RECOVERY TECHNIQUES TO IMPROVE FILE SYSTEM RELIABILITY

by

Swaminathan Sundararaman

B.E. (Hons.) Computer Science
(Birla Institute of Technology and Science (Pilani), India) 2005
M.S. Computer Sciences
(Stony Brook University) 2007

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy
in
Computer Sciences

University of Wisconsin-Madison

2011

Committee in charge:

- Prof. Andrea C. Arpaci-Dusseau (Co-chair)
- Prof. Remzi H. Arpaci-Dusseau (Co-chair)
- Prof. Ben Liblit
- Prof. Michael M. Swift
- Prof. Shan Lu
- Prof. Jon Eckhardt

To my parents

Acknowledgements

I would like to express my heart-felt thanks to my advisors, Andrea Arpaci-Dusseau and Remzi Arpaci-Dusseau, who guided me in this endeavor and taught me how to do research. I always wanted to build really cool systems during my Ph.D. Not only did they let me do this, but they also taught me the valuable lesson that solutions (or systems) are just a small part of doing research. I also learnt the importance of clarity in both writing and presentation from them.

I would like to thank Andrea for teaching me the values of patience, good questions, and paying attention to details. I would like to thank Remzi for helping me select interesting projects and, most importantly, for showing me that research is a lot of fun. I am ever grateful to Remzi for taking the time to write me a long email and talk with me on the phone while I was deciding between universities. If not for that email, I would have ended up taking a full-time position at one of the companies in the Bay Area – and regretted not doing a Ph.D. for the rest of my life.

I am grateful to my thesis committee members – Mike Swift, Ben Liblit, Shan Lu, and Jon Eckhardt – for their insightful comments and suggestions. Their feedback during my preliminary and final exams improved this thesis greatly.

I would like to thank Mike Swift for collaborating with us on the Membrane project. It was one of the best learning experiences for me. During our meetings, he would go into the smallest implementation detail and then quickly switch back to the big picture ideas. After my meetings with him, I would go back and spend a few hours trying to understand what he had been able to say in only a few seconds during the meetings.

I am thankful to Sriram Subramanian for putting up with me during my best and worst times in Madison. Even though both of us graduated together from BITS, we hardly knew each other at the time we came to the UW. The friendship began with me sending him an email asking if he wanted to be my roommate – to which he agreed. Since then, it has been a great journey; he has been my roommate for the last four years, office mate for three years, project mate and a collaborator for last four years, and last but not least, a great friend. I have spent many hours

brainstorming ideas with him and he has helped a lot in refining and validating my ideas and designs.

I am fortunate to have had the opportunity to work with smart and hardworking colleagues during the course of my Ph.D.: Leo Arulraj, Lakshmi Bairavasundaram, Abhishek Rajimwale, Sriram Subramanian, Laxman Visampalli, Yiying Zhang, and Yupu Zhang. I also have enjoyed interacting with other members in the group: Ishani Ahuja, Nitin Agrawal, Vijay Chidambaram, Thanh Do, Haryadi Gunawi, Tyler Harter, Ao Ma, Joe Meehean, Thanumallayan Pillai, Deepak Ramamurthi, Lanyue Lu, Suli Yang. I would especially like to thank Abhishek, Sriram, Laxman, and Yupu for their hard work, contributions, and often burning the midnight oil with me during deadline times.

A lot of my motivation for pursuing a Ph.D. came from my experiences at Stony Brook University and Birla Institute of Technology and Science (BITS). I would like to especially thank my advisors: Erez Zadok at Stony Brook, for fostering in me an interest in file and storage systems and teaching me that kernel programming is a lot of fun; Rahul Banerjee at BITS, for enabling me to think big and dream about projects that were beyond my capabilities; and Sundar Balasubramanian at BITS, for teaching me the importance of novelty and creativity in doing research.

I have benefited greatly from interning at VMware, Microsoft Research, and EPFL. I would like to thank the organizations as well as my mentors and managers there: Satyam Vaghani at VMware, Dennis Fetterly at Microsoft Research, and George Candea at EPFL. I would also like to thank the other collaborators at these organizations: Abhishek Rai, Yunlin Tan, and Mayank Rawat at VMware; Michael Isard and Chandu Thekkath at Microsoft Research; and Vitaly Chipounov and Vova Kuznetsov at EPFL.

I would like to thank my friends for making my stay in Madison enjoyable. To name a few: Koushik Narasimhan, Srinath Sridharan, Uthra Srinath, Piramanayagam Arumuga Nainar, Siddharth Barman, Asim Kadav, Janani Kalyanam, Ashok Anand, Deepthi Pachauri, Giri Ravipati, Rathijit Sen, Neelam Goyal, Nikhil Teletia, and Abhishek Iyer. They were happy even for my small successes and cheered me during difficult times. I am going to miss all the fun we had: evening movies, potluck dinners, tennis, cricket, poker sessions, and road trips. I would like to specially thank Kaushik for coming up with fancy names and for the constant nudging during deadlines, which kept it fun for me.

Finally, I would like to thank my family, without whose love and support this Ph.D. would have been impossible. Most of the credit, however, is due to my parents. They live their life to make me happy, and they have given me everything I have needed to find my way in the world and succeed. They have always been very

supportive of my decisions and have been the two pillars that I can always lean on. My sister, Harini, has cheered me on during the entire course of my Ph.D. I would also like to thank my cousin, Rajesh, and his wife, Niranjana, for their constant encouragement and support. Finally, I would also like to thank all my relatives who always encouraged and supported me in all of my endeavors.

Abstract

RECOVERY TECHNIQUES TO IMPROVE FILE SYSTEM RELIABILITY
Swaminathan Sundararaman

We implement selective restart and resource reservation for commodity file systems to improve their reliability. Selective restart allows file systems to quickly recover from failures; resource reservation enables file systems to avoid certain failures altogether. Together they enable a new class of more robust and reliable file systems to be realized.

In the first part of this dissertation (on selective restart), we develop Membrane, a generic framework built inside the operating system to selectively restart kernel-level 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. We also develop Re-FUSE, a generic framework designed to restart user-level file systems upon failures. Re-FUSE monitors the user-level file-system and on a crash restarts the file system and restores its state; the restart process is completely transparent to applications. We evaluate both Membrane and Re-FUSE, and show, through experimentation, that both of these frameworks induce little performance and space overhead and can tolerate a wide range of crashes with minimal code change.

In the second part of the dissertation (on resource reservation), we develop Anticipatory Memory Allocation (AMA), a technique that uses static analysis to simplify recovery code dealing with memory-allocation failures. AMA determines the memory requirements of a particular call into a file system, and then pre-allocates said amount immediately upon entry; subsequent allocation requests are serviced from the pre-allocated pool and thus guaranteed never to fail. We evaluate AMA by transforming Linux ext2 file system into a memory-failure robust version of itself (called ext2-mfr). Experiments reveal that ext2-mfr avoids memory-allocation failures successfully while incurring little space and time overheads.

Contents

Chapter 1

Introduction

> *"It's not the prevention of bugs but the recovery, the ability to gracefully exterminate them, that counts."* – Victoria Livschitz

With the advent of low-cost disk drives, storage capacities and disk utilization have grown at rapid rates [58]. It is now possible for users to store terabytes of data on a modern disk drive. As the amount of data increases, access to data is even more critical, with data unavailability costing users mental anguish and companies millions of dollars per hour [38, 97, 131].

File systems are the commonly used software for managing data on disk. Modern file systems are large code bases with tens of thousands of lines of code and they support many features, protocols, and operations [23, 191]. 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 [32], ext3 [183], reiserfs [145], and still there is great interest in next-generation file systems such as Linux ext4 [113], btrfs [191], and ZFS [23]. Thus, file systems are large, complex, and under development – the perfect storm for numerous bugs to arise.

A great deal of recent activity in systems research has focused on new techniques for finding bugs in file systems [37, 49, 50, 74, 195]. Researchers have built tools that use static analysis [49, 74], model checking [107, 197], symbolic execution [29, 196], machine learning [106], and other testing-based techniques [11, 14, 138], all of which have uncovered hundreds of bugs in commonly-used file systems.

The majority of the software defects are found in *recovery code*; i.e., code that is run in reaction to a failure. Although these failures, whether from hardware (e.g., a disk) or software (e.g., a memory allocation), tend to occur infrequently in practice, the correctness of recovery code is nevertheless critical. For example, Yang et al. found a large number of bugs in file-system recovery code; when such

bugs were triggered, the results were often catastrophic, resulting in data corruption or unmountable file systems [197]. Recovery code has the worst possible property: it is rarely run, but must work absolutely correctly.

It is challenging to implement robust recovery code for the following reasons. First, current file systems have poor failure models and policies [69]. The recovery code is distributed across the file system and the success of recovery depends on the correctness of recovery code in each involved function. Unfortunately, not all functions in file systems and the kernel handle and propagate errors correctly [71, 138]. Even if file system developers are aware of the specific problem, it is hard for them to implement the correct recovery strategy due to inherently complex file system designs [69].

Second, manual implementation of recovery code combined with the large number of error scenarios inhibits scalability of recovery code. For example, there are around 100 different error cases that can arise in the Linux operating system and currently, there is no single way of handling errors in a file system. A developer has to manually write recovery code for every function in the file system that correctly handles the error code and propagates the error back to the caller. In many cases, the developer needs to implement different recovery strategies for different types of errors [138]. Given the number of errors that one must handle in the file system, it is easy to miss an error scenario or implement an incorrect recovery strategy.

The implications of poor recovery code depend on the nature of the fault and the state of the file system. In worst case scenarios, failures can lead to file system crashes. There are two primary reasons that such file system crashes are harmful. First, when a file system crashes, manual intervention is often required to repair any damage and restart the file system; thus, crashed file systems stay down for noticeable stretches of time and decrease availability dramatically, requiring costly human time to repair. Second, crashes give users the sense that a file system "does not work" and thus decrease the chances for adoption of new file systems.

The fact that file systems do not have robust recovery codes, combined with the existence of bugs in file system code, leaves us with a significant challenge: How can we promise users that file systems work robustly in spite of their massive software complexity and all the complex failures that can arise? We propose that the right approach is to accept the fact that failures are inevitable in file systems; we must learn to cope with failures rather than hoping to avoid them. To respond to this challenging question in a manner that accepts inevitable failures, we build new recovery techniques on top of the increasingly complex and less reliable file systems.

The goals of this thesis are two-fold: first, to develop techniques to improve

availability of file systems in the event of failures; second, to develop techniques to minimize the complexity of recovery code in file systems.

We address the goals of this thesis as follows. To improve availability, we restart file systems on failures, and to simplify recovery code, we employ resource reservation. First, we develop Membrane, an operating system framework that restarts kernel-level file systems on failures [165, 166, 169], and Re-FUSE, a framework built inside the operating system and FUSE to restart user-level file systems on crashes [167]. Second, we develop Anticipatory Memory Allocator (AMA), a technique that combines static analysis, dynamic analysis, and file-system domain knowledge to simplify and minimize the recovery code needed to handle memory allocation failures in operating systems [168]. The following sections elaborate on each of these contributions of the thesis.

1.1 Reliability Through Restartability

Data availability can be improved by restarting file systems after failure. Though recent research work has developed techniques to tolerate mistakes in the other components with which file systems interact, these techniques still cannot tolerate failures inside the file system code [15, 69, 138]. Failures in file systems can be avoided through prevention techniques (such as detection and removal) or tolerated through restart mechanisms. Unfortunately, the prevention techniques that exist today do not uncover all possible bugs in file system code [49, 128]. Hence, a practical solution would be to selectively restart file systems on failures.

In the past, many restart mechanisms have been proposed for restarting both kernel-level and user-level components upon failure. Most of these techniques work only for stateless components such as device drivers or applications that do not have to maintain a persistent state on a disk [52, 79, 80, 104, 171, 172, 200]; hence, these techniques are not applicable for file systems. Other restart techniques that can handle persistent storage are heavyweight and require wide scale code changes in the operating system, file system, or both [31, 42, 103, 156]; the overheads and code changes make such techniques less attractive for commodity file systems.

In the first part of the dissertation, we explore the possibility of selective restart of kernel-level and user-level file systems on failures. File systems deployed inside the operating system are known as kernel-level file systems and file systems deployed outside the operating system (i.e., user space) are known as user-level file systems. As mentioned earlier, the selective restarting of file systems tolerates bugs in the file system code and hence improves data availability in systems. Also,

instead of providing a customized solution for individual file systems, we explore the possibility of providing a generic framework that can be leveraged by different commodity file systems.

Kernel-level file systems

To tolerate kernel-level file-system failures, we develop *Membrane*, an operating system framework to support lightweight, stateful recovery from file system crashes [165]. 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 allows the file system to resume service to application requests; applications are unaware of the crash and recover except for a small performance blip during restart.

Membrane achieves its performance and robustness through three novel mechanisms. First, 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. Second, a *page stealing* technique greatly reduces the 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 unwind protocol restores kernel state while preventing further file-system-induced damage from taking place.

Membrane does not add new fault detection mechanisms; instead, it leverages the existing fault-detection mechanisms in file systems. 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 [111]), 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 [33]) 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 [171].

We demonstrate the benefits of Membrane by evaluating it on three different file systems: ext2, VFAT, and ext3. Through experimentation we show that Membrane enables file systems to recover through a wide range of fault scenarios. We also show that Membrane incurs less than 5% performance overhead for commonly used benchmarks; furthermore only 5 lines of code needed to be changed to enable existing file systems to work with Membrane.

User-level file systems

User-level file systems are an alternative to kernel-level file systems and are commonly run using a platform like File systems in USEr space (FUSE) [141]. The FUSE simplifies the development and deployment of user-level file system as file systems run outside the operating system in a separate address space. In the last five years, around 200 different user-level file systems have been implemented using FUSE [192].

Faults in user-level file systems still impact their availability. Though faults in user-level file systems do not impact the correctness or availability of the operating system, applications that depend on the file system are still affected and, in almost all cases, such applications are prematurely terminated.

To understand how user-level file systems work in the real world, we first look at six representative user-level file systems: NTFS-3g, ext2fuse, SSHFS, AVFS, HTTPFS, and TagFS. Using these six file systems, we create a reference model for user-level file systems. From the reference model, we derive the common properties of this type of file system; we also find that by excluding the in-memory file system state, the rest of the state (including on-disk data) is preserved during a user-level file system crash.

We leverage the reference model to develop *Re-FUSE*, a framework built inside the operating system and FUSE that restarts user-level file systems on crashes [167]. During normal operations, Re-FUSE tracks the progress of in-flight file-system operations and caches the results of the system calls executed by the user-level file systems. On a file-system failure, Re-FUSE automatically restarts the file system and continues executing requests from their last execution point using the information recorded during normal operations. Similar to Membrane, the application that is using the user-level file system will not notice file-system failure, except perhaps for a small drop in performance during the restart.

Re-FUSE implements three basic techniques to enable lightweight restart of user-level file systems. The first is *request tagging*, which differentiates activities being performed on the behest of concurrent requests; the second is *system-call logging*, which carefully tracks the system calls executed by a user-level file system and caches their results; the third is *non-interruptible system calls*, which ensure atomicity of system-call execution by user-level file system threads. We also add *page versioning* and *socket buffering* optimizations to further reduce the performance overheads.

We evaluate Re-FUSE with three popular file systems: NTFS-3g, SSHFS, and AVFS. Through evaluation, we show that Re-FUSE can statefully restart the user-level file system, while hiding crashes from applications. We test the file systems with commonly used workloads and show that the space and performance overheads associated with running user-level file systems on Re-FUSE are minimal. Moreover, less than ten lines of code changes were required in each of the three file systems in order for them to work with Re-FUSE.

1.2 Reliability Through Reservation

Reservation is a popular technique that is used in many systems. In computer systems, reservation can be done for resources such as processors [93], memory [186], and network bandwidth [139]. The reservation of resources helps in improving fairness, quality of service, and reliability. In the context of reliability, the reservation of resources helps to simplify the recovery code, as all of the needed resources can be acquired at the beginning of an operation. In other words, through resource reservation, a system ensures that a resource-allocation failure can only happen during the reservation phase, which can be handled easily.

File systems extensively use heap-allocated memory to store in-memory copies of on-disk user data, on-disk metadata, as well as for other temporary objects. Unfortunately, the memory-allocation calls in operating systems can fail; commodity operating systems do not provide guarantees for the success of memory allocation calls. As a result, components (such as file systems) that use memory allocation routines are forced to handle memory-allocation failures at each calling site.

First, to understand the robustness of memory-allocation failure-handling code, we perform fault injection during memory allocation calls in commodity file systems. We test seven different Linux file systems: btrfs, ext2, ext3, ext4, jfs, reiserfs, and xfs. Interestingly, we found that all seven file systems are not robust to memory-allocation failure. In many cases, the memory-allocation failure resulted in an unusable or inconsistent file system. Moreover, the processes exe-

cuting file system requests are killed due to the inability of file systems to handle allocation failures correctly. These results are in alignment with previous work that has also shown that memory allocation failures are not handled properly in file systems and can lead to catastrophic results (such as data loss or data corruption) [50, 71, 120, 197].

Given that memory allocation failures are poorly handled in commodity file systems, we develop a technique called *Anticipatory Memory Allocation (AMA)* to simplify recovery code that is responsible for handling memory-allocation failures. The idea is simple: we move all the memory allocation calls to a single function that allocates all memory at the beginning of a request. If the pre-allocation succeeds, AMA guarantees that no file-system allocation fails downstream, as all subsequent memory allocation are serviced using the pre-allocated pool. Of course, allocation requests can fail during the pre-allocation phase, but unlike existing systems, we only need to perform shallow recovery wherein no state modification are done to the existing systems.

Pre-allocation of memory is challenging in commodity operating systems. File-system requests pass through different layers of the operating system; one example of this is through the virtual file system. In each layer, many functions can be invoked and any of these functions have the ability to potentially call a memory allocation routine. Moreover, commodity operating systems such as Linux allocates memory in a variety of ways (e.g., *kmalloc*, *kmem_cache_alloc*, and *alloc_pages*). Hence, to pre-allocate all objects, one needs to identify the potential allocation sites, object types, object sizes, and the total number of objects. Making this process somewhat more difficult is the fact that the sizes and parameters passed to the memory-allocation calls also depend on the on-disk state of the file system, the input parameters to the system call, and the cached state in the operating system.

To determine a conservative estimate of total memory allocation demand, AMA combines static analysis, dynamic analysis, and file-system domain knowledge. Using AMA, a developer can augment the file system code to pre-allocate all the memory at the entry of a system call. At run time, AMA transparently returns memory from the pre-allocated chunk for all memory-allocation calls. Thus, when a memory allocation takes place deep in the heart of the kernel subsystem, it is guaranteed never to fail.

With AMA, kernel code is written naturally, with memory allocations inserted wherever the developer requires them; however, with AMA, the developer need not be concerned with downstream memory-allocation failures and the scattered (and often buggy) recovery code that would otherwise be required. Furthermore, by allocating memory in one large chunk upon entry, failure of the anticipatory pre-

allocation is straightforward to handle. We also implement two uniform failure-handling polices (retry forever and fail fast) with little effort.

To show the practicality of AMA, we apply it to the ext2 file system in Linux. We transform ext2 to ext2-mfr, a memory-failure robust version of the ext2 file system. Through experimentation, we show that ext2-mfr is robust to memory allocation failure; even for an allocation-failure probability of .99, ext2-mfr is able to retry and eventually make progress, thus hiding failures from application processes. Moreover, we show that for many commonly-used benchmarks the performance and space overheads of running ext2-mfr are less than 7% and 8%, respectively.

1.3 Contributions

The contributions of this thesis are as follows:

- We implement Membrane, a framework inside operating systems to restart kernel-level file systems on failures. To the best of our knowledge, this is the first work that shows stateful restarts of file systems are possible in commodity operating systems.
- We show that is possible to checkpoint the state of kernel-level file systems in a generic way that requires minimal changes to the file system code. In contrast, existing solutions such as journaling or snapshotting require extensive file-system code changes and need to be implemented on a per-file-system basis.
- We develop a new protocol (skip/trust unwind) to selectively avoid recovery code in kernel-level file systems on failures. This protocol also helps clean up any residual state in the kernel and restores the kernel back to a consistent state. Though the protocol was developed in the context of file systems, it is applicable to other operating-system components and user-level programs.
- We develop Re-FUSE, a framework inside the operating system and FUSE to statefully restart user-level file systems on failures. This framework is generic, lightweight, and is independent of the underlying user-level file systems. To the best of our knowledge, no other solutions exist to restart user-level file systems on failures.
- We describe a reference model for user-level file systems. Our reference model helps guide the development of new file systems and modifications to existing file systems to work with Re-FUSE.

