Membrane: Operating System Support for 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

We introduce 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. Membrane provides transparent recovery through a lightweight logging and checkpoint infrastructure, and includes novel techniques to improve performance and correctness of its fault-anticipation and recovery machinery. We tested Membrane with ext2, ext3, and VFAT. Through experimentation, we show that Membrane induces little performance overhead and can tolerate a wide range of file system crashes. More critically, Membrane does so with little or no change to existing file systems thus improving robustness to crashes without mandating intrusive changes to existing file-system code.

1 Introduction

Operating systems crash. Whether due to software bugs [8] or hardware bit-flips [22], 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 [9, 10, 13, 14, 20, 31, 32, 37, 40]. Many of these approaches achieve some level of fault tolerance by building a *hard wall* around OS subsystems using address-space based isolation and microrebooting [2, 3] said drivers upon fault detection. For example, Nooks (and follow-on work with Shadow Drivers) encapsulate device drivers in their own protection domain, thus making it challenging for errant driver code to overwrite data in other parts of the kernel [31, 32]. Other approaches are similar, using variants of microkernel-based architectures [7, 13, 37] or virtual machines [10, 20] 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 [5, 8, 11, 25, 38, 39]. 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. For example, Linux has many established file systems, including ext2 [34], ext3 [35], reiserfs [27], and still there is great interest in next-generation file systems such as Linux ext4 and btrfs. Thus, file systems are large, complex, and under development, the perfect storm for numerous bugs to arise.

Because of the likely presence of flaws in their implementation, it is critical to consider how to recover from file system crashes as well. Unfortunately, we cannot directly apply previous work from the device-driver literature to improving file-system fault recovery. 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 so forth. On-disk state of the file system also needs to be consistent upon restart to avoid any damage to the stored data. Thus, when a file system crashes, a great deal more care is required to recover while keeping the rest of the OS intact.

In this paper, we introduce *Membrane*, an operating system framework to support lightweight, stateful recovery from file system crashes. During normal operation, Membrane logs file system operations, tracks file system objects, and periodically performs lightweight checkpoints of file system state. If a file system crash occurs, 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 file system. Once finished with recovery, Membrane begins to service application requests again; applications are unaware of the crash and restart except for a small performance blip during recovery.

Membrane achieves its performance and robustness through the application of a number of novel mechanisms. For example, a *generic checkpointing mechanism* enables low-cost snapshots of file system-state that serve as recovery points after a crash with minimal support from existing file systems. A *page stealing* technique greatly reduces logging overheads of write operations, which would otherwise increase time and space overheads. Finally, an intricate *skip/trust unwind protocol* is applied to carefully unwind in-kernel threads through both the crashed file system and kernel proper. This process restores kernel state while preventing further file-system-induced damage from taking place.

Interestingly, file systems already contain many explicit error checks throughout their code. When triggered, these checks crash the operating system (e.g., by calling panic) after which the file system either becomes unusable or unmodifiable. Membrane leverages these explicit error checks and invokes recovery instead of crashing the file system. We believe that this approach will have the propaedeutic side-effect of encouraging file system developers to add a higher degree of integrity checking 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 [21]), crashing and quickly restarting is a sensible manner in which to respond to them.

As performance is critical for file systems, Membrane only provides a lightweight fault detection mechanism and 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 [4]) will corrupt kernel data structures and thus prohibit complete recovery, an inherent weakness of Membrane's architecture. Users willing to trade performance for reliability could use Membrane on top of stronger protection mechanism such as Nooks [31].

We evaluated Membrane with the ext2, VFAT, and ext3 file systems. Through experimentation, we find that Membrane enables existing file systems to crash and recover from a wide range of fault scenarios (around 50 fault injection experiments). We also find that Membrane has less than 2% overhead across a set of file system benchmarks. Membrane achieves these goals with little or no intrusiveness to existing file systems: only 5 lines of code were added to make ext2, VFAT, and ext3 restartable. 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. Section 2 places Membrane in the context of other relevant work. Sections 3 and 4 present the design and implementation, respectively, of Membrane; finally, we evaluate Membrane in Section 5 and conclude in Section 6.

2 Background

Before presenting Membrane, we first discuss previous systems that have a similar goal of increasing operating system fault resilience. We classify previous approaches along two axes: *overhead* and *statefulness*.

We classify fault isolation techniques that incur little overhead as *lightweight*, while more costly mechanisms are classified as *heavyweight*. Heavyweight mechanisms are not likely to be adopted by file systems, which have been tuned for high performance and scalability [15, 30, 1], especially when used in server environments.

We also classify techniques based on how much system state they are designed to recover after failure. Techniques that assume the failed component has little in-memory state is referred to as *stateless*, which is the case with most device driver recovery techniques. Techniques that can handle components with in-memory and even persistent storage are *stateful*; when recovering from file-system failure, stateful techniques are required.

We now examine three particular systems as they are exemplars of three previously explored points in the design space. Membrane, described in greater detail in subsequent sections, represents an exploration into the fourth point in this space, and hence its contribution.

2.1 Nooks and Shadow Drivers

The renaissance in building isolated OS subsystems is found in Swift et al.'s work on Nooks and subsequently shadow drivers [31, 32]. In these works, the authors use memory-management hardware to build an isolation boundary around device drivers; not surprisingly, such techniques incur high overheads [31]. The kernel cost of Nooks (and related approaches) is high, in this one case spending nearly 6$\times$ more time in the kernel.

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.

2.2 SafeDrive

SafeDrive takes a different approach to fault resilience [40]. Instead of address-space based protection, SafeDrive automatically adds assertions into device drivers. When an assert is triggered (e.g., due to a null pointer or an out-of-bounds index variable), SafeDrive enacts a recovery process that restarts the driver and thus survives the would-be failure. 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 (up to 17% reduced performance in a network throughput test and 23% higher CPU utilization for the USB driver [40], Table 6.).

However, the SafeDrive recovery machinery does not handle stateful subsystems; as a result the driver will be in an initial state after recovery. Thus, while currently well-suited for a certain class of device drivers, SafeDrive recovery cannot be applied directly to file systems.

2.3 CuriOS

CuriOS, a recent microkernel-based operating system, also aims to be resilient to subsystem failure [7]. It achieves this end through classic microkernel techniques

	Heavyweight	**Lightweight**
Stateless	Nooks/Shadow[31, 32]* Xen[10], Minix[13, 14] L4[20], Nexus[37]	SafeDrive[40]* Singularity[19]
Stateful	CuriOS[7] EROS[29]	Membrane*

Table 1: **Summary of Approaches.** *The table performs a categorization of previous approaches that handle OS subsystem crashes. Approaches that use address spaces or full-system checkpoint/restart are too heavyweight; other language-based approaches may be lighter weight in nature but do not solve the stateful recovery problem as required by file systems. Finally, the table marks (with an asterisk) those systems that integrate well into existing operating systems, and thus do not require the widespread adoption of a new operating system or virtual machine to be successful in practice.*

(i.e., address-space boundaries between servers) with an additional twist: instead of storing session state inside a service, it places such state in an additional protection domain where it can remain safe from a buggy service. However, the added protection is expensive. Frequent kernel crossings, as would be common for file systems in data-intensive environments, would dominate performance.

As far as we can discern, CuriOS 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 [29]. In the paper there is a brief description of an "ext2 implementation"; unfortunately it is difficult to understand exactly how sophisticated this file service is or how much work is required to recover from failures. It also seems that there is little shared state as is common in modern systems (e.g., pages in a page cache).

