However, because the POSIX/VFS interface was not designed with N-versioning in mind, we do find that EnvyFS must account for differences between file systems.

Second, many diverse file systems are available for Linux today. For example, in Linux 2.6, there are at least 30 different file systems (depending upon how one counts), such as ext2, ext3, JFS, ReiserFS, XFS, FAT, and HFS; new ones are being implemented as well, such as btrfs. All have been built for the POSIX/VFS interface. These different file systems have drastically different data structures, both on disk and in memory, which reduces the chances of common file-system bugs. Furthermore, previous research has shown that file systems behave differently when they encounter partial-disk failures; for example, Prabhakaran et al. show that when directory data is corrupted, ReiserFS and JFS detect the problem while ext3 does not [34].

3.1 Design Goals and Assumptions

The design of EnvyFS is influenced by the following goals and assumptions:

Simplicity: As systems have shown time and again, complexity is the source of many bugs. Therefore, an N-version file system should be as simple as possible. In EnvyFS, this goal primarily translates to avoiding persistent metadata; this simplification allows us to not allocate disk blocks and to not worry about failures affecting EnvyFS metadata.

No application modifications: Applications should not need to be modified to use EnvyFS instead of a single local file system. This goal supports our decision to leverage the POSIX specification as our specification.

Single disk: The N-version file system is intended to improve the reliability of desktop systems in the face of file-system mistakes. Therefore, it replicates data across multiple local file systems that use the same disk drive. This goal translates to a need for reducing disk-space overheads; thus, we develop a new single-instance store (Section 4) for our environment.

Non-malicious file systems: We assume that child file systems are not malicious. Thus, we must only guard against accidents and not intentional attempts to corrupt user data or file-system metadata.

Bug isolation: We also assume that the bugs do not propagate to the rest of the kernel. If such corruption were indeed a major issue, one could apply isolation techniques as found in previous work to contain them [16, 44].

3.2 Basic Architecture

EnvyFS receives application file operations, issues the operations to multiple *child file systems*, compares the results of the operation on all file systems, and returns the majority result to the application. Each child stores its data and metadata in its own disk partition.

We have built EnvyFS within Linux 2.6, and Figure 1 shows the basic architecture. EnvyFS consists of a software layer that operates underneath the virtual file system (VFS) layer. This layer executes file operations that it receives on multiple children. We use ext3 [46], JFS [9], and ReiserFS [36] for this purpose. We chose these file systems due to their popularity and their differences in how they handle failures [34]. However, the EnvyFS design does not preclude the use of other file systems that use the VFS interface.

Similar to stackable file systems [22], EnvyFS interposes transparently on file operations; it acts as a normal file system to the VFS layer and as the VFS layer to the children. It thus presents file-system data structures and interfaces that the VFS layer operates with and in turn manages the data structures of the child file systems. We have implemented wrappers for nearly all file and directory operations. These wrappers verify the status of necessary objects in the children before issuing the operation to them. For example, for an unlink operation, EnvyFS first verifies that both the file and its parent directory are consistent with majority opinion.

Each operation is issued in series to the child file systems; issuing an operation in parallel to all file systems

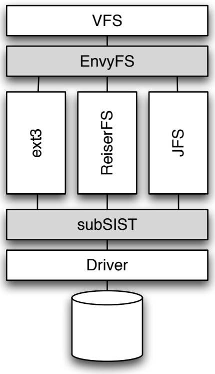

Figure 1: **N-version file system in Linux.** *The figure presents the architecture of a 3-version file system with ext3, ReiserFS and JFS as the children. The core layer is EnvyFS; it is responsible for issuing file operations to all three file systems, determining a majority result from the ones returned by the file systems, and returning it to the VFS layer. The optional layer beneath the file systems (SubSIST) is a single-instance store built to work in an N-version setting; it coalesces user data stored by the different file systems in order to reduce performance and space overheads.*

increases complexity and is unlikely to realize much, if any, performance benefit when the children share the same disk drive. When the operations complete, the results are semantically compared to determine the majority result; this result is then returned to the user. When no majority result is obtained, an I/O error is returned.

Our current implementation does not support the `mmap` operation. While supporting `mmap` is not fundamentally difficult, it does present a case where child file systems cannot be trivially leveraged. Specifically, an implementation of `mmap` in EnvyFS would likely involve the use of file `read` and `write` operations of children, rather than their `mmap` operations.

We now discuss how our implementation addresses each of the three steps of N-version programming. In particular, we discuss how EnvyFS deals with the complexities of leveraging the existing POSIX specification/VFS layer and of using existing Linux file systems while keeping EnvyFS simple.

3.3 Leveraging the POSIX Specification

EnvyFS leverages the existing POSIX specification and operates underneath VFS as it provides core functionality (like ordering of file operations) that is challenging to replicate without modifying applications. Thus, EnvyFS relies upon the correct operation of the VFS layer. We believe the VFS layer has been heavily tested over the years and is likely to have fewer bugs that the file systems themselves; this optimism is partially validated by Yang et al., who find two bugs in the VFS layer and nearly thirty in ext3, ReiserFS, and JFS [51].

One issue that EnvyFS must handle is that the POSIX specification is imprecise for use in an N-version setting; that is, the child file systems we leverage differ in various user-visible aspects that are not a part of the POSIX interface. For example, POSIX does not specify the order in which directory entries are to be returned when a directory is read; thus, different children may return directory entries in a different order. As another example, the inode number of a file is available to users and applications through the `stat` system call; yet, different file systems issue different inode numbers for the same file.

One approach to addressing this problem would be to make the specification more precise and change the file systems to adhere to the new specification. This approach has a number of problems. First, refining the specification discourages diversity across the different file systems. For example, if the specification details how inode numbers should be assigned to files, then all file systems will be forced to use the same algorithm to allocate inode numbers, perhaps causing them to also use the same data structures and inviting common bugs. Second, even given a more precise specification, nondeterminism and differences in operation ordering can easily cause different results. Finally, changing the specification would greatly increase the amount of development effort to produce an N-version file system, since each existing Linux file system would need to be changed to use it as a child file system.

3.3.1 Semantic Result Comparison

Our solution is to have EnvyFS deal with the imprecise POSIX specification: when EnvyFS compares and returns results from the child file systems, it does so using semantic knowledge of how the POSIX/VFS interface operates. In other words, EnvyFS examines the VFS data structures returned by each child file system and does a semantic comparison of individual fields.