- We develop AMA to show that it is possible to have shallow recovery for memory-allocation failures during file system requests in commodity operating systems. We also show that it is possible to combine static analysis, dynamic analysis, and domain knowledge to estimate the heap-allocated memory required to satisfy requests in commodity operating systems.

1.4 Outline

The rest of this thesis is organized as follows.

- **Background:** Chapter 2 provides background on different aspects of file systems: components, request handling, on-disk state, consistency mechanisms, and deployment types.

- **Reliability through Restartability:** To help understand the design choices of our restartable systems, we first provide a taxonomy on fault models, describe the restart process in file systems, and components of a restartable framework in Chapter 3. We then begin presenting the first contribution of this thesis, where we describe the design and implementation of Membrane in Chapter 4; Membrane is a generic framework built inside the operating system to restart kernel-level file systems on failures. Then, in Chapter 5, we describe the design and implementation of Re-FUSE, a generic framework built inside the operating system and FUSE to restart user-level file systems on crashes.

- **Reliability through Reservation:** Chapter 6 presents the second major contribution of this thesis, where we explore reservation as a mechanism to improve file system reliability. In this chapter, we present the design and implementation of AMA, a solution that combines static analysis, dynamic analysis, and domain knowledge to estimate and hence, preallocate the amount of memory required to satisfy file system requests.

- **Related Work:** Chapter 7 summarizes research efforts related to building restartable systems and systems that use reservation as a mechanism to improve their reliability.

- **Conclusions and Future Work:** Chapter 8 concludes the thesis, first summarizing our work and highlighting the lessons learned, and then discussing various avenues for future work that arise from our research.

Chapter 2

File Systems

"There are two ways to write error-free programs; only the third works."
– Alan J. Perlis

This chapter provides a background on the various aspects of file systems that are integral to this dissertation. It begins with an overview of the role of file systems in managing user data on a disk, in Section 2.1. Next, Section 2.2 presents the common file-system components and their inter-component interactions. Section 2.3 describes how application requests are processed in file systems. In Section 2.4, important objects that constitute the on-disk state of file systems and the need for on-disk consistency are described, and Section 2.5 provides an overview of existing consistency mechanisms in file systems. Finally, Section 2.6 presents user-level and kernel-level file systems – the two common ways of deploying commodity file systems.

2.1 Overview

It is difficult for users to directly manage data on a disk. Users are accustomed to the notion of files and directories. Unfortunately, commodity disk systems do not come with interfaces that can deal with files and directories. Rather, commodity disks have a simplified block-based interface, where a disk presents stored data as a sequence of fixed-sized blocks containing bytes of information [55]. The disk interface typically supports read and write operations; the arguments to these operations are a block number and a buffer, and the result is data transfer from the buffer to the disk location or vice versa.

File systems are software modules that help users to organize their data on one or more disks. File systems allow users to access data in the granularity of files

and directories and internally translate user requests to reads and writes of disk blocks. File systems internally maintain some metadata to map files to a list of block locations on a disk. Figure 2.1 shows an example of how a file denoted by the name *Document A* could be stored on a disk by a file system. In addition to the mapping information, file systems also maintain additional information about directory contents, free blocks, and other relevant bookkeeping information.

The fundamental unit of data storage in file systems is a file. A file can be a regular file or a directory. A regular file contains user data and is represented by a name assigned by the user along with a unique identifier that is internally generated by the file system. The unique identifiers help to locate files efficiently on a disk. A directory is a special file whose contents could be file or directory names along with their unique identifiers. Directories help in grouping related entities inside a file system for easier access.

For efficient data organization, files are typically maintained in a hierarchical structure within a file system. At the topmost level, there is the main directory, popularly known as the *root* directory. The purpose of the root directory is to serve as an origin from which the file system grows. Figure 2.2 shows an example of a file-system hierarchy. From the figure, we can see that all files and directories can be reached from the root directory.

File systems need to support a variety of operations in order to enable users to effectively access their data on a disk. The operations that file systems need to support are well-defined by the operating system and library (such as FUSE) developers for kernel-level and user-level file systems, respectively. These operations are required to manipulate file system contents by users and applications. Table 2.1 shows some of the common operations that are supported by file systems.

A file system caches frequently accessed data in the memory and periodically synchronizes file system updates to the disk to improve performance. Disks are a few orders magnitude slower than memory and have significantly longer access times. Because of this, frequent access to the disk significantly slows down the performance of file systems. To improve performance, file systems (or operating systems) typically maintain a cache of frequently accessed data; when data is read or written to a disk, it is first cached in memory by the file system, so that subsequent requests can be serviced from the memory instead of going to the disk. Moreover, the caching of frequent updates and lazy writes of dirty data is beneficial because file accesses typically represent both spatial and temporal locality [16, 127, 148].

File systems are initialized and removed using mount and unmount operations, respectively. The mount operation initializes and loads the file system state after reading the disk contents. The unmount operation is used to safely persist recent

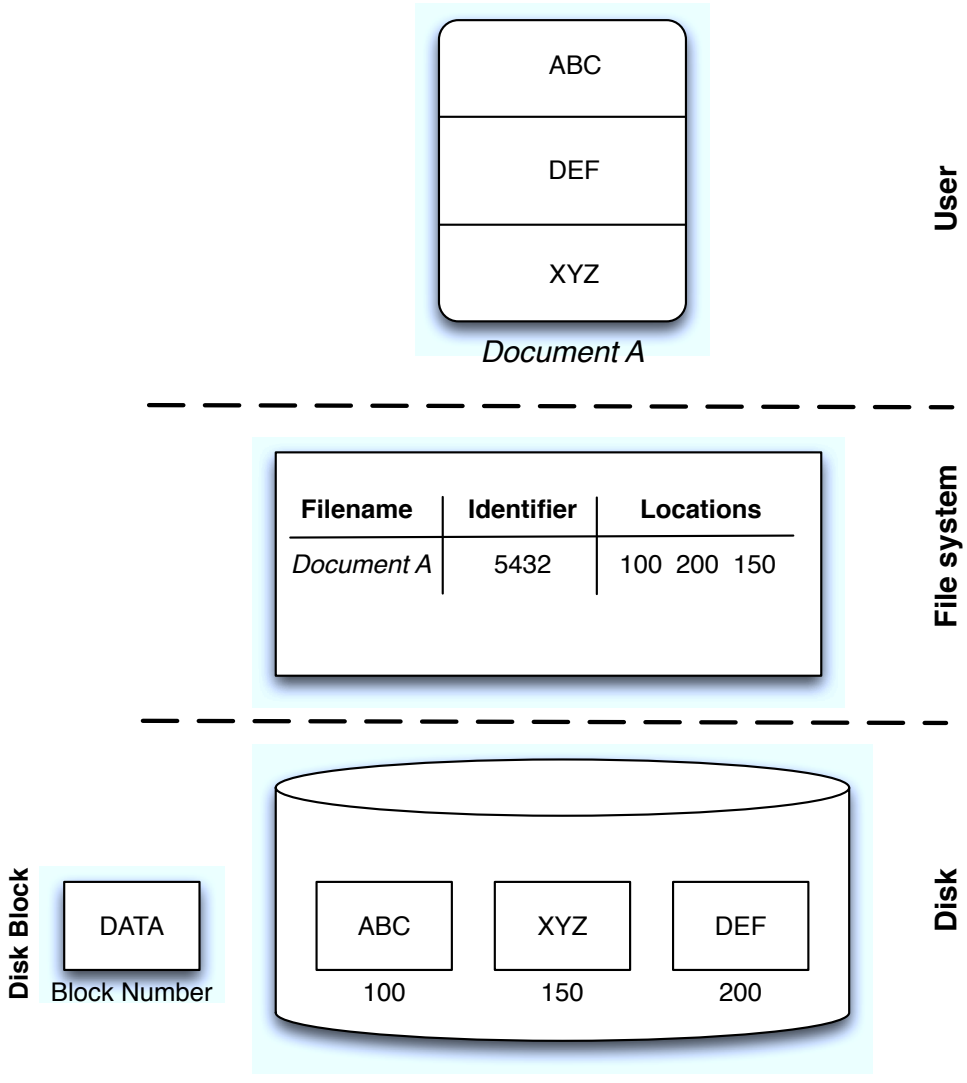

Figure 2.1: **Mapping Files to Disk Blocks.** *This figure shows a user file being internally mapped by a file system and stored in different locations on disk. The user file shown here is Document A which contains three blocks of data (i.e., ABC, DEF, and XYZ). These three blocks of data are internally mapped by the file system to three different disk locations: 100, 200, and 150. Each disk block contains data and is identified by a unique block number.*

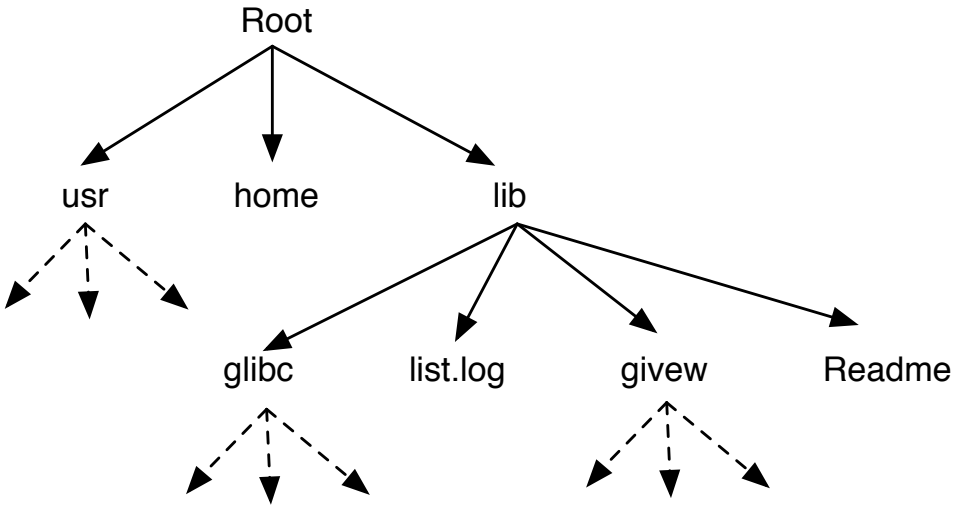

Figure 2.2: **File System Hierarchy Example.** *The figure shows an example of a simple file-system hierarchy. At the topmost level, we have the root directory. Underneath the root directory, we have usr, home, and lib directories. Further down, the lib directory has both directories (gview and glibc) and files (list.log and Readme) in it.*

Operation	Functionality	Arguments	Return Value
open	opens a file	path to a file	fh
close	closes a file	fh	status
read	Reads data from a file	fh, offset, bytes	data
write	Writes data to a file	fh, offset, buffer, bytes	status
mkdir	create a directory	dir path	status
rmdir	deletes a directory	dir path	status
readdir	return directory contents	dir path	entries
fsync	flush dirty data of a file	fh	status
sync	flush dirty data to disk	none	none

Table 2.1: **Common File-system Operations.** *The tables shows a few common operations supported by file systems. The argument column denotes the parameters that must be passed with the file system operation. Various symbols have been used to condense the presentation: fh - file handle, entries - directory entries, and dir path - path to a directory.*

changes to the disk and to clean up the in-memory file system state. More specifically, on unmount, file systems write back all dirty data that has not yet been written to the disk, free any cached data (or objects) in other operating-system components, and destroy any in-memory objects allocated by file systems.

2.2 File System Components

File systems can contain a variety of components depending on the features and guarantees they provide. In general, the majority of file systems contain the following four components: namespace management, data management, consistency management, and recovery. Figure 2.3 shows these file system components along with their inter-component interactions.

The namespace-management component handles all file system requests and is responsible for mapping user-visible filenames to unique identifiers that are internal to the file system. As mentioned earlier, other file system and operating system components do not understand filenames and require the unique identifiers in order to process requests. The common operation that this component handles is path traversal. In path traversal, a user sends in a list of directory names that need to be looked up in the file system hierarchy. Upon a path traversal request, the namespace component checks the permission and validity of each entry in the directory name list and returns the file handle to the caller if the request succeeds.

The data management component is responsible for managing storage space on disks. The responsibilities of the data management component include locating and reclaiming disk blocks used by the file system (i.e., freespace management), managing file-system metadata and data locations, and persistent storage of file-system metadata on the disk (see Section 2.4 for details). This component is essential in file systems, as disk systems are very simple and do not provide any support for high-level functionality such as freespace management [157].

The consistency management component is responsible for recording stable file system states to the disk. These stable states are used during the mount operation of file systems. To create such stable states, this component periodically groups file-system updates and atomically writes the updates to the disk (see Section 2.5 for details).

The recovery component is responsible for cleaning up file system states on errors. An error arises for various reasons: a file system mistake or a bug [49, 195, 197], data corruption [12, 13, 154], or unexpected behavior from the components with which file systems interact [153, 173, 196]. Ideally, file systems should be able to handle or tolerate such errors and continue servicing requests. Hence, this

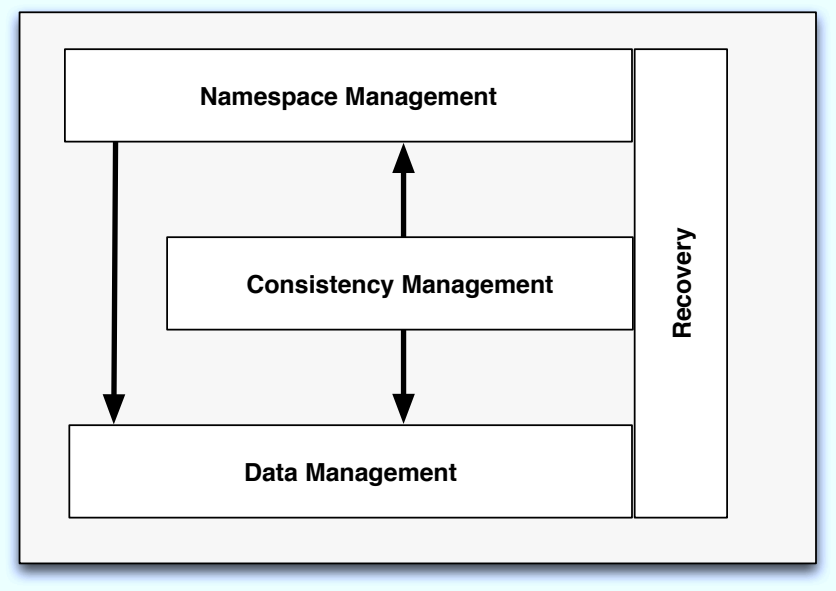

Figure 2.3: **Components of a File System.** *The figure shows the common components of a file system and its inter-component interactions. The common components are namespace management, data management, consistency management, and recovery. The namespace-management component interacts with the data-management component to retrieve directory contents. The consistency-management component interacts with both namespace- and data-management components to record file-system state on disk. The recovery component is spread across all file-system components.*

component is critical for ensuring file system availability even in the presence of errors. To the best of our knowledge, this component is never implemented as a stand-alone component, and is always tightly integrated with other file-system components [138].

2.3 Handling Application Requests

Applications interact with file systems through requests. As mentioned earlier, file systems support a variety of requests and these requests provide different functionality and guarantees depending on the type and parameters passed to it.

File system requests go through multiple layers in the storage stack when they are executed. Figure 2.4 shows how requests are executed in a kernel-level file system. From the figure, we can see that requests enter the operating system through the system call (or syscall) layer, which checks the validity of requests and then forwards requests to a virtual file system or an equivalent layer [100]. The virtual file system layer acts as a switch and forwards the request to the corresponding file system. The file system then processes these requests by accessing its in-memory contents, on-disk contents, or both. File systems might also have to interact with other components in order to reserve resources (such as memory) or use their services to complete its requests.

File system requests can be executed in either synchronous or asynchronous mode. In synchronous mode, modifications to user and file-system data are immediately written to the disk. In asynchronous mode, modifications to user and file system data are first cached in memory; a worker thread (or daemon) periodically writes the modifications to the disk in the background. By default, file system requests are executed in asynchronous mode, which helps improve file system performance [89, 125].

2.4 On-Disk State

The on-disk state of file systems consists of both file system metadata and user data. File system metadata consists of data structures such as inodes, bitmaps, extents, superblocks, etc. These metadata objects help locate and maintain user data and system metadata efficiently on disks. To improve their performance, file systems create in-memory copies of their on-disk objects and cache them in memory.

The correctness of the on-disk state is critical to the proper operation of the file system. As mentioned earlier, on-disk data is used to bootstrap the file system to its initial state on a mount operation, and all further actions on the file system depend on this initial state. As a result, file systems constantly write back their in-memory changes to disk and also use additional mechanisms (such as checksums) to ensure the correctness of file system objects on disk [23, 198].

The on-disk format of file system objects differs from one file system to the next. Despite these differences, the functionality and use of such objects remain the same across file systems. To give an overview of the on-disk objects of file systems, we will briefly discuss the on-disk state of the ext2/3 file system.

Figure 2.5 shows the ext2/3 on-disk layout. In this on-disk organization (based loosely on FFS [114]), the disk space is split into a number of block groups; within each block group are bitmaps, an inode table, and data blocks. Each block group also contains a redundant copy of crucial file-system control information such as the superblock and the group descriptors.