2.4 Summary

We now classify these systems along the two axes of overhead and statefulness, as shown in Table 1. From the table, we can see that many systems use methods that are simply too costly for file systems; placing address-space boundaries between the OS and the file system greatly increases the amount of data copying (or page remapping) that must occur and thus is untenable. We can also see that fewer lightweight techniques have been developed. Of those, we know of none that work for stateful subsystems such as file systems. Thus, there is a need for a lightweight, transparent, and stateful approach to fault recovery.

3 Design

Membrane is designed to transparently restart the affected file system upon a crash, while applications and the rest of the OS continue to operate normally. A primary challenge in restarting file systems is to correctly manage the state associated with the file system (e.g., file descriptors, locks in the kernel, and in-memory inodes and directories).

In this section, we first outline the high-level goals for our system. Then, we discuss the nature and types of faults Membrane will be able to detect and recover from. Finally, we present the three major pieces of the Membrane system: fault detection, fault anticipation, and recovery.

3.1 Goals

We believe there are five major goals for a system that supports restartable file systems.

Fault Tolerant: A large range of faults can occur in file systems. Failures can be caused by faulty hardware and buggy software, can be permanent or transient, and can corrupt data arbitrarily or be fail-stop. The *ideal* restartable file system recovers from all possible faults.

Lightweight: Performance is important to most users and most file systems have had their performance tuned over many years. Thus, adding significant overhead is not a viable alternative: a restartable file system will only be used if it has comparable performance to existing file systems.

Transparent: We do not expect application developers to be willing to rewrite or recompile applications for this environment. We assume that it is difficult for most applications to handle unexpected failures in the file system. Therefore, the restartable environment should be completely transparent to applications; applications should not be able to discern that a file-system has crashed.

Generic: A large number of commodity file systems exist and each has its own strengths and weaknesses. Ideally, the infrastructure should enable any file system to be made restartable with little or no changes.

Maintain File-System Consistency: File systems provide different crash consistency guarantees and users typically choose their file system depending on their requirements. Therefore, the restartable environment should not change the existing crash consistency guarantees.

Many of these goals are at odds with one another. For example, higher levels of fault resilience can be achieved with heavier-weight fault-detection mechanisms. Thus in designing Membrane, we explicitly make the choice to favor performance, transparency, and generality over the ability to handle a wider range of faults. We believe that heavyweight machinery to detect and recover from relatively-rare faults is not acceptable. Finally, although Membrane should be as generic a framework as possible, a few file system modifications can be tolerated.

3.2 Fault Model

Membrane's recovery 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* [26, 32, 40].

Deterministic faults, such as memory corruption, are challenging to recover from without altering file-system code. We assume that testing and other standard code-hardening techniques have eliminated most of these bugs. Faults such as a bug that is triggered on a given input sequence could be handled by failing the particular request. Currently, we return an error (-EIO) to the requests triggering such deterministic faults, thus preventing the same fault from being triggered again and again during recovery. Transient faults, on the other hand, are caused by race conditions and other environmental factors [33]. Thus, our aim is to mainly cope with transient faults, which can be cured with recovery and restart.

We feel that many faults and bugs can be caught with lightweight hardware and software checks. Other solutions, such as extremely large address spaces [17], could help reduce the chances of wild writes causing harm by hiding kernel objects ("needles") in a much larger addressable region ("the haystack").

Recovering a stateful file system with lightweight mechanisms is especially challenging when faults are not fail-stop. For example, consider buggy file-system code that attempts to overwrite important kernel data structures. If there is a heavyweight address-space boundary between the file system and kernel proper, then such a stray write can be detected immediately; in effect, the fault becomes fail-stop. If, in contrast, there is no machinery to detect stray writes, the fault can cause further silent damage to the rest of the kernel before causing a detectable fault; in such a case, it may be difficult to recover from the fault.

We strongly believe that once a fault is detected in the file system, no aspect of the file system should be trusted: no more code should be run in the file system and its in-memory data structures should not be used.

The major drawback of our approach is that the boundary we use is soft: some file system bugs can still corrupt kernel state outside the file system and recovery will not succeed. However, this possibility exists even in systems with hardware boundaries: data is still passed across boundaries, and no matter how many integrity checks one makes, it is possible that bad data is passed across the boundary and causes problems on the other side.

3.3 Overview

The main design challenge for Membrane is to recover file-system state in a lightweight, transparent fashion. At a high level, Membrane achieves this goal as follows.

Once a fault has been detected in the file system, Membrane rolls back the state of the file system to a point in the past that it trusts: this trusted point is a consistent file-system image that was checkpointed to disk. This checkpoint serves to divide file-system operations into distinct epochs; no file-system operation spans multiple epochs.

To bring the file system up to date, Membrane replays the file-system operations that occurred after the checkpoint. In order to correctly interpret some opera-

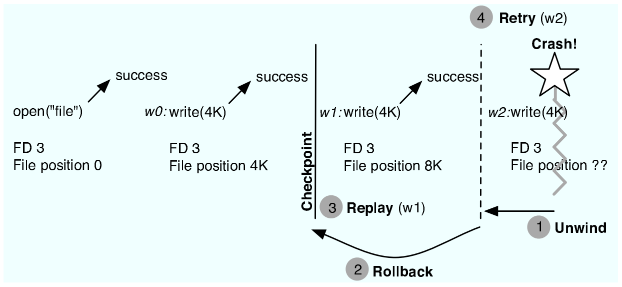

Figure 1: **Membrane Overview.** *The figure shows a file being created and written to on top of a restartable file system. Halfway through, Membrane creates a checkpoint. After the checkpoint, the application continues to write to the file; the first succeeds (and returns success to the application) and the program issues another write, which leads to a file system crash. For Membrane to operate correctly, it must (1) unwind the currently-executing write and park the calling thread, (2) clean up file system objects (not shown), restore state from the previous checkpoint, and (3) replay the activity from the current epoch (i.e., write* w1*). Once file-system state is restored from the checkpoint and session state is restored, Membrane can (4) unpark the unwound calling thread and let it reissue the write, which (hopefully) will succeed this time. The application should thus remain unaware, only perhaps noticing the timing of the third write (*w2*) was a little slow.*

tions, Membrane must also remember small amounts of application-visible state from before the checkpoint, such as file descriptors. Since the purpose of this replay is only to update file-system state, non-updating operations such as reads do not need to be replayed.

Finally, to clean up the parts of the kernel that the buggy file system interacted with in the past, Membrane releases the kernel locks and frees memory the file system allocated. All of these steps are transparent to applications and require no changes to file-system code. Applications and the rest of the OS are unaffected by the fault. Figure 1 gives an example of how Membrane works during normal file-system operation and upon a file system crash.

Thus, there are three major pieces in the Membrane design. First, *fault detection* machinery enables Membrane to detect faults quickly. Second, *fault anticipation* mechanisms record information about current file-system operations and partition operations into distinct epochs. Finally, the *fault recovery* subsystem executes the recovery protocol to clean up and restart the failed file system.

3.4 Fault Detection

The main aim 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, divide-by-zero, and many other exceptions are caught by the hardware and routed to the Membrane recovery subsystem. More expensive hardware machinery, such as address-space-based isolation, is not used.

The software techniques leverage the many checks that already exist in file system code. For example, file systems contain assertions as well as calls to `panic()` and similar functions. We take advantage of such internal integrity checking and transform calls that would crash the system into calls into our recovery engine. An approach such as that developed by SafeDrive [40] could be used to automatically place out-of-bounds pointer and other checks in the file system code.

