Refuse to Crash with Re-FUSE

Swaminathan Sundararaman
Computer Sciences Department,
University of Wisconsin-Madison
swami@cs.wisc.edu

Laxman Visampalli
Qualcomm Inc.
laxmanv@qualcomm.com

Andrea C. Arpaci-Dusseau
Remzi H. Arpaci-Dusseau
Computer Sciences Department,
University of Wisconsin-Madison
{dusseau,remzi}@cs.wisc.edu

Abstract

We introduce Re-FUSE, a framework that provides support for restartable user-level file systems. Re-FUSE monitors the user-level file-system and on a crash transparently restarts the file system and restores its state; the restart process is completely transparent to applications. Re-FUSE provides transparent recovery through a combination of novel techniques, including request tagging, system-call logging, and non-interruptible system calls. We tested Re-FUSE with three popular FUSE file systems: NTFS-3g, SSHFS, and AVFS. Through experimentation, we show that Re-FUSE induces little performance overhead and can tolerate a wide range of file-system crashes. More critically, Re-FUSE does so with minimal modification of existing FUSE file systems, thus improving robustness to crashes without mandating intrusive changes.

Categories and Subject Descriptors D.0 [*Software*]: General—File system Reliability

General Terms Reliability, Fault tolerance, Performance

Keywords FUSE, Restartable, User-level File Systems

1. Introduction

File system deployment remains a significant challenge to those developing new and interesting file systems designs [Ganger 2010]. Because of their critical role in the long-term management of data, organizations are sometimes reluctant to embrace new storage technology even though said innovations may address current needs. Similar problems exist in industry, where venture capitalists are loathe to fund storage startups, as it is well known that it takes three to five years for storage products to "harden" and thus become ready for real commercial usage [Vahdat 2010].

One reason for this reluctance to adopt new technology is that unproven software often still has bugs in it, beyond those that are discovered through testing [Lu 2008]. Such "heisenbugs" [Gray 1987] often appear only in deployment, are hard to reproduce, and can lead to system unavailability in the form of a crash.

File system crashes are harmful for two primary reasons. First, when a file system crashes, manual intervention is often required to repair any damage and restart the file system; thus, crashed file systems stay down for noticeable stretches of time and decrease availability dramatically, requiring costly human time to repair. Second, crashes give users the sense that a file system "does not work" and thus decrease the chances for adoption.

To address this problem, we introduce Restartable FUSE (Re-FUSE), a restartable file system layer built as an extension to the Linux FUSE user-level file system infrastructure [Sourceforge 2010a]. Nearly 200 FUSE file systems have already been implemented [Sourceforge 2010b, Wikipedia 2010], indicating that the move towards user-level file systems is significant. In this work, we add a transparent restart framework around FUSE which hides many file-system crashes from users; Re-FUSE simply restarts the file system and user applications continue unharmed.

Restart with Re-FUSE is based on three basic techniques. The first is *request tagging*, which differentiates activities that are being performed on the behalf of concurrent requests; the second is *system-call logging*, which carefully tracks the system calls issued by a user-level file system and caches their results; the third is *non-interruptible system calls*, which ensures that no user-level file-system thread is terminated in the midst of a system call. Together, these three techniques enable Re-FUSE to recover correctly from a crash of a user-level file system by simply re-issuing the calls that the FUSE file system was processing when the crash took place; no user-level application using a user-level file system will notice the failure, except perhaps for a small drop in performance during the restart. Additional performance optimizations, including *page versioning* and *socket*

Eurosys'11, April 10-13, Salzburg, Austria.

buffering, are employed to lower the overheads of logging and recovery mechanisms.

We evaluate Re-FUSE with three popular file systems, NTFS-3g, SSHFS, and AVFS, which differ in their data-access mechanisms, on-disk structures, and features. Less than ten lines of code were added to each of these file systems to make them restartable, showing that the modifications required to use Re-FUSE are minimal. We tested these file systems with both micro- and macro-benchmarks and found that performance overhead during normal operation is minimal. Moreover, recovery time after a crash is small, on the order of a few hundred milliseconds in our tests.

Overall, we find that Re-FUSE successfully detects and recovers from a wide range of fail-stop and transient failures. By doing so, Re-FUSE increases system availability, as most crashes no longer make the entire file system unavailable for long periods of time. Re-FUSE thus removes one critical barrier to the deployment of future file-system technology.

The rest of the paper is organized as follows. Section 2 gives an overview of FUSE and user-level file systems. Section 3 discusses the essentials of a restartable user-level file system framework. Section 4 presents Re-FUSE, and Section 5 describes the three modified FUSE file systems. Section 6 evaluates the robustness and performance of Re-FUSE. Section 7 concludes the paper.

2. FUSE Background

Before delving into Re-FUSE, we first present background on the original FUSE system. We discuss the rationale for such a framework and present its basic architecture.

2.1 Rationale

FUSE was implemented to bridge the gap between features that users want in a file system and those offered in kernel-level file systems. Users want simple yet useful features on top of their favorite kernel-level file systems. Examples of such features are encryption, de-duplication, and accessing files inside archives. Users also want simplified file-system interfaces to access systems like databases, web servers, and new web services such as Amazon S3. The simplified file-system interface obviates the need to learn new tools and languages to access data. Such features and interfaces are lacking in many popular kernel-level file systems.

Kernel-level file-system developers may not be open to the idea of adding all of the features users want in file systems for two reasons. First, adding a new feature requires a significant amount of development and debugging effort [Zadok 2000]. Second, adding a new feature in a tightly coupled system (such as a file system) increases the complexity of the already-large code base. As a result, developers are likely only willing to include functionality that will be useful to the majority of users.

FUSE enables file systems to be developed and deployed at user level and thus simplifies the task of creating a new file system in a number of ways. First, programmers no longer need to have an in-depth understanding of kernel internals (e.g., memory management, VFS, block devices, and network layers). Second, programmers need not understand how these kernel modules interact with others. Third, programmers can easily debug user-level file systems using standard debugging tools such as gdb [GNU 2010] and valgrind [Nethercote 2007]. All of these improvements combine to allow developers to focus on the features they want in a particular file system.

In addition to Linux, FUSE has been developed for FreeBSD [Creo 2010], Solaris [Open Solaris 2010], and OS X [Google Code 2010] operating systems. Though most of our discussion revolves around the Linux version of FUSE, the issues faced herein are likely applicable to FUSE within other systems.

2.2 Architecture

To better understand how FUSE file systems are different than traditional kernel-level file systems, we begin by giving a bit of background on how kernel-level file-systems are structured. In the majority of operating systems, requests to file systems from applications begin at the system-call layer and eventually are routed to the proper underlying file system through a virtual file system (VFS) layer [Kleiman 1986]. The VFS layer provides a unified interface to implement file systems within the kernel, and thus much common code can be removed from the file systems themselves and performed instead within the generic VFS code. For example, VFS code caches file-system objects, thus greatly improving performance when objects are accessed frequently.

FUSE consists of two main components: the *Kernel File-system Module* (KFM) and a user-space library *libfuse* (see Figure 1). The KFM acts as a pseudo file system and queues *application requests* that arrive through the VFS layer. The libfuse layer exports a simplified file-system interface that each user-level file system must implement and acts as a liaison between user-level file systems and the KFM.