For example, for a file read operation, EnvyFS compares (a) the size of data read (or the error code returned), (b) the actual content read, and (c) the file position at the end of the read. For all file operations where inodes may be updated, EnvyFS compares (and copies to its VFS inode) the contents of the individual inodes. We have developed comparators for different file-system data types like superblocks, inodes, and directories. For example, an inode comparator checks whether the fields `i_nlink`, `i_mode`, `i_uid`, and so forth in the child inodes are the same. While EnvyFS compares results returned to it, it does not verify that the operation completed correctly in each file system; for example, it does not re-read data written to a file to verify that all file systems actually wrote the correct data.

As mentioned above, directory entries and inodes are especially interesting cases. We now describe how

EnvyFS handles these structures in more detail and we also describe how EnvyFS optimizes its comparison of data blocks across file systems.

Directory Entries: POSIX does not specify the order in which directory entries are to be returned. Thus, EnvyFS reads all directory entries from all file systems; it then returns individual entries that occur in a majority of file systems. The disadvantage of this approach is that it increases the overhead for the `getdirentries` system call for very large directories. We note that we could optimize the performance of this case (at the expense of code simplicity), by reading from child file systems only until EnvyFS finds matches for exactly as many entries as the user provides space for.

Inode Numbers: POSIX does not specify how inode numbers should be assigned to files, yet inode numbers are visible to user applications. Since EnvyFS cannot always use the inode number produced by any one child file system (because it may fail), it assigns a virtual inode number when a new object is encountered and tracks this mapping. Keeping with our simplicity goal, inode numbers so assigned are not persistent; that is, an object has a specific virtual inode number only between a mount and the corresponding unmount. This decision impacts only a few applications that depend on the persistence of file-system inode numbers. If applications using EnvyFS do require persistent inode numbers, one simple solution that could be explored is to store the inode mapping in a hidden file in the root directory of each file system and load the mapping at mount time. A specific example in this context is an NFS server using protocol versions 2 or 3; the server uses persistent inode numbers to create file handles for clients that can be used across server crashes. Even so, in protocol version 4, a "volatile file handle" option was introduced, thereby eliminating the need for persistent inode numbers. Interestingly, some local file systems, like the High Sierra file system for CD-ROMs, do not have persistent inode numbers [32].

Reads of Data Blocks: In performing read operations, we would like to avoid the performance overhead of allocating memory to store the results returned by all of the file systems (especially when the data read is already in cache). Therefore, EnvyFS reuses the memory provided by the application for the `read` system call. Reusing the memory influences two subsequent decisions. First, to determine whether the child file systems return the same data from the `read`, EnvyFS computes checksums on the data returned by the child file systems and compares them; a more thorough byte-by-byte comparison would require memory for all copies of data. Second, EnvyFS issues the read operation in series to child file systems only until a majority opinion is reached (i.e., usually to two children); this choice eliminates the problem of issuing reads again in case the last file system returns incorrect data; in addition, in the common case, when file systems agree, the third read is avoided. It is important to note that we choose not to take the same issue-only-until-majority approach with other VFS operations such as lookup since the limited performance gain for such operations is not worth the complexity involved, say in tracking and issuing a sequence of lookups for the entire path when a lookup returns erroneous results in one file system. A future implementation could include a "verify-all" option that causes EnvyFS to issue the read to all file systems ignoring the performance cost.

In choosing the checksum algorithm for comparing data, one must remember that the cost of checksumming can be significant for reads that are satisfied from the page cache. We have measured that this cost is especially high for cryptographic checksums such as MD5 and SHA-1; therefore, in keeping with our goal of protecting against bugs but not maliciousness, we use a simple TCP-like checksum (sum of bytes) for comparisons.

3.3.2 Operation Ordering

Our placement of EnvyFS beneath VFS simplifies the issue of ordering file operations. As in many replication-based fault tolerance schemes, determining an ordering of operations is extremely important; in fact, recent work in managing heterogeneous database replicas focuses primarily on operation ordering [47]. In the context of a file system, consider the scenario where multiple file operations are issued for the same object: if an ordering is not predetermined for these operations, their execution may be interleaved such that the different children perform the operations in a different order and therefore produce different results even in the absence of bugs.

Unlike databases, the dependence between operations can be predetermined for file systems. In EnvyFS, we rely on the locking provided by the Linux VFS layer to order metadata operations. As explained earlier, this reliance cannot be avoided without modifying applications (to issue operations to multiple replicas of VFS that execute an agreement algorithm). In addition to the VFS-level locking, we perform file locking within EnvyFS for reads and writes to the same file. This locking is necessary since the VFS layer does not (and has no need to) order file reads and writes.

3.4 Using Existing File Systems

Our decision to leverage existing Linux file systems for child file systems greatly simplifies the development costs of the system. However, it does restrict our behavior in some cases.

One problem with using multiple local file systems is that the different file systems execute within the same address space. This exposes EnvyFS to two problems: (a) a kernel panic induced by a child file system, and (b) a memory bug in a child file system that corrupts the rest of the kernel. A solution to both problems would be to completely isolate the children using a technique such as Nooks [43]. However, due to the numerous interactions between the VFS layer and the file systems, such isolation comes at a high performance cost.

Therefore, we explore a more limited solution to handle kernel panics. We find the current practice of file systems issuing a call to `panic` whenever they encounter errors to be too drastic, and developers seem to agree. For example, ext3 code had the following comment: *"Given ourselves just enough room to cope with inodes in which i_blocks is corrupt: we've seen disk corruptions in the past which resulted in random data in an inode which looked enough like a regular file for ext3 to try to delete it. Things will go a bit crazy if that happens, but at least we should try not to panic the whole kernel"*. In the case of ext3 and JFS, a mount option (*errors*) can specify the action to take when a problem is encountered; one could specify *errors=continue* to ensure that panic is not called by the file systems. However, this option is not available on all file systems. Our solution is to replace calls to `panic`, `BUG`, and `BUG_ON` by child file systems with a call to a `nvfs_child_panic` routine in EnvyFS. This simple replacement is performed in file-system source code. The `nvfs_child_panic` routine disables issuing of further file operations to the failed file system.

Another limitation of using existing file systems is that different file systems use different error codes for the same underlying problems (e.g., "Input/output error", "Permission denied", or "Read-only file system"). A consistent error code representing each scenario would enable EnvyFS to take further action. In our current implementation EnvyFS simply reports the majority error code or reports an I/O error if there is no majority.