Membrane provides further software-based protection by adding extensive parameter checking on any call from the file system into the kernel proper. These *lightweight boundary wrappers* protect the calls between the file system and the kernel and help ensure such routines are called with proper arguments, thus preventing file system from corrupting kernel objects through bad arguments. Sophisticated tools (e.g., Ballista[18]) could be used to generate many of these wrappers automatically.

3.5 Fault Anticipation

As with any system that improves reliability, there is a performance and space cost to enabling recovery when a fault occurs. We refer to this component as *fault anticipation*. Anticipation is pure overhead, paid 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 *epochs* (or *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 *parallel stacks*. The recovery subsystem (described below) utilizes these pieces in tandem to restart the file system after failure.

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

3.5.1 Checkpointing

Checkpointing is critical because a checkpoint represents a point in time to which Membrane can safely roll back and initiate recovery. We define a checkpoint as a consistent boundary between epochs where no operation spans multiple epochs. By this definition, file-system state at a checkpoint is consistent as no file system operations are in flight.

We require such checkpoints for the following reason: file-system state is constantly modified by operations such as writes and deletes and file systems lazily write back the modified state to improve performance. As a result, at any point in time, file system state is comprised of (i) dirty pages (in memory), (ii) in-memory copies of its meta-data objects (that have not been copied to its on-disk pages), and (iii) data on the disk. Thus, the file system is in an inconsistent state until all dirty pages and meta-data objects are quiesced to the disk. For correct operation, one needs to ensure that the file system is in a consistent state at the beginning of the mount process (or the recovery process in the case of Membrane).

Modern file systems take a number of different approaches to the consistency management problem: some group updates into transactions (as in journaling file systems [12, 27, 30, 35]); others define clear consistency intervals and create snapshots (as in shadow-paging file systems [1, 15, 28]). All such mechanisms periodically create checkpoints of the file system in anticipation of a power failure or OS crash. Older file systems do not impose any ordering on updates at all (as in Linux ext2 [34] and many simpler file systems). In all cases, Membrane must operate correctly and efficiently.

The main challenge with checkpointing is to accomplish it in a lightweight and non-intrusive manner. For modern file systems, Membrane can leverage the in-built journaling (or snapshotting) mechanism to periodically checkpoint file system state; as these mechanisms atomically write back data modified within a checkpoint to the disk. To track file-system level checkpoints, Membrane only requires that these file systems explicitly notify the beginning and end of the file-system transaction (or snapshot) to it so that it can throw away the log records before the checkpoint. Upon a file system crash, Membrane uses the file system's recovery mechanism to go back to the last known checkpoint and initiate the recovery process. Note that the recovery process uses on-disk data and does not depend on the in-memory state of the file system.

For file systems that do not support any consistent-management scheme (e.g., ext2), Membrane provides a generic checkpointing mechanism at the VFS layer. Membrane's checkpointing mechanism groups several file-system operations into a single transaction and commits it atomically to the disk. A transaction is created by temporarily preventing new operations from entering into the file system for a small duration in which dirty meta-data objects are copied back to their on-disk pages and all dirty pages are marked copy-on-write. Through copy-on-write support for file-system pages, Membrane improves performance by allowing file system operations to run concurrently with the checkpoint of the *previous* epoch. Membrane associates each page with a checkpoint (or epoch) number to prevent pages dirtied in the current epoch from reaching the disk. It is important to note that the checkpointing mechanism in Membrane is implemented at the VFS layer; as a result, it can be leveraged by all file system with little or no modifications.

3.5.2 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 handling: file system operations, application-visible sessions, mallocs, locks, and execution state.

First, an in-memory *operation log (op-log)* records all state-modifying file system operations (such as open) that have taken place during the epoch or are currently in progress. The op-log records enough information about requests to enable full recovery from a given checkpoint.

Membrane also requires a small *session log (s-log)*. The s-log tracks which files are open at the beginning of an epoch and the current position of the file pointer. The op-log is not sufficient for this task, as a file may have been opened in a previous epoch; thus, by reading the op-log alone, one can only observe reads and writes to various file descriptors without the knowledge of which files such operations refer to.

Third, an in-memory *malloc table (m-table)* tracks heap-allocated memory. Upon failure, the m-table can be consulted to determine which blocks should be freed. If failure is infrequent, an implementation could ignore memory left allocated by a failed file system; although memory would be leaked, it may leak slowly enough not to impact overall system reliability.

Fourth, lock acquires and releases are tracked by the *lock stack (l-stack)*. When a lock is acquired by a thread executing a file system operation, information about said lock is pushed onto a per-thread l-stack; when the lock is released, the information is popped off. Unlike memory allocation, the exact order of lock acquires and releases is critical; by maintaining the lock acquisitions in LIFO order, recovery can release them in the proper order as required. Also note that only locks that are global kernel locks (and hence survive file system crashes) need to be tracked in such a manner; private locks internal to a file system will be cleaned up during recovery and therefore require no such tracking.

Finally, an *unwind stack (u-stack)* is used to track the execution of code in the file system and kernel. By pushing register state onto the per-thread u-stack when the file system is first called on kernel-to-file-system calls, Membrane records sufficient information to unwind threads after a failure has been detected in order to enable restart.

Note that the m-table, l-stack, and u-stack are *compensatory* [36]; they are used to compensate for actions that have already taken place and must be undone before proceeding with restart. On the other hand, both the op-log and s-log are *restorative* in nature; they are used by recovery to restore the in-memory state of the file system before continuing execution after restart.

3.6 Fault Recovery

The *fault recovery* subsystem is likely the largest subsystem within Membrane. Once a fault is detected, control is transferred to the recovery subsystem, which executes the recovery protocol. This protocol has the following phases:

Halt execution and park threads: Membrane first halts the execution of threads within the file system. Such “in-flight” threads are prevented from further execution within the file system in order to both prevent further damage as well as to enable recovery. Late-arriving threads (i.e., those that try to enter the file system after the crash takes place) are parked as well.

Unwind in-flight threads: Crashed and any other in-flight thread are unwound and brought back to the point where they are about to enter the file system; Membrane uses the u-stack to restore register values before each call into the file system code. During the unwind, any held global locks recorded on l-stack are released.

Commit dirty pages from previous epoch to stable storage: Membrane moves the system to a clean starting point at the beginning of an epoch; all dirty pages from the previous epoch are forcefully committed to disk. This action leaves the on-disk file system in a consistent state. Note that this step is not needed for file systems that have their own crash consistency mechanism.

“Unmount” the file system: Membrane consults the m-table and frees all in-memory objects allocated by the the file system. The items in the file system buffer cache (e.g., inodes and directory entries) are also freed. Conceptually, the pages from this file system in the page cache are also released mimicking an unmount operation.

“Remount” the file system: In this phase, Membrane reads the super block of the file system from stable storage and performs all other necessary work to reattach the FS to the running system.

Roll forward: Membrane uses the s-log to restore the sessions of active processes to the state they were at the last checkpoint. It then processes the op-log, replays previous operations as needed and restores the active state of the file system before the crash. Note that Membrane uses the regular VFS interface to restore sessions and to replay logs. Hence, Membrane does not require any explicit support from file systems.

Restart execution: Finally, Membrane wakes all parked threads. Those that were in-flight at the time of the crash begin execution as if they had not entered the file system; those that arrived after the crash are allowed to enter the file system for the first time, both remaining oblivious of the crash.