A typical application request is processed as follows. First, the application issues a system call, which is routed through VFS to the KFM. The KFM queues this application request (e.g., to read a block from a file) and puts the calling thread to sleep. The user-level file system, through the libfuse interface, retrieves the request off of the queue and begins to process it; in doing so, the user-level file system may issue a number of system calls itself, for example to read or write the local disk, or to communicate with a remote machine via the network. When the request processing is complete, the user-level file system passes the result back through libfuse, which places it within a queue, where the KFM can retrieve it. Finally, the KFM copies the result into the page cache, wakes the application blocked on the request, and returns the desired data to it. Subsequent accesses to the same block will be retrieved from the page cache, without involving the FUSE file system.

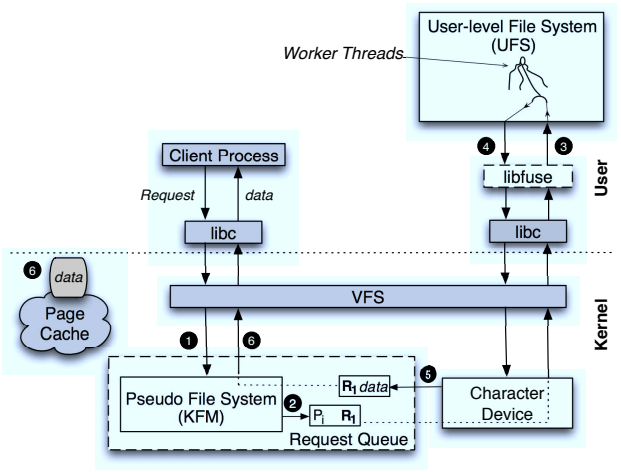

Figure 1. FUSE Framework. *The figure presents the FUSE framework. The user-level file system (in solid white box) is a server process that uses libfuse to communicate with the Kernel-level FUSE Module (KFM). The client process is the application process invoking operations on the file system. File-system requests are processed in the following way: (1) the application sends a request through the KFM via the VFS layer; (2) the request gets tagged and is put inside the request queue; (3) the user-level file-system worker thread dequeues the request; (4) the worker services the request and returns the response; (5) the response is added back to the queue; (6) finally, the KFM copies the data into the page cache before returning it to the application.*

Unlike kernel file systems, where the calling thread executes the bulk of the work, FUSE has a *decoupled* execution model, in which the KFM queues application requests and a separate user-level file system process handles them. As we will see in subsequent sections, this decoupled model is useful in the design of Re-FUSE. In addition, FUSE uses multi-threading to improve concurrency in user-level file systems. Specifically, the libfuse layer allows user-level file-systems to create worker threads to concurrently process file-system requests; as we will see in subsequent sections, such concurrency will complicate Re-FUSE.

The caching architecture of FUSE is also of interest. Because the KFM pretends to be a kernel file system, it must create in-memory objects for each user-level file system object accessed by the application. Doing so improves performance greatly, as in the common case, cached requests can be serviced without consulting the user-level file system.

3. Restartable User-Level File Systems

In this section, we discuss the essentials of a restartable user-level file system framework. We present our goals, and then discuss both our assumptions of the fault model as well as assumptions we make about typical FUSE file systems. We conclude by discussing some challenges a restartable system must overcome, as well as some related approaches.

3.1 Goals

We now present our goals in building a restartable file-system framework for FUSE. Such a framework should have the following four properties:

Generic: A gamut of user-level file-systems exist today. These file systems have varied underlying data-access mechanisms, features, and reliability guarantees. Ideally, the framework should enable any user-level file system to be made restartable with little or no changes.

Application-Transparent: We believe it is difficult for applications using a user-level file system to handle file-system crashes. Expecting every application developer to change and recompile their code to work with a restartable file-system framework is likely untenable. Thus, any restartable framework should be completely transparent to applications and hide failures from them.

Lightweight: FUSE already has significant overheads compared to kernel-level file systems. This additional overhead is attributed to frequent context switching from user to kernel and back as well as extra data copying [Rajgarhia 2010]. Thus, adding significant overhead on top of already slower file-systems is not palatable; a restartable framework should strive to minimize or remove any additional overheads.

Consistent: User-level file systems use different underlying systems (such as databases, web servers, file systems, etc.) to access and store their data. Each of these systems provide different consistency guarantees. The restartable framework should function properly with whatever underlying consistency mechanisms are in use.

3.2 The Fault Model

Faults in a user-level file-system impact availability. A fault could occur due to developer mistakes, an incomplete implementation (such as missing or improper error handling), or a variety of other reasons. On a fault, a user-level file system becomes unavailable until it is restarted.

We believe that user-level file systems are likely to be less reliable than kernel-level file systems, due to a number of factors. First, unlike kernel-level file systems, most user-level file systems are written by novice programmers. Second, no common testing infrastructure exists to detect problems; as a result, systems are likely not stress-tested as carefully as kernel file systems are before release. Finally, no FUSE documentation exists to inform user-level file-system developers about the errors, corner cases, and failure scenarios that a file system should handle.

Our goal is to tolerate a broad class of faults that occur due to programming mistakes and transient changes in the environment. Examples of sources of such faults include sloppy or missing error handling, temporary resource unavailability, memory corruption, and memory leaks. Given the relative inexperience of the developers of many user-level file systems, it is hard to eliminate such failures.

The subset of these failures we seek to address are those that are "fail-stop" and transient [Qin 2005, Swift 2004, Zhou 2006]. In these cases, when such faults are triggered, the system crashes quickly, before ill effects such as permanent data loss can arise; upon retry, the problem is unlikely to re-occur. Faulty error-handling code and certain programming bugs are thus avoided on restart, as the fault that caused these errors to manifest does not take place again.

As with many systems, our goal is not to handle faults caused by basic logic errors and fail-silent bugs. Avoiding logic errors is critical to the correct operation of the file-system; we believe that such bugs should (and likely would) be detected and eliminated during development. On the other hand, fail-silent bugs are more problematic, as they do not crash the system but silently corrupt the in-memory state of the file system. Such corruption could slowly propagate to other components in the system (e.g., the page cache); recovery from such faults is difficult if not impossible. To the best of our knowledge, all previous restartable solutions make the same fail-stop and transient assumption that we make [Candea 2004, David 2008, Qin 2005, Sundararaman 2010, Swift 2003; 2004].

In our failure model, we assume that user-level file-system crashes are due to transient, fail-stop faults. We also assume that all the other components (i.e., the operating system, FUSE itself, and any remote host) work correctly. We believe it is reasonable to make this assumption as the rest of the components that the user-level file system interacts with (i.e., kernel components) are more rigorously tested and used by a larger number of users.

3.3 The User-level File-System Model

To design a restartable framework for FUSE, we must first understand how user-level file systems are commonly implemented; we refer to these assumptions as our *reference model* of a user-level file system.

It is infeasible to examine all FUSE file systems to obtain the "perfect" reference model. Thus, to derive a reference model, we instead analyze six diverse and popular file systems. Table 1 presents details on each of the six file systems we chose to study. NTFS-3g and ext2fuse each are kernel-like file systems "ported" to user space. AVFS allows programs to look inside archives (such as tar and gzip) and TagFS allows users to organize documents using tags inside existing file systems. Finally, SSHFS and HTTPFS allow users to mount remote file systems or websites through the SSH and HTTP protocols, respectively. We now discuss the properties of the reference file-system model.

Simple Threading Model: A single worker thread is responsible for processing a file-system request from start to finish, and only works on a single request at any given time. Amongst the reference-model file systems, only NTFS-3g is single-threaded by default; the rest all operate in multi-threaded mode.