3.5 Keeping EnvyFS Simple

EnvyFS has its own data structures (e.g., in-memory inodes and *dentry* structures), which are required for interacting with the VFS layer. In turn, EnvyFS manages the allocation and deallocation of such structures for child file systems; this management includes tracking the status of each object: whether it matches with the majority and whether it needs to be deallocated.

In keeping with our simplicity goal, we have designed EnvyFS so that it does not maintain any persistent data structures of its own. This decision affects various parts of the design; we previously discussed how this impacts the management of inode numbers (Section 3.3.1); we now discuss how it impacts the handling of faulty file systems and system crashes.

3.5.1 Handling Disagreement

An important part of EnvyFS is the handling of cases where a child file system disagrees with the majority result. This part is specifically important for local file systems since the ability to perform successive operations may depend on the result of the current operation (e.g., a file read cannot be issued when open fails).

When an error is detected, in order to restore EnvyFS to full replication, the erroneous child file system should be repaired. The repair functionality within EnvyFS fixes incorrect data blocks and inodes in child file systems. Specifically, if EnvyFS observes that the file contents in one file system differs from the other file systems during a file read, it issues a write of the correct data to the corrupt file system before returning the data to the user. With respect to inodes, EnvyFS repairs a subset of various possible corruptions; it fixes inconsistencies in the permission flags (which are i_mode, i_uid, i_gid) with the majority result from other file systems. It also fixes size mismatches where the correct size is larger than the corrupt one by copying the data from correct file systems. On the other hand, issuing a file truncate for the case where the correct size is smaller may result in more corruption in an already corrupt file system (e.g., the blocks being freed by truncate may actually be in use by a different file as a result of a prior corruption).

As the above example demonstrates, efficient repair for all inconsistencies is challenging. If EnvyFS cannot repair the erroneous object in a child file system, it operates in *degraded-mode* for the associated object. In degraded mode, future operations are not performed for that object in the file system with the error, but EnvyFS continues to perform operations on other objects for that file system. For example, if a child's file inode is declared faulty, then read operations for that file are not issued to that file system. As another example, if a lookup operation completes successfully for only one file system, its corresponding in-memory dentry data structure is deallocated, and any future file create operation for that dentry is not issued to that file system.

For simplicity, the validity information for objects is not maintained persistently. With this approach, after a reboot, the child file system will try to operate on the faulty objects again. If the object is faulty due to a permanent failure, then the error is likely to be detected again, as desired. Alternately, if the problem was due to a transient error, the child will return to normal operation as long as the object has not been modified in the interim. Our current approach to fully repair inconsistencies that cannot be repaired in-flight requires that the entire erroneous child file system be re-created from the other (correct) children, an expensive process.

Some further challenges with efficient repair may arise from limitations of the VFS layer. Consider the following scenario. A file with two hard links to it may have incorrect contents. If EnvyFS detects the corruption through one of the links, it may create a new file in the file system to replace the erroneous one. However, there is no simple way to identify the directory where the other link is located, so that it can be fixed as well (except through an expensive scan of the entire file system). In the future, we plan to investigate how one can provide hooks into the file system to enable fast repair.

3.5.2 System Crashes

When a system crash occurs, EnvyFS file-system recovery consists of performing file-system recovery for all child file systems before EnvyFS is mounted again. In our current approach, EnvyFS simply leverages the recovery methods inherent to each individual file system, such as replaying the journal. This approach leads to a consistent state within each of the children, but it is possible for different file systems to recover to different states. Specifically, when a crash occurs in the middle of a file operation, EnvyFS could have issued (and completed) the operation for only a subset of the file systems, thereby causing children to recover to different states. In addition, file systems like ext3 maintain their journal in memory, flushing the blocks to disk periodically; journaling thus provides consistency and not durability.

An alternative approach for solving this problem would be for EnvyFS itself to journal operations and replay them during recovery. However, this would require EnvyFS to maintain persistent state.

In EnvyFS, the state modifications that occur durably for a majority of file systems before the crash are considered to have completed. The differences in the minority set can be detected when the corresponding objects are read, either during user file operations or during a proactive file-system scan. There are corner cases where a majority result will not be obtained when a system crash occurs. In these cases, choosing the result of any one file system will not affect file-system semantics. At the same time, these cases cannot be distinguished from other real file-system errors. Therefore, EnvyFS returns an error code when these differences are detected; future implementations could choose to use the result from a designated "primary" child.

4 SubSIST: A Single-Instance Store

Two issues that arise in using an N-version file system are the disk-space and performance overheads. Since data is stored in N file systems, there is an N-fold increase (approximately) in disk space used. Since each file operation is performed on all file systems (except for file reads), the likely disk traffic is N times that for a single file system. For those environments where the user is willing to trade-off some data reliability for disk space and performance, we develop a variant of single-instance storage [11, 17, 35]. Note that SubSIST is not mandatory; if the performance and space overheads of EnvyFS are acceptable, there is no reason to make use of SubSIST (indeed, the less code relied upon the better).

With SubSIST, the disk operations of the multiple children pass through SubSIST, which is implemented as a block-level layer. As is common in single-instance stores, SubSIST computes a content hash (MD5) for all disk blocks being written and uses the content hash to detect duplicate data.

Using an SIS greatly reduces disk usage underneath an N-version file system. At the same time, despite coalescing data blocks, an SIS retains much of the benefit of EnvyFS for two reasons. First, the reliability of file-system *metadata* is not affected by the use of an SIS. Since metadata forms the access path to multiple units of data, its reliability may be considered more important than that of data blocks. Because the format of file-system metadata is different across different file systems, metadata blocks of different file systems have different hash values and are stored separately; thus, the SIS layer can distinguish between data and metadata blocks *without* any knowledge of file-system data structures. Second, since file systems maintain different in-memory copies of data, file-system bugs that corrupt data blocks in-memory cause the data in different file systems to have different content hashes; therefore, individual file systems are still protected against each other's in-memory file-data corruptions.

4.1 Requirements and Implications

The design of SubSIST for an N-version file system should satisfy slightly different requirements than a conventional SIS. We discuss four important observations and their impact on the design of SubSIST.

First, child file systems often replicate important metadata blocks so that they can recover from failures. For example, JFS replicates its superblock and uses the replica to recover from a latent sector error to the primary. Thus, SubSIST does not coalesce disk blocks with the same content if they belong to the same file system.

Second, an SIS coalesces common data written at approximately the same time by different file systems. Therefore, in SubSIST, the content hash information for each disk block is not stored persistently; the content hashes are maintained in memory and deleted after some time has elapsed (or after N file systems have written the same content). This ephemeral nature of content hashes also reduces the probability of data loss or corruption due hash collisions [10, 23].