4 Implementation

We now present the implementation of Membrane. We first describe the operating system (Linux) environment, and then present each of the main components of Membrane. Much of the functionality of Membrane is encapsulated within two components: the *checkpoint manager (CPM)* and the *recovery manager (RM)*. Each of these subsystems is implemented as a background thread and is needed during anticipation (CPM) and recovery (RM). Beyond these threads, Membrane also makes heavy use of *interposition* to track the state of various in-memory objects and to provide the rest of its functionality. We ran Membrane with ext2, VFAT, and ext3 file systems.

In implementing the functionality described above, Membrane employs three key techniques to reduce overheads and make lightweight restart of a stateful file systems feasible. The techniques are (i) *page stealing*: for low-cost operation logging, (ii) *COW-based checkpointing*: for fast in-memory partitioning of pages across epochs using copy-on-write techniques for file systems that do not support transactions, and (iii) *control-flow capture* and *skip/trust unwind protocol*: to halt in-flight threads and properly unwind in-flight execution.

4.1 Linux Background

Before delving into the details of Membrane's implementation, we first provide some background on the operating system in which Membrane was built. Membrane is currently implemented inside Linux 2.6.15.

Linux provides support for multiple file systems via the VFS interface [16], much like many other operating systems. Thus, the VFS layer presents an ideal point of interposition for a file system framework such as Membrane.

Like many systems [6], Linux file systems cache user data in a unified page cache. The page cache is thus tightly integrated with file systems and there are frequent crossings between the generic page cache and file system code.

Writes to disk are handled in the background (except when forced to disk by applications). A background I/O daemon, known as `pdflush`, wakes up, finds old and dirty pages, and flushes them to disk.

4.2 Fault Detection

There are numerous fault detectors within Membrane, each of which, when triggered, immediately begins the recovery protocol. We describe the detectors Membrane currently uses; because they are lightweight, we imagine more will be added over time, particularly as file-system developers learn to trust the restart infrastructure.

4.2.1 Hardware-based Detectors

The hardware provides the first line of fault detection. In our implementation inside Linux on x86 (64-bit) architecture, we track the following runtime exceptions: null-pointer exception, invalid operation, general protection fault, alignment fault, divide error (divide by zero), segment not present, and stack segment fault. These exception conditions are detected by the processor; software fault handlers, when run, inspect system state to determine whether the fault was caused by code executing in the file system module (i.e., by examining the faulting instruction pointer). Note that the kernel already tracks these runtime exceptions which are considered kernel errors and triggers panic as it doesn't know how to handle them. We only check if these exceptions were generated in the context of the restartable file system to initiate recovery, thus preventing kernel panic.

File System	`assert()`	`BUG()`	`panic()`
xfs	2119	18	43
ubifs	369	36	2
ocfs2	261	531	8
gfs2	156	60	0
jbd	120	0	0
jbd2	119	0	0
afs	106	38	0
jfs	91	15	6
ext4	42	182	12
ext3	16	0	11
reiserfs	1	109	93
jffs2	1	86	0
ext2	1	10	6
ntfs	0	288	2
fat	0	10	16

Table 2: **Software-based Fault Detectors.** *The table depicts how many calls each file system makes to `assert()`, `BUG()`, and `panic()` routines. The data was gathered simply by searching for various strings in the source code. A range of file systems and the ext3 journaling devices (jbd and jbd2) are included in the micro-study. The study was performed on the latest stable Linux release (2.6.26.7).*

4.2.2 Software-based Detectors

A large number of explicit error checks are extant within the file system code base; we interpose on these macros and procedures to detect a broader class of semantically-meaningful faults. Specifically, we redefine macros such as `BUG()`, `BUG_ON()`, `panic()`, and `assert()` so that the file system calls our version of said routines.

These routines are commonly used by kernel programmers when some unexpected event occurs and the code cannot properly handle the exception. For example, Linux ext2 code that searches through directories often calls `BUG()` if directory contents are not as expected; see `ext2_add_link()` where a failed scan through the directory leads to such a call. Other file systems, such as reiserfs, routinely call `panic()` when an unanticipated I/O subsystem failure occurs [25]. Table 2 presents a summary of calls present in existing Linux file systems.

In addition to those checks within file systems, we have added a set of checks across the file-system/kernel boundary to help prevent fault propagation into the kernel proper. Overall, we have added roughly 100 checks across various key points in the generic file system and memory management modules as well as in twenty or so header files. As these checks are low-cost and relatively easy to

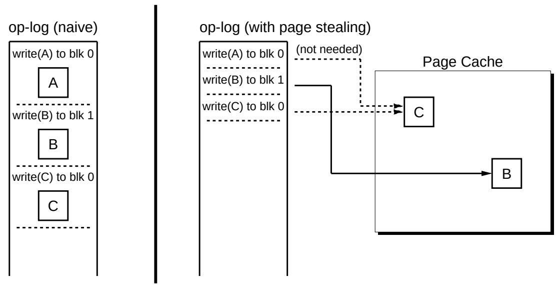

Figure 2: **Page Stealing.** *The figure depicts the op-log both with and without page stealing. Without page stealing (left side of the figure), user data quickly fills the log, thus exacting harsh penalties in both time and space overheads. With page stealing (right), only a reference to the in-memory page cache is recorded with each write; further, only the latest such entry is needed to replay the op-log successfully.*

add, we will continue to "harden" the file-system/kernel interface as our work continues.

4.3 Fault Anticipation

We now describe the fault anticipation support within the current Membrane implementation. We begin by presenting our approach to reducing the cost of operation logging via a technique we refer to as *page stealing*.

4.3.1 Low-Cost Op-Logging via Page Stealing

Membrane interposes at the VFS layer in order to record the necessary information to the op-log about file-system operations during an epoch. Thus, for any restartable file system that is mounted, the VFS layer records an entry for each operation that updates the file system state in some way.

One key challenge of logging is to minimize the amount of data logged in order to keep interpositioning costs low. A naive implementation (including our first attempt) might log all state-updating operations and their parameters; unfortunately, this approach has a high cost due to the overhead of logging write operations. For each write to the file system, Membrane has to not only record that a write took place but also log the *data* to the op-log, an expensive operation both in time and space.

Membrane avoids the need to log this data through a novel *page stealing* mechanism. Because dirty pages are held in memory before checkpointing, Membrane is assured that the most recent copy of the data is already in memory (in the page cache). Thus, when Membrane needs to replay the write, it steals the page from the cache (before it is removed from the cache by recovery) and writes the stolen page to disk. In this way, Membrane avoids the costly logging of user data. Figure 2 shows how page stealing helps in reducing the size of op-log.

When two writes to the same block have taken place, note that only the last write needs to be replayed. Earlier writes simply update the file position correctly. This strategy works because reads are not replayed (indeed, they have already completed); hence, only the current state of the file system, as represented by the last checkpoint and current op-log and s-log, must be reconstructed.

4.3.2 Other Logging and State Tracking

Membrane also interposes at the VFS layer to track all necessary session state in the s-log. There is little information to track here: simply which files are open (with their pathnames) and the current file position of each file.

Membrane also needs to track memory allocations performed by a restartable file system. We added a new allocation flag, `GFP_RESTARTABLE`, in Membrane. We also provide a new header file to include in file-system code to append `GFP_RESTARTABLE` to all memory allocation call. This enables the memory allocation module in the kernel to record the necessary per-file-system information into the m-table and thus prepare for recovery.

Tracking lock acquisitions is also straightforward. As we mentioned earlier, locks that are private to the file system will be ignored during recovery, and hence need not be tracked; only global locks need to be monitored. Thus, when a thread is running in the file system, the instrumented lock function saves the lock information in the thread's private l-stack for the following locks: the global kernel lock, super-block lock, and the inode lock.