File System	Category	LOC	Downloads
NTFS-3g	block-based	32K	N/A
ext2fuse	block-based	19K	40K
AVFS	pass-through	39K	70K
TagFS	pass-through	2K	400
SSHFS	network-based	4K	93K
HTTPFS	network-based	1K	8K

Table 1. Reference Model File Systems.

Request Splitting: Each request to a user-level file system is eventually translated into one or more system calls. For example, an application-level write request to a NTFS-3g file-system is translated to a sequence of block reads and writes where NTFS-3g reads in the meta-data and data blocks of the file and writes them back after updating them.

Access Through System Calls: Any external calls that the user-level file system needs to make are issued through the system-call interface. These requests are serviced by either the local system (e.g., the disk) or a remote server (e.g., a web server); in either case, system calls are made by the user-level file system in order to access such services.

Output Determinism: For a given request, the user-level file system always performs the same sequence of operations. Thus, on replay of a particular request, the user-level file system outputs the same values as the original invocation [Altekar 2009].

Synchronous Writes: Both dirty data and meta-data generated while serving a request are immediately written back to the underlying system. Unlike kernel-level file systems, a user-level file system does not buffer writes in memory; doing so makes a user-level file system stateless, a property adhered to by many user-level file systems in order to afford a simpler implementation.

Our reference model clearly does not describe all possible user-level file-system behaviors. The FUSE framework does not impose any rules or restrictions on how one should implement a file system; as a result, it is easy to deviate from our reference model, if one desires. We discuss this issue further at the end of Section 4.

3.4 Challenges

FUSE in its current form does not tolerate any file-system mistakes. On a user-level file system crash, the kernel cleans up the resources of the killed file-system process, which forces FUSE to abort all new and in-flight requests of the user-level file system and return an error (a "connection abort") to the application process. The application is thus left responsible for handling failures from the user-level file system. FUSE also prevents any subsequent operations on the crashed file system until a user manually restarts it. As a result, the file system remains unavailable to applications during this process. Three main challenges exist in restarting user-level file systems; we now discuss each in detail.

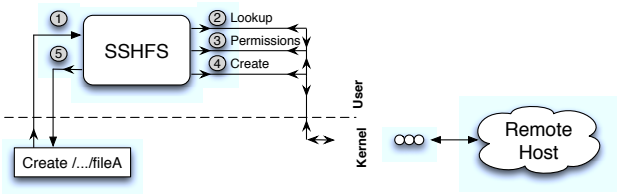

Figure 2. SSHFS Create Operation. *The figure shows a simplified version of SSHFS processing a create request. The number within the gray circle indicates the sequence of steps SSHFS performs to complete the operation. The FUSE, application process, and network components of the OS are not shown for simplicity.*

Generic Recovery Mechanism: Currently there are hundreds of user-level file systems and most of them do not have in-built crash-consistency mechanisms. Crash consistency mechanisms such as journaling or snapshotting could help restore file-system state after a crash. Adding such mechanisms would require significant implementation effort, not only for user-level file-systems but also to the underlying data-management system. Thus, any recovery mechanism should not depend upon the user-level file system itself in order to perform recovery.

Synchronized State: Even if a user-level file system has some in-built crash-consistency mechanism, leveraging such a mechanism could still lead to a disconnect between application perceived file-system state and the state of the recovered file system. This discrepancy arises because crash-consistency mechanisms group file-system operations into a single transaction and periodically commit them to the disk; they are designed only for power failures and not for soft crashes. Hence, on restart, a crash-consistency mechanism only ensures that the file system is restored back to the last known consistent state, which results in a loss of updates that occurred between the last checkpoint and the crash. As applications are not killed on a user-level file-system crash, the file-system state recovered after a crash may not be the same as that perceived by applications. Thus, any recovery mechanism must ensure that the file system and application eventually realize the same view of file system state.

Residual State: The non-idempotent nature of system calls in user-level file systems can leave *residual state* on a crash. This residual state prevents file systems from recreating the state of partially-completed operations. Both undo or redo of partially completed operations through the user-level file system thus may not work in certain situations. The create operation in SSHFS is a good example of such an operation. Figure 2 shows the sequence of steps performed by SSHFS during a create request. SSHFS can crash either before file create (Step 4) or before it returns the result to the FUSE module (Step 5). Undo would incorrectly delete a file if it was already present at the remote host if the crash happened before Step 4; redo would incorrectly return an error to the application if it crashed before Step 5. Thus any recovery mechanism must properly handle residual state.

3.5 Existing Solutions

There has been a great deal of research on restartable systems. Solutions such as CuriOS [David 2008], Rx [Qin 2005], and Microreboot [Candea 2004] help restart and recover application processes from crashes. These solutions require significant implementation effort to both the file system and underlying data-access system and also have high performance overheads. For example, CuriOS heavily instruments file-system code to force the file system to store its state in a separate address space. On restart, CuriOS uses the stored state to rebuild in-memory file-system state, but does not take care of on-disk consistency.

Solutions that use either roll-back [Hitz 1994] or roll-forward [Hagmann 1987, Sweeney 1996, Ts'o 2002] do not work well for user-level file systems. The residual state left by non-idempotent operations coupled with utilization of an underlying data-access system (such as a database) prevent proper recovery using these techniques.

Our earlier work on Membrane [Sundararaman 2010] shows how to restart kernel-level file systems. However, the techniques developed therein are highly tailored to the in-kernel environment and have no applicability to the FUSE context. Thus, a new FUSE-specific approach is warranted.

4. Re-FUSE: Design and Implementation

Re-FUSE is designed to transparently restart the affected user-level file system upon a crash, while applications and the rest of the operating system continue to operate normally. In this section, we first present an overview of our approach. We then discuss how Re-FUSE anticipates, detects, and recovers from faults. We conclude with a discussion of how Re-FUSE leverages many existing aspects of FUSE to make recovery simpler, and some limitations of our approach.

4.1 Overview

The main challenge for Re-FUSE is to restart the user-level file system without losing any updates, while also ensuring the restart activity is both lightweight and transparent. File systems are *stateful*, and as a result, both in-memory and on-disk state needs to be carefully restored after a crash. Three types of work must be done by the system to ensure correct recovery. First is *anticipation*, which is the additional work that must be done during the normal operation of a file system to prepare the file system for a failure. The second is *detection*, which notices a problem has occurred. The third component, *recovery*, is the additional work performed after a failure is detected to restore the file system back to its fully-operational mode.

Unlike existing solutions, Re-FUSE takes a different approach to restoring the consistency of a user-level file system after a file-system crash. After a crash, most existing systems rollback their state to a previous checkpoint and attempt to restore the state by re-executing operations from the beginning [Candea 2004, Qin 2005, Sundararaman 2010]. In con-

trast, Re-FUSE does not attempt to rollback to a consistent state, but rather continues forward from the inconsistent state towards a new consistent state. Re-FUSE does so by allowing partially-completed requests to continue executing from where they were stopped at the time of the crash. This action has the same effect as taking a snapshot of the user-level file system (including on-going operations) just before the crash and resuming from the snapshot during the recovery.

Most of the complexity and novelty in Re-FUSE comes in the fault anticipation component of the system. We now discuss this piece in greater detail, before presenting the more standard detection and recovery protocols.

4.2 Fault Anticipation

In anticipation of faults, Re-FUSE must perform a number of activities in order to ensure it can properly recover once the said fault arises. Specifically, Re-FUSE must track the progress of application-level file-system requests in order to continue executing them from their last state once a crash occurs. The inconsistency in file-system state is caused by partially-completed operations at the time of the crash; fault anticipation must do enough work during normal operation in order to help the file system move to a consistent state during recovery.