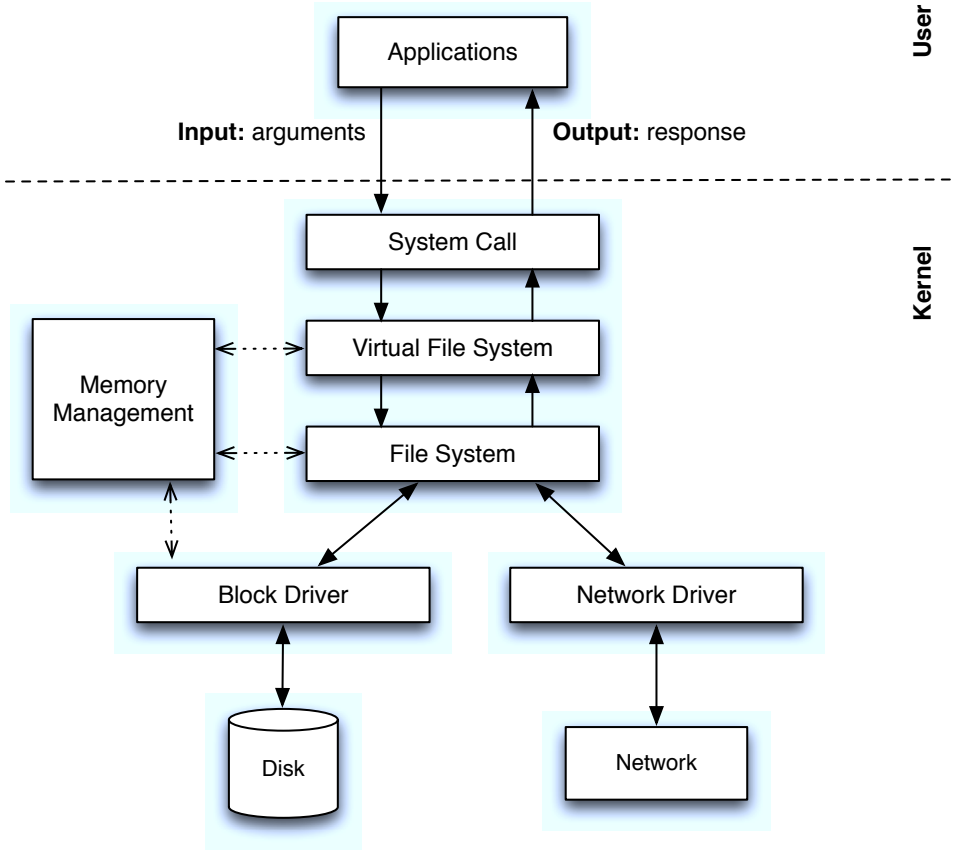

Figure 2.4: **Processing Requests in File Systems.** *The figure presents a simplified version of a storage stack and shows how requests flows through different layers in the operating system to execute a file-system operation. Requests enter through the system call layer and after execution returns to applications with the response. The dotted lines represent call(s) to the memory-management component by different layers while executing a file-system request.*

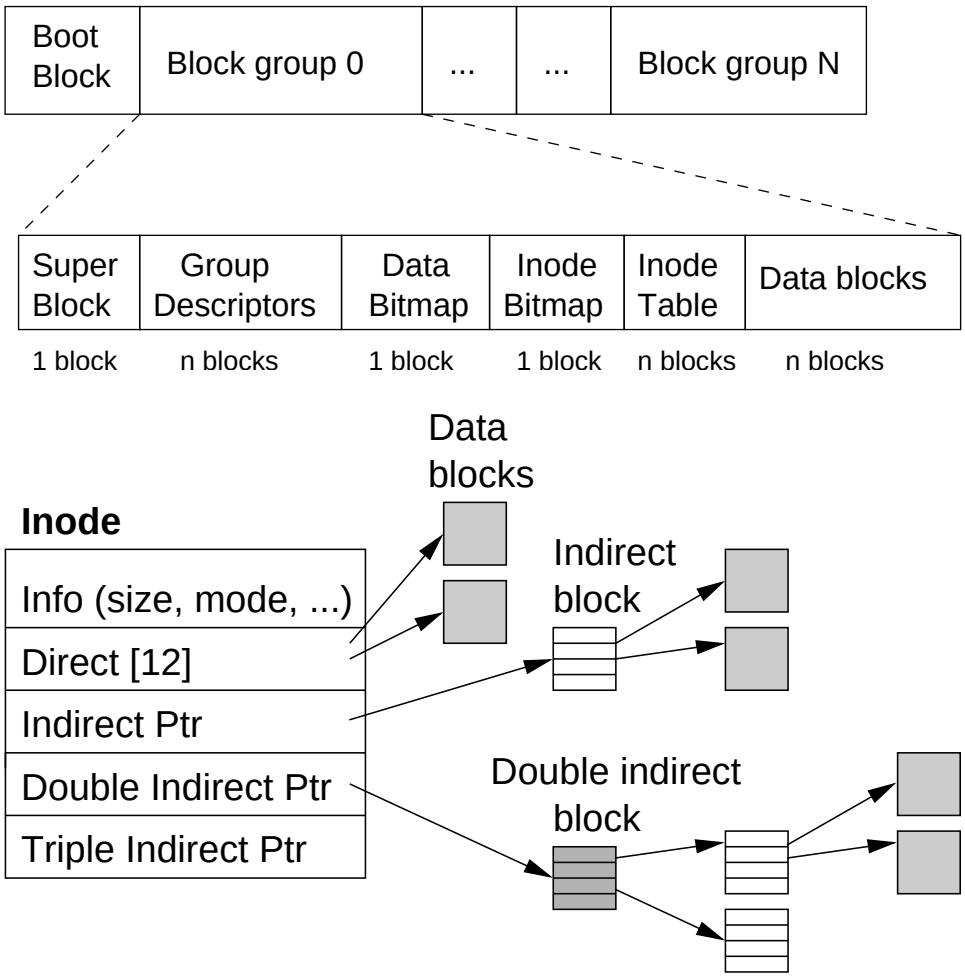

Figure 2.5: **Ext2/3 Ondisk Layout.** *The figure shows the layout of an ext2/3 file system. The disk address space is broken down into a series of block groups (similar to FFS cylinder groups), each of which is described by a group descriptor and has bitmaps to track allocations and regions for inodes and data blocks. The lower figure shows the organization of an inode. An ext2/3 inode has some attributes and twelve direct pointers to data blocks. If the file is large, indirect pointers are used.*

The super block is the most important block and is used to bootstrap the ext2/3 file system. The superblock contains important layout information such as inodes count, blocks count, and how the block groups are laid out. Without the information in the superblock, the file system cannot be mounted properly and would be unusable.

Group descriptor blocks maintain the summary for the block group they represent. Each one contains information about the location of the inode table, block bitmap, and inode bitmap for the corresponding group. In addition, each group descriptor also keeps track of allocation information such as the number of free blocks, free inodes, and used directories in the group.

Inodes contain necessary information to locate file and directory data on disk. An inode table consists of an array of inodes, and it can span multiple blocks. An inode can represent a user file, a directory, or other special files (e.g., symbolic link). An inode primarily stores file attributes (e.g., size, access control list) and pointers to its data blocks. An ext2/3 inode has 12 direct pointers to its data blocks. If more blocks are required (to hold a larger file), the inode will use its indirect pointer that points to an indirect block which contains pointers to data blocks. If the indirect block is not enough, the inode will use a double indirect block which contains pointers to indirect blocks. At most, an ext2/3 inode can use a triple indirect block which contains pointers to double indirect blocks.

A data block can contain user data or directory entries. If an inode represents a user file, its data blocks contain user data. If an inode represents a directory, its data blocks contain directory entries. Directory entries are managed as linked lists of variable length entries. Each directory entry contains the inode number, the entry length, the file name and its length.

2.5 Consistency

A file system is said to be in a consistent state if the following conditions are met. First, all of its metadata must correctly point to their respective metadata and data blocks; for example, in the ext2/3 file system, an inode block should correctly point to the indirect and direct blocks that belong to that particular file. Second, the internal set of tables and bitmaps (i.e., file-system-specific metadata) that are used for various pieces of bookkeeping information should match the count or locations of the allocated on-disk objects. For example, in the ext2/3 file system, the summary information in the group descriptor should match the actual utilization and locations of the objects within that group. At a high level, a consistent file-system state ensures that the semantics of the file-system state is respected, which also includes

proper security and privacy attributes on users' data.

The file system state can become inconsistent due to a variety of reasons. For example, power failures can result in only partially updated file-system state on a disk [115], and incorrect behavior can in wrong updates [138]. Another reason can be hardware failures that lead to file-system updates being lost [61, 173]. Finally, bugs in file and operating-system code can also result in an inconsistent state [49].

A consistent on-disk state is critical for the correct operation of a file system. File systems are stateful; they use their on-disk state to bootstrap the file system during the mount operation and depend on this boot-strapped state to perform subsequent operations. If the on-disk state is inconsistent, the file system will always remain inconsistent, as the same state is observed across reboots. In extreme cases, file systems can perform incorrect operations, which have the potential to lead to catastrophic results (e.g., data loss, data corruption, unusable file systems, or unrecoverable file systems) [15, 70, 196, 197].

The file system state can be fixed using a consistency checker (such as fsck) [19, 76, 99, 115]. Consistency checkers are offline tools or programs that read the on-disk file-system state, validate the contents, and fix any inconsistencies that might exist. Although the repair might lose some file-system updates, there is an attempt to ensure that the fixed file system state is consistent [70]. The main drawback of such consistency checkers is that they are very slow and their scanning and repairing of the on-disk state of file systems can take hours or even days [76–78].

Modern file systems have different crash-consistency mechanisms to help avoid file system inconsistencies due to power failures. The popular crash-consistency techniques that exist today are journaling [18, 113, 170, 183] and snapshotting [82, 191, 198]. These crash-consistency mechanisms are orthogonal to fsck, where the file system records additional information during regular operations to help restore it to a consistent state on reboots after power failures.

2.5.1 Journaling

In journaling file systems, extra information is recorded on the disk in the form of a write-ahead log or a journal [67]. The common approach taken by journaling file systems is to group multiple file-system updates into a single transaction and atomically commit the updates to the journal. File system updates are typically pinned in memory until the journal records are safely committed to stable storage. By forcing journal updates to disk before updating complex file system structures, this write-ahead logging technique enables efficient crash recovery; a simple scan of the journal and a redo of any incomplete committed operations bring the file system to a consistent state. During normal operation, the journal is treated as a

circular buffer; once the necessary information has been propagated to its fixed location structures, journal space can be reclaimed.

Different modes of journaling exist in modern file systems. These modes provide different recovery guarantees to the stored file-system metadata and data. In some cases, a file system support multiple journaling modes, where users have the freedom to choose their journaling mode depending on their requirements. The common modes in journaling are writeback, ordered, and data.

In writeback mode, only updates to file system metadata are journaled; data blocks are written directly to their fixed location. The writeback mode does not enforce any ordering between the journal and fixed-location data writes, and as a result, writeback mode has the weakest integrity and consistency semantics of the three modes. File-system metadata and data could be out of sync after crash recovery. Although it guarantees integrity and consistency for file system metadata, it does not provide any corresponding guarantees to the data blocks.

In ordered journaling mode, again only metadata writes are journaled; however, data writes to their fixed location are ordered before the journal writes of the metadata. Though the data blocks are not journaled, the ordering attempts to keep file-system metadata and data to in sync after recovery. In contrast to writeback mode, the ordered mode provides better integrity semantics where a metadata block is guaranteed not to point to a block that does not belong to the file.

In data journaling mode, the file system logs both metadata and data to the journal. In this mode, typically both metadata and data will be written out to disk twice: once to the journal, and then later to their fixed location. Of the three modes, data journaling mode provides the strongest integrity and consistency guarantees. However, it has a different performance characteristics compared to the other nodes, and the performance depends heavily on the workload [137].

2.5.2 Snapshotting

Snapshotting or Copy-On-Write (COW), is an alternative technique to journaling that helps preserve consistent file-system state on crashes. Many recently-developed file systems have resorted to snapshots as their mechanism for enforcing file-system consistency during crash-recovery [23, 82, 191].

Snapshotting works on a simple principle: never overwrite existing metadata or data contents on disk. All updates to file-system data are first cached in memory and are periodically written to a separate location on disk in an atomic fashion. The idea behind this approach is that the original data contents are always preserved, and on recovery, the recent updates are visible to users if and only if the snapshot that they belong to are committed to disk.

Snapshotting works well due to its copy-on-write approach. Since snapshotting does not overwrite any existing data on disk, it always preserves file-system consistency by writing updates to a new location. Once all updates of a snapshot are written to disk, the file system atomically switches to the new snapshot. In the event of a failure, the file system simply discards (i.e., reclaims) the partially written blocks of any uncommitted snapshots and bootstraps the file system from the last consistent (i.e., committed) snapshot on disk. It is easy to see that the recovery through snapshots could be lossy; recent updates that are part of an uncommitted snapshot are simply discarded during recovery.

In summary, these crash-consistency approaches always ensure that file system can be restored to the most recent consistent state using the associated meta-data maintained by them. The recovered file-system state is guaranteed to be consistent if there are no errors or bugs in the underlying storage system [138]. The disadvantages of these approaches are that they do not recover all file-system updates before a crash, but only until the last committed checkpoint (or transaction). Moreover, crash-consistency mechanisms are specialized for a particular file system and are not generic enough to be applied to a variety of file systems.

2.6 Deployment Types

File systems can be deployed in two ways: inside the kernel or as a user-level process. We will now discuss both types of deployments.

2.6.1 Kernel-level File Systems

File systems that are deployed and executed as part of an operating system are knows as kernel-level file systems. Kernel-level file systems are widely deployed and used in real systems [61, 83, 87]. Kernel-level file systems run in the same address space as the operating system and directly manage the data stored on disks or across the network. Examples of kernel-level file systems are ext3 [183], xfs [170], FAT32 [116], NTFS [158], and HFS [6].

Kernel-level file systems come with many advantages. They can leverage the design and features of the operating system to the full extent, as they are tightly integrated with the operating system code. These file systems can effectively control when data gets written to the disk or sent across the network, as they have direct access to operating system components, such as block and network drivers. Moreover, these file systems also have better control over the cached in-memory data. The combination of tight integration with the operating system, direct access to operating system components, and control over cached data helps provide better

performance in kernel-level file systems. Kernel-level file systems are also considered secure, as it is difficult for a malicious user to change the file-system code or manipulate its in-memory objects.

Though kernel-level file systems are widely used in real systems, they do have a few limitations. First, it is hard to port kernel-level file systems to other operating systems. Second, kernel-level file systems have longer development cycles [199]. Third, a bug in the kernel-level file system code has the potential to easily bring down the entire operating system, as kernel- level file systems run in the same address space as the operating system. Fourth, one needs skilled programmers with in-depth operating-system knowledge to design and develop these file systems. Finally, it is difficult to add all user-desired features to these kernel-level file systems.

2.6.2 User-level File Systems

File systems that are deployed and executed in user space are known as user-level file systems. User-level file systems provide better flexibility in terms of the features that they offer. Moreover, these file systems are completely isolated from the operating system and are run as regular processes with no special privileges. Examples of user-level file systems are SSHFS [163], NTFS-3g [181], AVFS [159], and HTTPFS [161].

User-level file systems have many advantages. They are relatively easy to develop and deploy, as most of them are only a few thousand lines of code. They can easily be ported to other operating systems with little to no effort. They can provide specialized functionality on top of existing kernel-level file systems. These file systems can be developed by a regular programmer and do not require a highly skilled developer with a deep understanding of operating system components. User-level file-system failures no longer impact the availability, correctness, and consistency of the entire operating system, as user-level file systems are completely isolated from the operating system.

Though user-level file systems are simpler than kernel-level file systems, they are not widely deployed for the following reasons. First, they are less secure than kernel-level file systems, as they run in user space. Second, the performance overheads are higher than kernel-level file systems due to the additional copying of data across the file-system-kernel boundary and context switches in the operating system [141]. Finally, they do not have control over the quantity of dirty data that gets written to the disk, nor do they have control over when dirty data gets written to the disk; hence, they cannot provide good crash-consistency guarantees.

2.7 Summary

In this chapter, we provided a brief background on how file systems manage user data on a disk. We then gave an overview of namespace management, data management, consistency management, and recovery in file systems along with their interactions to persist file-system changes to disk. We then described the different on-disk objects in file systems along with the need for consistency of the on-disk state. We concluded the chapter by giving an overview of kernel-level and user-level file systems. In the following chapters we will explore different solutions to tolerate failures in file systems.

Chapter 3

Reliability through Restartability

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

It is difficult for a file system to recover from a failure, as file systems maintain a significant amount of in-memory state, on-disk state, and OS state. Upon a file system failure, the common recovery solution is to crash and restart the affected file system. Such crash-restart recovery mechanisms may require an entire operating system reboot, manual restart of the file system followed by a consistency check (fsck) of the file system, or both. This process of recovering from a file-system failure through an explicit restart is slow and applications can no longer use the crashed file system and hence are forcefully killed.

A popular way to improve reliability is to restart systems on failures [31, 174], with the goal being to selectively restart a particular component (or a sub-component) on failures. Such selective restarts can potentially help hide file-system failures from applications and other operating system components. Moreover, selective restarts allow for applications to survive file system crashes.

Recent research, such as EROS and CuriOS, has proposed solutions to tolerating file system bugs through stateful restart mechanisms [42, 156]. Unfortunately, these solutions have required complete redesign and rewrite of both OS and file system code. Moreover, solutions that require extensive code restructuring are not viable for commodity operating systems and file systems, as extensive code changes take a long time to become adopted in the mainline kernel. This is attributed to the fact that extensive changes tend to reduce the stability – and hence the reliability – of the system.

In this chapter, we explore the possibility of creating generic frameworks in order to restart both kernel- and user-level file systems. The intuition behind our

approach is that we do not want to customize the restart process to suit a particular file system. Instead, our intention is to create a generic restart mechanism that is suited for most file systems. We believe that because this would more likely require a one-time change to the operating system code, it could easily be leveraged by those file systems that run on it.

The rest of this chapter is organized as follows. First, in Section 3.1, we explain the fault space and failure model in file systems. Section 3.2 gives an overview of the restart process in file systems. Finally, in Section 3.3, we look at the three important components of a restartable framework for file systems: detection, anticipation, and recovery.

3.1 Failure Model

File systems can fail in a variety of ways [90, 138, 173]. Failure in a file system impacts its availability. A failure may be caused by developer mistakes, an incomplete implementation (such as missing or improper error handling), or a variety of other issues. When a failure occurs, a file system becomes unavailable until it is restarted. It is important to understand how systems fail to help determine the tradeoffs between performance and reliability in the framework designed for restartable file systems.

Before we present the failure model, we first define the common terms used in this research and provide a taxonomy of faults. We then present our system model, behavior of systems on failures, failure occurrence pattern, and operating system response to a failure. Finally, we present our approach to handling file system failure.

3.1.1 Definitions

- **System:** An entity that interacts with other entities. Entity may refer to hardware, software, a human, etc.
- **System Boundary:** The common frontier between the system and its environment.
- **Fault:** A fault is a flaw in the software or hardware.
- **Error:** In a system, any deviation from the system's correct state is defined as an error. The correct state of a system is defined as the state that is achieved when the system's functionality is implemented correctly. Alternatively, we

can define error as a condition that occurs when a fault is executed or activated and the system state is corrupted.

- **Failure:** We define failure as an event that occurs when the observable output of a system deviates from its correct state. Alternatively, a failure is said to have happened if the error causes the system to behave incorrectly.

3.1.2 Taxonomy of Faults

All faults that can affect a system when activated are classified into eight fundamental types, which are shown in Figure 3.1. Faults need not necessarily be restricted within these eight classes; combinations of faults from different categories are also possible.

Most relevant to file and storage systems are the following faults categories: system boundary, dimension, persistence, capability, and occurrence phase [12, 49, 95, 128, 138, 173, 176].

3.1.3 System Model

In the context of our research, the system is a file system. The system boundary is the file system interface that file systems use to interact with other components. Our goal is not to attempt to handle failures outside the file systems, but rather, to improve the fault-tolerance of file systems and therefore focus on failures inside file systems. Faults occur either within or outside of the file system, but in this research, we assume that the consequence of a fault results in a file-system failure.

Figure 3.2 shows a system model in user-level and kernel-level file systems. For kernel-level file systems, we assume that only the file system state is affected by the fault, and that the failure is isolated within the file system; we trust that the data that is available in the other operating system components (such as memory management, virtual file system, block layer, etc.) will be able to recover the crashed file system. For user-level file systems, we assume that fault results in a file-system failure and affects the file system state; all the other components (i.e., the operating system, FUSE, and any remote host) work correctly after a user-level file system failure.