Finally, Membrane must also track register state across certain code boundaries to unwind threads properly. To do so, Membrane wraps all calls from the kernel into the file system; these wrappers push and pop register state, return addresses, and return values onto and off of the u-stack.

4.3.3 COW-based Checkpointing

Our goal of checkpointing was to find a solution that is lightweight and works correctly despite the lack of transactional machinery in file systems such as Linux ext2, many UFS implementations, and various FAT file systems; these file systems do not include journaling or shadow paging to naturally partition file system updates into transactions.

One could implement a checkpoint using the following strawman protocol. First, during an epoch, prevent dirty pages from being flushed to disk. Second, at the end of an epoch, checkpoint file-system state by first halting file system activity and then forcing all dirty pages to disk. At this point, the on-disk state would be consistent. If a file-system failure occurred during the next epoch, Membrane could rollback the file system to the beginning of the epoch, replay logged operations, and thus recover the file system.

The obvious problem with the strawman is performance: forcing pages to disk during checkpointing makes checkpointing slow, which slows applications. Further,

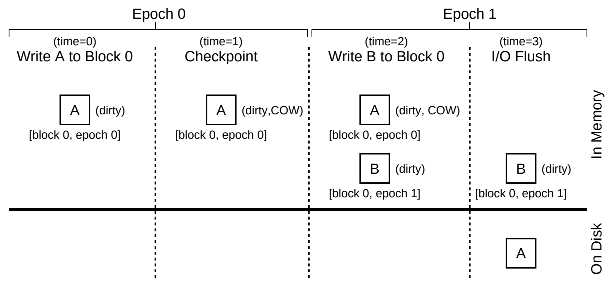

Figure 3: **COW-based Checkpointing.** *The picture shows what happens during COW-based checkpointing. At time=0, an application writes to block 0 of a file and fills it with the contents "A". At time=1, Membrane performs a checkpoint, which simply marks the block copy-on-write. Thus, Epoch 0 is over and a new epoch begins. At time=2, block 0 is over-written with the new contents "B"; the system catches this overwrite with the COW machinery and makes a new in-memory page for it. At time=3, Membrane decides to flush the previous epoch's dirty pages to disk, and thus commits block 0 (with "A" in it) to disk.*

update traffic is bunched together and must happen during the checkpoint, instead of being spread out over time; as is well known, this can reduce I/O performance [23].

Our lightweight checkpointing solution instead takes advantage of the page-table support provided by modern hardware to partition pages into different epochs. Specifically, by using the protection features provided by the page table, the CPM implements a *copy-on-write-based checkpoint* to partition pages into different epochs. This COW-based checkpoint is simply a lightweight way for Membrane to partition updates to disk into different epochs. Figure 3 shows an example on how COW-based checkpointing works.

We now present the details of the checkpoint implementation. First, at the time of a checkpoint, the checkpoint manager (CPM) thread wakes and indicates to the *session manager* (SM) that it intends to checkpoint. The SM parks new VFS operations and waits for in-flight operations to complete; when finished, the SM wakes the CPM so that it can proceed.

The CPM then walks the lists of dirty objects in the file system, starting at the superblock, and finds the dirty pages of the file system. The CPM marks these kernel pages *copy-on-write*; further updates to such a page will induce a copy-on-write fault and thus direct subsequent writes to a new copy of the page. Note that the copy-on-write machinery is present in many systems, to support (among other things) fast address-space copying during process creation. This machinery is either implemented within a particular subsystem (e.g., file systems such as ext3cow [24], WAFL [15] manually create and track their COW pages) or inbuilt in the kernel for application pages. To our knowledge, copy-on-write machinery is not available for kernel pages. Hence, we explicitly added support for copy-on-write machinery for kernel pages in Membrane; thereby avoiding extensive changes to file systems to support COW machinery.

The CPM then allows these pages to be written to disk (by tracking a checkpoint number associated with the page), and the background I/O daemon (`pdflush`) is free to write COW pages to disk at its leisure during the next epoch. Checkpointing thus groups the dirty pages from the previous epoch and allows only said modifications to be written to disk during the next epoch; newly dirtied pages are held in memory until the complete flush of the previous epoch's dirty pages.

There are a number of different policies that can be used to decide when to checkpoint. An ideal policy would likely consider a number of factors, including the time since last checkpoint (to minimize recovery time), the number of dirty blocks (to keep memory pressure low), and current levels of CPU and I/O utilization (to perform checkpointing during relatively-idle times). Our current policy is simpler, and just uses time (5 secs) and a dirty-block threshold (40MB) to decide when to checkpoint. Checkpoints are also initiated when an application forces data to disk.

4.4 Fault Recovery

We now describe the last piece of our implementation which performs fault recovery. Most of the protocol is implemented by the recovery manager (RM), which runs as a separate thread. The most intricate part of recovery is how Membrane gains control of threads after a fault occurs in the file system and the unwind protocol that takes place as a result. We describe this component of recovery first.

4.4.1 Gaining Control with Control-Flow Capture

The first problem encountered by recovery is how to gain control of threads already executing within the file system. The fault that occurred (in a given thread) may have left the file system in a corrupt or unusable state; thus, we would like to stop all other threads executing in the file system as quickly as possible to avoid any further execution within the now-untrusted file system.

Membrane, through the RM, achieves this goal by immediately marking all code pages of the file system as non-executable and thus ensnaring other threads with a technique that we refer as *control-flow capture*. When a thread that is already within the file system next executes an instruction, a trap is generated by the hardware; Membrane handles the trap and then takes appropriate action to unwind the execution of the thread so that recovery can proceed after all these threads have been unwound. File systems in Membrane are inserted as loadable kernel modules, this ensures that the file system code is in a 4KB page and not part of a large kernel page which could potentially be shared among different kernel modules. Hence, it is straightforward to transparently identify code pages of file systems.

4.4.2 Intertwined Execution and The Skip/Trust Unwind Protocol

Unfortunately, unwinding a thread is challenging, as the file system interacts with the kernel in a tightly-coupled fashion. Thus, it is not uncommon for the file system to call into the kernel, which in turn calls into the file system, and so forth. We call such execution paths *intertwined*.

Intertwined code puts Membrane into a difficult position. Ideally, Membrane would like to unwind the execution of the thread to the beginning of the first kernel-to-file-system call as described above. However, the fact that (non-file-system) kernel code has run complicates the unwinding; kernel state will *not* be cleaned up during recovery, and thus any state modifications made by the kernel must be undone before restart.

For example, assume that the file system code is executing (e.g., in function `f1()`) and calls into the kernel (function `k1()`); the kernel then updates kernel-state in some way (e.g., allocates memory or grabs locks) and then calls back into the file system (function `f2()`); finally, `f2()` returns to `k1()` which returns to `f1()` which completes. The tricky case arises when `f2()` crashes; if we simply unwound execution naively, the state modifications made while in the kernel would be left intact, and the kernel could quickly become unusable.

To overcome this challenge, Membrane employs a careful *skip/trust unwind protocol*. The protocol *skips* over file system code but *trusts* the kernel code to behave reasonable in response to a failure and thus manage kernel state correctly. Membrane coerces such behavior by carefully arranging the return value on the stack, mimicking an error return from the failed file-system routine to the kernel; the kernel code is then allowed to run and clean up as it sees fit. We found that the Linux kernel did a good job of checking return values from the file-system function and in handling error conditions. In places where it did not (12 such instances), we explicitly added code to do the required check.