To create light-weight continuous snapshots of a user-level file system, Re-FUSE fault anticipation uses three different techniques: request tagging, system-call logging, and uninterruptible system calls. Re-FUSE also optimizes its performance through page versioning. We now discuss each of these in detail.

4.2.1 Request Tagging

Tracking the progress of each file-system request is difficult in the current FUSE implementation. The decoupled execution model of FUSE combined with request splitting at the user-level file system makes it hard for Re-FUSE to correlate an application request with the system calls performed by a user-level file system to service said application request.

Request tagging enables Re-FUSE to correlate application requests with the system calls that each user-level file system makes on behalf of the request. As the name suggests, request tagging transparently adds a request ID to the task structure of the file-system process (i.e., worker thread) that services it.

Re-FUSE instruments the libfuse layer to automatically set the ID of the application request in the task structure of the file-system thread whenever it receives a request from the KFM. Re-FUSE adds an additional attribute to the task structure to store the request ID. Any system call that the thread issues on behalf of the request thus has the ID in its task structure. On a system call, Re-FUSE inspects the tagged request ID in the task structure of the process to correlate the system call with the original application request. Re-FUSE also uses the tagged request ID in the task structure of the file-system process to differentiate system calls made by the user-level file system from other processes in the operating system. Figure 3 presents these steps in more detail.

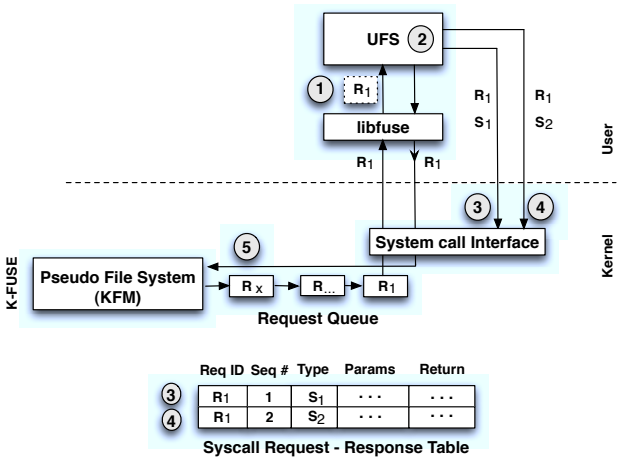

Figure 3. Request Tagging and System-call Logging. *The figure shows how Re-FUSE tracks the progress of individual file-system request. When KFM queues the* application requests *(denoted by R with a subscript). Re-FUSE tracks the progress of the request in the following way: (1) the request identifier is transparently attached to the task structure of the worker thread at the libfuse layer; (2) the user-level file system worker thread issues one or more system calls (denoted by S with a subscript) while processing the request; (3 and 4) Re-FUSE (at the system call interface) identifies these calls through the request ID in the caller's task structure and logs the input parameters along with the return value; (5) the KFM, upon receiving the response from the user-level file system for a request, deletes its entries from the log.*

4.2.2 System-Call Logging

Re-FUSE checkpoints the progress of individual application requests inside the user-level file system by logging the system calls that the user-level file system makes in the context of the request. On a restart, when the request is re-executed by the user-level file system, Re-FUSE returns the results from recorded state to mimic its execution.

The logged state contains the type, input arguments, and the response (return value and data), along with a request ID, and is stored in a hash table called the *syscall request-response table*. This hash table is indexed by the ID of the application request. Figure 3 shows how system-call logging takes place during regular operations.

Re-FUSE maintains the number of system calls that a file-system process makes to differentiate between user-level file-system requests to the same system call with identical parameters. For example, on a create request, NTFS-3g reads the same meta-data block multiple times between other read and write operations. Without a sequence number, it would be difficult to identify its corresponding entry in the syscall request-response table. Additionally, the sequence number also serves as a sanity check to verify that

the system calls happen in the same order during replay. Re-FUSE removes the entries of the application request from the hash table when the user-level file system returns the response to the KFM.

4.2.3 Non-interruptible System Calls

The threading model in Linux prevents this basic logging approach from working correctly. Specifically, the threading model in Linux forces all threads of a process to be killed when one of the thread terminates (or crashes) due to a bug. Moreover, the other threads are killed independent of whether they are executing in user or kernel mode. Our logging approach only works if the system call issued by the user-level file system finishes completely, as a partially-completed system call could leave some residual state inside the kernel, thus preventing correct replay of in-flight requests.

To remedy this problem, Re-FUSE introduces the notion of *non-interruptible system calls*. Such a system call provides the guarantee that if a system call starts executing a request, it continues until its completion. Of course, the system call can still complete by returning an error, but the worker thread executing the system call cannot be killed prematurely when one of its sibling threads is killed within the user-level file-system. In other words, by using non-interruptible system calls, Re-FUSE allows a user-level file-system thread to continue to execute a system call to completion even when another file-system thread is terminated due to a crash.

Re-FUSE implements non-interruptible system calls by changing the default termination behavior of a thread group in Linux. Specifically, Re-FUSE modifies the termination behavior in the following way: when a thread abruptly terminates, Re-FUSE allows other threads in the group to complete whatever system call they are processing until they are about to return the status (and data) to the user. Re-FUSE then terminates said threads after logging their responses (including the data) to the syscall request-response table.

Re-FUSE eagerly copies input parameters to ensure that the crashed process does not infect the kernel. Lazy copying of input parameters to a system call in Linux could potentially corrupt the kernel state as non-interruptible system calls allow other threads to continue accessing the process state. Re-FUSE prevents access to corrupt input arguments by eagerly copying in parameters from the user buffer into the kernel and also by skipping COPY_FROM_USER and COPY_TO_USER functions after a crash. It is important to note that the process state is never accessed within a system call except for copying arguments from the user to the kernel at the beginning. Moreover, non-interruptible system calls are enforced only for user-level file system processes (i.e., only for processes that have a FUSE request ID set in their task structure). As a result, other application processes remain unaffected by non-interruptible system calls.

4.2.4 Performance Optimizations

Logging responses of read operations has high overheads in terms of both time and space as we also need to log the data returned with each read request. To reduce these overheads, instead of storing the data as part of the log records, Re-FUSE implements *page versioning*, which can greatly improve performance. Re-FUSE first tracks the pages accessed (and also returned) during each read request and then marks them as copy-on-write. The operating system automatically creates a new version whenever a subsequent request modifies the previously-marked page. The copy-on-write flag on the marked pages is removed when the response is returned back from the user-level file system to the KFM layer. Once the response is returned back, the file-system request is removed from the request queue at the KFM layer and need not be replayed back after a crash.

Page versioning does not work for network-based file systems, which use socket buffers to send and receive data. To reduce the overheads of logging read operations, Re-FUSE also caches the socket buffers of the file-system requests until the request completes.

4.3 Fault Detection

Re-FUSE detects faults in a user-level file-system through file-system crashes. As discussed earlier, Re-FUSE only handles faults that are both transient and fail-stop. Unlike kernel-level file systems, detection of faults in a user-level file system is simple. The faults Re-FUSE attempts to recover crash the file-system as soon as they are triggered (see Section 3.2). Re-FUSE inspects the return value and the signal attached to the killed file-system process to differentiate between regular termination and a crash.

