Why panic()? Improving Reliability with Restartable File Systems

Swaminathan Sundararaman, Sriram Subramanian, Abhishek Rajimwale,
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Michael M. Swift
Computer Sciences Department, University of Wisconsin, Madison

ABSTRACT
The file system is one of the most critical components of the operating system. Almost all applications running in the operating system require file systems to be available for their proper operation. Though file-system availability is critical in many cases, very little work has been done on tolerating file system crashes. In this paper, we propose Membrane, a set of changes to the operating system to support restartable file systems. Membrane allows an operating system to tolerate a broad class of file system failures and does so while remaining transparent to running applications; upon failure, the file system restarts, its state is restored, and pending application requests are serviced as if no failure had occurred. Our initial evaluation of Membrane with ext2 shows that Membrane induces little performance overhead and can tolerate a wide range of file system crashes. More critically, Membrane does so with few changes to ext2, thus improving robustness to crashes without mandating intrusive changes to existing file-system code.

1. INTRODUCTION
Operating systems crash. Whether due to software bugs or hardware bit-flips, the reality is clear: large code bases are brittle and the smallest problem in software implementation or hardware environment can lead the entire monolithic operating system to fail.

Recent research has made great headway in operating-system crash tolerance, particularly in surviving device driver failures [17, 18, 23, 25]. Many of these approaches achieve some level of fault resilience by building a *hard wall* around OS subsystems using address-space based isolation and microrebooting said drivers upon fault detection [17, 18]. Other approaches are similar, using variants of microkernel-based architectures [2, 23] or virtual machines [5, 9] to isolate drivers from the kernel.

Device drivers are not the only OS subsystem, nor are they necessarily where the most important bugs reside. Many recent studies have shown that *file systems* contain a large number of bugs [4, 6, 11, 24]. Perhaps this is not surprising, as file systems are one of the largest and most complex code bases in the kernel. Further, file systems are still under active development, and new ones are introduced quite frequently.

Due to the presence of flaws in file system implementation, it is critical to consider how to handle their crashes. Unfortunately, two problems prevent us from directly applying previous work from the device-driver literature to improve file-system fault tolerance. First, most approaches mentioned above are *heavyweight* due to the high costs of data movement and page-table manipulation across address-space boundaries. Second, file systems, unlike device drivers, are extremely *stateful*, as they manage vast amounts of both in-memory and persistent data; making matters worse is the fact that file systems spread such state across many parts of the kernel, including the page cache, dynamically-allocated memory, and locks. Thus, when a file system crashes (typically by calling `panic()`), a great deal of care is required to recover from the crash while keeping the rest of the OS intact.

In this paper, we propose *Membrane*, an operating system framework to support lightweight, stateful recovery from file system crashes. During normal operation, Membrane logs file system operations and periodically performs lightweight checkpoints of file system state. If a file system crashes, Membrane parks pending requests, cleans up existing state, restarts the file system from the most recent checkpoint, and replays the in-memory operation log to restore the state of the running file system. Once finished with recovery, Membrane begins to service on-going application requests again; applications are kept unaware of the crash and restart except for the small performance blip during recovery.

Membrane does not place an address-space boundary between the file system and the rest of the kernel. Hence, it is possible that some types of crashes (e.g., wild writes) will corrupt kernel data structures and thus prohibit Membrane from properly recovering from a file system crash, an inherent weakness (and strength!) of Membrane's architecture. However, we believe that this approach will have the propaedeutic side-effect of encouraging file system developers to add a higher degree of integrity checking (e.g., via assertions, either by hand or through automated techniques such as SafeDrive [25]) into their code in order to fail quickly rather than run the risk of further corrupting the system. If such faults are transient (as many important classes of bugs are), crashing and quickly restarting is a sensible course of action.