Third, in an N-version file system, reads of the same data blocks occur at nearly the same time. Thus, SubSIST services reads from different file systems by maintaining a small read cache. This read cache holds only those disk blocks whose reference count (number of file systems that use the block) is more than one. It also tracks the number of file systems that have read a block and removes a block from cache as soon as this number reaches the reference count for the block.

Finally, the child file systems using SubSIST are unmodified and therefore have no knowledge of content addressing; therefore, SubSIST virtualizes the disk address space; it exports a virtual disk to the file system, and maintains a mapping from each file system's virtual disk address to the corresponding physical disk address, along with a reference count for each physical disk block. SubSIST uses file-system virtual addresses as well as previously mapped physical addresses as hints when assigning physical disk blocks to maintain as much sequentiality and spatial locality as possible. When these hints do not provide a free disk block, SubSIST selects the closest free block to the previously mapped physical block.

4.2 Implementation

SubSIST has numerous important data structures, including: (i) a table of virtual-to-physical mappings, (ii) allocation information for each physical disk block in the form of reference count maps, (iii) a content-hash cache of recent writes and the identities of the file systems that performed the write, and (iv) a small read cache.

We have built SubSIST as a pseudo-device driver in Linux. It exports virtual disks that are used by the file systems. Our current implementation does not store virtual-to-physical mappings and reference-count maps persistently; in the future, we plan to explore reliably writing this information to disk.

5 Reliability Evaluation

We evaluate the reliability improvements of a 3-version EnvyFS (EnvyFS_3) that uses ext3, JFS, and ReiserFS (v3) as children. All our experiments use the versions of these file systems that are available as part of the Linux 2.6.12 kernel.

We evaluate the reliability of EnvyFS_3 in two ways: First, we examine whether it recovers from scenarios where file-system content is different in one of the three children. Second, we examine whether it can recover from corruption to on-disk data structures of one child.

5.1 Differing File System Content

The first set of experiments is intended to mimic the scenario where one of the file systems has an incorrect disk image. Such a scenario might occur either when (i) a system crash occurs and one of the children has written more or less to disk than the others, (ii) a bug causes one of the file systems to corrupt file data, say by performing a misdirected write of data belonging to one file to another file, or (iii) soft memory errors cause corruption.

Difference in content	**Num Tests**	**Correct success**	**Correct error code**
None	28	17 / 17	11 / 11
Dir contents differ in one	13	6 / 6	7 / 7
Dir present in only two	13	6 / 6	7 / 7
Dir present in only one	9	4 / 4	5 / 5
File contents differ in one	15	11 / 11	4 / 4
File metadata differ in one	45	33 / 33	12 / 12
File present in only two	15	11 / 11	4 / 4
File present in only one	9	3 / 3	6 / 6
Total	**147**	**91 / 91**	**56 / 56**

Table 1: **File-system Content Experiments.** *This table presents the results of issuing file operations to EnvyFS_3 objects that differ in data or metadata content across the different children. The first column describes the difference in file-system content. The second column presents the total number of experiments performed for this content difference; this is the number of applicable file operations for the file or directory object. For metadata differences, 15 operations each are performed for differences in mode, nlink, and size fields of the inode. The third column is the fraction of operations that return correct data and/or successfully complete. The fourth column is the fraction of operations that correctly return an error code (and it is the expected error code) (e.g., ENOENT when an unlink operation is performed for a non-existent file). We see that EnvyFS_3 successfully uses the majority result in all 147 experiments.*

We first experiment by creating different file-system images as the children and executing a set of file operations on EnvyFS_3 that uses the children. We have explored various file-system content differences, including extra or missing files or directories, and differences in file or directory content. The different file operations performed include all possible file operations for the object (irrespective of whether the operation causes the different content to be read). Our file operations include those that are expected to succeed as well as those that are expected to fail with a specific error code.

Table 1 shows that EnvyFS_3 correctly detects all differences and always returns the majority result to the user (whether the expected data or error code). EnvyFS_3 can also be successfully mounted and unmounted in all cases. We find that the results are the same irrespective of which child (ext3, JFS, ReiserFS) has incorrect contents.

We then explore whether EnvyFS_3 continues to detect and recover from differences caused by in-memory corruption when SubSIST is added. We experiment by modifying data (or metadata) as it being written to a child file system and then causing the data (or metadata) to be read back. Table 2 presents the results of the experiments. We find that EnvyFS_3 used along with SubSIST returns the correct results in all scenarios. Also, in most sce-

Corruption Type	Num Tests	Correct success	Fix
File contents differ in one	3	3 / 3	3 / 3
Dir contents differ in one	3	3 / 3	0 / 3
Inode contents differ in one	15	15 / 15	9 / 15
Total	**21**	**21 / 21**	**12 / 21**

Table 2: **File-system Corruption Experiments.** *This table presents the results of corrupting one of the file objects in EnvyFS$_3$ that results in different data or metadata content across the different children with SubSIST underneath it. The first column describes the type of corruption. The second column presents the total number of experiments performed; The third column is the fraction of operations that return correct data and/or successfully complete (which also include identification of mismatch in file system contents). The fourth column is the fraction of operations that EnvyFS was able to repair after detecting corruption.*

narios when file contents or inode contents are different, EnvyFS$_3$ successfully repairs the corrupt child during file-system operation (Section 3.5.1 describes scenarios in which EnvyFS repairs a child during file-system operation). The use of SubSIST does not affect protection against in-memory corruption; a data block corrupted in memory will cause SubSIST to generate a different content hash for the bad block when it is written out, thereby avoiding the usual coalescing step.

5.2 Disk Corruption

The second set of experiments analyzes whether EnvyFS$_3$ recovers when a child's on-disk data structures are corrupt. Such corruption may be due to a bug in the file system or the rest of the storage stack. We inject corruption into JFS and ext3 data structures by interposing a pseudo-device driver that has knowledge of the data structures of each file system. This driver zeroes the entire buffer being filled in response to a disk request by the file system, but does not return an error code (i.e., the corruption is silent). All results, except that for data blocks, are applicable to using EnvyFS$_3$ with SubSIST.

5.2.1 Corruption in JFS

Figures 2a and 2b compare the user-visible results of injecting corruptions into JFS data structures when JFS is used stand-alone and when EnvyFS$_3$ is used (that is composed of JFS, ext3, and ReiserFS).