Re-FUSE currently only implements a lightweight fault-detection mechanism. Fault detection can be further hardened in user-level file systems by applying techniques used in other systems [Cowan 1998, Necula 2005, Zhou 2006]. Such techniques can help to automatically add checks (by code or binary instrumentation) to crash file systems more quickly when certain types of bugs are encountered (e.g., out-of-bounds memory accesses).

4.4 Fault Recovery

The recovery subsystem is responsible for restarting and restoring the state of the crashed user-level file system. To restore the in-memory state of the crashed user-level file system, Re-FUSE leverages the information about the file-system state available through the KFM. Recovery after a crash mainly consists of the following steps: cleanup, re-initialize, restore the in-memory state of the user-level file system, and re-execute the in-flight file-system requests at the time of the crash. The decoupled execution model in the FUSE preserves application state on a crash. Hence, application state need not be restored. We now explain the steps in the recovery process in detail.

The operating system automatically cleans up the resources used by a user-level file system on a crash. The file system is run as a normal process with no special privileges by the FUSE. On a crash, like other killed user-level processes, the operating system cleans up the resources of the file system, obviating the need for explicit state clean up.

Re-FUSE holds an extra reference on the FUSE device file object owned by the crashed process. This file object is the gateway to the request queue that was being handled by the crashed process and KFM's view of the file system. Instead of doing a new mount operation, the file-system process sends a restart message to the KFM to attach itself to the old instance of the file system in KFM. This action also informs the KFM to initiate the recovery process for the particular file system.

The in-memory file-system state required to execute file-system requests is restored using the state cached inside the kernel (i.e., the VFS layer). Re-FUSE then exploits the following property: an access on a user-level file-system object through the KFM layer recreates it. Re-FUSE performs a lookup for each of the object cached in the VFS layer, which recreates the corresponding user-level file-system object in memory. Re-FUSE also uses the information returned in each call to point the cached VFS objects to the newly created file-system object. It is important to note that lookups do not recreate all file-system objects but only those required to re-execute both in-flight and new requests. To speed up recovery, Re-FUSE looks up file-system objects lazily.

Finally, Re-FUSE restores the on-disk consistency of the user-level file-system by re-executing in-flight requests. To re-execute the crashed file-system requests, a copy of each request that is available in the KFM layer is put back on the request queue for the restarted file system. For each replayed request, the FUSE request ID, sequence number of the external call, and input arguments are matched with the entry in the syscall request-response table and if they match correctly, the cached results are returned to the user-level file system. If the previously encountered fault is transient, the user-level file system successfully executes the request to completion and returns the results to the waiting application.

On an error during recovery, Re-FUSE falls back to the default FUSE behavior, which is to crash the user-level file system and wait for the user to manually restart the file system. An error could be due to a non-transient fault or a mismatch in one or more input arguments in the replayed system call (i.e., violating our assumptions about the reference file-system model). Before giving up on recovering the file system, Re-FUSE dumps useful debugging information about the error for the file-system developer.

4.5 Leveraging FUSE

The design of FUSE simplifies the recovery process in a user-level file system for the following four reasons. First, in FUSE, the file-system is run as a stand-alone user-level process. On a file-system crash, only the file-system process is killed and other components such as FUSE, the operating system, local file system, and even a remote host are not corrupted and continue to work normally.

Second, the decoupled execution model blocks the application issuing the file-system request at the kernel level (i.e., inside KFM) and a separate file-system process executes the request on behalf of the application. On a crash, the decoupled execution model preserves application state and also provides a copy of file-system requests that are being serviced by the user-level file system.

Third, requests from applications to a user-level file system are routed through the VFS layer. As a result, the VFS layer creates an equivalent copy of the in-memory state of the file system inside the kernel. Any access (such as a lookup) to the user-level file system using the in-kernel copy recreates the corresponding in-memory object.

Finally, application requests propagated from KFM to a user-level file system are always idempotent (i.e., this idempotency is enforced by the FUSE interface). The KFM layer ensures idempotency of operations by changing all relative arguments from the application to absolute arguments before forwarding it to the user-level file system. The idempotent requests from the KFM allow requests to be re-executed without any side effects. For example, the read system call does not take the file offset as an argument and uses the current file offset of the requesting process; the KFM converts this relative offset to an absolute offset (i.e., an offset from beginning of the file) during a read request.

4.6 Limitations

Our approach is obviously not without limitations. First, one of the assumptions that Re-FUSE makes for handling non-idempotency is that operations execute in the same sequence every time during replay. If file systems have some internal non-determinism, additional support would be required from the remote (or host) system to undo the partially-completed operations of the file system. For example, consider block allocation inside a file system. The block allocation process is deterministic in most file systems today; however, if the file system randomly picked a block during allocation, the arguments to the subsequent replay operations (i.e., the block number of the bitmap block) would change and thus could potentially leave the file system in an inconsistent state.

Re-FUSE does not currently support all I/O interfaces. For example, file systems cannot use mmap to write back data to the underlying system as updates to mapped files are not immediately visible through the system-call interface. Similarly, page versioning does not work in direct-I/O mode; Re-FUSE needs the data to be cached within the page cache.

Multi-threading can also limit the applicability of Re-FUSE. For example, multi-threading in block-based file systems could lead to race conditions during replay of in-flight requests and hence data loss after recovery. Different threading models could also involve multiple threads to handle a single request. For such systems, the FUSE request ID needs

Component	Original	Added	Modified
libfuse	9K	250	8
KFM	4K	750	10
Total	**13K**	**1K**	**18**

FUSE Changes

Component	Original	Added	Modified
VFS	37K	3K	0
MM	28K	250	1
NET	16K	60	0
Total	**81K**	**3.3K**	**1**

Kernel Changes

Table 2. Implementation Effort. *The table presents the code changes required to transform FUSE and Linux 2.6.18 into their restartable counterparts.*

to be explicitly transferred between the (worker) threads so that the operating system can identify the FUSE request ID for which the corresponding system call is issued.

The file systems in our reference model do not cache data in user space, but user-level file systems certainly could do so to improve performance (e.g., to reduce the disk or network traffic). For such systems, the assumption about the completion of requests (by the time the response is written back) would be broken and result in lost updates after a restart. One solution to handle this issue is to add a commit protocol to the request-handling logic, where in addition to sending a response message back, the user-level file system should also issue a commit message after the write request is completed. Requests in the KFM could be safely thrown away from the request queue only after a commit message is received from the user-level file system. In the event of a crash, all cached requests for which the commit message has not been received will be replayed to restore file-system state. For multi-threaded file systems, Re-FUSE would also need to maintain the execution order of requests to ensure correct replay. Moreover, if a user-level file system internally maintains a special cache (for some reason), for correct recovery, the file system would need to to explicitly synchronize the contents of the cache with Re-FUSE.

4.7 Implementation Statistics

Our Re-FUSE prototype is implemented in Linux 2.6.18 and FUSE 2.7.4. Table 2 shows the code changes done in both FUSE and the kernel proper. For Re-FUSE, around 3300 and 1000 lines of code were added to the Linux kernel and FUSE, respectively. The code changes in libfuse include request tagging, fault detection, and state restoration; changes in KFM center around support for recovery. The code changes in the VFS layer correspond to the support for system-call logging, and modifications in the MM and NET modules correspond to page versioning and socket-buffer caching respectively.

5. Re-FUSE File Systems

Re-FUSE is not entirely transparent to user-level file systems. We briefly describe the minor changes required in the three file systems employed in this work.