In the example above, when the fault is detected in `f2()`, Membrane places an error code in the appropriate location on the stack and returns control immediately to `k1()`. This trusted kernel code is then allowed to execute, hopefully freeing any resources that it no longer needs (e.g., memory, locks) before returning control to `f1()`. When the return to `f1()` is attempted, the control-flow capture machinery again kicks into place and enables Membrane to unwind the remainder of the stack. A real example from Linux is shown in Figure 4.

Throughout this process, the u-stack is used to capture the necessary state to enable Membrane to unwind properly. Thus, both when the file system is first entered as well as any time the kernel calls into the file system, wrapper functions push register state onto the u-stack; the values are subsequently popped off on return, or used to skip back through the stack during unwind.

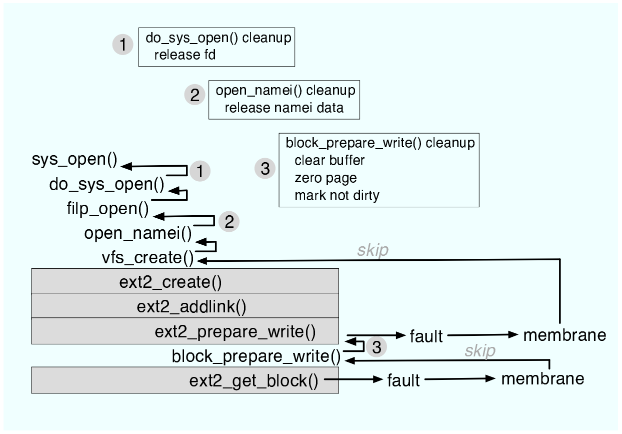

Figure 4: **The Skip/Trust Unwind Protocol.** *The figure depicts the call path from the `open()` system call through the ext2 file system. The first sequence of calls (through `vfs_create()`) are in the generic (trusted) kernel; then the (untrusted) ext2 routines are called; then ext2 calls back into the kernel to prepare to write a page, which in turn may call back into ext2 to get a block to write to. Assume a fault occurs at this last level in the stack; Membrane catches the fault, and skips back to the last trusted kernel routine, mimicking a failed call to `ext2_get_block()`; this routine then runs its normal failure recovery (marked by the circled "3" in the diagram), and then tries to return again. Membrane's control-flow capture machinery catches this and then* skips *back all the way to the last trusted kernel code (`vfs_create`), thus mimicking a failed call to `ext2_create()`. The rest of the code unwinds with Membrane's interference, executing various cleanup code along the way (as indicated by the circled 2 and 1).*

4.4.3 Other Recovery Functions

There are many other aspects of recovery which we do not discuss in detail here for sake of space. For example, the RM must orchestrate the entire recovery protocol, ensuring that once threads are unwound (as described above), the rest of the recovery protocol to unmount the file system, free various objects, remount it, restore sessions, and replay file system operations recorded in the logs, is carried out. Finally, after recovery, RM allows the file system to begin servicing new requests.

4.4.4 Correctness of Recovery

We now discuss the correctness of our recovery mechanism. Membrane throws away the corrupted in-memory state of the file system immediately after the crash. Since faults are fail-stop in Membrane, on-disk data is never corrupted. We also prevent any new operation from being issued to the file system while recovery is being performed. The file-system state is then reverted to the last known checkpoint (which is guaranteed to be consistent). Next, successfully completed op-logs are replayed to restore the file-system state to the crash time. Finally, the unwound processes are allowed to execute again.

Non-determinism could arise while replaying the completed operations. The order recorded in op-logs need not be the same as the order executed by the scheduler. This new execution order could potentially pose a problem while replaying completed write operations as applications could have observed the modified state (via *read*) before the crash. On the other hand, operations that modify the file-system state (such as create, unlink, etc.) would not be a problem as conflicting operations are resolved by the file system through locking.

Membrane avoids non-deterministic replay of completed write operations through page stealing. While replaying completed operations, Membrane reads the final version of the page from the page cache and re-executes the write operation by copying the data from it. As a result, write operations while being replayed will end up with the same final version no matter what order they are executed. Lastly, as the in-flight operations have not returned back to the application, Membrane allows the scheduler to execute them in arbitrary order.

5 Evaluation

We now evaluate Membrane in the following three categories: transparency, performance, and generality. All experiments were performed on a machine with a 2.2 GHz Opteron processor, two 80GB WDC disks, and 2GB of memory running Linux 2.6.15. We evaluated Membrane using ext2, VFAT, and ext3. The ext3 file system was mounted in data journaling mode in all the experiments.

5.1 Transparency

We employ fault injection to analyze the transparency offered by Membrane in hiding file system crashes from applications. The goal of these experiments is to show the inability of current systems in hiding faults from application and how using Membrane can avoid them.

Our injection study is quite targeted; we identify places in the file system code where faults may cause trouble, and inject faults there, and observe the result. These faults represent transient errors from three different components: virtual memory (e.g., kmap, d_alloc_anon), disks (e.g., write_full_page, sb_bread), and kernel-proper (e.g., clear_inode, iget). In all, we injected 47 faults in different code paths in three file systems. We believe that many more faults could be injected to highlight the same issue.

Table 3 presents the results of our study. The caption explains how to interpret the data in the table. In all experiments, the operating system was always usable after fault injection (not shown in the table). We now discuss our major observations and conclusions.