Membrane, being a lightweight and generic operating system framework, requires little or no change to existing Linux file systems. We believe that Membrane can be used to restart most of the existing file systems. We have prototyped Membrane with the ext2 file system. From our initial evaluation, we find that Membrane enables ext2 to recover from a wide range of fault scenarios. We also find that Membrane is relatively lightweight, adding a small amount of overhead across a set of file system benchmarks. Membrane achieves these goals with little or no intrusiveness: only five lines were added to transform ext2 into its restartable counterpart. Finally, Membrane improves robustness with complete application transparency; even though the underlying file system has crashed, applications continue to run.

The rest of this paper is organized as follows. Sections 2, 3, and 4 describe the motivation, the challenges in building restartable file systems, and the design of Membrane respectively. Section 5 and Section 6 discuss the consequence of having Membrane in the operating system and evaluates Membrane's robustness and performance. Section 7 places Membrane in the context of other relevant work; finally Section 8 concludes the paper.

2. MOTIVATION

We first motivate the need for restartability in file systems.The file system is one of the critical components of the operating system that is responsible for storing and retrieving user, application, and OS data from the disk. File systems are also one of the largest and complex codes in the operating system; for example, modern file systems such as Sun's Zettabyte File System (ZFS) [1], SGI's XFS [16], and older code bases such as Sun's UFS [10] contain nearly 100,000 lines of code [14]. Further, file systems are still under active development, and new ones are introduced quite frequently. For example, Linux has many established file systems, including ext2 [19], ext3 [20], reiserfs [13], and still there is great interest in next-generation file systems such as Linux ext4 [21] and Btrfs [22]. Thus, file systems are large, complex, and under development: the perfect storm for numerous bugs to arise.

Existing file systems only provide data consistency in the presence of system crashes (power failures or operating system crashes) through techniques such as journaling [7, 16] and snapshotting [1]. However, little has been done to tolerate transient failures in file systems (such as those caused by bugs in file system code and memory corruption), and to prevent them from causing total system failures. Recent research on restarting individual OS component such as device drivers [17, 18, 25] cannot be directly applied to file systems because, unlike device drivers, file systems are very stateful and spread their state across many operating system components (e.g., in-memory objects, data in page cache, locks, and meta-data cached in memory) and are fine tuned for performance.

Consequently, when file systems crash (due to a bug in the file system code), the entire operating system effectively becomes useless, as a majority of the applications running in the operating system depend on the file system for their proper operation. The currently available solution for handling file system crashes is to restart the operating system and run recovery (or a repair utility such as *fsck* for file systems that do not have any in-built crash consistency mechanism). Moreover, applications that were using the file systems are killed, making their services unavailable during this period. This motivates the need for a restartable framework in the operating system, to tolerate failures in file systems.

3. CHALLENGES

Building restartable file systems is hard. We now discuss the challenges in building such a system.

Fault Tolerant. A gamut of faults can occur in file systems. Failures can be caused due to faulty hardware and/or buggy software. These failures can be permanent or transient, and can corrupt data arbitrarily or be fail-stop. The *ideal* restartable file system should recover from all possible faults.

Transparency. File-system failures must be transparent to applications. Applications should not require any modifications to work with restartable file systems. Moreover, multiple applications (or threads) could be modifying the file system state at the same time and one of them could trigger a crash (through a bug).

Performance. Since file systems are fine-tuned for performance, the infrastructure to restart file systems should have little or no overhead during regular operations. Also, the recovery time should be as small as possible to hide file-system crashes from applications.

Consistency. Care must be taken to ensure that the on-disk state of the file system is consistent after recovery. Hence, infrastructure for creating light-weight recovery points should be built as most file systems do not have any inbuilt crash consistency mechanism (e.g., only 8 out of 30 on-disk file system have such feature in Linux 2.6.27). Also, the kernel data structures (locks, memory, list, etc.) should be kept consistent after recovery.

Generic. A large number of commodity file systems exist and each has its own strengths and weaknesses. The infrastructure should not be designed for a specific file system. Ideally, the infrastructure should enable any file system to be transformed into a restartable file system with little or no changes.