NTFS-3g: NTFS-3g reads a few key metadata pages into memory during initialization, just after the creation of the file system, and uses these cached pages to handle subsequent requests. However, any changes to these key metadata pages are immediately written back to disk while processing requests. On a restart of the file-system process, NTFS-3g would again perform the same initialization process. However, if we allow the process to read the current version of the metadata pages, it could potentially access inconsistent data and may thus fail. To avoid this situation, we return the oldest version of the metadata page on restart, as the oldest version points to the version that existed before the handling of a particular request (note that NTFS-3g operates in single-threaded mode).

AVFS: To make AVFS work with Re-FUSE, we simply increment the reference count of open files and cache the file descriptor so that we can return the same file handle when it is reopened again after a restart.

SSHFS: To make SSHFS work correctly with Re-FUSE, we made the following changes to SSHFS. SSHFS internally generates its own request IDs to match the responses from the remote host with waiting requests. The request IDs are stored inside SSHFS and are lost on a crash. After restart, on replay of an in-flight request, SSHFS generates new request IDs which could be different than the old ones. In order to match new request IDs with the old ones, Re-FUSE uses the FUSE request ID tagged in the worker thread along with the sequence number. Once requests are matched, Re-FUSE correctly returns the cached response. Also, to mask the SSHFS crash from the remote server, Re-FUSE holds an extra reference count on the network socket, and re-attaches it to the new process that is created. Without this action, upon a restart, SSHFS would start a new session, and the cached file handle would not be valid in the new session.

6. Evaluation

We now evaluate Re-FUSE in the following three categories: generality, robustness, and performance. Generality helps to demonstrate that our solution can be easily applied to other file systems with little or no change. Robustness helps show the correctness of Re-FUSE. Performance results help us analyze the overheads during regular operations and during a crash to see if they are acceptable.

All experiments were performed on a machine with a 2.2 GHz Opteron processor, two 80GB WDC disks, and 2GB of memory running Linux 2.6.18. We evaluated Re-FUSE with FUSE (2.7.4) using NTFS-3g (2009.4.4), AVFS (0.9.8), and SSHFS (2.2) file systems. For SSHFS, we use public-key authentication to avoid typing the password on restart.

File System	Original	Added	Modified
NTFS-3g	32K	10	1
AVFS	39K	4	1
SSHFS	4K	3	2

Table 3. Implementation Complexity. *The table presents the code changes required to transform NTFS-3g, AVFS and SSHFS into their restartable counterparts.*

6.1 Generality

To show Re-FUSE can be used by many user-level file systems, we chose NTFS-3g, AVFS, and SSHFS. These file systems are different in their underlying data access mechanism, reliability guarantees, features, and usage. Table 3 shows the code changes required in each of these file systems to work with Re-FUSE.

From the table, we can see that file-system specific changes required to work with Re-FUSE are minimal. To each user-level file system, we have added less than 10 lines of code, and modified a few more. Some of these lines were added to daemonize the file system and to restart the process in the event of a crash. A few further lines were added or modified to make recovery work properly, as discussed previously in Section 5.

6.2 Robustness

To analyze the robustness of Re-FUSE, we use fault injection. We employ both controlled and random fault-injection to show the inability of current user-level file systems to tolerate faults and how Re-FUSE helps them.

The injected faults are fail-stop and transient. These faults try to mimic some of the possible crash scenarios in user-level file systems. We first run the fault injection experiments on a vanilla user-level file system running over FUSE and then compare the results by repeating them over the adapted user-level file system running over Re-FUSE both with and without kernel modifications. The experiments without the kernel modifications are denoted by *Restart* and those with the kernel changes are denoted by *Re-FUSE*. We include the restart column to show that, without the kernel support, simple restart and replay of in-flight operations does not work well for FUSE.

6.2.1 Controlled Fault Injection

We employ controlled fault injection to understand how user-level file systems react to failures. In these experiments, we exercise different file-system code paths (e.g., `create()`, `mkdir()`, etc.) and crash the file system by injecting transient faults (such as a null-pointer dereference) in these code paths. We performed a total of 60 fault-injection experiments for all three file systems; we present the user-visible results.

User-visible results help analyze the impact of a fault both at the application and the file-system level. We choose *application state*, *file-system consistency*, and *file-system state* as the user-visible metrics of interest. Application state indicates how a fault affects the execution of the application that uses the user-level file system. File-system consistency indicates if a potential data loss could occur as a result of a fault. File-system state indicates if a file system can continue servicing subsequent requests after a fault.

Table 4 summarizes the results of our fault-injection experiments. The caption explains how to interpret the data in the table. We now discuss the major observations and the conclusions of our fault-injection experiments.

First, we analyze the vanilla versions of the file systems running on vanilla FUSE and a vanilla Linux kernel. The results are shown in the leftmost result column in Table 4. We observe that the vanilla versions of user-level file systems and FUSE do a poor job in hiding failures from applications. In all experiments, the user-level file system is unusable after the fault; as a result, applications have to prematurely terminate their requests after receiving an error (a "software-caused connection abort") from FUSE. Moreover, in 40% of the cases, crashes lead to inconsistent file system state.

Second, we analyze the usefulness of fault-detection and simple restart at the KFM *without* any explicit support from the operating system. The second result column (denoted by Restart) of Table 4 shows the result. We observe that a simple restart of the user-level file system and replay of in-flight requests at the KFM layer ensures that the application completes the failed operation in the majority of the cases (around 60%). It still cannot, however, re-execute a significant amount (around 40%) of partially-completed operations due to the non-idempotent nature of the particular file-system operation. Moreover, an error is wrongly returned to the application and the crashes leave the file system in an inconsistent state.

Finally, we analyze the usefulness of Re-FUSE that includes restarting the crashed user-level file system, replaying in-flight requests, and has support from the operating system for re-executing non-idempotent operations (i.e., all the support described in Section 4). The results of the experiments are shown in the rightmost column of Table 4. From the table, we can see that all faults are handled properly, applications successfully complete the operation, and the file system is always left in a consistent state.

6.2.2 Random Fault Injection

In order to stress the robustness of our system, we use random fault injection. In the random fault-injection experiments, we arbitrarily crash the user-level file system during different workloads and observe the user-visible results. The sort, Postmark, and OpenSSH macro-benchmarks are used as workloads for these experiments; each is described further below. We perform the experiments on the vanilla versions of the user-level file systems, FUSE and Linux kernel, and on the adapted versions of the user-level file systems that run with Re-FUSE.