3.1.4 Behavior of Systems on Failures

The failure behavior of a component determines how a system contains and detects faults. For example, the operating system detects user-level file system crashes

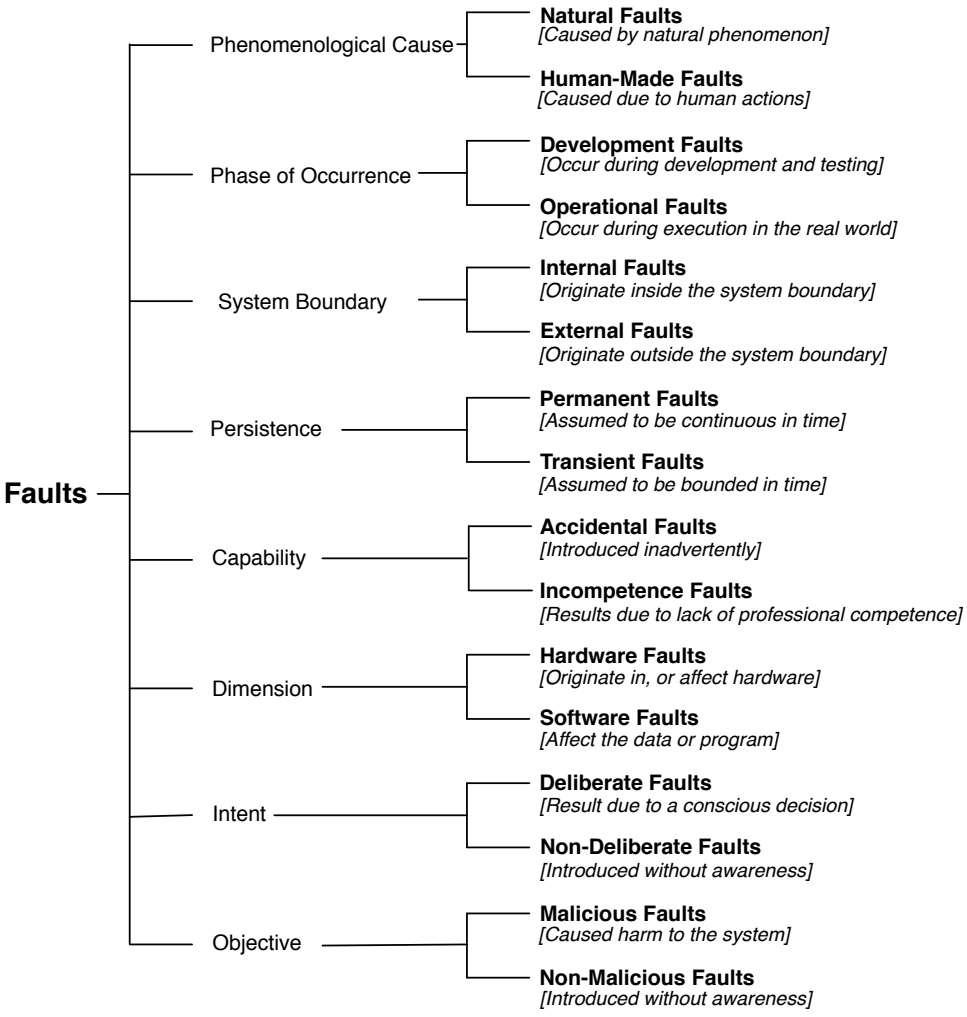

Figure 3.1: **Basic Fault Classes.** *The figure shows the elementary fault classes in systems. The first level in the hierarchy shows the base fault classes and the second level shows further classification within the base classes.*

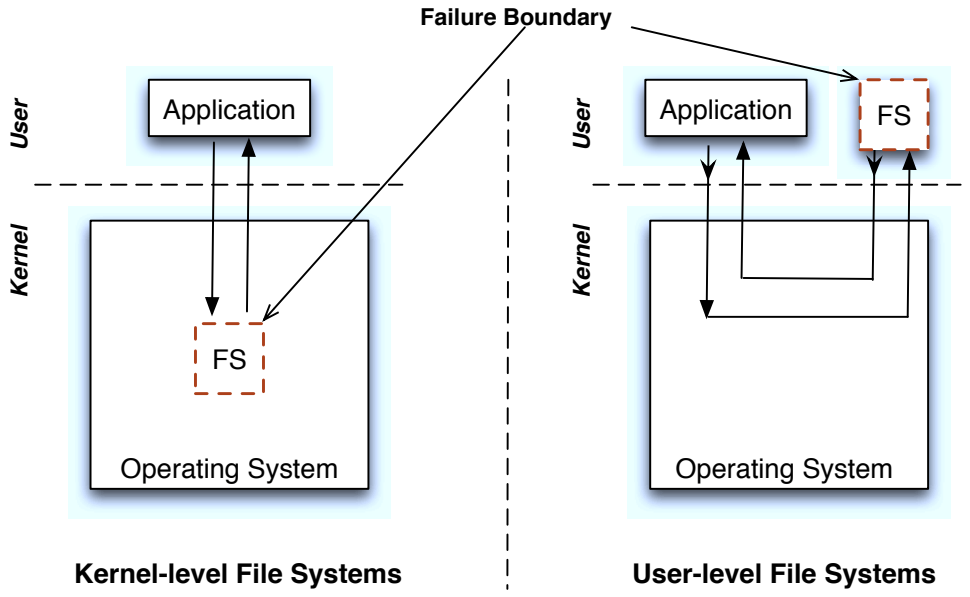

Figure 3.2: **System Model.** *The figure shows the deployment of user-level and kernel-level file systems. The system consists of the operating system and the file system. In our model, we assume that faults could occur within or outside the file system but failures only happen inside file systems.*

through process termination. Failures are categorized into five different groupings based on failure behavior [41]:

- **Omission Failures:** which occur when a system fails to produce an expected output.
- **Crash Failures:** which occur when the system stops producing any output.
- **Arbitrary Failures:** which occur when some or all of the system users perceive incorrect service (or output).
- **Response Failures:** which occur when a system outputs an incorrect value; and
- **Timing Failures:** which occur when a system violates timing constraints.

Each failure category could require its own detection and isolation mechanism, with such mechanisms also requiring additional support in terms of software and hardware implementation.

3.1.5 Occurrence Pattern

The activation of a fault that causes a failure determines the possible strategy for tolerating that failure. A failure caused by a fault that is activated by a regularly occurring event is defined as a *deterministic* failure. Since failures are deterministic, the recovery strategy should ensure that the triggering event does not occur again on recovery. In the context of file systems, deterministic failures can occur due to a variety of reasons: hardware failures [12, 138], corrupt file-systems, or operating-system state [13, 14].

A failure caused by an irregularly occurring fault is defined as a *transient failure*. Transient failures are usually triggered by a combination of inputs, such as request interleaving or rarely-occurring environment conditions [66]. In the context of file systems, transient failures may be caused by faulty SCSI back plane, cables, or Ethernet cables [173]. These faults typically do not occur on subsequent requests or retries. Empirical evidence also suggests that many failures are transient in nature [65].

3.1.6 Operating System Response to a Failure

When a file system (or a component) failure occurs, the operating system may translate the failure from one category to another. The translation of failure also depends on the failure policy described in the file system. For example, Linux either crashes or remounts the file system as read-only when it detects file-system failures. Moreover, a file system could first attempt to fix the failure by itself. For example, ZFS attempts to fix disk corruption by using redundant copies of the data stored in other disks. Failure translation also helps in simplifying the failure handling policy. A higher-level system – such as an operating system – must only handle a single category of component failure rather than manage the variety of component failures that may occur.

Many fault-tolerant systems, such as Phoenix and Hive, perform failure translation in order to simplify recovery [33, 54]. These systems, on detection of a failure in a component, halt the component as a means of preventing further corruption. In order to halt a component (i.e., translate to a fail-stop failure), a system must meet the following three conditions that are defined in the fail-stop processor model [193]:

- **Halt on failure:** the system halts the component before performing an erroneous state transformation.

- **Failure status:** the failure of a component can be detected.

- **Stable storage:** the component state can be separated into volatile storage, which is lost after a failure, and stable storage, which is preserved and unaffected by the failure.

The fail-stop model of handling failure simplifies the design of reliable systems. The system and its components deal only with correct and incorrect states and not with other failure modes. In other words, the simplicity in design is achieved through the ability to differentiate between correct and incorrect states, and to partition memory into correct and corrupted storage.

The drawback of the fail-stop model is that it is difficult to achieve in commodity operating systems. Commodity operating systems typically run their components in the operating system address space, allowing components to spread their state throughout the operating system. To transform component failures into fail-stop faults, the operating system might require additional support, such as address-space isolation [171].

3.1.7 Our Approach

In designing our restartable systems, we make several assumptions about file system failures. First, we assume that most file system failures are crash failures and response failures. Furthermore, we assume that these failures are fail-stop and can be detected and stopped before the kernel or other components are corrupted. When possible, we also add mechanisms that can transform detected faults in file systems in order to fail-stop failures.

Second, we also assume that most file system failures are transient. Thus, a possible recovery solution would be to restart the file system and to retry requests in progress at the time of failure, as the failure is unlikely to occur again. We believe that this is a reasonable solution as most of the failures in the real world. Including storage system failures, are transient in nature [66, 173]. Although we assume transient failures, we attempt to handle deterministic failures if they occur in the context of a file system request.

Our goal is not to handle malicious faults, natural faults, human-made faults, development faults, incompetence faults, and deliberate faults. For example, avoiding development faults (such as logic errors) is critical for the correct operation of

the file system; we believe that such bugs should (and likely will) be detected and eliminated during development and testing.

3.2 Restarting a file system

Restarting a crashed file system is not straightforward. File systems have a great deal of state spread across memory and on disk. Moreover, applications also have some in-memory state (such as file descriptors) that are associated with a particular file system. Additional care must be taken to restore all such state after a file-system restart.

In this section, we first describe the goals of restartable frameworks designed for user- and kernel-level file systems. We then describe states associated with a running file system. Finally, we describe the different ways of restarting a crashed file system.

3.2.1 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 restartable file systems, we explicitly make the choice to favor performance, transparency, and generality over the ability to handle a wider range of faults.

In this thesis, we investigate the following questions:

- Is it possible to implement a generic framework to restart file systems?
- Whether a light-weight solution is sufficient?
- How transparent is the restart process to applications?
- How many modifications are required to transform commodity file systems to work with a framework that support restartability?
- Can a restartable framework respect file-system-consistency guarantees?

3.2.2 State Associated with File Systems

File systems have a great deal of state spread across different components. The number of components depend on the file system deployment. File systems running inside the operating system (i.e., kernel-level file systems) have their state spread across other operating system components, and also they share the same address space as the operating system. File systems running in user space have most of their state spread across their own address space and some of their state is spread in the underlying FUSE and storage systems (such as disks or networks).

The state that needs to be restored on a file system restart are application-specific state, in-memory file-system system, on-disk file-system state, and operating-system state. Restoring all of the above-mentioned states after a failure makes restarting file systems challenging. We now discuss the file-system-associated states in detail.

Application-specific state

An application (or a process) using the file system has state inside the operating system that is specific to the file or directory it is using. The state typically associated with an application are file handles, file positions, file locks, and callbacks. All of the above mentioned state are typically maintained by the virtual file system or an equivalent layer inside the operating system [100]. A file handle, as the name suggests, serves as a unique identifier to access a file that has been previously opened by the application. A file position is an index to a file's data and is updated at the end of a read or write operation. A file lock is used to provide atomicity for updated by user-level processes. Finally, a callback (such as inotify) helps utilities (such as desktop search) to easily identify modified or updated files or directories [112]. All such state needs to be tracked and restored after a restart.

Operating-system state

File systems leverage operating system components (such as memory management, network, block layer, and virtual file system) to execute file-system requests to completion. For example, during the execution of a file-system request, in-memory objects in the operating system could be created or updated, or a lock could have been acquired or released. In the event of a file-system failure, one needs to ensure that changes done in the operating system by partially completed requests are properly cleaned up. The cleanup will help ensure that the operating system is restored to an consistent state, which would allow it to service subsequent requests.

In-memory File-system State

The in-memory state of the file system mainly consist of three components. First, application-specific file-system state gets created in the process of executing application requests. A good example of such a state is a file object. Second, file systems cache recently accessed data from the disk to improve overall performance. Such cached in-memory state need not be restored as they would be recreated again during subsequent access. In other words, since read-only data does not affect the correctness of the file-system state, it is not necessary to restore all of the cached in-memory state. Finally, file systems also maintain their own metadata (such as bitmaps, extents, etc.) in memory. Only the dirty metadata that have not yet written back to the disk need to be restored, as in-memory copies of metadata are also persistently stored on disks.

On-Disk File-system State

Changes to the file-system state need to be persistently written to the disk. The persistent writes help ensure that the modified or updated data can survive across operating system reboots. The on-disk state of file systems consists of meta-data specific to the file system (described in Section 2.4) and data written by applications. The file-system specific meta-data needs to be consistent on disk to prevent further damage to the stored data [13, 15, 138].

3.2.3 Different Ways to Restart a File System

File systems can be restarted in different ways depending on the user's needs and requirements. We characterize the restart process into three different categories depending on the recovered state. The recovered state includes file system (both in-memory and on-disk), application, and the operating-system state.

Primitive Restart

Primitive restart is the currently used restart mechanism in commodity file systems. The goal of primitive restart is to restore the file system to a consistent state after a crash. This "consistent" state need not be the state that existed at the time of the crash. In other words, some of the updates to the file system prior to a crash could be lost during the restart process.

Upon a fault in file systems, the operating system is restarted (applicable only to kernel-level file systems) and then file-system state is restored by using the recently recorded file-system state on disk. The recorded file-system state could be created using any of the crash-consistency mechanisms (see Section 2.5). In the event that a file system does not have any crash consistency mechanism, a file-system utility (such as fsck) is run to repair and recover the file system state.

The advantage of primitive restart is that is does not require any explicit support from the operating system or the file system. The drawbacks of this approach are that it is lossy, manual, and slow. The application-state is lost and applications have to be manually restarted to use the file systems. More over, for kernel-level file systems, the operating-system state is also lost and all other applications or processes are also killed and have to be manually restarted.

Stateless Restart

We define stateless restart as an automatic restart and restore of crashed file systems but not applications or the underlying operating system. Stateless restart can be

achieved if the faults or failures are isolated within the file system. In other words, the other components that the file system interacts with should not be affected by the fault. One can restore the file-system to a consistent state by simply leveraging a crash-consistency mechanism, an offline checker, or both.

Stateless restart is better than primitive restart for the following reasons. First, application or processes that are not using the file system are not affected by the file system restart. Second, the entire operating system need not be restarted (i.e., smaller down time). Third, no manual intervention is required to restart a crashed file system.

The drawback of stateless restart is that application-specific state is not restored after a file-system restart. The disconnected application state (*e.g.*, file position pointing to a non-existent file location) forces developers to handle incorrect file-system behavior inside applications. Also, like primitive restart, some of the recent file-system updates could be lost. The magnitude of loss depends on the time of the crash and the parameters defined inside the crash consistency mechanism.

Stateless restart is difficult to implement for kernel-level file systems and is easier for user-level file systems. Implementing stateless restart is a bit tricky for kernel-level file systems, as they have state that is spread across operating system components. Moreover, we still require sophisticated techniques to isolate failures within file systems to correctly restart and restore back its state without corrupting the underlying operating-system state. Stateless restart can be easily achieved in the case of user-level file systems as they do not have any explicit state inside the operating system.

Stateful Restart

Stateful restart, as the name indicates, restarts and restores the state associated with applications, file systems, and the operating system on a file-system failure. The restored state is closer (or equivalent) to the state that existed prior to the file-system crash. The idea behind stateful restart is to recover the file system in a way that applications and other operating system components are oblivious to file system failures and restarts. Also, after a restart, file systems can continue servicing both pending and new requests.

There are many advantages of stateful restart. First, applications can be made oblivious to file-system failures. Second, the services running inside the operating system, including those that depend on file systems, can continue to work correctly. Third, the downtime on faults could be minimized to a large extent. Fourth, no user intervention is required to restart the application or file systems after a failure.

The major drawback of stateful restart is that one needs to track many updates

(or states) across file systems and the operating system in order to correctly restore the file system to the state it was in before the crash. Moreover, one also needs to ensure that the side effects of all in-flight requests are correctly undone; this undo process allows in-flight requests to be re-executed again.

Our goal while designing restartable frameworks is to support stateful restart of file systems. As mentioned earlier, stateful restarts enable applications, file systems, and the operating system to gracefully tolerate a wide variety of file-system failures. In most cases, when we have perfect recovery, applications can continue to use file systems even after a crash. In other cases, file system can continue to service new request that arrive after the crash. Either way, file system reliability is significantly improved in comparison with the existing commodity file systems.

3.3 Components of a Restartable Framework

A framework that provides restartability for stateful systems needs to first identify faults when they occur, continually record system state in preparation for a failure, and recover systems when faults happen. Thus, fault detection, fault anticipation, and fault recovery are the three fundamental components of a framework for restartable file systems. Functionality of the three components are common across kernel- and user-level file systems but their implementations could significantly differ depending on the system (or subsystem) that they interact with.

3.3.1 Fault Detection

The fault-detection component is responsible for identifying occurrences of faults within file systems. As seen before, faults can be detected after arbitrary periods of time. Without timely fault detection (i.e., absence of fail-stop faults), the file system or the operating system could become corrupted and become unrecoverable after a failure. Hence, the goal of fault detection is to reduce the time delay between the occurrence and detection of faults. The detection of faults can be implemented inside (such as assertions) and outside (such as hardware checks) of a file system.

Fault detection can be implemented at different granularities. In terms of granularity, a file system developer can detect faults at the level of instructions, functions, requests, or modules. At instruction-level, a file system developer can check if syntax and semantics of the operations are respected. Hardware-level checks such as segment-violation checks are a good example of instruction-level fault detection. At function-level, a file system developer can add simple checks (such as assertions) in one or more statements or can add a higher-level semantic check at the

beginning or at the end of a function. At the request-level, a file system developer can add syntactic and semantic checks at various execution points. Finally, at module-level, a file system developer detect faults by monitoring its liveliness and responses. For a flexible fault-detection mechanism, a file system developer can also use a combination of checks at different granularities.

Fault detection can also be implemented at different boundaries. Boundaries act as a natural divide between two entities. An entity could be an instruction, a function, or a module. At the instruction-level boundary, a file system developer can check the input parameters for the next instruction. At the function-level boundary, a file system developer can check the input and the return values between function calls. At the module-level boundary, we can add checks for the input and the output values. It is important to note that the above-mentioned checks could also include checks for verifying or validating the state of different objects before or after a boundary crossing.

One can further improve fault-detection techniques by adding heavyweight mechanisms. A good example of a heavyweight mechanism is running specialized file system checks after a few operations that verify both the data and the semantics of the operations. Another example is to add address-space protection [171]. Unfortunately, such checks come at a very high performance cost, and hence, are not in alignment with our goals.

We must use checks for fault detection with caution. On one extreme, too many checks could significantly slow down the system. On the other extreme, too few checks might not be able to catch many faults. There is always the trade-off between performance and reliability. Since our focus is more on anticipation and recovery, we leave the choice of implementing such checks to file system developers, who can make an informed decision based on users (or applications) needs. Ideally, we envision many lightweight checks that would improve fault detection and not impact the overall performance of the system.

3.3.2 Fault Anticipation

Anticipation in the context of file systems involves recording file-system state, along with application- and OS-specific associated state. 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.

Anticipation can be implemented at different layers and granularities depending on how much of the state needs to be restored after a fault has occurred. In terms of implementation location (i.e., layer), anticipation can be performed at the system call layer [171, 172], the file system layer [5, 82, 113, 183, 191, 198], the

block driver layer [130, 132], and virtual file system or an equivalent layer [15, 94]. In the context of granularities, we can implement anticipation at the instruction level [152, 188], the file-system request level [5, 82], and the epoch level (*i.e.*, granularity of time) [132].

The correct layer and granularity of anticipation depends on the design and deployment of the file system of interest. It is important to keep in mind that with any system that improves reliability, there is a performance and space cost to enabling recovery when a fault occurs.

3.3.3 Fault Recovery

Fault recovery is likely the most complex component of a restartable framework. Correct recovery is critical for proper operation of applications, file systems, and the OS after a fault. For stateful recovery, we need to restore the system to the state it was just before the fault. The fault recovery component is responsible for cleaning residual state, restarting file systems and restoring their state, and restoring application-specific state. We now discuss the responsibilities of the fault recovery component in detail.

Cleanup of Residual State

Cleanup of residual state is critical for correct recovery. Upon a failure, the file system, the operating system, or both could be in an inconsistent (or corrupt) state. As a result, one must first perform a cleanup before any repair is attempted on the failed system.

The goal of the cleanup process is to eliminate any residual state and restore the operating system to a consistent state. The residual state is created by in-flight requests, which could generate dirty data, create or modify existing OS objects, acquire locks, and modify on-disk contents. If requests are prematurely terminated, the corresponding cleanup actions are never run and thus could leave the system in an inconsistent or a corrupt state.

Cleanup of residual state includes the following actions. First, undo any actions (or effects) of in-flights requests that were executing at the time of the fault. Second, free up any in-memory objects of the file system; examples of such objects are bitmaps, extents, and inodes. Finally, cleanup any file-system objects that are stored in operating system components; examples of such objects are files, directory entries, and locks.

Restore and Restart Failed File System

After a fault, a file system typically becomes inconsistent, unusable, or both. Cleanup typically discards all file-system state and undoes the residual state in the OS or other components (if any). For stateful recovery, we must first restore this lost state.

It is difficult to recreate the lost state in file systems. The difficulty is caused by the fact that the file-system state is spread across many components and needs to be consistent in memory and on disk (see Section 3.2.2). To recreate the lost file-system state, we could leverage the state recorded by the fault-anticipation component along with the state maintained in other components that the file system interacts with.

One must then restart the file system to service pending and new requests. A file-system restart mainly consists of reinitialization of the failed file system. Typically, this reinitialization would be a mount (or an equivalent) operation along with some repair of operating-system state.

Restore Application-specific State

The connection between the application-specific state (such as file descriptors) and the actual file-system state is lost on a file-system restart. The cleanup and restore process discards and recreates file system objects, but, applications still have pointers to the old file-system objects, which could result in wrong (or faulty) behavior.

To restore the application-specific state, the recovery component should reattach the application-specific metadata in the OS with the newly created file-system objects. The application-specific state typically includes the open files (i.e., file descriptors), file positions, locks, and handles registered with notification daemons (such as inotify and dnotify).

3.4 Summary

In this chapter, we first discussed the fault space, and described our failure model. Then, we described the restart process in file systems, goals of our restartable framework, and different ways to restart a file system. Finally we described the three components of a restartable framework: fault detection, fault anticipation, and fault recovery.

Chapter 4

Restartable Kernel-level File Systems

> "*The best performance improvement is the transition from the nonworking state to the working state.*" – John Osterhout

File systems have traditionally been built inside the OS. Modern operating systems support a variety of file systems. For example, Linux supports 30 or so different block-based file systems. These file systems differ from each other in terms of the features, performance, and reliability guarantees they provide.