Each row in the figures corresponds to the JFS data structure for which the fault is injected. Each column in the figures corresponds to different file operations. The different symbols represent the user-visible results of the fault; examples of user-visible results include data loss, and a non-mountable file system. For example, in Figure 2a, when an inode block is corrupted during path traversal (column 1), the symbol indicates that (i) the operation fails and (ii) the file system is remounted in read-only mode. In addition to the symbols for each column, the symbol next to the data-structure name for all the rows indicates whether or not the loss of the disk block causes irreparable data or metadata loss.

As shown in Figure 2a, JFS is rarely able to recover from corruptions: JFS can continue normal operation when the read to the block-allocation bitmap fails during truncate and unlink. Often, the operation fails and JFS remounts the file system in read-only mode. The corruption of some data structures also results in a file system that cannot be mounted. In one interesting case, JFS detects the corruption to an internal (indirect) block of a file and remounts the file system in read-only mode, but still returns corrupt data to the user. Data loss is indicated for many of the JFS rows.

In comparison to stand-alone JFS, EnvyFS$_3$ recovers from all but one of the corruptions (Figure 2b). EnvyFS$_3$ detects errors reported by JFS and also detects corrupt data returned by JFS when the internal block or data block is corrupted during file read. In all these cases, EnvyFS$_3$ uses the two other file systems to continue normal operation. Therefore, no data loss occurs when any of the data structures is corrupted.

In one interesting fault-injection experiment, a system crash occurs both when using JFS stand-alone and when using it in EnvyFS$_3$. In this experiment, the first aggregate inode block (`AGGR-INODE-1`) is corrupted, and the actions of JFS lead to a kernel panic during paging. Since this call to `panic` is not in JFS code, it cannot been replaced as described in Section 3.4. Therefore, the kernel panic occurs both when using JFS stand-alone and when using EnvyFS$_3$. Thus, we find a case where EnvyFS$_3$ is not completely resilient to underlying child failure; faults that lead to subsequent panics in the main kernel cannot be handled with N-version techniques.

5.2.2 Corruption in Ext3

Figures 2c and 2d show the results of injecting corruption into ext3 data structures. As in the case of JFS, the figures compare ext3 against EnvyFS$_3$.

Overall, we find that ext3 does not handle corruption well. Figure 2c shows that no corruption error leads to normal operation without data loss for ext3. In most cases, there is unrecoverable data loss and either the operation fails (ext3 reports an error) or the file system is remounted in read-only mode or both. In some cases, the file system cannot even be mounted. In other cases, ext3 fails to detect corruption (e.g., `IMAP`, `INDIRECT`), thereby either causing data loss (`IMAP`) or returning corrupt data to the user (`INDIRECT`). Finally, in one scenario (corrupt `INODE` during `unlink`), the failure to handle corruption leads to a system crash upon unmount.

In comparison, Figure 2d shows that EnvyFS$_3$ contin-

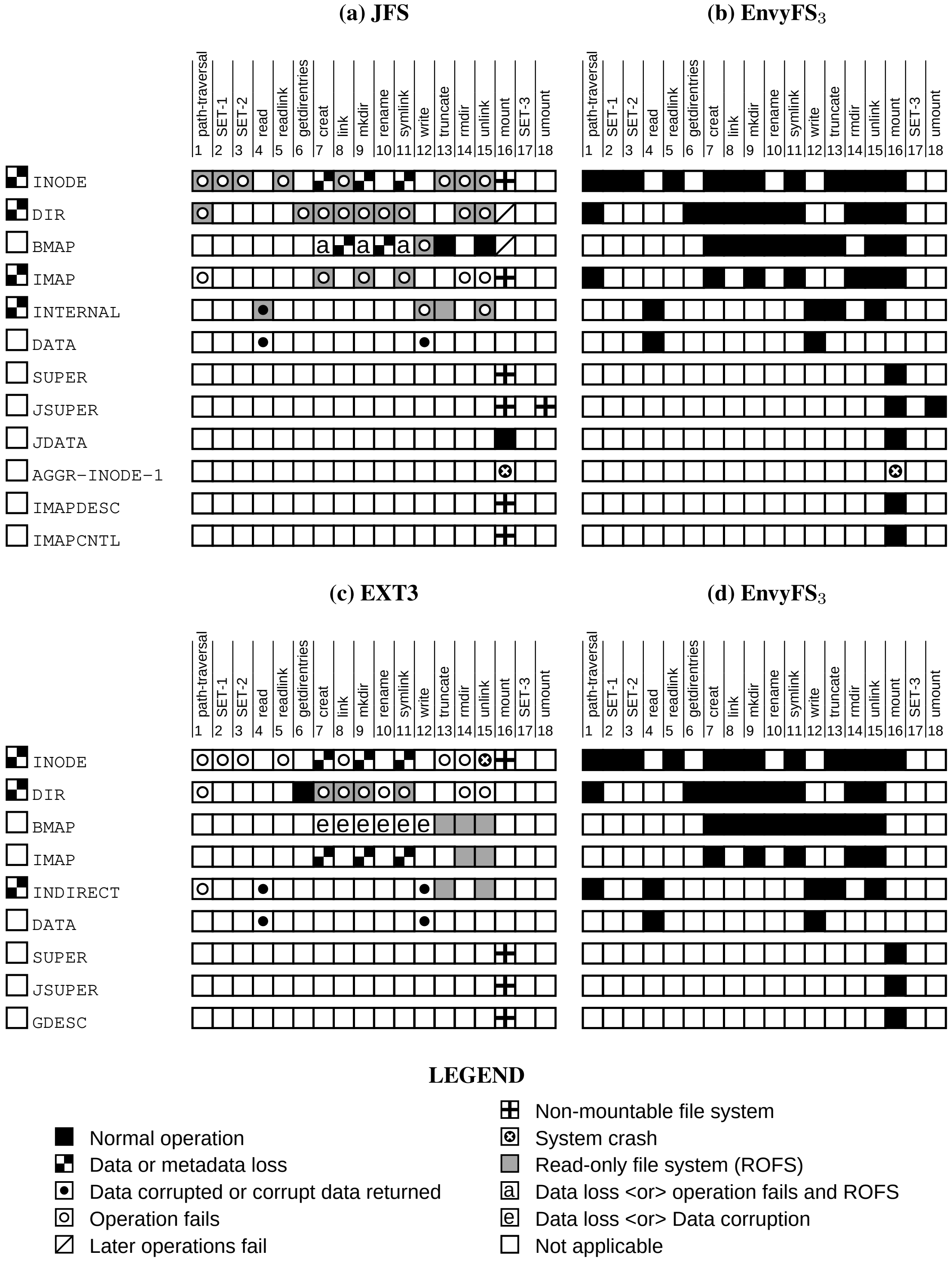

Figure 2: **Disk corruption experiments.** *The figures show the results of injecting corruption for JFS and ext3 on-disk data structures. JFS is used stand-alone in (a) and is one of the children in EnvyFS$_3$ in (b). ext3 is used stand-alone in (c) and is one of the children in EnvyFS$_3$ in (d). Each row in the figures corresponds to the data structure for which the fault is injected; each column corresponds to a file operation; each symbol represents the user-visible result of the fault injection. Note that (i) the column SET-1 denotes file operations access, chdir, chroot, stat, statfs, lstat, and open; SET-2 denotes chmod, chown, and utimes; SET-3 denotes fsync and sync, (ii) some symbols are a combination of two symbols, one of which is the light-gray square for "read-only file system."*