		NTFS-3g								
		Regular			Restart			Re-Fuse		
Operation	NTFS_fn	Application?	FS:Consistent?	FS:Usable?	Application?	FS:Consistent?	FS:Usable?	Application?	FS:Consistent?	FS:Usable?
create	fuse_create	×	×	×	e	×	√	√	√	√
mkdir	fuse_create	×	×	×	e	×	√	√	√	√
symlink	fuse_create	×	×	×	e	×	√	√	√	√
link	link	×	×	×	e	×	√	√	√	√
rename	link	×	×	×	e	×	√	√	√	√
open	fuse_open	×	√	×	√	√	√	√	√	√
read	fuse_read	×	√	×	√	√	√	√	√	√
readdir	fuse_readdir	×	√	×	√	√	√	√	√	√
readlink	fuse_readlink	×	√	×	√	√	√	√	√	√
write	fuse_write	×	×	×	√	×	√	√	√	√
unlink	delete	×	×	×	e	×	√	√	√	√
rmdir	inode_sync	×	×	×	e	×	√	√	√	√
truncate	fuse_truncate	×	×	×	√	×	√	√	√	√
utime	inode_sync	×	√	×	√	√	√	√	√	√
		SSHFS								
	SSHFS_fn	Regular			Restart			Re-FUSE		
create	open_common	×	√	×	e	√	√	√	√	√
mkdir	mkdir	×	√	×	e	√	√	√	√	√
symlink	symlink	×	√	×	e	√	√	√	√	√
rename	rename	×	√	×	e	√	√	√	√	√
open	open_common	×	√	×	√	√	√	√	√	√
read	sync_read	×	√	×	√	√	√	√	√	√
readdir	getdir	×	√	×	√	√	√	√	√	√
readlink	readlink	×	√	×	√	√	√	√	√	√
write	write	×	√	×	√	√	√	√	√	√
unlink	unlink	×	√	×	e	√	√	√	√	√
rmdir	rmdir	×	√	×	e	√	√	√	√	√
truncate	truncate	×	√	×	√	√	√	√	√	√
chmod	chmod	×	√	×	√	√	√	√	√	√
stat	getattr	×	√	×	√	√	√	√	√	√
		AVFS								
	AVFS_fn	Regular			Restart			Re-FUSE		
create	mknod	×	×	×	e	×	√	√	√	√
mkdir	mkdir	×	×	×	e	×	√	√	√	√
symlink	symlink	×	×	×	e	×	√	√	√	√
link	link	×	×	×	e	×	√	√	√	√
rename	rename	×	×	×	e	×	√	√	√	√
open	open	×	√	×	√	√	√	√	√	√
read	read	×	√	×	√	√	√	√	√	√
readdir	readdir	×	√	×	√	√	√	√	√	√
readlink	readlink	×	√	×	√	√	√	√	√	√
write	write	×	×	×	√	×	√	√	√	√
unlink	unlink	×	×	×	e	×	√	√	√	√
rmdir	rmdir	×	×	×	e	×	√	√	√	√
truncate	truncate	×	×	×	√	×	√	√	√	√
chmod	chmod	×	√	×	√	√	√	√	√	√
stat	getattr	×	√	×	√	√	√	√	√	√

Table 4. Fault Study. *The table shows the affect of fault injections on the behavior of NTFS-3g, SSHFS and AVFS, respectively. Each row presents the results of a single experiment, and the columns show (in left-to-right order) the intended operation, the file system function that was fault injected, how it affected the application, whether the file system was consistent after the fault, and whether the file system was usable for other operations. Various symbols are used to condense the presentation. For application behavior, "√": application observed successful completion of the operation; "×": application received the error "software caused connection abort"; "e": application incorrectly received an error.*

File System + Re-FUSE	Injected Faults	Sort (Survived)	OpenSSH (Survived)	Postmark (Survived)
NTFS-3g	100	100	100	100
SSHFS	100	100	100	100
AVFS	100	100	100	100

Table 5. Random Fault Injection. *The table shows the affect of randomly injected crashes on the three file systems supported with Re-FUSE. The second column refers to the total number of random (in terms of the crash point in the code) crashes injected into the file system during the span of time it is serving a macro-benchmark. The crashes are injected by sending the signal SIGSEGV to the file system process periodically. The right-most three columns indicate the number of survived crashes by the reinforced file systems during each macro-benchmark. We do not include the results of the experiments on the vanilla versions of these file systems in the table; those file systems remain unusable after the first crash even though we inject the crash at varied time-points during the workload.*

We use three commonly-used macro-benchmarks to help analyze file-system robustness (and later, performance). Specifically, we utilize the sort utility, Postmark [Katcher 1997], and OpenSSH [Sourceforge 2010c]. The sort benchmark represents data-manipulation workloads, Postmark represents I/O-intensive workloads, and OpenSSH represents user-desktop workloads.

Table 5 presents the result of our study. From the table, Re-FUSE ensures that the application continues executing through the failures, thus making progress. We also found that a vanilla user-level file system with no support for fault handling cannot tolerate crashes (not shown in the table).

In summary, both from controlled and random fault injection experiments, we clearly see the usefulness of Re-FUSE in recovering from user-level file system crashes. In a standard environment, a user-level file system is always unusable after the crash and applications using the user-level file system are killed. Moreover, in many cases, the file system is also left in an inconsistent state. In contrast, Re-FUSE, upon detecting a user-level file system crash, transparently restarts the crashed user-level file system and restores it to a consistent and usable state. It is important to understand that even though Re-FUSE recovers cleanly from both controlled and random faults, it is still limited in its applicability (i.e., Re-FUSE only works for faults that are both fail-stop and transient and for file systems that strictly adhere to the reference file-system model described in Section 3.3).

6.3 Performance

Though fault-tolerance is our primary goal, we also evaluate the performance of Re-FUSE in the context of regular operations and recovery time.

Benchmark	ntfs	ntfs+ Re-FUSE	overhead %	sshfs	sshfs+ Re-FUSE	Overhead %	avfs	avfs+ Re-FUSE	Overhead %
Sequential read	9.2	9.2	0.0	91.8	91.9	0.1	17.1	17.2	0.6
Sequential write	13.1	14.2	8.4	519.7	519.8	0.0	17.9	17.9	0.0
Random read	150.5	150.5	0.0	58.6	59.5	1.5	154.4	154.4	0.0
Random write	11.3	12.4	9.7	90.4	90.8	0.4	53.2	53.7	0.9
Create	20.6	23.2	12.6	485.7	485.8	0.0	17.1	17.2	0.6
Delete	1.4	1.4	0.0	2.9	3.0	3.4	1.6	1.6	0.0

Table 6. Microbenchmarks. *This table compares the execution time (in seconds) for various benchmarks for restartable versions of ntfs-3g, sshfs, avfs (on Re-FUSE) against their regular versions on the unmodified kernel. Sequential reads/writes are 4 KB 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. Create/delete copies/removes 1000 files each of size 1MB to/from the file system respectively. All workloads use a cold file-system cache.*

Benchmark	ntfs	ntfs+ Re-FUSE	Overhead %	sshfs	sshfs+ Re-FUSE	Overhead %	avfs	avfs+ Re-FUSE	Overhead %
Sort	133.5	134.2	0.5	145.0	145.2	0.1	129.0	130.3	1.0
OpenSSH	32.5	32.5	0.0	55.8	56.4	1.1	28.9	29.3	1.4
PostMark	112.0	113.0	0.9	5683	5689	0.1	141.0	143.0	1.4

Table 7. Macrobenchmarks. *The table presents the performance (in seconds) of different benchmarks running on both standard and restartable versions of ntfs-3g, sshfs, and avfs. The sort benchmark (CPU intensive) sorts roughly 100MB of text using the command-line sort utility. For the OpenSSH benchmark (CPU+I/O intensive), we measure the time to copy, untar, configure, and make the OpenSSH 4.51 source code. PostMark (I/O intensive) parameters are: 3000 files (sizes 4KB to 4MB), 60000 transactions, and 50/50 read/append and create/delete biases.*

6.3.1 Regular Operations

We now evaluate the performance of Re-FUSE. Specifically, we measure the overhead of our system during regular operations and also during user-level file system crashes to see if a user-level file system running on Re-FUSE has acceptable overheads. We use both micro- and macro-benchmarks to evaluate the overheads during regular operation.