4. DESIGN

In designing Membrane, we explicitly make the choice to favor performance, transparency, consistency, and generality over the ability to handle a wider range of faults. Membrane does not attempt to handle all types of faults. Like most work in subsystem fault detection and recovery, Membrane best handles failures that are *transient* and *fail-stop* [12, 18, 25]. We now present the three major pieces in the Membrane design namely fault detection, fault anticipation, and recovery.

4.1 Fault Detection

The goal of fault detection within Membrane is to be lightweight while catching as many faults as possible. Membrane uses both hardware and software techniques to catch faults. The hardware support is simple: null pointers and other exceptions are caught by the hardware and routed to the Membrane recovery subsystem.

The software techniques leverage the checks that already exist in file system code. For example, file systems contain assertions, calls to `panic()`, and similar functions. We take advantage of such internal integrity checking and modify these functions and macros to call into our recovery engine instead of crashing the system. Membrane provides further protection by adding extensive parameter checking on calls from the file system into the kernel.

4.2 Fault Anticipation

Anticipation is the overhead incurred even when the system is behaving well; it should be minimized to the greatest extent possible while retaining the ability to recover.

In Membrane, there are two components of fault anticipation. First, the *checkpointing* subsystem partitions file system operations into different *transactions* and ensures that the checkpointed image on disk represents a consistent state. Second, updates to data structures and other state are tracked with a set of *in-memory logs* and *per thread stacks*. The recovery subsystem utilizes these pieces in tandem to recover the file system after failure.

Checkpointing and state-tracking are central to the management of file system state. File system operations use many core kernel services (e.g., memory allocation), are heavily intertwined with kernel subsystems (e.g., the page cache), and have application-visible state (e.g., file descriptors). Careful state-tracking and checkpointing is thus required to enable clean recovery after a fault.

4.2.1 Checkpointing

Checkpointing is critical because a checkpoint defines a point in time to which Membrane can roll back and thus initiate recovery. A checkpoint also represents a consistent boundary where no file system operation is in flight.

Checkpointing must integrate with the file system's own consistency management scheme. Modern file systems take a number of different approaches such as journaling [7, 16]) and snapshotting [1]); some file systems such as ext2 do not impose any ordering on updates. In all cases, Membrane must operate correctly and efficiently.

4.2.2 Tracking State with Logs and Stacks

Membrane must track changes to various aspects of file system state that transpired after the last checkpoint. This is accomplished with five different types of logs or stacks that handle file system operations, application-visible sessions, mallocs, locks, and execution state.

First, an in-memory *operation log* records file system operations that have taken place during the epoch or are currently in progress together with relevant information. Second, Membrane maintains a small *session log* that tracks open files at the beginning of an epoch along with their file position. Third, an in-memory *malloc table* tracks heap-allocated memory so that upon failure, it can determine which blocks should be freed. Fourth, lock acquired and released are tracked by the *lock stack*. When a lock is acquired/released by a thread executing a file system operation, information about the lock is pushed/popped to a per-thread stack. Finally, an *unwind stack* pushes register state during kernel-to-file system calls onto the per-thread stack to track the execution of code in the file system and kernel. Thus, by using these data structures, Membrane tracks the file system state after the last checkpoint.

4.3 Fault Recovery

Once a fault is detected, control is transferred to the recovery subsystem, which executes the recovery protocol. The six steps in the recovery protocol are described below.

First, Membrane halts the execution of threads within the file system as well as late-arriving threads to prevent further damage. Second, the crashed thread and all other in-flight threads are unwound and brought back to a point such that they appear to be just about to enter the file system call they are making. Third, Membrane moves the system to a clean starting point at the beginning of an epoch, and commits any dirty pages from the previous epoch to disk. Fourth, Membrane frees any operating system state corresponding to the file system and releases any in-memory file-system objects by simulating an unmount operation; Membrane then mimics the mount operation to recreate state for the restarted file system. Fifth, Membrane reopens sessions of active processes and replays operations after the last checkpoint to restore the file system to the state it was before crashing. Finally, Membrane wakes all parked threads which will behave as if a crash never occurred.