Kernel-level file systems come with both advantages and disadvantages. The primary advantage of using a kernel-level file system is that it eliminates the additional data copying and context switching costs that are associated with user-level file systems. The main drawback with kernel-level file systems is that they spread their state across other OS components and run in the same address space as the OS.

The design of kernel-level file systems makes recovery difficult to implement. When a fault occurs inside a file system, it is hard to isolate the fault within the file system, and restore the OS to a consistent state by undoing the actions of the file system. As a result, the recovery solution that exists today is to simply crash the entire OS and hope that the problem goes away on a reboot. Though this is practical, we believe it is not acceptable as applications and other services running in the OS are forcefully killed, making them unavailable to users.

In this chapter, we explore the possibility of implementing a generic framework inside OS to restart kernel-level file systems. Such a generic framework helps eliminate the need for an entire OS reboot and a tailored solution to restart individual file systems on crashes. Moreover, as performance is critical for file systems,

we also explore the possibility of minimizing the reliability tradeoffs while still maintaining similar performance characteristics of vanilla file systems.

Our solution to restarting file systems is *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.

The rest of this chapter is organized as follows. Sections 4.1 and 4.2 present the design and implementation, respectively, of Membrane. Section 4.3 discuss the consequence of having Membrane in the operating system; finally, we evaluate Membrane in Section 4.4.

4.1 Design

Membrane is designed to transparently restart the affected kernel-level 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 give an overview of our solution. We then present the three major pieces of the Membrane system: fault detection, fault anticipation, and recovery.

4.1.1 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 operations,

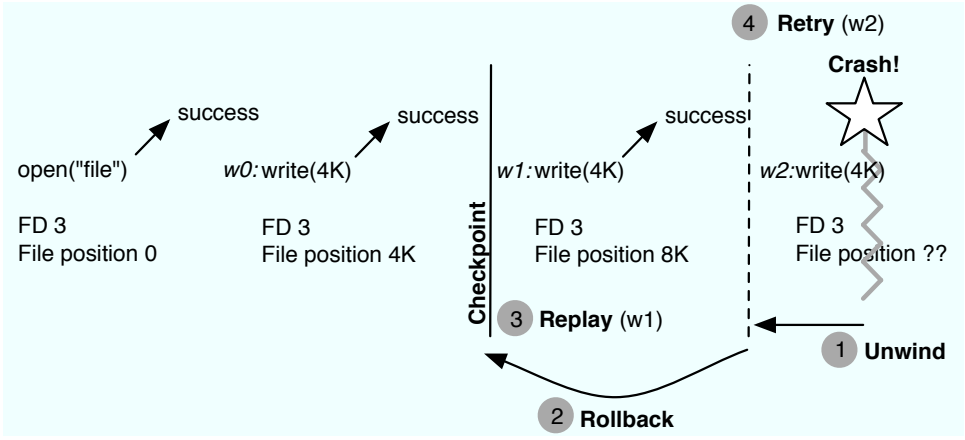

Figure 4.1: **Membrane Overview.** *The figure shows a file being created and written to on top of a restartable file system. Halfway through, Membrane creates a checkpoint. After the checkpoint, the application continues to write to the file; the first succeeds (and returns success to the application) and the program issues another write, which leads to a file system crash. Steps 1 to 4 denoted by gray circles indicate the sequence of operation that Membrane performs to restart the file system after a crash.*

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 4.1 gives an example of how Membrane works during normal file-system operation and upon a file system crash.

From the figure, we can see that Membrane creates a checkpoint after the open and the first write operation (w0) on the file system. After the checkpoint, the second write operation (w1) also successful completes. But the next write operation (w2) causes the file system to crash. Upon a 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.

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.

4.1.2 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 [200] 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[102]) could be used to generate many of these wrappers automatically.

4.1.3 Fault Anticipation

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.

Fault anticipation is difficult due to the complex interactions between the file system and the core kernel services. 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). In order to selectively restart the crashed file system and restore the operating-system state, careful checkpointing and state-tracking are thus required. We now discuss our checkpointing and state-tracking mechanisms in detail.

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 consistent 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 [73, 145, 170, 179]); others define clear consistency intervals and create snapshots (as in shadow-paging file systems [23, 82, 149]). 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 [178] 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 check-

point 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 modification.

Tracking State with Logs and Stacks

Membrane must track changes to file system state that transpired after the last checkpoint. This tracking is accomplished with five different types of logs or stacks to track: file system operations, application-visible sessions, memory allocations, 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; memory 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* [189]; 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.

4.1.4 Fault Recovery

The *fault recovery* subsystem is 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.

4.2 Implementation

We now present the implementation of Membrane. We first present each of the main components of Membrane, and then describe the operating system (Linux) changes. 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.

4.2.1 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.

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.

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 ex-

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
nfs	0	54	0
fat	0	10	16

Table 4.1: **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).*

ample, 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 [138]. Table 4.1 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 add, we believe that operating system developers will continue to "harden" the file-system/kernel interface when Membrane gets integrated inside commodity operating systems.

4.2.2 Fault Anticipation

We now describe the fault anticipation support within the current Membrane implementation. Anticipation consists of the following techniques: page stealing, state

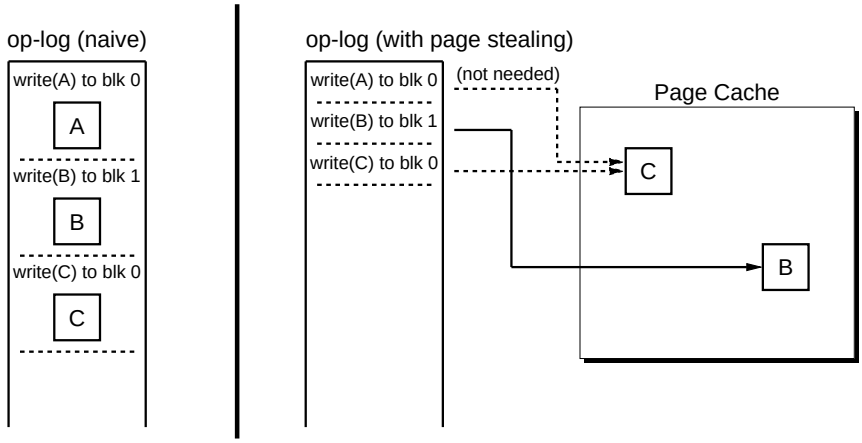

Figure 4.2: **Page Stealing.** *The figure depicts the op-log both with and without page stealing. Without page stealing (left side of the figure), user data quickly fills the log, thus exacting harsh penalties in both time and space overheads. With page stealing (right), only a reference to the in-memory page cache is recorded with each write; further, only the latest such entry is needed to replay the op-log successfully.*

tracking, and COW-based checkpointing. We begin by presenting our approach to reducing the cost of operation logging via a technique we refer to as *page stealing*.

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 4.2 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.

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 add a new allocation flag, GFP_RESTARTABLE and provide a new header file to include in file-system code to append GFP_RESTARTABLE to all memory allocation call in Membrane. This enables the memory allocation module in the kernel to transparently 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 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.

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

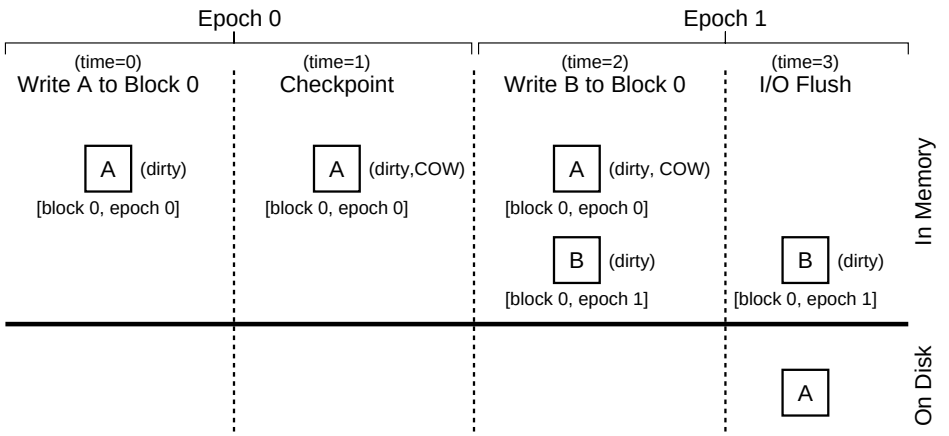

Figure 4.3: **COW-based Checkpointing.** *The picture shows what happens during COW-based checkpointing. At time=0, an application writes to block 0 of a file and fills it with the contents "A". At time=1, Membrane performs a checkpoint, which simply marks the block copy-on-write. Thus, Epoch 0 is over 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.*

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, update traffic is bunched together and must happen during the checkpoint, instead of being spread out over time; as is well known, forcing pages to disk can reduce I/O performance [117].

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 4.3 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 [133], WAFL [82] manually create and track their COW pages) or built 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.

4.2.3 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.

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.

Intertwined Execution and The Skip/Trust Unwind Protocol

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 changes 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 4.4.

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.

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, the control-flow capture mechanism in Membrane ensures that the on-disk data is never corrupted after a fault. Membrane 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.

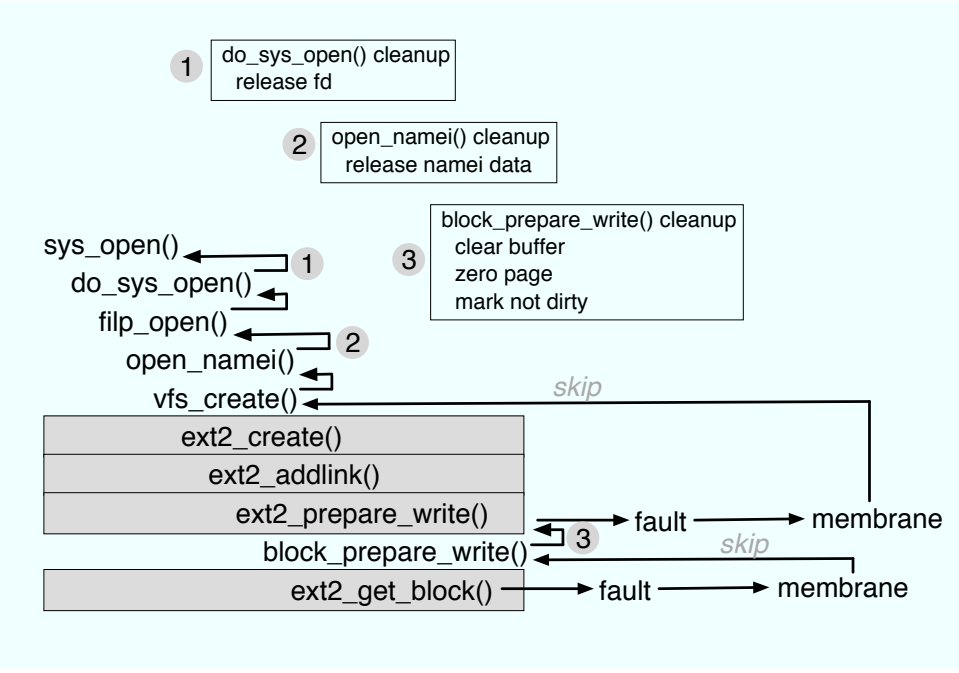

Figure 4.4: **The Skip/Trust Unwind Protocol.** *The figure depicts the call path from the* `open()` *system call through the ext2 file system. The first sequence of calls (through* `vfs_create()`*) are in the generic (trusted) kernel; then the (untrusted) ext2 routines are called; then ext2 calls back 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).*

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

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**

Table 4.2: **Implementation Complexity.** *The table presents the code changes required to tranform vanilla Linux 2.6.15 x86_64 kernel to support restartable file systems. 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.*

(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.

4.2.4 Implementation Statistics

Our prototype is implemented in Linux 2.6.15 kernel. Table 4.2 shows the code changes required to transform a vanilla 2.6.15 kernel into an operating system that implements Membrane. We now briefly describe the code changes in different OS components to support Membrane.

The FS changes include support for op-logging, COW-based checkpointing, and skip/trust unwind protocol. The op-logging support mainly consists of logging input arguments, return values, and lock acquisitions or releases of file-system requests. The COW-based checkpointing changes mainly consists of support for parking (or releasing) threads executing file-system operations during (or after) checkpoint and maintaining the count of requests that are currently executing inside the file system of interest. The skip/trust unwind support consist of calls to macros that records the current operating-system state before a request makes a transition from the OS into the file system.

The MM changes include support for COW-based checkpointing and page stealing mechanisms. The COW-based checkpoint changes consist of support to manage attributes of COW pages in the operating system. The page stealing support consist of tracking and freeing pages that belong to the file system before a crash.

Finally, the arch code changes correspond to COW support for kernel pages, and the module changes include support to locate file-system code pages and methods to change the page protection bits.

4.3 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 the 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 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 [98].

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 [200]), 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 [43]. The motivation to check for violations is low in current file systems since there is little recourse when a problem is detected. The ability to recover from the problem in Membrane gives greater motivation.

4.4 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 experiments.

4.4.1 Generality

We chose ext2, VFAT, and ext3 to evaluate the generality of our approach. In other words, we want to understand the effort involved in porting existing file systems to work with Membrane. ext2 and VFAT file systems were chosen for their lack of crash consistency machinery and for their completely different on-disk layout. The ext3 file system was selected for its journaling machinery that provides better crash consistency guarantees than ext2. Table 4.3 shows the code changes required in each file system.

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

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

Table 4.3: **File system code changes.** *The table presents the code changes required to transform a ext2, VFAT, and ext3 file systems in Linux 2.6.15 kernel into their restartable counterparts.*

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 deletion) 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.

4.4.2 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.

We use four different metrics to understand the impact of each fault injection experiment. The metrics used in our experiments are: how detected, application, FS:consistent, and FS:usable. How detected denotes how (or if at all) the fault was detected, and the reaction of the operating system to that fault. Application denotes the state of the application (or process) that is executing the file system request. FS:consistent denotes whether the file system was consistent after the injected fault. FS:usable denotes whether the file system was able to service subsequent requests after the injected fault.

Tables 4.4, 4.5, and 4.6 present 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 observations and conclusions.

Vanilla OS and File Systems

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 Tables 4.4, 4.5, and 4.6.

For ext2, 85% of faults triggered a kernel "oops". An oops typically indicates that something seriously went wrong within the file system. The other 15% of faults resulted in general protection error inside the OS. The file system was only consistent for 55% of the fault injection experiments. In cases where the file system was consistent, the operation triggering the fault did not internally modify any file-system state to cause an inconsistency. Finally, the file system was unusable in 80% of the fault injection experiments. In the remaining 20% of experiments, the file system was unmountable 75% of the time, even though it was still usable.

For VFAT, all of the injected faults triggered an oops and the application (i.e., process executing the file system request) was killed after the fault was triggered in the file system. Only in a few experiments (around 38%) was the file system consistent after a fault injection. Finally, the file system was usable after fault injection in 54% of experiments. In cases where the file system was usable, it was not unmountable, indicating that file-system state was corrupted and did not get correctly cleaned up after the fault-injection experiment.

For ext3, 93% of the fault-injection experiments result in a kernel oops. After every fault-injection experiment, the application was killed as the file system or the OS was unable to recover from the fault correctly. The file system was left in a consistent state only in 43% of experiements and was both usable and not unmountable in 7% of experiments.

Overall, 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*).

ext2_Function	Fault	ext2 How Detected?	ext2 Application?	ext2 FS:Consistent?	ext2 FS:Usable?	ext2+ boundary How Detected?	ext2+ boundary Application?	ext2+ boundary FS:Consistent?	ext2+ boundary FS:Usable?	ext2+ Membrane How Detected?	ext2+ Membrane Application?	ext2+ Membrane FS:Consistent?	ext2+ Membrane 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	√	√	√

Table 4.4: **Fault Study of ext2.** *The table shows the results of fault injections on the behavior of Linux ext2. 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.*

vfat_Function	Fault	vfat				vfat+ boundary				vfat+ Membrane			
		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	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	√	√	√

Table 4.5: **Fault Study of VFAT.** *The table shows the results of fault injections on the behavior of Linux VFAT. 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.*

ext3_Function	Fault	ext3 How Detected?	ext3 Application?	ext3 FS:Consistent?	ext3 FS:Usable?	ext3+ boundary How Detected?	ext3+ boundary Application?	ext3+ boundary FS:Consistent?	ext3+ boundary FS:Usable?	ext3+ Membrane How Detected?	ext3+ Membrane Application?	ext3+ Membrane FS:Consistent?	ext3+ Membrane FS:Usable?
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 4.6: **Fault Study of ext3.** *The table shows the results of fault injections on the behavior of Linux 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.*

Hardening Through Parameter Checks

Second, we analyzed the usefulness of fault detection without recovery by hardening the kernel and file-system boundary through parameter checks. The goal of this experiment is to understand if we really need Membrane to handle file system crashes or could hardening the kernel and the file-system boundary suffice. The second result column (denoted by +boundary) of Tables 4.4, 4.5, and 4.6 shows the results.

For ext2, faults were correctly detected in 25% of the fault-injection experiments. In 35% of experiments, the injected fault went undetected and later triggered an oops when the corrupted data was subsequently accessed. Moreover, in 5% and 10% of cases, the injected fault resulted in an invalid operation and triggered a protection fault, respectively. An error was correctly returned to the application in 10% of the fault-injection experiments. In 15% of cases, errors were silently discarded by the operating system and application was falsely notified that the request was successfully completed. The consistency percentages for file systems in the hardened kernel were the same as that of the vanilla OS. Finally, the file system was usable in 30% of cases; a slight improvement compared to the vanilla OS. In 50% of cases, where the file system was usable, it was unmountable.

For VFAT, in 46% of the fault-injection experiments, faults were successfully detected inside the OS. Only in 8% of cases, the faults initially went undetected, but later resulted in an oops. In 38% of cases, a failure was incorrectly propagated as a successful operation to the application. Only in one experiment (i.e., 8% of cases), an error was correctly returned back to the application. File system consistency improved to 54% compared to 38% for the vanilla OS. Finally, around 70% of cases, the file system was usable after a fault. In 44% of experiements where the file system was usable, it was not unmountable after the fault injection.

For ext3, around 36% of cases, faults were correctly detected in the OS. In 21% of the cases, an injected fault was incorrectly detected as an invalid opcode in the OS. In terms of application state, only in 21% of cases, an error was correctly returned back to the application. In 14% of cases, error was silent ignored inside the OS and a success was incorrectly propagated to the application. The file system was consistent in 86% of cases, which was much more than that of the vanilla OS running ext3 (around 43%). Finally, in 57% of cases, the file system was usable after the fault-injection experiment. Amongst the 57% of the usable file-system states, the file system was not unmountable for 50% of experiments.

Overall, 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

Benchmark	ext2	ext2+ Membrane	o/h %	VFAT	VFAT+ Membrane	o/h %	ext3	ext3+ Membrane	o/h %
Seq. read	17.8	17.8	0	17.7	17.7	0	17.8	17.8	0
Seq. write	25.5	25.7	0.8	18.5	19.4	4.9	56.3	56.3	0
Rand. read	163.2	163.5	0.2	163.5	163.6	0	163.2	163.2	0
Rand. write	20.3	20.5	1	18.9	18.9	0	65.5	65.5	0
create	34.1	34.1	0	32.4	34.0	4.9	33.9	34.3	1.2
delete	20.0	20.1	0.5	20.8	21.0	0.9	18.6	18.7	0.5

Table 4.7: **Microbenchmarks.** *This table compares the execution time (in seconds) for various benchmarks for restartable versions of ext2, VFAT, and ext3 (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.*

the file system was left inconsistent, unusable, or both.

Recovery Using Membrane

Finally, we focused on file systems surrounded by Membrane. The results of the experiments are shown in the rightmost column of Tables 4.4, 4.5, and 4.6. In all cases, for all file systems, 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.

4.4.3 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. The goal of our benchmarks is to understand the overheads of running the above-mentioned file systems on top of Membrane during regular operations (i.e., measuring anticipation costs) and during crash recovery.