ues normal operation in every single experiment, including in the system-crash case. EnvyFS_3 again shows great resilience to faults in a single child file system.

We also found EnvyFS_3 to be surprisingly helpful in isolating a non-trivial bug in ext3. As reported above, when an ext3 inode block is corrupted before an `unlink`, the system crashes when the file system is later unmounted. The system crash does not occur in EnvyFS_3; one might suspect that EnvyFS_3 is robust because ext3 was modified to call `nvfs_child_panic`. However, this is not the case; instead, EnvyFS_3 completely avoids the code paths that cause the panic; in particular, EnvyFS_3 detects that the inode returned by ext3 in response to a lookup (that is performed by VFS prior to the actual unlink) is faulty (i.e., semantically differs from the inodes returned by the other file systems). Therefore, it does not issue the subsequent unlink operation to ext3, hence avoiding actions that cause the panic. Interestingly, the bug that causes the crash is actually in the lookup operation, the first point where EnvyFS_3 detects a problem. Note that in the absence of an N-version file system, one would find that the system crashed on an unmount, but will not have information linking the crash to the unlink system call or the bug in `ext3_lookup`. Checking the ext3 source code, we found that this bug in Linux 2.6.12 was subsequently fixed in 2.6.23. This experience highlights the potential for using N-versioning to localize bugs in file systems.

5.3 Discussion

Our experiments show that EnvyFS_3 can recover from various kinds of corruptions in a child file system. Since this improvement in reliability is achieved through additional layers of code, any bugs in these layers could offset the reliability improvements. Therefore, an important goal in our design is to keep EnvyFS simple. We now compare the amount of code used to construct EnvyFS and SubSIST against other file systems in order to estimate the complexity (and therefore, the likelihood of bugs) in such a system.

The EnvyFS layer is about 3,900 lines of code, while SubSIST is about 2,500 lines of code. In comparison, ext3 contains 10,423 lines, JFS has 15,520 lines, ReiserFS has 18,537 lines, and XFS, a complex file system, has 44,153 lines.

6 Time and Space Overheads

Although reliable file-system operation is our major goal, we are also concerned with the overheads innate to an N-version approach. In this section, we quantify the performance costs of EnvyFS and the reduction in disk-space overheads due to SubSIST.

	ext3	**JFS**	**Reiser**	**Envy_3**	**+SIS**
Cached read	2.1	2.1	2.2	5.7	5.7
Cached write	3.7	2.5	2.2	8.8	8.8
Seq. read-4K	17.8	17.7	18.2	424.1	33.7
Seq. read-1M	17.8	17.7	18.2	75.4	33.7
Seq. write	26.0	18.7	24.4	74.9	29.7
Rand. read	163.6	163.5	165.1	434.2	164.2
Rand. write	20.4	18.9	20.4	61.4	7.0
OpenSSH	25.3	25.7	25.6	26.4	26.0
Postmark-10K	14.7	39.0	9.6	128.8	26.4
Postmark-100K	29.0	107.2	33.6	851.4	430.0
Postmark-100K*	128.3	242.5	78.3	405.5	271.1

Table 3: **Performance.** *This table compares the execution time (in seconds) for various benchmarks for EnvyFS_3 (without and with SubSIST) against the child file systems, ext3, JFS, and ReiserFS. All our experiments use Linux 2.6.12 installed on a machine with an AMD Opteron 2.2 GHz processor, 2 GB RAM, Hitachi Deskstar 7200-rpm SATA disks, and 4-GB disk partitions for each file system. Cached reads and writes involve 1 million reads/writes to 1 file data block. Sequential read-4K/writes are 4 KB at a time to a 1-GB file. Sequential read-1M is 1MB at a time to a 1-GB file. Random reads/writes are 4 KB at a time to 100 MB of a 1-GB file. OpenSSH is a copy, untar, and make of OpenSSH-4.5. Postmark was configured to create 2500 files of sizes between 4KB and 40KB. We ran it with 10K and 100K transactions. All workloads except ones named "Cached" use a cold file-system cache.*

We now quantify the performance overheads of EnvyFS_3 both with and without SubSIST, in contrast to each of the child file systems (ext3, JFS, and ReiserFS) running alone. Table 3 presents the results.

We now highlight the interesting points from the table:

- When reads hit in the cache (*cached reads*), EnvyFS_3 pays a little more than twice the cost (as it accesses data from only two children and performs a checksum comparison to find a majority).
- EnvyFS_3 performance under *cached writes* is roughly the sum across the children; such writes go to all three child file systems, and thus are replicated in the buffer cache three times. This aspect of EnvyFS_3 is bad for performance (and increases cache pressure), but at the same time increases fault resilience; a corruption to one copy of the data while in memory will not corrupt the other two copies.
- SubSIST does not help with either cached workload as it only interposes on disk traffic.
- EnvyFS_3 has terrible performance under *sequential disk reads*, as it induces seeks (and loses disk track prefetches) between two separate sequential streams especially with small block sizes; much of this cost could be alleviated with additional prefetching or with larger block sizes. Increasing

the read size from 4KB to 1MB significantly improves the performance of $EnvyFS_3$.