We have implemented a prototype of Membrane in the Linux 2.6.15 kernel. This prototype includes the features described above including fault detection, checkpointing, state tracking, and recovery.

5. DISCUSSION

The major negative of the Membrane approach is that, without address-space-based protection, file system faults may corrupt other components of the system. If the file system corrupts other kernel data or code or data that resides on disk, Membrane will not be able to recover the system. Thus, an important factor in Membrane's success will be minimizing the latency between when a fault occurs and when it is detected.

An assumption we make is that kernel code is trusted to work properly, even when the file system code fails and returns an error. We found that this is true in most of the cases across the kernel proper code. But in twenty or so places, we found that kernel proper did not check the return value from the file system and additional code was added to clean up the kernel state and propagate the error back to the callee.

A potential limitation of our implementation is that, in some scenarios, a file system restart can be visible to applications. For instance, when a file is created, the file system assigns it a specific inode number which an application may query (e.g., `rsync` and similar tools may use this low-level number for backup and archival purposes). If a crash occurs before the end of the epoch, Membrane will replay the file create; during replay, the file system may assign a different inode number to the file (based on in-memory state). In this case, the application would possess what it thinks is the inode number of the file, but what may be in fact either unallocated or allocated to

ext2_Function	Fault	ext2 Vanilla: How Detected?	Application?	FS:Consistent?	FS:Usable?	ext2+ Membrane: How Detected?	Application?	FS:Consistent?	FS:Usable?
create	mark_inode_dirty(∅)	o	×	×	×	d	√	√	√
free_inode	mark_buffer_dirty(bad)	o	×	×	×	d	√	√	√
readdir	page_address(bad)	G	×	×	×	d	√	√	√
add_nondir	d_instantiate(bad)	o	×	×	×	d	√	√	√
symlink	null-pointer exception	o	×	×	×	d	√	√	√
readpages	mpage_readpages(bad)	o	×	√	√	d	√	√	√

Table 1: Fault Study. ***The table shows the results of some fault injections on the behavior of Linux ext2. Each row represents a fault injection experiment, and the columns show (in left-to-right order): routine the fault was injected into, the exact nature of the fault, fault detection and its affect on the application, and file system. Symbols used to condense the presentation are as follows: "o": kernel oops was triggered; "G": general protection fault occurred; "d": fault was successfully detected. An "×" or "√" implies a no or yes to the question.***

a different file. Thus, to guarantee that the user-visible inode number is valid, an application must sync the file system state after the create operation.

On the brighter side, we believe Membrane will encourage two positive fault-detection behaviors among file-system developers. First, we believe that quick-fix *bug patching* will become more prevalent. Imagine a scenario where an important customer has a workload that is causing the file system to occasionally corrupt data, thus reducing the reliability of the system. After some diagnosis, the development team discovers the location of the bug in the code, but unfortunately there is no easy fix. With the Membrane infrastructure, the developers may be able to transform the corruption into a fail-stop crash. By installing a quick patch that crashes the code instead of allowing further corruption to continue, the deployment can operate correctly while a longer-term fix is developed. Even more interestingly, if such problems can be detected but would require extensive code restructuring to fix, then a patch may be the best possible permanent solution. As Tom West said: not all problems worth solving are worth solving well [8].

Second, with Membrane, file-system developers will see significant benefits to putting integrity checks into their code. Some of these lightweight checks could be automated (as was nicely done by SafeDrive [25]), but we believe that developers will be able to place much richer checks as they have a deep knowledge about expectations at various locations. For example, developers understand the exact meaning of a directory entry and can signal a problem if one has gone awry; automating such a check is a great deal more complicated [3]. The motivation to check for violations is low in current file systems since their is little recourse when a problem is detected. The ability to recover from the problem in Membrane gives greater motivation.

6. EVALUATION

We evaluated Membrane in terms of transparency and performance. All experiments were performed on Linux 2.6.15 on a machine with a 2.2 GHz Opteron processor, 1GB of memory and 2 80 GB Hitachi 7200 rpm disks.