Micro-benchmarks help analyze file-system performance for frequently performed operations in isolation. We use sequential read/write, random read/write, create, and delete operations as our micro benchmarks. These operations exercise the most frequently accessed code paths in file systems. The caption in Table 6 describes our micro-benchmark configuration in more detail. We also use the previously-described macro-benchmarks sort, Postmark, and OpenSSH; the caption in Table 7 describes the exact configuration parameters for our experiments.

Table 6 and Table 7 show the results of micro- and macro-benchmarks respectively. From the tables, we can see that for both micro- and macro-benchmarks, Re-FUSE has minimal overhead, often less than 3%. The overheads are small due to in-memory logging and our optimization through page versioning (or socket buffer caching in the context of SSHFS). These results show that the additional reliability Re-FUSE achieves comes with negligible overhead for common file-system workloads, thus removing one important barrier of adoption for Re-FUSE.

	Vanilla	Re-FUSE	
File System	Total Time (s)	Total Time (s)	Restart Time (ms)
NTFS-3g	133.5	134.45	65.54
SSHFS	145.0	145.4	255.8
AVFS	129.0	130.7	6.0

Table 8. Restart Time in Re-FUSE. *The table shows the impact of a single restart on the restartable versions of the file systems. The benchmark used is sort and the restart is triggered approximately mid-way through the benchmark.*

6.3.2 Recovery Time

We now measure the overhead of recovery time in Re-FUSE. Recovery time is the time Re-FUSE takes to restart and restore the state of the crashed user-level file system. To measure the recovery-time overhead, we ran the sort benchmark for ten times and crashed the file system half-way through each run. Sort is a good benchmark for testing recovery as it makes many I/O system calls and both reads and updates file-system state.

Table 8 shows the elapsed time and the average time Re-FUSE spent in restoring the crashed user-level file system state. The restoration process includes restart of the user-level file-system process and restoring its in-memory state. From the table, we can see that the restart time is in the order of a few milliseconds. The application also does not see any observable increase in its execution time due to the user-level file-system crash.

7. Conclusions

"*Failure is not falling down but refusing to get up.*"
–Chinese Proverb

Software imperfections are common and are a fact of life especially for code that has not been well tested. Even though user-level file systems crashes are isolated from the operating system by FUSE, the reliability of individual file systems has not necessarily improved. File systems still remain unavailable to applications after a crash. Re-FUSE embraces the fact that failures sometimes occur and provides a framework to transparently restart crashed file systems.

We develop a number of new techniques to enable efficient and correct user-level file system restartability. In particular, request tagging allows Re-FUSE to differentiate between concurrently-serviced requests; system-call logging enables Re-FUSE to track (and eventually, replay) the sequence of operations performed by a user-level file system; non-interruptible system calls ensure that user-level file-system threads move to a reasonable state before file system recovery begins. Through experiments, we demonstrate that our techniques are reasonable in their performance overheads and effective at detection and recovery from a certain class of faults.

In the future, much work can be done to enhance Re-FUSE. More file systems can be ported to use it, and more experience with the real pitfalls of running a file system within such a framework can be obtained. It is unlikely developers will ever build the "perfect" file system; Re-FUSE presents one way to tolerate these imperfections.

Acknowledgments

We thank the anonymous reviewers and Steve Gribble (our shepherd) for their feedback and comments, which have substantially improved the content and presentation of this paper. We also thank Sriram Subramanian, Ashok Anand, Sankaralingam Panneerselvam, Mohit Saxena, and Asim Kadav for their comments on earlier drafts of the paper.

This material is based upon work supported by the National Science Foundation under the following grants: CCF-0621487, CNS-0509474, CNS-0834392, CCF-0811697, CCF-0811697, CCF-0937959, as well as by generous donations from NetApp and Google.

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

References

[Altekar 2009] Gautam Altekar and Ion Stoica. Odr: output-deterministic replay for multicore debugging. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '07)*, pages 193–206, Big Sky, Montana, October 2009.

[Candea 2004] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot – A Technique for Cheap Recovery. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 31–44, San Francisco, California, December 2004.

[Cowan 1998] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium (Sec '98)*, San Antonio, Texas, January 1998.

[Creo 2010] Creo. Fuse for FreeBSD. http://fuse4bsd.creo.hu/, 2010.

[David 2008] Francis M. David, Ellick M. Chan, Jeffrey C. Carlyle, and Roy 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.

[Ganger 2010] Greg Ganger. File System Virtual Machines. http://www.pdl.cmu.edu/FSVA/index.shtml, 2010.

[GNU 2010] GNU. The GNU Project Debugger. http://www.gnu.org/software/gdb, 2010.

[Google Code 2010] Google Code. MacFUSE. http://code.google.com/p/macfuse/, 2010.

[Gray 1987] Jim Gray. Why Do Computers Stop and What Can We Do About It? In *6th International Conference on Reliability and Distributed Databases*, June 1987.

[Hagmann 1987] Robert 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.

[Hitz 1994] Dave Hitz, James Lau, and Michael 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.

[Katcher 1997] Jeffrey Katcher. PostMark: A New File System Benchmark. Technical Report TR-3022, Network Appliance Inc., October 1997.

[Kleiman 1986] Steve R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *Proceedings of the USENIX Summer Technical Conference (USENIX Summer '86)*, pages 238–247, Atlanta, Georgia, June 1986.

[Lu 2008] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from Mistakes — A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*, Seattle, Washington, March 2008.

[Necula 2005] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: Type-Safe

Retrofitting of Legacy Software. *ACM Transactions on Programming Languages and Systems*, 27(3), May 2005.

[Nethercote 2007] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, 2007. ISSN 0362-1340.

[Open Solaris 2010] Open Solaris. Fuse on Solaris. http://hub.opensolaris.org/bin/view/Project+fuse/, 2010.

[Qin 2005] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating Bugs As Allergies. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, Brighton, United Kingdom, October 2005.

[Rajgarhia 2010] Aditya Rajgarhia and Ashish Gehani. Performance and extension of user space file systems. In *SAC '10: Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 206–213, New York, NY, USA, 2010. ACM.

[Sourceforge 2010a] Sourceforge. AVFS: A Virtual Filesystem. http://sourceforge.net/projects/avf/, 2010.

[Sourceforge 2010b] Sourceforge. File systems using FUSE. http://sourceforge.net/apps/mediawiki/fuse/index.php?title=FileSystems, 2010.

[Sourceforge 2010c] Sourceforge. OpenSSH. http://www.openssh.com/, 2010.

[Sundararaman 2010] Swaminathan Sundararaman, Sriram Subramanian, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Membrane: Operating System Support for Restartable File Systems. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST '10)*, San Jose, California, February 2010.

[Sweeney 1996] Adan Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS File System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '96)*, San Diego, California, January 1996.

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

[Swift 2004] Michael M. Swift, Brian N. Bershad, and Henry 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.

[Ts'o 2002] Theodore Ts'o and Stephen Tweedie. Future Directions for the Ext2/3 Filesystem. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, Monterey, California, June 2002.

[Vahdat 2010] Amin Vahdat. VCs loathe to fund storage startups. Personal Communication, October 2010.

[Wikipedia 2010] Wikipedia. Filesystem in Userspace. http://en.wikipedia.org/wiki/Filesystem_in_Userspace, 2010.

[Zadok 2000] E. Zadok and J. Nieh. FiST: A language for stackable file systems. In *Proc. of the Annual USENIX Technical Conference*, pages 55–70, San Diego, CA, June 2000. USENIX Association.

[Zhou 2006] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, Washington, November 2006.