		ext2				ext2+ boundary				ext2+ Membrane			
ext2_Function	**Fault**	**How Detected?**	**Application?**	**FS:Consistent?**	**FS:Usable?**	**How Detected?**	**Application?**	**FS:Consistent?**	**FS:Usable?**	**How Detected?**	**Application?**	**FS:Consistent?**	**FS:Usable?**
create	null-pointer	o	×	×	×	o	×	×	×	d	√	√	√
create	mark_inode_dirty	o	×	×	×	o	×	×	×	d	√	√	√
writepage	write_full_page	o	×	√	√[a]	d	s	×	√[a]	d	√	√	√
writepages	write_full_page	o	×	×	√[a]	d	s	×	√[a]	d	√	√	√
free_inode	mark_buffer_dirty	o	×	×	×	o[b]	×	×	√[a]	d	√	√	√
mkdir	d_instantiate	o	×	×	×	d	s	√	√	d	√	√	√
get_block	map_bh	o	×	×	√[a]	o[b]	×	×	×	d	√	√	√
readdir	page_address	G	×	×	×	G	×	×	×	d	√	√	√
get_page	kmap	o	×	√	×	o[b]	×	√	×	d	√	√	√
get_page	wait_page_locked	o	×	√	×	o[b]	×	√	×	d	√	√	√
get_page	read_cache_page	o	×	√	×	o	×	√	×	d	√	√	√
lookup	iget	o	×	√	×	o[b]	×	√	×	d	√	√	√
add_nondir	d_instantiate	o	×	×	×	d	e	√	√	d	√	√	√
find_entry	page_address	G	×	√	×	G[b]	×	√	×	d	√	√	√
symlink	null-pointer	o	×	×	×	o	×	√	×	d	√	√	√
rmdir	null-pointer	o	×	√	×	o	×	√	×	d	√	√	√
empty_dir	page_address	G	×	√	×	G	×	√	×	d	√	√	√
make_empty	grab_cache_page	o	×	√	×	o[b]	×	×	×	d	√	√	√
commit_chunk	unlock_page	o	×	√	×	d	e	×	×	d	√	√	√
readpage	mpage_readpage	o	×	√	√	i	×	√	√	d	√	√	√
vfat_Function	**Fault**	**vfat**				**vfat+ boundary**				**vfat+ Membrane**			
create	null-pointer	o	×	×	×	o	×	×	×	d	√	√	√
create	d_instantiate	o	×	×	×	o	×	×	×	d	√	√	√
writepage	blk_write_fullpage	o	×	×	√[a]	d	s	×	√[a]	d	√	√	√
mkdir	d_instantiate	o	×	√	×	d	s	√	√	d	√	√	√
rmdir	null-pointer	o	×	√	×	o	×	√	√[a]	d	√	√	√
lookup	d_find_alias	o	×	√	×	d	e	√	√	d	√	√	√
get_entry	sb_bread	o	×	√	×	o	×	√	×	d	√	√	√
get_block	map_bh	o	×	×	√[a]	o	×	×	√[a]	d	√	√	√
remove_entries	mark_buffer_dirty	o	×	×	√[a]	d	s	×	√	d	√	√	√
write_inode	mark_buffer_dirty	o	×	×	√[a]	d	s	√	√	d	√	√	√
clear_inode	is_bad_inode	o	×	×	√[a]	d	s	√	√	d	√	√	√
get_dentry	d_alloc_anon	o	×	×	√[a]	o[b]	×	×	×	d	√	√	√
readpage	mpage_readpage	o	×	√	√[a]	o	×	√	√[a]	d	√	√	√
ext3_Function	**Fault**	**ext3**				**ext3+ boundary**				**ext3+ Membrane**			
create	null-pointer	o	×	×	×	o	×	√	×	d	√	√	√
get_blk_handle	bh_result	o	×	×	×	d	s	×	√[a]	d	√	√	√
follow_link	nd_set_link	o	×	×	√[a]	d	e	√	√	d	√	√	√
mkdir	d_instantiate	o	×	×	×	d	s	√	√	d	√	√	√
symlink	null-pointer	o	×	×	×	d	×	√	×	d	√	√	√
readpage	mpage_readpage	o	×	×	√[a]	d	×	√	√[a]	d	√	√	√
add_nondir	d_instantiate	o	×	√	×	o	×	√	×	d	√	√	√
prepare_write	blk_prepare_write	o	×	√	×	i	e	√	√	d	√	√	√
read_blk_bmap	sb_bread	o	×	√	×	o	×	√	×	d	√	√	√
new_block	dquot_alloc_blk	o	×	√	×	o	×	√	×	d	√	√	√
readdir	null-pointer	o	×	×	×	o	×	√	√[a]	d	√	√	√
file_write	file_aio_write	G	×	√	√	i	e	√	√	d	√	√	√
free_inode	clear_inode	o	×	×	×	o	×	√	×	d	√	√	√
new_inode	null-pointer	o	×	√	×	i	×	×	√[a]	d	√	√	√

Table 3: **Fault Study.** *The table shows the results of fault injections on the behavior of Linux ext2, VFAT and ext3. Each row presents the results of a single experiment, and the columns show (in left-to-right order): which routine the fault was injected into, the nature of the fault, how/if it was detected, how it affected the application, whether the file system was consistent after the fault, and whether the file system was usable. Various symbols are used to condense the presentation. For detection, "o": kernel oops; "G": general protection fault; "i": invalid opcode; "d": fault detected, say by an assertion. For application behavior, "×": application killed by the OS; "√": application continued operation correctly; "s": operation failed but application ran successfully (silent failure); "e": application ran and returned an error. Footnotes: [a]- file system usable, but un-unmountable; [b] - late oops or fault, e.g., after an error code was returned.*

Benchmark	ext2	ext2+ Membrane	ext3	ext3+ Membrane	VFAT	VFAT+ Membrane
Seq. read	17.8	17.8	17.8	17.8	17.7	17.7
Seq. write	25.5	25.7	56.3	56.3	18.5	20.2
Rand. read	163.2	163.5	163.2	163.2	163.5	163.6
Rand. write	20.3	20.5	65.5	65.5	18.9	18.9
create	34.1	34.1	33.9	34.3	32.4	34.0
delete	20.0	20.1	18.6	18.7	20.8	21.0

Table 4: **Microbenchmarks.** *This table compares the execution time (in seconds) for various benchmarks for restartable versions of ext2, ext3, VFAT (on Membrane) against their regular versions on the unmodified kernel. Sequential read/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	ext2	ext2+ Membrane	ext3	ext3+ Membrane	VFAT	VFAT+ Membrane
Sort	142.2	142.6	152.1	152.5	146.5	146.8
OpenSSH	28.5	28.9	28.7	29.1	30.1	30.8
PostMark	46.9	47.2	478.2	484.1	43.1	43.8

Table 5: **Macrobenchmarks.** *The table presents the performance (in seconds) of different benchmarks running on both standard and restartable versions of ext2, VFAT, and ext3. 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.*

First, we analyzed the vanilla versions of the file systems on standard Linux kernel as our base case. The results are shown in the leftmost result column in Table 3. We observed that Linux does a poor job in recovering from the injected faults; most faults (around 91%) triggered a kernel "oops" and the application (i.e., the process performing the file system operation that triggered the fault) was always killed. Moreover, in one-third of the cases, the file system was left unusable, thus requiring a reboot and repair (*fsck*).

Second, we analyzed the usefulness of fault detection without recovery by hardening the kernel and file-system boundary through parameter checks. The second result column (denoted by +boundary) of Table 3 shows the results. Although assertions detect the bad argument passed to the kernel proper function, in the majority of the cases, the returned error code was not handled properly (or propagated) by the file system. The application was always killed and the file system was left inconsistent, unusable, or both.

Finally, we focused on file systems surrounded by Membrane. The results of the experiments are shown in the rightmost column of Table 3; faults were handled, applications did not notice faults, and the file system remained in a consistent and usable state.

In summary, even in a limited and controlled set of fault injection experiments, we can easily realize the usefulness of Membrane in recovering from file system crashes. In a standard or hardened environment, a file system crash is almost always visible to the user and the process performing the operation is killed. Membrane, on detecting a file system crash, transparently restarts the file system and leaves it in a consistent and usable state.

5.2 Performance

To evaluate the performance of Membrane, we run a series of both microbenchmark and macrobenchmark workloads where ext2, VFAT, and ext3 are run in a standard environment and within the Membrane framework.

Tables 4 and 5 show the results of our microbenchmark and macrobenchmark experiments respectively. From the tables, one can see that the performance overheads of our prototype are quite minimal; in all cases, the overheads were between 0% and 2%.

(a)

Data (MB)	Recovery time (ms)
10	12.9
20	13.2
40	16.1

(b)

Open Sessions	Recovery time (ms)
200	11.4
400	14.6
800	22.0

(c)

Log Records	Recovery time (ms)
1K	15.3
10K	16.8
100K	25.2

Table 6: **Recovery Time.** *Tables a, b, and c show recovery time as a function of dirty pages (at checkpoint), s-log, and op-log respectively. Dirty pages are created by copying new files. Open sessions are created by getting handles to files. Log records are generated by reading and seeking to arbitrary data inside multiple files. The recovery time was 8.6ms when all three states were empty.*

Recovery Time. Beyond baseline performance under no crashes, we were interested in studying the performance of Membrane during recovery. Specifically, how long does it take Membrane to recover from a fault? This metric is particularly important as high recovery times may be noticed by applications.

We measured the recovery time in a controlled environment by varying the amount of state kept by Membrane and found that the recovery time grows sub-linearly with the amount of state and is only a few milliseconds in all the cases. Table 6 shows the result of varying the amount of state in the s-log, op-log and the number of dirty pages from the previous checkpoint.