Benchmark	ext2	ext2+Membrane
OpenSSH	57.7s	62.3s
PostMark	21.0s	25.0s
Sort (100 Mb)	143.0s	145.0s

Table 2: Performance. ***The table presents the performance of a number of benchmarks (in seconds) running on both standard ext2 as well as ext2 within the Membrane framework. PostMark parameters are: 100 files, 1000 transactions, file sizes of 160k to 4MB, and 50/50 read/append and create/delete biases.***

Transparency. To analyze the ability of Membrane to hide file system crashes from applications, we systematically inject faults in the file system code where faults may cause trouble. Table 1 presents the results of our study, the caption explains how to interpret the raw data one sees in the table. Due to lack of space, we only show the results of a few representative fault injections. We injected a total of 15 faults, the results of which are similar to the ones shown here.

We begin our analysis with vanilla ext2, the results of which are shown in the leftmost result column. We can see that in all the cases, an unhandled exception such as "oops" is generated within the file system. This results in the application being killed, frequently leaving the file system inconsistent and/or unusable. A complete reboot is required to fix the system.

In our second set of fault injections, we analyze ext2 with Membrane, which also includes stateful restart of file system. We find that Membrane is able to detect all the errors that we injected. Upon detection, Membrane transparently restarts the file system while the applications keep running normally. Further, Membrane is able to keep the file system in a consistent and usable state.

Performance. To measure performance, we ran a series of workloads on both standard ext2 as well as ext2 with Membrane. Table 2 shows the results of our experiments. From the table, we can see that the performance overheads of our untuned prototype are quite minimal. We expect that with further effort, the small overheads observed here could be reduced noticeably.

In summary, our initial evaluation suggests that Membrane is fault resilient, lightweight and transparent. Further, in the current prototype for ext2, we made only a minor modification to the file system (5 lines of code for flushing super blocks at checkpoints) suggesting that Membrane infrastructure could be generic and not tailored for a specific file system.

7. RELATED WORK

We now discuss three previous systems that have the similar goal of increasing operating system fault resilience. First, the restarting of OS subsystems was started by Swift et al.'s work on Nooks, followed by shadow drivers [17, 18]. The authors use memory-management hardware to build an isolation boundary around device drivers; not surprisingly, such techniques incur high overheads [17]. The subsequent shadow driver work shows how recovery can be transparently achieved by restarting failed drivers and diverting clients by passing them error codes and related tricks. However, such recovery is relatively straightforward: only a simple reinitialization must occur before reintegrating the restarted driver into the OS.

Second, SafeDrive takes a completely different approach to fault resilience [25]. Instead of address-space based protection, SafeDrive automatically adds assertions in the device driver code. Because the assertions are added in a C-to-C translation pass and the final driver code is produced through the compilation of this code, SafeDrive is lightweight and induces relatively low overheads. However, the SafeDrive recovery machinery does not handle stateful subsystems and the recovery is not transparent to applications. Thus, while currently well-suited for a certain class of device drivers, SafeDrive recovery cannot be applied directly to file systems.

Finally, CuriOS, a microkernel-based operating system, also aims to be resilient to subsystem failure [2]. It achieves this end through address-space boundaries between servers along with storing session state in an additional protection domain. Frequent kernel crossings (an expensive operation) are common for file systems in data-intensive environments and would dominate performance. CuriOS also represents one of the few systems that attempt to provide failure resilience for more stateful services such as file systems; other heavyweight checkpoint/restart systems also share this property [15]. In the paper there is a brief description of an "ext2 implementation"; unfortunately, it is difficult to understand how sophisticated this file service is or how much work is required to recover from the failure of such a service.

In summary, we see that few lightweight techniques have been developed. Of those, we know of none that work for stateful subsystems such as file systems.