Benchmark	ext2	ext2+ Membrane	o/h %	VFAT	VFAT+ Membrane	o/h %	ext3	ext3+ Membrane	o/h %
Sort	142.2	142.6	0.3	146.5	146.8	0.2	152.1	152.5	0.3
OpenSSH	28.5	28.9	1.4	30.1	30.8	2.3	28.7	29.1	1.4
PostMark	46.9	47.2	0.6	43.1	43.8	1.6	478.2	484.1	1.2

Table 4.8: **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: 3,000 files (sizes 4KB to 4MB), 60,000 transactions, and 50/50 read/append and create/delete biases.*

Regular Operations

Micro-benchmarks help analyze file-system performance for frequently performed operations in isolation. We use sequential read/write, random read/write, create, and delete operations as our micro benchmarks. These operations exercise the most frequently accessed code paths in file systems. The caption in Table 4.7 describes our micro-benchmark configuration in more detail.

We also use commonly-used macro-benchmarks to help analyze file-system performance. Specifically, we use the sort utility, Postmark [96], and OpenSSH [162]. The sort benchmark represents data-manipulation workloads, Postmark represents I/O-intensive workloads, and OpenSSH represents user-desktop workloads. Table 4.8 show the results of our macrobenchmark experiments.

From the tables, one can see that the performance overheads of our prototype for both micro- and macro-benchmarks are quite minimal; in all cases, the overheads were between 0% and 5%.

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 important as high recovery times may be noticed by applications.

The recovery time in Membrane depends on the amount of dirty data, open sessions (or file handles), and log records (i.e., completed operations after last checkpoint). Dirty data denotes the number of dirty pages of recent checkpoint that has not yet been written to the disk. Open sessions denote the number of file handles

Data (MB)	Recovery time (ms)
0	8.6
10	12.9
20	13.2
40	16.1

(a)

Open Sessions	Recovery time (ms)
0	8.6
200	11.4
400	14.6
800	22.0

(b)

Log Records	Recovery time (ms)
0	8.6
1K	15.3
10K	16.8
100K	25.2

(c)

Table 4.9: **Recovery Time.** *Tables a, b, and c show recovery time as a function of dirty pages (at checkpoint), s-log, and op-log respectively for ext2 file system. 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.*

that need to be opened and re-attached to the file descriptor table of applications after a restart. Log records denote the number of file system requests logged in the op-log that need to be re-executed from the VFS layer during recovery.

We measured the recovery time in a controlled environment by varying the amount of state kept by Membrane. To vary the amount of dirty data, we execute a write-intensive workload and forcefully create a checkpoint after the required amount of data is written to the file system. To vary the number of open sessions, we simply open files in application (i.e., through the syscall layer) and crash the file system when required number of file handles have been created. To vary the amount of log records, we run the postmark benchmark and crash the file system after the required number of log records were created. In all experiments, the two other parameters were maintained at 0.

Table 4.9 shows the result of varying the amount of dirty pages from the previous checkpoint, open sessions (i.e., s-log), and completed file system operations which are not yet part of a checkpoint (i.e., op-log) for ext2 file system. From the table, we can see that the recovery time is less than 17 ms even when the amount of dirty data was varied between 0 to 40 MB. This gives the upper bound on the time that would be needed to write back dirty data. It is important to note that in our current prototype, 40MB is the upper watermark before Membrane forcefully create a new checkpoint to limit the amount of dirty data in memory. It is also important to note that the dirty pages are not synchronously written back to the disk and hence does not incur a large overhead.

For session logs, the recovery time did not significantly vary with the number of open file handles. Recovering open file handles include performing a path lookup, creating a new file object, and associating the file object with the entry in the file

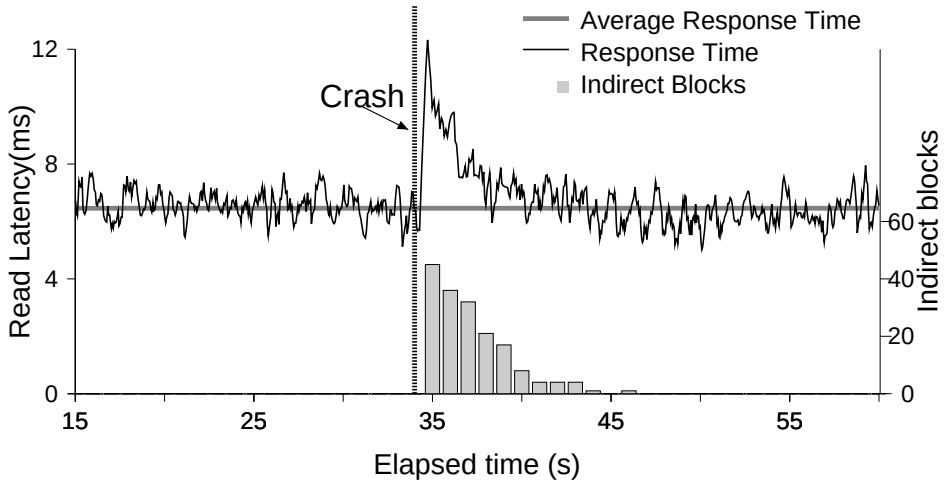

Figure 4.5: **Recovery Overhead.** *The figure shows the overhead of restarting ext2 while running random-read microbenchmark. The x axis represents the overall elapsed time of the microbenchmark in seconds. The primary y axis contains the execution time per read operation as observed by the application 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.*

descriptor table of an application. Even when the number of open file handles were varied between 0 and 800, the recovery time was still in the order of a few milliseconds.

For log records, we can see that the recovery time does not increase linearly with increase in the number of records. Log records used for this experiment consisted of lookups, reads, creates, deletes, and a few writes. One might observe larger recovery times depending on the type and number of log records that need to be replayed.

In summary, from these experiments, we found that the recovery time grows sub-linearly with the amount of state. Moreover, the recovery time is only a few milliseconds in all the cases. Hence, application need not necessarily see any significant performance drop that would be caused by the recovery process.

Figure 4.5 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 10 ms 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).

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.

4.5 Summary

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 continue to arise, but those that are rare and hard to reproduce will remain where they belong, automatically "fixed" by a system that can tolerate them.

Chapter 5

Restartable User-level File Systems

"Retrofitting reliability to an existing design is very difficult."
– Butler Lampson

File System in USEr Space (FUSE) was designed to simplify the development and deployment of file systems [192]. FUSE provides fault isolation by moving file systems out of the kernel and running them in a separate address space. FUSE also simplifies the file-system interface and minimizes the interaction with the operating system components. Nearly 200 FUSE file systems have already been implemented [160, 192], indicating that the move towards user-level file systems is significant.

Unfortunately, support for recovery in file systems using FUSE does not exist today. The current solution is to crash the user-level file system on a fault, and wait for the user (or administrator) to manually repair and restart the file system; in the meantime, FUSE returns an error to applications if they attempt to access the crashed file system. This solution is not useful when applications and users depend on these file systems to access their data.

In this chapter, we explore the possibility of implementing a generic framework inside the operating system and FUSE to restart user-level file systems. Such a generic framework helps eliminate the need for any tailored solution to restart individual user-level file systems on crashes. We also come up with a user-level file-system model that represents some of the popular user-level file systems. We believe this will help current and future file-system developers to modify and design their file systems to work with Re-FUSE.

Our solution to a generic framework is Restartable FUSE (Re-FUSE), a restartable file system layer built as an extension to the Linux FUSE user-level file system infrastructure [159]. In our solution, we add a transparent restart framework around FUSE which hides many file-system crashes from users; Re-FUSE simply restarts the file system and user applications continue unharmed.

The rest of the chapter is organized as follows. Section 5.1 presents the FUSE framework. Section 5.2 discusses the essentials of a restartable user-level file system framework. Section 5.3 presents the design and implementation of Re-FUSE.Section 5.4 evaluates the robustness and performance of Re-FUSE.

5.1 FUSE

FUSE is a framework that enables users to create and run their own file systems as user-level processes [177]. In this section, we discuss the rationale for such a framework and present its basic architecture.

5.1.1 Rationale

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

Kernel-level file-system developers may not be open to the idea of adding all of the features users want in file systems for two reasons. First, adding a new feature requires a significant amount of development and debugging effort [199]. Second, adding a new feature in a tightly coupled system (such as a file system) increases the complexity of the already-large code base. As a result, developers are likely only willing to include functionality that will be useful to the majority of users.

FUSE enables file systems to be developed and deployed at user level and thus simplifies the task of creating a new file system in a number of ways. First, programmers no longer need to have an in-depth understanding of kernel internals (e.g., memory management, VFS, and block devices). Second, programmers need not understand how these kernel modules interact with others. Third, programmers can easily debug user-level file systems using standard debugging tools such as

gdb [63] and valgrind [123]. All of these improvements combine to allow developers to focus on the features they want in a particular file system.

In addition to Linux, FUSE has been developed for FreeBSD [40], Solaris [126], and OS X [64] operating systems. Though most of our discussion revolves around the Linux version of FUSE, the issues faced herein are likely applicable to FUSE within other systems.

5.1.2 Architecture

FUSE consists of two main components: the *Kernel File-system Module* (KFM) and a user-space library *libfuse* (see Figure 5.1). The KFM acts as a pseudo file system and queues *application requests* that arrive through the VFS layer. The libfuse layer exports a simplified file-system interface that each user-level file system must implement and acts as a liaison between user-level file systems and the KFM.

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

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

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

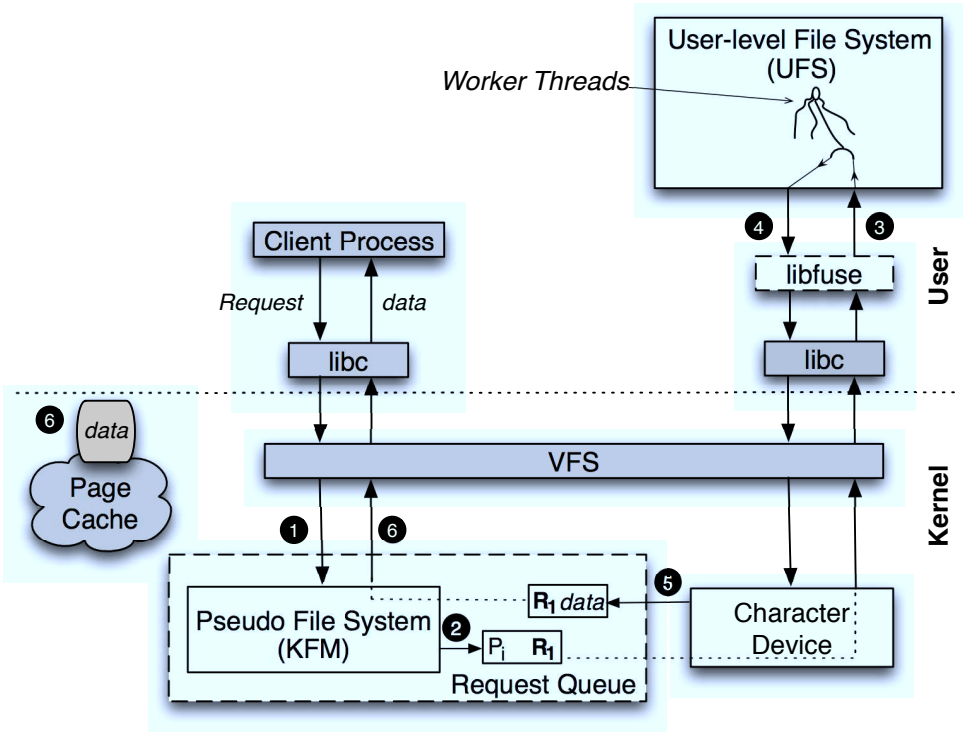

Figure 5.1: **FUSE Framework.** *The figure presents the FUSE framework. The user-level file system (in solid white box) is a server process that uses libfuse to communicate with the Kernel-level FUSE Module (KFM). The client process is the application process invoking operations on the file system. File-system requests are processed in the following way: (1) the application sends a request through the KFM via the VFS layer; (2) the request gets tagged and is put inside the request queue; (3) the user-level file-system worker thread dequeues the request; (4) the worker services the request and returns the response; (5) the response is added back to the queue; (6) finally, the KFM copies the data into the page cache before returning it to the application.*

5.2 User-level File Systems

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

5.2.1 The User-level File-System Model

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

It is infeasible to examine all FUSE file systems to obtain the "perfect" reference model. Thus, to derive a reference model, we instead analyze six diverse and popular file systems. Table 5.1 presents details on each of the six file systems we chose to study. NTFS-3g (2009.4.4) and ext2fuse (0.8.1) each are kernel-like file systems "ported" to user space. AVFS (0.9.8) allows programs to look inside archives (such as tar and gzip) and TagFS (0.1) allows users to organize documents using tags inside existing file systems. Finally, SSHFS (2.2) and HTTPFS (2.06) allow users to mount remote file systems or websites through the SSH and HTTP protocols, respectively. We now discuss the properties of the reference file-system model.

- **Simple Threading Model:** A single worker thread is responsible for processing a file-system request from start to finish, and only works on a single request at any given time. Amongst the reference-model file systems, only NTFS-3g is single-threaded (i.e., a worker thread to service all requests) by default; the rest all operate in multi-threaded mode (i.e., multiple worker threads to concurrently process file system requests). All reference-model file systems adhere to the simple threading model.

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

- **Access Through System Calls:** Any external calls that the user-level file system needs to make are issued through the system-call interface. These requests are serviced by either the local system (e.g., the disk) or a remote

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

Table 5.1: **Reference Model File Systems.** *The table shows different aspects of the reference-model file systems. Category column indicates the underlying communication mechanisms used by user-level file systems to persist data. LOC column indicates the lines of code in each of the file system. Downloads column indicates the download count for the user-level file system mentioned in the user-level file system websites as of September 1st, 2010. Various definitions have been used to compress the description. Block-based denotes that file systems use the raw block-device interface to store data; pass-through denotes that user-level file system is running on top of a kernel-level file system and stores data in the kernel-level file system. Network-based denotes that the user-level file system uses sockets to communicate with the underlying data access mechanism to store data.*

server (e.g., a web server); in either case, system calls are made by the user-level file system in order to access such services.

- **Output Determinism:** For a given request, the user-level file system always performs the same sequence of operations. Thus, on replay of a particular request, the user-level file system outputs the same values as the original invocation [2].

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

Our reference model clearly does not describe all possible user-level file-system behaviors. The FUSE framework does not impose any rules or restrictions on how one should implement a file system; as a result, it is easy to deviate from our reference model, if one desires. We discuss this issue further at the end of Section 5.3.

5.2.2 Challenges

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

Generic Recovery Mechanism

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

Synchronized State

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

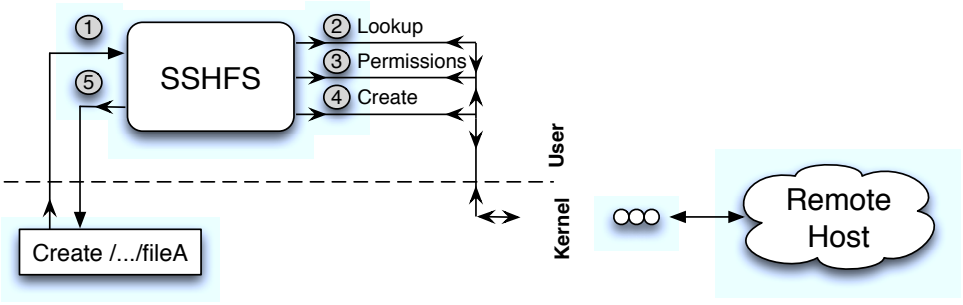

Figure 5.2: **SSHFS Create Operation.** *The figure shows a simplified version of SSHFS processing a create request. The number within the gray circle indicates the sequence of steps SSHFS performs to complete the operation. The FUSE, application process, and network components of the OS are not shown for simplicity.*

Residual State

The non-idempotent nature of system calls in user-level file systems can leave *residual state* on a crash. This residual state prevents file systems from recreating the state of partially-completed operations. Both undo or redo of partially completed operations through the user-level file system thus may not work in certain situations. The create operation in SSHFS is a good example of such an operation. Figure 5.2 shows the sequence of steps performed by SSHFS during a create request. SSHFS can crash either before file create (Step 4) or before it returns the result to the FUSE module (Step 5). Undo would incorrectly delete a file if it was already present at the remote host if the crash happened before Step 4; redo would incorrectly return an error to the application if it crashed before Step 5. Thus, any recovery mechanism for user-level file systems must properly handle residual state.

5.3 Re-FUSE: Design and Implementation

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

5.3.1 Overview

The main challenge for Re-FUSE is to restart the user-level file system without losing any updates, while also ensuring the restart activity is both lightweight and transparent. File systems are *stateful*, and as a result, both in-memory and on-disk state needs to be carefully restored after a crash.

Unlike existing solutions, Re-FUSE takes a different approach to restoring the consistency of a user-level file system after a file-system crash. After a crash, most existing systems rollback their state to a previous checkpoint and attempt to restore the state by re-executing operations from the beginning [31, 140, 165]. In contrast, Re-FUSE does not attempt to rollback to a consistent state, but rather continues forward from the inconsistent state towards a new consistent state. Re-FUSE does so by allowing partially-completed requests to continue executing from where they were stopped at the time of the crash. This action has the same effect as taking a snapshot of the user-level file system (including on-going operations) just before the crash and resuming from the snapshot during the recovery.

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

5.3.2 Fault Anticipation

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

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

Request Tagging

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

application request with the system calls performed by a user-level file system to service said application request (see Sections 5.1.2 and 5.2.1 for details).

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

Re-FUSE instruments the libfuse layer to automatically set the ID of the application request in the task structure of the file-system thread whenever it receives a request from the KFM. Re-FUSE adds an additional attribute to the task structure to store the request ID. Any system call that the thread issues on behalf of the request thus has the ID in its task structure. On a system call, Re-FUSE inspects the tagged request ID in the task structure of the process to correlate the system call with the original application request. Re-FUSE also uses the tagged request ID in the task structure of the file-system process to differentiate system calls made by the user-level file system from other processes in the operating system. Figure 5.3 presents these steps in more detail.

System-Call Logging

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

The logged state contains the type, input arguments, and the response (return value and data), along with a request ID, and is stored in a hash table called the *syscall request-response table*. This hash table is indexed by the ID of the application request. Figure 5.3 shows how system-call logging takes place during regular operations.

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

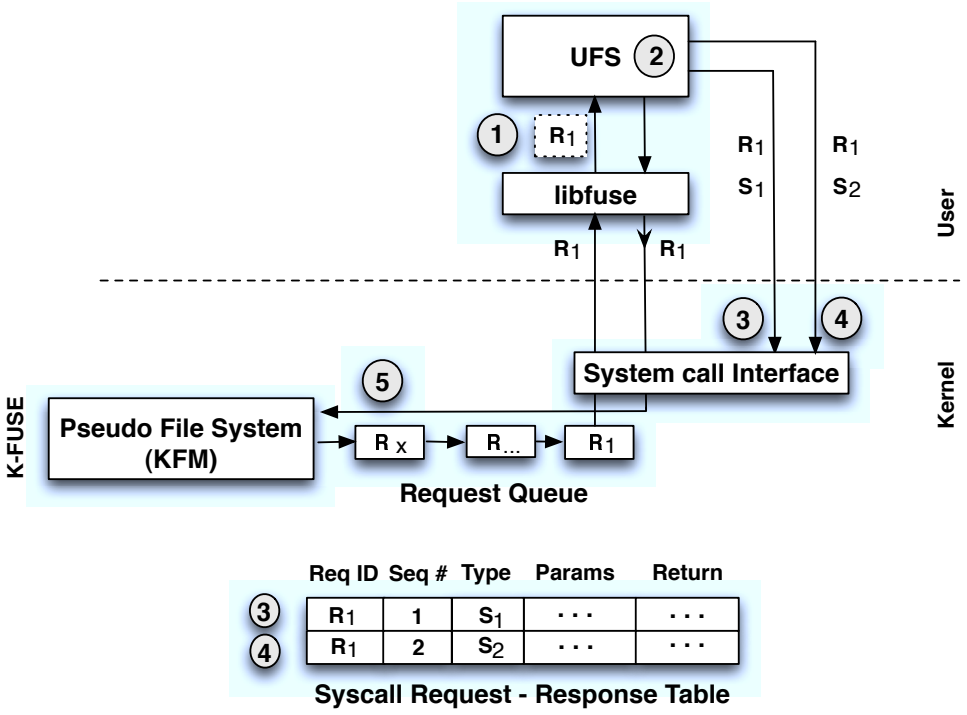

Figure 5.3: **Request Tagging and System-call Logging.** *The figure shows how Re-FUSE tracks the progress of individual file-system request. When KFM queues the* application requests *(denoted by R with a subscript). Re-FUSE tracks the progress of the request in the following way: (1) the request identifier is transparently attached to the task structure of the worker thread at the libfuse layer; (2) the user-level file system worker thread issues one or more system calls (denoted by S with a subscript) while processing the request; (3 and 4) Re-FUSE (at the system call interface) identifies these calls through the request ID in the caller's task structure and logs the input parameters along with the return value; (5) the KFM, upon receiving the response from the user-level file system for a request, deletes its entries from the log.*

Non-interruptible System Calls

The threading model in Linux prevents this basic logging approach from working correctly. Specifically, the threading model in Linux forces all threads of a process to be killed when one of the thread terminates (or crashes) due to a bug. Moreover, the other threads are killed independent of whether they are executing in user or kernel mode. Our logging approach only works if the system call issued by the user-level file system finishes completely, as a partially-completed system call could leave some residual state inside the kernel, thus preventing correct replay of in-flight requests (see Section 3.3.3 for details).