- *Sequential writes* perform much better on $EnvyFS_3$ compared to sequential reads, due to batching of operations (and hence fewer seeks).
- In many cases where $EnvyFS_3$ performance suffers (*sequential reads and writes, random reads*), SubSIST greatly improves performance through coalescing of I/O. Indeed, in one case (*random writes*), SubSIST improves performance of $EnvyFS_3$ as compared to any other single file system, as for this specific case its layout policy transforms random writes into a more sequential pattern to disk (see Section 4.1). These performance improvements likely represent the best case since the numbers do not show the costs that would be incurred in a SubSIST implementation that maintains data structures persistently.
- Application performance, as measured on the *OpenSSH* benchmark, is quite acceptable, even without SubSIST.
- In the case of *Postmark* benchmark, both workload size and dirty page writeout intervals affect the performance of $EnvyFS_3$. For smaller workloads (i.e., Postmark-10K), performance of $EnvyFS_3$ with SubSIST is comparable with other file systems. But with increase in workload size (Postmark-100K), performance of $EnvyFS_3$ worsens as it is forced to write back more data due to increase in cache pressure along with shorter dirty page writeout internals. If we provide $EnvyFS_3$ with thrice the amount of memory and change the writeback intervals accordingly, we see that $EnvyFS_3$ performance (with SubSIST) is comparable to the slowest of the three children (JFS).

We also tracked the storage requirement across these benchmarks. For those workloads that generated writes to disk, we found that SubSIST reduced the storage requirement of EnvyFS by roughly a factor of three.

7 Related Work

Over the years, N-version programming has been used in various real systems and research prototypes to reduce the impact of software bugs on system reliability. As noted by Avižienis [1], N-version computing has very old roots (going back to Babbage and others in the 1800s).

The concept was (re)introduced in computer systems by Avižienis and Chen in 1977 [2]. Since then, various other efforts, many from the same research group, have explored the process as well as the efficacy of N-version programming [3, 5, 4, 13, 27].

Avižienis and Kelly [4] study the results of using different specification languages; they use 3 different specification languages to develop 18 different versions of an airport scheduler program. They perform 100 demanding transactions with different sets of 3-version units and determined that while at least one version failed in 55.1% of the tests, a collective failure occurred only in 19.9% of the cases. This demonstrates that the N-version approach reduces the chances of failure. Avižienis et al.also determine the usefulness of developing the different software versions in different languages like Pascal, C etc. [5]. As in the earlier study, the different versions developed had faults, but only very few of these faults were common and the source of the common faults were traced to ambiguities in the initial specification.

N-version computing has been employed in many systems. For many years, such uses have primarily been in mission-critical or safety-critical systems [48, 52]. More recently, with the increasing cost of system failures and the rising impact of software bugs, many research efforts have focused on solutions that use N-version programming for improving system security and for handling failures [15, 26, 37, 47]. Joukov et al. [26] store data across different local file systems with different options for storing the data redundantly. However, unlike our approach, they do not protect against file-system bugs, and inherently rely on each individual file system to report any errors, so that data recovery may be initiated in RAID-like fashion. Rodrigues et al. [37] develop a framework to allow the use heterogeneous network file systems as replicas for Byzantine-fault tolerance. Vandiver et al. [47] explore the use of heterogeneous database systems for Byzantine-fault tolerance. They specifically address the issue of ordering of operations using *commit barriers*. In EnvyFS, this issue is made simpler due to two reasons: (i) in the absence of transactions, file systems are not expected to provide atomicity across multiple operations on the same file, and (ii) the VFS layer can easily identify conflicts through locking of file-system data structures.

8 Conclusion

"A three-ply cord is not easily severed."
King Solomon [Ecclesiastes 4:12]

We have proposed EnvyFS, an approach that harnesses the N-version approach to tolerate file-system bugs. Central to our approach is building a reliable whole out of existing and potentially unreliable parts, thereby significantly reducing the cost of development. We have also proposed the use of a single-instance store to reduce the performance and disk-space overheads of an N-version approach. SubSIST, the single-instance store, is designed to retain much of the reliability improvements obtained from EnvyFS. We have built and evaluated EnvyFS for Linux file systems and shown that it is significantly more reliable than file systems of which it is composed; with SubSIST, performance and capacity overheads are brought into the acceptable range. As a fringe benefit, we also show that the N-version approach can be used to locate bugs in file systems.

Modern file systems are becoming more complex by the day; mechanisms to achieve data-structure consistency [45], scalability and flexible allocation of disk blocks [9, 42], and the capability to snapshot the file system [25, 40] significantly increase the amount of code and complexity in a file system. Such complexity could lead to bugs in the file system that render any data protection further down the storage stack useless. N-versioning can help; by building reliability on top of existing pieces, EnvyFS takes an end-to-end approach and thus delivers reliability in spite of the unreliability of the underlying components.

Of course, our approach is not a panacea. Each file system may have features that N-versioning hides or makes difficult to realize. For example, some file systems are tailored for specific workloads (e.g., LFS[38]). In the future, it would be interesting if one could enable the N-version layer to be cognizant of such differences; for example, if one file system is optimized for write performance, all writes could initially be directed to it, and only later (in the background) would other file systems be updated. In such a manner, we could truly achieve the best of both worlds: reliability of the N-version approach but without the loss of characteristics that makes each file system unique.

Acknowledgments

We thank the anonymous reviewers and Sean Rhea (our shepherd) for their tremendous feedback and comments, which have substantially improved the content and presentation of this paper. We also thank the members of the ADSL research group for their insightful comments.

This material is based upon work supported by the National Science Foundation under the following grants: CCF-0621487, CNS-0509474, CCR-0133456, as well as by generous donations from NetApp and Sun Microsystems.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

† Author worked on EnvyFS as a UW-Madison student.

References

[1] A. A. Avižienis. The Methodology of N-Version Programming. In M. R. Lyu, editor, *Software Fault Tolerance*, chapter 2. John Wiley & Sons Ltd., 1995.

[2] A. A. Avižienis and L. Chen. On the Implementation of N-Version Programming for Software Fault Tolerance During Execution. In *Proceedings of 1st Annual International Computer Software and Applications Conference (COMPSAC'77)*, Chicago, USA, 1977.

[3] A. A. Avižienis, P. Gunningberg, J. P. J. Kelly, L. Strigini, P. J. Traverse, K. S. Tso, and U. Voges. The UCLA DEDIX system: A Distributed Testbed for Multiple-version Software. In *Digest of 15th International Symposium on Fault-Tolerant Computing (FTCS'85)*, pages 126–134, Ann Arbor, MI, June 1985.

[4] A. A. Avižienis and J. P. J. Kelly. Fault Tolerance by Design Diversity: Concepts and Experiments. *IEEE Computer*, 17(8), August 1984.

[5] A. A. Avižienis, M. R. Lyu, and W. Schütz. In Search of Effective Diversity: A Six-Language Study of Fault-Tolerant Flight Control Software. In *Digest of 18th International Symposium on Fault-Tolerant Computing (FTCS '88)*, Tokyo, Japan, June 1988.