8. CONCLUSIONS

File systems fail. With Membrane, failure is transformed from a show-stopping event into a small performance issue. The benefits are many. Membrane enables file-system developers to ship file systems sooner, as small bugs will not cause massive user headaches. Membrane similarly enables customers to install new file systems with knowledge that it won't bring down their entire operation. Bugs will still arise, but those that are rare and hard to reproduce will remain where they belong, automatically "fixed" by a system that can tolerate them.

Future work. We wish to employ the Membrane framework for different file systems including other simple ones like VFAT and more complex journaling file systems like ext3, JFS, and ReiserFS. An interesting aspect of journaling file systems is the fact that they are designed to group file operations into transactions and Membrane can leverage it to checkpoint file system state.

We also wish to carefully evaluate the robustness and performance of Membrane. This includes comprehensive fault injection tests and detailed characterization of the performance overheads and restart delay as perceived by applications.

9. ACKNOWLEDGMENTS

We thank the anonymous reviewers and Richard Golding (our shepherd) for their feedback and comments. 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, Inc 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.

10. REFERENCES

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

[2] Francis M. David, Ellick M. Chan, Jeffrey C. Carlyle, and Roy H. Campbell. CuriOS: Improving Reliability through Operating System Structure. In *OSDI '08*, San Diego, CA, December 2008.

[3] Brian Demsky and Martin Rinard. Automatic Detection and Repair of Errors in Data Structures. In *OOPSLA '03*, Anaheim, California, October 2003.

[4] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *SOSP '01*, pages 57–72, Banff, Canada, October 2001.

[5] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe Hardware Access with the Xen Virtual Machine Monitor. In *Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure*, 2004.

[6] Haryadi S. Gunawi, Cindy Rubio-Gonzalez, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Ben Liblit. EIO: Error Handling is Occasionally Correct. In *FAST '08*, pages 207–222, San Jose, California, February 2008.

[7] Robert Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *SOSP '87*, Austin, TX, November 1987.

[8] Tracy Kidder. *Soul of a New Machine*. Little, Brown, and Company, 1981.

[9] J. LeVasseur, V. Uhlig, J. Stoess, and S. Gotz. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Proceedings of the 6th USENIX OSDI*, 2004.

[10] L. W. McVoy and S. R. Kleiman. Extent-like Performance from a UNIX File System. In *USENIX Winter '91*, pages 33–43, Dallas, Tx, January 1991.

[11] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON File Systems. In *SOSP '05*, pages 206–220, Brighton, UK, October 2005.

[12] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating Bugs As Allergies. In *SOSP '05*, Brighton, UK, October 2005.

[13] Hans Reiser. ReiserFS. www.namesys.com, 2004.

[14] Eric Schrock. UFS/SVM vs. ZFS: Code Complexity. http://blogs.sun.com/eschrock/, November 2005.

[15] J. S. Shapiro and N. Hardy. EROS: A Principle-Driven Operating System from the Ground Up. *IEEE Software*, 19(1), January/February 2002.

[16] Adan Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS File System. In *USENIX 1996*, San Diego, CA, January 1996.

[17] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. In *SOSP '03*, Bolton Landing, NY, October 2003.

[18] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. In *OSDI '04*, pages 1–16, San Francisco, CA, December 2004.

[19] Theodore Ts'o. http://e2fsprogs.sourceforge.net/ext2.html, June 2001.

[20] Theodore Ts'o and Stephen Tweedie. Future Directions for the Ext2/3 Filesystem. In *FREENIX '02*, Monterey, CA, June 2002.

[21] Wikipedia. ext4. en.wikipedia.org/wiki/Ext4, 2008.

[22] Wikipedia. Btrfs. en.wikipedia.org/wiki/Btrfs, 2009.

[23] Dan Williams, Patrick Reynolds, Kevin Walsh, Emin Gün Sirer, and Fred B. Schneider. Device Driver Safety Through a Reference Validation Mechanism. In *Proceedings of the 8th USENIX OSDI*, 2008.

[24] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using Model Checking to Find Serious File System Errors. In *OSDI '04*, San Francisco, CA, December 2004.

[25] 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 *OSDI '06*, Seattle, WA, November 2006.