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

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

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

Performance Optimizations

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

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

5.3.3 Fault Detection

Re-FUSE detects faults in a user-level file-system through file-system crashes. As discussed earlier, Re-FUSE only handles faults that are both transient and fail-stop. Unlike kernel-level file systems, detection of faults in a user-level file system is simple; Re-FUSE inspects the return value and the signal attached to the killed file-system process to differentiate between regular termination and a crash.

Re-FUSE currently only implements a lightweight fault-detection mechanism. Fault detection can be further hardened in user-level file systems by applying techniques used in other systems [39, 121, 200]. Such techniques can help to automatically add checks (by code or binary instrumentation) to crash file systems more quickly when certain types of bugs are encountered (e.g., out-of-bounds memory accesses).

5.3.4 Fault Recovery

The recovery subsystem is responsible for restarting and restoring the state of the crashed user-level file system. To restore the in-memory state of the crashed user-level file system, Re-FUSE leverages the information about the file-system state available through the KFM. Recovery after a crash mainly consists of the following steps: cleanup, re-initialize, restore the in-memory state of the user-level file system, and re-execute the in-flight file-system requests at the time of the crash

(see Section 3.3.3 for details). The decoupled execution model in the FUSE preserves application state on a crash. Hence, application state need not be restored. We now explain the steps in the recovery process in detail.

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

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

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

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

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

system, Re-FUSE dumps useful debugging information about the error for the file-system developer.

5.3.5 Leveraging FUSE

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

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

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

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

5.3.6 Limitations

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

a block during allocation, the arguments to the subsequent replay operations (i.e., the block number of the bitmap block) would change and thus could potentially leave the file system in an inconsistent state.

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

Multi-threading can also limit the applicability of Re-FUSE. For example, multi-threading in block-based file systems could lead to race conditions during replay of in-flight requests and hence data loss after recovery. Different threading models could also involve multiple threads to handle a single request. For such systems, the FUSE request ID needs to be explicitly transferred between the (worker) threads so that the operating system can identify the FUSE request ID for which the corresponding system call is issued.

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

5.3.7 Implementation Statistics

Our Re-FUSE prototype is implemented in Linux 2.6.18 and FUSE 2.7.4. Table 5.2 shows the code changes done in both FUSE and the kernel proper. For Re-FUSE, around 3300 and 1000 lines of code were added to the Linux kernel and FUSE, respectively. The code changes in libfuse include request tagging, fault detection, and state restoration; changes in KFM center around support for recovery. The

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

FUSE Changes

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

Kernel Changes

Table 5.2: **Implementation Effort.** *The table presents the code changes required to transform FUSE and Linux 2.6.18 into their restartable counterparts.*

code changes in the VFS layer correspond to the support for system-call logging, and modifications in the MM and NET modules correspond to page versioning and socket-buffer caching respectively.

5.4 Evaluation

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

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

5.4.1 Generality

To show Re-FUSE can be used by many user-level file systems, we chose NTFS-3g, AVFS, and SSHFS. These file systems are different in their underlying data access mechanism, reliability guarantees, features, and usage. Table 5.3 shows the code changes required in each of these file systems to work with Re-FUSE.

From the table, we can see that file-system specific changes required to work with Re-FUSE are minimal. To each user-level file system, we have added less

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

Table 5.3: **Implementation Complexity.** *The table presents the code changes required to transform NTFS-3g, AVFS and SSHFS into their restartable counterparts.*

than 10 lines of code, and modified a few more. Some of these lines were added to daemonize the file system and to restart the process in the event of a crash. A few further lines were added or modified to make recovery work properly. We now discuss the changes in individual file systems.

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

AVFS: AVFS caches file handles from open requests to help speedup subsequent accesses in the underlying file systems. On a restart, the cached file handles are thrown away, this prevent requests from being properly executed through AVFS. To make AVFS work with Re-FUSE, we simply increment the reference count of open files and cache the file descriptor so that we can return the same file handle when it is reopened again after a restart.

SSHFS: SSHFS internally generates its own request IDs to match the responses from the remote host with waiting requests. The request IDs are stored inside SSHFS and are lost on a crash. After restart, on replay of an in-flight request, SSHFS generates new request IDs which could be different than the old ones. The mismatch in request IDs would prevent system-call logging from operating correctly, as all parameters need to be exactly matched for SSHFS to process the response. To make SSHFS work correctly with Re-FUSE, we made Re-FUSE uses

the FUSE request ID tagged in the worker thread along with the sequence number to match new request IDs with the old ones. Once requests are matched, Re-FUSE correctly returns the cached response. Also, to mask the SSHFS crash from the remote server, Re-FUSE holds an extra reference count on the network socket, and re-attaches it to the new process that is created. Without this action, upon a restart, SSHFS would start a new session, and the cached file handle would not be valid in the new session.

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

5.4.2 Robustness

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

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

Controlled Fault Injection

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

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

fault. File-system state indicates if a file system can continue servicing subsequent requests after a fault.

Tables 5.4, 5.5, and 5.6 summarizes the results of our fault-injection experiments. The caption explains how to interpret the data in the table. We now discuss the major observations and the conclusions of our fault-injection experiments.

		Regular			Restart			Re-Fuse		
Operation	**NTFS_fn**	**Application?**	**FS:Consistent?**	**FS:Usable?**	**Application?**	**FS:Consistent?**	**FS:Usable?**	**Application?**	**FS:Consistent?**	**FS:Usable?**
create	fuse_create	×	×	×	e	×	√	√	√	√
mkdir	fuse_create	×	×	×	e	×	√	√	√	√
symlink	fuse_create	×	×	×	e	×	√	√	√	√
link	link	×	×	×	e	×	√	√	√	√
rename	link	×	×	×	e	×	√	√	√	√
open	fuse_open	×	√	×	√	√	√	√	√	√
read	fuse_read	×	√	×	√	√	√	√	√	√
readdir	fuse_readdir	×	√	×	√	√	√	√	√	√
readlink	fuse_readlink	×	√	×	√	√	√	√	√	√
write	fuse_write	×	×	×	√	×	√	√	√	√
unlink	delete	×	×	×	e	×	√	√	√	√
rmdir	inode_sync	×	×	×	e	×	√	√	√	√
truncate	fuse_truncate	×	×	×	√	×	√	√	√	√
utime	inode_sync	×	√	×	√	√	√	√	√	√

Table 5.4: **NTFS-3g Fault Study.** *The table shows the affect of fault injections on the behavior of NTFS-3g. Each row presents the results of a single experiment, and the columns show (in left-to-right order) the intended operation, the file system function that was fault injected, how it affected the application, whether the file system was consistent after the fault, and whether the file system was usable for other operations. Various symbols are used to condense the presentation. For application behavior, "*√*": application observed successful completion of the operation; "*×*": application received the error "software caused connection abort"; "e": application incorrectly received an error.*

		Regular			Restart			Re-Fuse		
Operation	**SSHFS_fn**	**Application?**	**FS:Consistent?**	**FS:Usable?**	**Application?**	**FS:Consistent?**	**FS:Usable?**	**Application?**	**FS:Consistent?**	**FS:Usable?**
create	open_common	×	√	×	e	√	√	√	√	√
mkdir	mkdir	×	√	×	e	√	√	√	√	√
symlink	symlink	×	√	×	e	√	√	√	√	√
rename	rename	×	√	×	e	√	√	√	√	√
open	open_common	×	√	×	√	√	√	√	√	√
read	sync_read	×	√	×	√	√	√	√	√	√
readdir	getdir	×	√	×	√	√	√	√	√	√
readlink	readlink	×	√	×	√	√	√	√	√	√
write	write	×	√	×	√	√	√	√	√	√
unlink	unlink	×	√	×	e	√	√	√	√	√
rmdir	rmdir	×	√	×	e	√	√	√	√	√
truncate	truncate	×	√	×	√	√	√	√	√	√
chmod	chmod	×	√	×	√	√	√	√	√	√
stat	getattr	×	√	×	√	√	√	√	√	√

Table 5.5: **SSHFS Fault Study.** *The table shows the affect of fault injections on the behavior of SSHFS. Each row presents the results of a single experiment, and the columns show (in left-to-right order) the intended operation, the file system function that was fault injected, how it affected the application, whether the file system was consistent after the fault, and whether the file system was usable for other operations. Various symbols are used to condense the presentation. For application behavior, "√": application observed successful completion of the operation; "×": application received the error "software caused connection abort"; "e": application incorrectly received an error.*

		Regular			Restart			Re-Fuse		
Operation	**AVFS_fn**	**Application?**	**FS:Consistent?**	**FS:Usable?**	**Application?**	**FS:Consistent?**	**FS:Usable?**	**Application?**	**FS:Consistent?**	**FS:Usable?**
create	mknod	×	×	×	e	×	√	√	√	√
mkdir	mkdir	×	×	×	e	×	√	√	√	√
symlink	symlink	×	×	×	e	×	√	√	√	√
link	link	×	×	×	e	×	√	√	√	√
rename	rename	×	×	×	e	×	√	√	√	√
open	open	×	√	×	√	√	√	√	√	√
read	read	×	√	×	√	√	√	√	√	√
readdir	readdir	×	√	×	√	√	√	√	√	√
readlink	readlink	×	√	×	√	√	√	√	√	√
write	write	×	×	×	√	×	√	√	√	√
unlink	unlink	×	×	×	e	×	√	√	√	√
rmdir	rmdir	×	×	×	e	×	√	√	√	√
truncate	truncate	×	×	×	√	×	√	√	√	√
chmod	chmod	×	√	×	√	√	√	√	√	√
stat	getattr	×	√	×	√	√	√	√	√	√

Table 5.6: **AVFS Fault Study.** *The table shows the affect of fault injections on the behavior of AVFS. Each row presents the results of a single experiment, and the columns show (in left-to-right order) the intended operation, the file system function that was fault injected, how it affected the application, whether the file system was consistent after the fault, and whether the file system was usable for other operations. Various symbols are used to condense the presentation. For application behavior, "√": application observed successful completion of the operation; "×": application received the error "software caused connection abort"; "e": application incorrectly received an error.*

Vanilla OS, FUSE, and File Systems

First, we analyze the vanilla versions of the file systems running on vanilla FUSE and a vanilla Linux kernel. The results are shown in the leftmost result columns in Tables 5.4, 5.5, and 5.6.

For NTFS-3g, in all experiments, the application always received a connection abort error after the fault injection and was killed as it could not handle the error from the file system. In all experiments, the file system was unusable after a fault injection. Finally, the file system was only consistent for 36% of the fault-injection experiments. The inconsistency was caused due to partially completed operations inside the file system (see request splitting in Section 5.2.1 for details).

For SSHFS, in all experiments, the application received a connection abort error and the file system was unusable after a fault injection. Unlike NTFS-3g, the file system was always consistent after fault injection. By design, SSHFS atomically updates the changes to the remote host. The remote host is unaffected by the fault, as faults are localized within the user-level file system (see Section **??** for details).

For AVFS, in all experiments, application received a connection abort error and the file system was unusable after a fault injection. The file system was only consistent in 40% of the experiments. This is because unlike SSHFS, not all file system requests are atomic. Hence, some of the partially completed file system requests resulted in the underlying system being inconsistent.

Overall, we observe that the vanilla versions of user-level file systems and FUSE do a poor job in hiding failures from applications. In all experiments, the user-level file system is unusable after the fault; as a result, applications have to prematurely terminate their requests after receiving an error (a “software-caused connection abort”) from FUSE. Moreover, in 40% of the cases, crashes lead to inconsistent file system state.

Simple Restart

Second, we analyze the usefulness of fault-detection and simple restart at the KFM *without* any explicit support from the operating system. The second result columns (denoted by Restart) of Tables 5.4, 5.5, and 5.6 shows the result.

For NTFS-3g, in 50% of the fault injection experiments, an error was incorrectly returned to the application on a request retry after a restart. The file system was consistent in 38% of the experiments; the percentage of consistent file-system state is the same as that of the vanilla NTFS-3g file system. Finally, unlike the vanilla file systems, the restart versions of the file system were usable after all fault injection experiments.

For SSHFS, in 43% of the experiments, an error was incorrectly returned to the application on a request retry after a restart. In all experiments, the file system was consistent and usable after a fault injection. The simple restart mechanism added to FUSE was sufficient to restart the crashed file system and enabled it to service subsequent requests.

For AVFS, in 47% of the experiments, an error was incorrectly returned to the application on a requet retry after a restart. The file system was always usable after the fault injection experiments. But, in 40% of the fault injection experiments, the file system was left in an inconsistent state.

Overall, we observe that a simple restart of the user-level file system and replay of in-flight requests at the KFM layer ensures that the application completes the failed operation in the majority of the cases (around 60%). It still cannot, however, re-execute a significant amount (around 40%) of partially-completed operations due to the non-idempotent nature of the particular file system operation. Moreover, an error is wrongly returned to the application and the crashes leave the file system in an inconsistent state.

Restart Using Re-FUSE

Finally, we analyze the usefulness of Re-FUSE that includes restarting the crashed user-level file system, replaying in-flight requests, and has support from the operating system for re-executing non-idempotent operations (i.e., all the support described in Section 5.3). The results of the experiments are shown in the rightmost columns of Tables 5.4, 5.5, 5.6. From the table, we can see that all faults are handled properly, applications successfully complete the operation, and the file system is always left in a consistent state.

Random Fault Injection

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

We use three commonly-used macro-benchmarks to help analyze file-system robustness (and later, performance). Specifically, we utilize the sort utility, Postmark [96], and OpenSSH [162]. The sort benchmark represents data-manipulation

File System + FUSE	Injected Faults	Sort (Survived)	OpenSSH (Survived)	Postmark (Survived)
NTFS-3g	100	0	0	0
SSHFS	100	0	0	0
AVFS	100	0	0	0

Table 5.7: **Random Fault Injection in FUSE.** *The table shows the affect of randomly injected crashes on the three file systems running on FUSE. The second column refers to the total number of random (in terms of the crash point in the code) crashes injected into the file system during the span of time it is serving a macro-benchmark. The crashes are injected by sending the signal SIGSEGV to the file system process periodically. The right-most three columns indicate the number of survived crashes by the reinforced file systems during each macro-benchmark.*

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

Table 5.8: **Random Fault Injection in Re-FUSE.** *The table shows the affect of randomly injected crashes on the three file systems supported with Re-FUSE. The details of the fault injection and data interpretation are in Table 5.7.*

workloads, Postmark represents I/O-intensive workloads, and OpenSSH represents user-desktop workloads.

Tables 5.7 and 5.8 present the result of our study. From Table 5.8, we see that Re-FUSE ensures that the application continues executing through the failures, thus making progress. We also found that a vanilla user-level file system with no support for fault handling cannot tolerate crashes (shown in the Table 5.7).

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

5.4.3 Performance

Though fault-tolerance is our primary goal, we also evaluate the performance of Re-FUSE in the context of regular operations and recovery time. Performance evaluation enables us to understand the overhead of running the system in the absence and presence of faults. Specifically, we measure the overhead of our system during regular operations and also during user-level file system crashes to see if a user-level file system running on Re-FUSE has acceptable overheads.

Regular Operations

We use both micro- and macro-benchmarks to evaluate the overheads during regular operation. Micro-benchmarks help analyze file-system performance for frequently executed operations in isolation. We use sequential read/write, random read/write, create, and delete operations as our micro benchmarks. These operations exercise the most frequently accessed code paths in file systems. The caption in Table 5.9 describes our micro-benchmark configuration in more detail. We also use the previously-described macro-benchmarks sort, Postmark, and OpenSSH; the caption in Table 5.10 describes the configuration parameters for our experiments.

Tables 5.9 and 5.10 show the results of micro- and macro-benchmarks, respectively. From the tables, we can see that for both micro- and macro-benchmarks, Re-FUSE has minimal overhead, often less than 3%. The overheads are small due to in-memory logging and our optimization through page versioning (or socket buffer caching in the context of SSHFS). The overheads of running NTFS-3g on Re-FUSE are noticeable in some of the write-intensive micro-benchmark experiments due to page versioning. These results show that the additional reliability Re-FUSE achieves comes with negligible overhead for common file-system workloads, thus removing one important barrier of adoption for Re-FUSE.

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

Table 5.9: **Microbenchmarks.** *This table compares the execution time (in seconds) for various benchmarks for restartable versions of ntfs-3g, sshfs, avfs (on Re-FUSE) against their regular versions on the unmodified kernel. Sequential reads/writes are 4 KB at a time to a 1-GB file. Random reads/writes are 4 KB at a time to 100 MB of a 1-GB file. Create/delete copies/removes 1000 files each of size 1MB to/from the file system respectively. All workloads use a cold file-system cache.*

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

Table 5.10: **Macrobenchmarks.** *The table presents the performance (in seconds) of different benchmarks running on both standard and restartable versions of ntfs-3g, sshfs, and avfs. The sort benchmark (CPU intensive) sorts roughly 100MB of text using the command-line sort utility. For the OpenSSH benchmark (CPU+I/O intensive), we measure the time to copy, untar, configure, and make the OpenSSH 4.51 source code. PostMark (I/O intensive) parameters are: 3000 files (sizes 4KB to 4MB), 60000 transactions, and 50/50 read/append and create/delete biases.*

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

Table 5.11: **Restart Time in Re-FUSE.** *The table shows the impact of a single restart on the restartable versions of the file systems. The benchmark used is sort and the restart is triggered approximately mid-way through the benchmark.*

Recovery Time

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

Table 5.11 shows the elapsed time and the average time Re-FUSE spent in restoring the crashed user-level file system state. The restoration process includes restart of the file-system process and restoring its in-memory state. From the table, we can see that the restart time is on the order of a few milliseconds. The application also does not see any observable increase in its execution time due to the file-system crash.

In summary, both micro- and macro-benchmark results show that the performance overheads during regular operations are minimal. Even in the event of a file system crash, Re-FUSE restarts the user-level file system within a few hundred milliseconds.

5.5 Summary

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

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

It is unlikely developers will ever build the "perfect" file system; Re-FUSE presents one way to tolerate these imperfections.

Chapter 6

Reliability through Reservation

"Act as if it were impossible to fail."
– Dorothea Brande

Memory is one of the important resources that is widely used within the file system and also across other operating-system components. Moreover, in a complex system such as Linux, memory allocations can happen in a variety of ways (*e.g.*, `kmalloc`, `kmem_cache_alloc`, etc.). Previous studies have shown that memory-allocation failure can lead to catastrophic results in file systems [50, 71, 120, 197].

In this chapter, we explore the possibility of improving the reliability of file systems through resource reservation. We take a new approach to solving the problem presented by memory-allocation failures by following a simple mantra: *the most robust recovery code is recovery code that never runs at all*. In other words, our goal is to eliminate the recovery code that deals with memory-allocation failures to the largest possible extent.

There are a few challenges in doing it this way. First, we need a mechanism to identify all possible memory allocation calls during each system call. Second, we need to know the type and parameters of the objects that need to be allocated. Third, seamlessly pre-allocate and return the corresponding objects at runtime. Four, safe way to clean up any unused pre-allocated objects at the end of each system call.

Our approach is called *Anticipatory Memory Allocation (AMA)*. The basic idea behind AMA is simple. First, using both a static analysis tool and domain knowledge, the developer determines a conservative estimate of the total memory allocation demand of each call into the kernel subsystem of interest. Using this information, the developer then augments the code to pre-allocate the requisite amount of memory at run-time, immediately upon entry into the kernel subsystem. The AMA

run-time then transparently redirects existing memory-allocation calls to use memory from the pre-allocated chunk. Thus, when a memory allocation takes place deep in the heart of the kernel subsystem, it is guaranteed never to fail.