[6] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *Proceedings of the 2007 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '07)*, San Diego, California, June 2007.

[7] L. N. Bairavasundaram, G. R. Goodson, B. Schroeder, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, pages 223–238, San Jose, California, February 2008.

[8] S. Best. JFS Overview. www.ibm.com/developerworks/library/l-jfs.html, 2000.

[9] S. Best. JFS Overview. http://jfs.sourceforge.net/project/pub/jfs.pdf, 2000.

[10] J. Black. Compare-by-hash: a reasoned analysis. In *Proceedings of the USENIX Annual Technical Conference (USENIX '06)*, pages 7–12, Boston, Massachusetts, June 2006.

[11] W. J. Bolosky, S. Corbin, D. Goebel, and J. R. Douceur. Single Instance Storage in Windows 2000. In *Proceedings of the 4th USENIX Windows Systems Symposium*, Seattle, Washington, August 2000.

[12] J. Bonwick and B. Moore. ZFS: The Last Word in File Systems. http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf, 2007.

[13] L. Chen and A. A. Avižienis. N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation. In *Digest of 8th International Symposium on Fault-Tolerant Computing (FTCS'78)*, Toulouse, France, 1978.

[14] A. Chou, J.-F. Yang, B. Chelf, S. Hallem, and D. Engler. An Empirical Study of Operating System Errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 73–88, Banff, Canada, October 2001.

[15] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davdison, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-Variant Systems - A Secretless Framework for Security through Diversity. In *Proceedings of the 15th USENIX Security Symposium (Sec '06)*, Vancouver, British Columbia, Aug. 2006.

[16] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell. CuriOS: Improving Reliability through Operating System Structure. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI '08)*, San Diego, California, December 2008.

[17] EMC. Centera Family. http://www.emc.com/products/family/emc-centera-family.htm, 2009.

[18] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 57–72, Banff, Canada, October 2001.

[19] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[20] H. S. Gunawi, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. SQCK: A Declarative File System Checker. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI '08)*, San Diego, California, December 2008.

[21] R. Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP '87)*, Austin, Texas, November 1987.

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

[23] V. Henson. An Analysis of Compare-by-hash. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS'03)*, Lihue, Hawaii, May 2003.

[24] V. Henson. The Many Faces of fsck. http://lwn.net/Articles/248180/, September 2007.

[25] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '94)*, San Francisco, California, January 1994.

[26] N. Joukov, A. Rai, and E. Zadok. Increasing Distributed Storage Survivability with a Stackable RAID-like File System. In *Proceedings of the 1st International Workshop on Cluster Security (Cluster-Sec'05)*, Cardiff, UK, 2005.

[27] J. P. J. Kelly and A. A. Avižienis. A Specification-Oriented Multiversion Software Experiment. In *Digest of 13th International Symposium on Fault-Tolerant Computing (FTCS '83)*, Milano, Italy, June 1983.

[28] A. Krioukov, L. N. Bairavasundaram, G. R. Goodson, K. Srinivasan, R. Thelen, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Parity Lost and Parity Regained. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, pages 127–141, San Jose, California, February 2008.

[29] Z. Li, Z. Chen, S. M. Srivivasan, and Y. Zhou. C-miner: Mining block correlations in storage systems. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, pages 173–186, San Francisco, California, April 2004.

[30] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. Fsck - The UNIX File System Check Program. Unix System Manager's Manual - 4.3 BSD Virtual VAX-11 Version, April 1986.

[31] D. Milojicic, A. Messer, J. Shau, G. Fu, and A. Munoz. Increasing Relevance of Memory Hardware Errors: A Case for Recoverable Programming Models. In *9th ACM SIGOPS European Workshop 'Beyond the PC: New Challenges for the Operating System'*, Kolding, Denmark, September 2000.

[32] B. Pawlowski, S. Shepler, C. Beame, B. Callaghan, M. Eisler, D. Noveck, D. Robinson, and R. Thurlow. The NFS Version 4 Protocol. In *Proceedings of the 2nd International System Administration and Networking Conference (SANE 2000)*, May 2000.

[33] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Model-Based Failure Analysis of Journaling File Systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '05)*, pages 802–811, Yokohama, Japan, June 2005.

[34] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 206–220, Brighton, United Kingdom, October 2005.

[35] S. Quinlan and S. Dorward. Venti: A New Approach to Archival Storage. In *Proceedings of the 1st USENIX Symposium on File and Storage Technologies (FAST '02)*, Monterey, California, January 2002.

[36] H. Reiser. ReiserFS. www.namesys.com, 2004.

[37] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using Abstraction to Improve Fault Tolerance. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, October 2001.

[38] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[39] B. Schroeder, E. Pinheiro, and W.-D. Weber. DRAM errors in the wild: A Large-Scale Field Study. In *Proceedings of the 2009 Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/Performance '09)*, Seattle, Washington, June 2007.

[40] Sun Microsystems. ZFS: The last word in file systems. www.sun.com/2004-0914/feature/, 2006.

[41] R. Sundaram. The Private Lives of Disk Drives. http://www.netapp.com/go/techontap/matl/sample/0206tot_resiliency.html, February 2006.

[42] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS File System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '96)*, San Diego, California, January 1996.

[43] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing (Lake George), New York, October 2003.

[44] M. M. Swift, B. N. Bershad, and H. M. Levy. Recovering device drivers. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 1–16, San Francisco, California, December 2004.

[45] T. Ts'o and S. Tweedie. Future Directions for the Ext2/3 Filesystem. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, Monterey, California, June 2002.

[46] S. C. Tweedie. Journaling the Linux ext2fs File System. In *The Fourth Annual Linux Expo*, Durham, North Carolina, May 1998.

[47] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating Byzantine Faults in Transaction Processing Systems using Commit Barrier Scheduling. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, Stevenson, Washington, October 2007.

[48] U. Voges, editor. *Software Diversity in Computerized Control Systems*. Springer, Wien, New York, Dec. 1988.

[49] J. Yang, C. Sar, and D. Engler. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, Washington, November 2006.

[50] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. Engler. Automatically Generating Malicious Disks using Symbolic Execution. In *IEEE Security and Privacy (SP '06)*, Berkeley, California, May 2006.

[51] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using Model Checking to Find Serious File System Errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, California, December 2004.

[52] Y. C. Yeh. Triple-Triple Redundant 777 Primary Flight Computer. In *Proceedings of the 1996 IEEE Aerospace Applications Conference*, 1996.