We also ran microbenchmarks and forcefully crashed ext2, ext3, and VFAT file systems during execution to measure the impact in application throughput inside Membrane. Figure 5 shows the results for performing recovery during the random-read microbenchmark for the ext2 file system. From the figure, we can see that Membrane restarts the file system within 10ms from the point of crash. Subsequent read operations are slower than the regular case because the indirect blocks, that were cached by the file system, are thrown away at recovery time in our current prototype and have to be read back again after recovery (as shown in the graph).

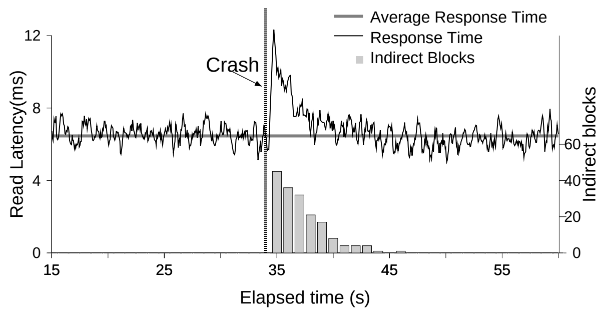

Figure 5: **Recovery Overhead.** *The figure shows the overhead of restarting ext2 while running random-read microbenchmark. The x axis represents the overall elapsed time of the microbenchmark in seconds. The primary y axis contains the execution time per read operation as observed by the application in milliseconds. A file-system crash was triggered at 34s, as a result the total elapsed time increased from 66.5s to 67.1s. The secondary y axis contains the number of indirect blocks read by the ext2 file system from the disk per second.*

In summary, both micro and macrobenchmarks show that the fault anticipation in Membrane almost comes for free. Even in the event of a file system crash, Membrane restarts the file system within a few milliseconds.

5.3 Generality

We chose ext2, VFAT, and ext3 to evaluate the generality of our approach. ext2 and VFAT were chosen for their lack of crash consistency machinery and for their completely different on-disk layout. ext3 was selected for its journaling machinery that provides better crash consistency guarantees than ext2. Table 7 shows the code changes required in each file system.

File System	**Added**	**Modified**
ext2	4	0
VFAT	5	0
ext3	1	0
JBD	4	0

Individual File-system Changes

Components	**No Checkpoint**		**With Checkpoint**	
	Added	Modified	Added	Modified
FS	1929	30	2979	64
MM	779	5	867	15
Arch	0	0	733	4
Headers	522	6	552	6
Module	238	0	238	0
Total	**3468**	**41**	**5369**	**89**

Kernel Changes

Table 7: **Implementation Complexity.** *The table presents the code changes required to transform a ext2, VFAT, ext3, and vanilla Linux 2.6.15 x86_64 kernel into their restartable counterparts. Most of the modified lines indicate places where vanilla kernel did not check/handle errors propagated by the file system. As our changes were non-intrusive in nature, none of existing code was removed from the kernel.*

From the table, we can see that the file system specific changes required to work with Membrane are minimal. For ext3, we also added 4 lines of code to JBD to notify the beginning and the end of transactions to the checkpoint manager, which could then discard the operation logs of the committed transactions. All of the additions were straightforward, including adding a new header file to propagate the `GFP_RESTARTABLE` flag and code to write back the free block/inode/cluster count when the `write_super` method of the file system was called. No modification (or deletions) of existing code were required in any of the file systems.

In summary, Membrane represents a generic approach to achieve file system restartability; existing file systems can work with Membrane with minimal changes of adding a few lines of code.

6 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, knowing that it won't bring down their entire operation.

Membrane further encourages developers to harden their code and catch bugs as soon as possible. This fringe benefit will likely lead to more bugs being triggered in the field (and handled by Membrane, hopefully); if so, diagnostic information could be captured and shipped back to the developer, further improving file system robustness.

We live in an age of imperfection, and software imperfection seems a fact of life rather than a temporary state of affairs. With Membrane, we can learn to embrace that imperfection, instead of fearing it. 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.

7 Acknowledgments

We thank the anonymous reviewers and Dushyanth Narayanan (our shepherd) for their feedback and comments, which have substantially improved the content and presentation of this paper. We also thank Haryadi Gunawi for his insightful comments.

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, Sun Microsystems, 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 NSF or other institutions.

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] George Candea and Armando Fox. Crash-Only Software. In *The Ninth Workshop on Hot Topics in Operating Systems (HotOS IX)*, Lihue, Hawaii, May 2003.

[3] 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.

[4] John Chapin, Mendel Rosenblum, Scott Devine, Tirthankar Lahiri, Dan Teodosiu, and Anoop Gupta. Hive: Fault Containment for Shared-Memory Multiprocessors. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, Copper Mountain Resort, Colorado, December 1995.

[5] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An Empirical Study of Operating System Errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 73–88, Banff, Canada, October 2001.

[6] Charles D. Cranor and Gurudatta M. Parulkar. The UVM Virtual Memory System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '99)*, Monterey, California, June 1999.

[7] 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.

[8] 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 *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 57–72, Banff, Canada, October 2001.

[9] Ulfar Erlingsson, Martin Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software Guards for System Address Spaces. In *Proceedings of the 7th USENIX OSDI*, pages 6–6, 2006.

[10] 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.

[11] Haryadi S. Gunawi, Cindy Rubio-Gonzalez, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Ben Liblit. EIO: Error Handling is Occasionally Correct. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, pages 207–222, San Jose, California, February 2008.

[12] 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.

[13] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Construction of a Highly Dependable Operating System. In *Proceedings of the 6th European Dependable Computing Conference*, October 2006.

[14] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Failure Resilience for Device Drivers. In *Proceedings of the 2007 IEEE International Conference on Dependable Systems and Networks*, pages 41–50, June 2007.

[15] 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.

[16] 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.

[17] E. Koldinger, J. Chase, and S. Eggers. Architectural Support for Single Address Space Operating Systems. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, Boston, Massachusetts, October 1992.

[18] Nathan P. Kropp, Philip J. Koopman, and Daniel P. Siewiorek. Automated Robustness Testing of Off-the-Shelf Software Components. In *Proceedings of the 28th International Symposium on Fault-Tolerant Computing (FTCS-28)*, Munich, Germany, June 1998.

[19] James Larus. The Singularity Operating System. Seminar given at the University of Wisconsin, Madison, 2005.

[20] 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.

[21] 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.

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

[23] Jeffrey C. Mogul. A Better Update Policy. In *Proceedings of the USENIX Summer Technical Conference (USENIX Summer '94)*, Boston, Massachusetts, June 1994.

[24] Zachary Peterson and Randal Burns. Ext3cow: a time-shifting file system for regulatory compliance. *Trans. Storage*, 1(2):190–212, 2005.

[25] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 206–220, Brighton, United Kingdom, October 2005.

[26] 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.

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

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

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

[30] 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.

[31] 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.

[32] 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.

[33] Nisha Talagala and David Patterson. An Analysis of Error Behaviour in a Large Storage System. In *The IEEE Workshop on Fault Tolerance in Parallel and Distributed Systems*, San Juan, Puerto Rico, April 1999.

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

[35] 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.

[36] W. Weimer and George C. Necula. Finding and Preventing Runtime Error-Handling Mistakes. In *The 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '04)*, Vancouver, Canada, October 2004.

[37] 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.

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

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

[40] 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 USENIX OSDI*, Seattle, Washington, November 2006.