The rest of this chapter is organized as follows. First, in Section 6.1, we present a background on memory allocation in Linux. Then in Section 6.2, we present our study of how Linux file systems react to memory failure. Then we give an overview of our approach in Section 6.3. We present the design and implementation of AMA in Section 6.4 and Section 6.5, respectively. Finally, in Section 6.6 we evaluate AMA's robustness and performance.

6.1 Linux Memory Allocators

We provide some background on kernel memory allocation. We describe the many different ways in which memory is explicitly allocated within the kernel. Our discussion revolves around the Linux kernel (with a focus on file systems), although in our belief the allocation types shown here likely to exist in other modern operating systems.

6.1.1 Memory Zones

At the lowest level of memory allocation within Linux is a buddy-based allocator of physical pages [25]. The buddy-based allocator uses low-level routines such as `alloc_pages()` and `free_pages()` to request and return pages, respectively. These functions serve as the basis for the allocators used for kernel data structures (described below), although they can be called directly if so desired.

6.1.2 Kernel Allocators

Most dynamic memory requests in the kernel use the Linux *slab allocator*, which is based on Bonwick's original slab allocator for Solaris [22] (a newer SLUB allocator provides the same interfaces but is internally simpler). The premise behind the slab allocator is that certain objects are repeatedly created and destroyed by the kernel; the allocator thus keeps separate caches for a range of allocation sizes (from 32 bytes to 128 KB, in powers of 2), and thus can readily recycle freed memory and avoid fragmentation. One simply calls the generic memory allocation routines `kmalloc()` and `kfree()` to use these facilities.

For particularly popular objects, specialized caches can be explicitly created. To create such a cache, one calls `kmem_cache_create()`, which (if successful) returns a reference to the newly-created object cache; subsequent calls to

	kmalloc	kmem_cache_alloc	vmalloc	mempool create	alloc pages
btrfs	93	7	3	0	1
ext2	8	1	0	0	0
ext3	12	1	0	0	0
ext4	26	10	1	0	0
jfs	18	1	2	1	0
reiser	17	1	5	0	0
xfs	11	1	0	1	1

Table 6.1: **Usage of Different Allocators.** *The table shows the number of different memory allocators used within Linux file systems. Each column presents the number of times a particular routine is found in each file system.*

`kmem_cache_alloc()` are passed this reference and return memory for the specific object. Hundreds of these specialized allocation caches exist in a typical system (see `/proc/slabinfo`); a common usage for a file system, for example, is an inode cache.

Beyond these commonly-used routines, there are a few other ways to request memory in Linux. A *memory pool* interface allows one to reserve memory for use in emergency situations. Finally, the *virtual malloc* interface requests in-kernel pages that are virtually (but not necessarily physically) contiguous.

To demonstrate the diversity of allocator usage, we present a study of the popularity of these interfaces within a range of Linux file systems. Table 6.1 presents our results. As one can see, although the generic interface `kmalloc()` is most popular, the other allocation routines are used as well. For kernel code to be robust, it must handle failures from all of these allocation routines.

6.1.3 Failure Modes

When calling into an allocator, flags determine the exact behavior of the allocator, particularly in response to failure. Of greatest import to us is the use of the `__GFP_NOFAIL` flag, which a developer can use when they know their code cannot handle an allocation failure; using the flag is the only way to guarantee that an allocator will either return successfully or not return at all (*i.e.*, keep trying forever). However, this flag is rarely used. As lead Linux kernel developer Andrew Morton said [119]: "`__GFP_NOFAIL` should only be used when we have no way

of recovering from failure. ... Actually, nothing in the kernel should be using __GFP_NOFAIL. It is there as a marker which says 'we really shouldn't be doing this but we don't know how to fix it'." In all other uses of kernel allocators, failure is thus a distinct possibility.

6.2 Bugs in Memory Allocation

Memory-allocation failures are an issues in many systems, as developers do not always handle rare events like allocation failures. Earlier work has repeatedly found that memory-allocation failure is often mishandled [49, 197]. In Yang *et al.*'s model-checking work, one key to finding bugs is to follow the code paths where memory allocation has failed [197].

We now perform a brief study of memory-allocation failure handling within Linux file systems. We use fault injection to fail calls to the various memory allocators and determine how the code reacts as the number of such failures increases. Our injection framework picks a certain allocation call (*e.g.*, `kmalloc()`) within the code and fails it probabilistically; we then vary the probability and observe how the kernel reacts as an increasing percentage of memory-allocation calls fail. For the workload, we created a micro-benchmark that performs a mix of file-system operations (such as read, write, create, delete, open, close, and lookup). Table 6.2 presents our results, which sums the failures seen in 15 runs per file system, while changes the probability of an allocation request failing for 0%, 10% and 50% of the time.

We report the outcomes in two categories: process state and file-system state. The process state results are further divided into two groups: the number of times (in 15 runs) that a running process received an error (such as ENOMEM), and the number of times that a process was terminated abnormally (i.e., killed). The file system results are split into two categories as well: a count of the number of times that the file system became unusable (i.e., further use of the file system was not possible after the trial), and the number of times the file system became inconsistent as a result, possible losing user data.

From the table, we can make the following observations. First, we can see that even a simple, well-tested, and slowly-evolving file system such as Linux ext2 still does not handle memory-allocation failures very well; we take this as evidence that doing so is challenging. Second, we observe that all file systems have difficulty handling memory-allocation failure, often resulting in an unusable or inconsistent file system.

An example of how a file-system inconsistency can arise is found in Figure 6.1.

	Process State		File-System State	
	Error	Abort	Unusable	Inconsistent
$btrfs_0$	0	0	0	0
$btrfs_{10}$	0	14	15	0
$btrfs_{50}$	0	15	15	0
$ext2_0$	0	0	0	0
$ext2_{10}$	10	5	5	0
$ext2_{50}$	10	5	5	0
$ext3_0$	0	0	0	0
$ext3_{10}$	10	5	5	4
$ext3_{50}$	10	5	5	5
$ext4_0$	0	0	0	0
$ext4_{10}$	10	5	5	5
$ext4_{50}$	10	5	5	5
jfs_0	0	0	0	0
jfs_{10}	15	0	2	5
jfs_{50}	15	0	5	5
$reiserfs_0$	0	0	0	0
$reiserfs_{10}$	10	4	4	0
$reiserfs_{50}$	10	5	5	0
xfs_0	0	0	0	0
xfs_{10}	13	1	0	3
xfs_{50}	10	5	0	5

Table 6.2: **Fault Injection Results.** *The table shows the reaction of the Linux file systems to memory-allocation failures as the probability of a failure increases. We randomly inject faults into the three most-used allocation calls:* `kmalloc()`, `kmem_cache_alloc()`, *and* `__alloc_pages()`. *For each file system and each probability (shown as subscript), we run a micro benchmark 15 times and report the number of runs in which certain failures happen in each column. We categorize all failures into process state and file-system state, in which 'Error' means that file system operations fail (gracefully), 'Abort' indicates that the process was terminated abnormally, 'Unusable' means the file system is no longer accessible, and 'Inconsistent' means file system metadata has been corrupted and data may have been lost. Ideally, we expect the file systems to gracefully handle the error (i.e., return error) or retry the failed allocation request. Aborting a process, inconsistent file-system state, and unusable file system are unacceptable actions on an memory allocation failure.*

```
empty_dir() [file: namei.c]
  if (...|| !(bh = ext4_bread(..., &err)))
    ...
    return 1; // XXX: should have returned 0


ext4_rmdir() [file: namei.c]
  retval = -ENOTEMPTY;
  if (!empty_dir(inode))
    goto end_rmdir;
  retval = ext4_delete_entry(handle, dir, de, bh);
  if (retval)
    goto end_rmdir;
```

Figure 6.1: **Improper Failure Propagation.** *The code shown in the figure is from the ext4 file system, and shows a case where a failed low-level allocation (in ext4_bread()) is not properly handled, which eventually leads to an inconsistent file system.*

In this example, while trying to remove a directory from the file system (*i.e.,* the `ext4_rmdir` function), the routine first checks if the directory is empty by calling `empty_dir()`. This routine, in turn, calls `ext4_bread()` to read the directory data. Unfortunately, due to our fault injection, `ext4_bread()` tries to allocate memory but fails to do so, and thus the call to `ext4_bread()` returns an error (correctly). The routine `empty_dir()` incorrectly propagates this error, simply returning a 1 and thus accidentally indicating that the directory is empty and can be deleted. Deleting a non-empty directory not only leads to a hard-to-detect file-system inconsistency (despite the presence of journaling), but also could render inaccessible a large portion of the directory tree.

Finally, a closer look at the code of some of these file systems reveals a third interesting fact: in a file system under active development (such as btrfs), there are many places within the code where memory-allocation failure is never checked for; our inspection has yielded over 20 places within btrfs such as this. Such trivial mishandling is rarer inside more mature file systems.

Overall, our results hint at a broader problem: developers write code as if memory allocation will never fail; only later do they (possibly) go through the code and attempt to "harden" it to handle the types of failures that might arise. Proper handling of such errors, as seen in the ext4 example, is a formidable task, and as a result, such hardening sometimes remains "softer" than desired.

Summary

Kernel memory allocation is complex, and handling failures still proves challenging even for code that is relatively mature and generally stable. We believe these problems are fundamental given the way current systems are designed; specifically, to handle failure correctly, a *deep recovery* must take place, where far downstream in the call path, one must either handle the failure, or propagate the error up to the appropriate error-handling location while concurrently making sure to unwind all state changes that have taken place on the way down the path. Earlier work has shown that the simple act of propagating an error correctly in a complex file system is challenging [71]; doing so and correctly reverting all other state changes presents further challenges. Although deep recovery is possible, we believe it is usually quite hard, and thus error-prone. More sophisticated bug-finding tools could be built, and further bugs unveiled; however, to truly solve the problem, an alternate approach to deep recovery is likely required.

6.3 Overview

We now present an overview of *Anticipatory Memory Allocation (AMA)*, a novel approach to solve the memory-allocation failure-handling problem. The basic idea is simple: first, we analyze the code paths of a kernel subsystem to determine what its memory requirements are. Second, we augment the code with a call to pre-allocate the necessary amounts. Third, we transparently redirect allocation requests during run-time to use the pre-allocated chunks of memory.

Figure 6.2 shows a simple example of the transformation. In the figure, a simple entry-point routine `f1()` calls one other *downstream* routine, `f2()`, which in turn calls `f3()`. Each of these routines allocates some memory during their normal execution, in this case 100 bytes by `f2()` and 25 bytes by `f3()`.

With AMA, we analyze the code paths to discover the worst-case allocation possible; in this example, the analysis is simple, and the result is that two memory chunks, of size 100 and 25 bytes, are required. Then, before calling into `f2()`, one calls into the anticipatory memory allocator to pre-allocate chunks of 100 and 25 bytes. The modified run-time then redirects all downstream allocation requests to use this pre-allocated pool. Thus the calls to allocate 100 and 25 bytes in `f2()` and `f3()` (respectively) will use memory already allocated by AMA, and are guaranteed not to fail.

The advantages of this approach are many. First, memory-allocation failures never happen downstream, and thus there is no need to handle said failures; the complex unwinding of kernel state and error propagation are thus avoided entirely.

```
void f2() {
  void *p = malloc(100);
  f3();
}

void f3() {
  void *q = malloc(25);
}

int f1() {
   // AMA: Pre-allocate 100- and 25-byte chunks
  f2();
   // AMA: Free any unused chunks
}
```

Figure 6.2: **Simple AMA Example.** *The code presents a simple example of how AMA is used. In the unmodified case, routine* `f1()` *calls* `f2()`*, which calls* `f3()`*, each of which allocate some memory (and perhaps incorrectly handle their failure). With AMA,* `f1()` *pre-allocates the full amount needed; subsequent calls to allocate memory are transparently redirected to use the pre-allocated chunks instead of calling into the real allocators, and any remaining memory is freed.*

Second, because allocation failure can only happen in only one place in the code (at the top), it is easy to provide a unified handling mechanism; for example, if the call to pre-allocate memory fails, the developer could decide to immediately return a failure, retry, or perhaps implement a more sophisticated exponential backoff-and-retry approach, all excellent examples of the *shallow recovery* AMA enables. Third, very little code change is required; except for the calls to pre-allocate and perhaps free unused memory, the bulk of the code remains unmodified, as the run-time transparently redirects downstream allocation requests to use the pre-allocated pool.

Unfortunately, code in real systems is not as simple as that found in the figure, and indeed, the problem of determining how much memory needs to be allocated given an entry point into a complex code base is generally undecidable. Thus, the bulk of our challenge is transforming the code and gaining certainty that we have done so correctly and efficiently. To gain a better understanding of the problem, we must choose a subsystem to focus upon, and transform it to use AMA.

```
void ext2_init_block_alloc_info(struct inode *inode)
{
  struct ext2_inode_info *ei = EXT2_I(inode);
   struct ext2_block_alloc_info *block_i = ei→i_block_alloc_info;
  block_i = kmalloc(sizeof(*block_i), GFP_NOFS);
  ...
}
```

Figure 6.3: **A Simple Call.** *This code is a good example of a simple call in the ext2 file system. The memory allocation routine kmalloc allocates an object (block_i) that is equal to the size of ext2_block_alloc_info structure. This size is determined at the compile time and does not change across invocations.*

6.3.1 A Case Study: Linux ext2-mfr

The case study we use is the Linux ext2 file system. Although simpler than its modern journaling cousins (i.e., ext3 and ext4), ext2 is a real file system and has enough complex memory-allocation behavior (as described below) to demonstrate the intricacies of developing AMA for a real kernel subsystem.

We describe our effort to transform the Linux ext2 file system into a memory-robust version of itself, which we call Linux ext2-mfr (*i.e.*, a version of ext2 that is Memory-Failure Robust). In our current implementation, the transformation requires some human effort and is aided by a static analysis tool that we have developed. The process could be further automated, thus easing the development of other memory-robust file systems; we leave such efforts to future work.

We now highlight the various types of allocation requests that are made, from simpler to more complex. By doing so, we are showing what work needs to be done to be able to correctly pre-allocate memory before calling into ext2 routines, and thus shedding light on the types of difficulties we encountered during the transformation process.

Simple Calls

Most of the memory-allocation calls made by the kernel are of a fixed size. Allocating file system objects such as dentry, file, inode, and page have pre-determined sizes. For example, file systems often maintain a cache of inode objects, and thus must have memory allocated for them before being read from disk. Figure 6.3 shows one example of such a call from ext2.

```
struct dentry *d_alloc(..., struct qstr *name) {
  ...
  if (name→len > DNAME_INLINE_LEN-1) {
    dname = kmalloc(name→len + 1, GFP_KERNEL);
    if (!dname)
      return NULL;
    ...
  }
}
```

Figure 6.4: **A Parameterized and Conditional Call.** *This code represents a simplified version of the dentry allocation function, which is a good example of both parameterized and conditional call. The size of the object (dname) that needs to be allocated depends on the input parameter (name) and the allocation will only happen if the condition (name→len > DNAME_INLINE_LEN-1) holds true.*

Parameterized and Conditional Calls

Some allocated objects have variable lengths (e.g., a file name and extended attributes) and the exact size of the allocation is determined at run-time; sometimes allocations are not performed due to conditionals. Figure 6.4 shows how ext2 allocates memory for a directory entry, which uses a length field (plus one for the end-of-string marker) to request the proper amount of memory. This allocation is only performed if the name is too long and requires more space to hold it.

Loops

In many cases, file systems allocate objects inside a loop or inside nested loops. In ext2, the upper bound of the loop execution is determined by the object passed to the individual calls. For example, allocating pages to search for directory entries are done inside a loop. Another good example is searching for a free block within the block bitmaps of the file system. Figure 6.5 shows the page allocation code during directory lookups in ext2.

Function Calls

Of course, a file system is spread across many functions, and hence any attempt to understand the total memory allocation of a call graph given an entry point must be able to follow all such paths, sometimes into other major kernel subsystems. For example, one memory allocation request in ext2 is invoked 21 calls deep; this

```
ext2_find_entry (struct inode * dir, ...)
{
  unsigned long npages = dir_pages(dir);
  unsigned long n = 0;
  do {
    page = ext2_get_page(dir, n,..); // allocate a page
    ...
    if (ext2_match_entry (...));
        goto found;
    ...
    n++;
  } while (n != npages); // worst case: n = npages
found:
 return entry;
}
```

Figure 6.5: **Loop Calls.** *This is a code snippet from the ext2 file system that belongs to the directory entry search function. In the worst case, the number of pages that need to be allocated before the entry is found (if present) depends on the size of the directory (dir) that is being searched.*

example path starts at `sys_open`, traverses through some link-traversal and lookup code, and ends with a call to `kmem_cache_alloc`.

Recursions

A final example of an in-kernel memory allocation is one that is performed within a recursive call. Some portions of file systems are naturally recursive (*e.g.*, path-name traversal), and thus perhaps it is no surprise that recursion is commonplace. Figure 6.6 shows the block-freeing code that is called when a file is truncated or removed in ext2; in the example, `ext2_free_branches` calls itself to recurse down indirect-block chains and free blocks as need be.

6.3.2 Summary

To be able to pre-allocate enough memory for a call, one must handle parameterized calls, conditionals, loops, function calls, and recursion. If file systems only contained simple allocations and minimal amounts of code, pre-allocation would be rather straightforward.

```
static void
ext2_free_branches(struct inode *inode, ..., int depth){
  if (depth--) {
    ...
    // allocate a page and buffer head
    bh = sb_bread(inode→i_sb, ..);
    ...
    ext2_free_branches(inode,
                  (__le32*) bh→b_data,
                  (__le32*) bh→b_data + addr_per_block,
                  depth);
  } else
    ext2_free_data(inode, ...);
}
```

Figure 6.6: **Recursion.** *This code snippet is an example of memory allocation invocation inside a recursion. The function shown here is the ext2_free_branches, which frees any triple-, double-, and single-indirect blocks associated with a data block that is being freed in the ext2 file system. The termination condition for the recursion is the depth of the branch (i.e., the input parameter depth), which also determines the number of buffer heads and pages that need to be pre-allocated to avoid any allocation failures while executing this function.*

6.4 Static Transformation

We now present the static-analysis portion of AMA, in which we develop a tool, *the AMAlyzer*, to help decide how much memory to pre-allocate at each entry point into the kernel subsystem that is being transformed (in this case, Linux ext2). The AMAlyzer takes in the entire relevant call graph and produces a skeletal version, from which the developer can derive the proper pre-allocation amounts. After describing the tool, we present two novel optimizations we employ, cache peeking and page recycling, to reduce memory demands. We end the section with a discussion of the limits of our current approach.

We build the AMAlyzer on top of CIL [122], a tool that allows us to readily analyze kernel source code. CIL does not resolve function pointers automatically, which we require for our complete call graph, and hence we perform a small amount of extra work to ensure we cover all calls made in the context of the file system; because of the limited and stylized use of function pointers within the kernel, this process is straightforward. The AMAlyzer in its current form is comprised of a few thousand lines of OCaml code.

6.4.1 The AMAlyzer

We now describe the AMAlyzer in more detail. AMA consists of two phases. In the first phase, the tool searches through the entire subsystem to construct the allocation-relevant call graph (*i.e.*, the complete set of downstream functions that contain kernel memory-allocation requests). In the second phase, a more complex analysis determines which variables and state are relevant to allocation calls, and prunes away other irrelevant code. The result is a skeletal form of the subsystem in question, from which the pre-allocation amounts are readily derived.

Phase 1: Allocation-Relevant Call Graph

We start our analysis with the entire call graph (shown in Figure 6.7). The relevant portion of the call graph for ext2 (and all related components of the kernel) contains nearly 2000 nodes (one per relevant function) and roughly 7000 edges (calls between functions) representing roughly 180,000 lines of kernel source code. Even for a relatively-simple file system such as ext2, the task of manually computing the pre-allocation amount would be daunting, without automated assistance.

The first step of our analysis prunes the entire call graph and generates what we refer to as the *allocation-relevant call graph (ARCG)*. The ARCG contains only nodes and edges in which a memory allocation occurs, either within a node of the graph or somewhere downstream of it.

We perform a Depth First Search (DFS) on the call graph to generate ARCG. An additional attribute namely *calls_memory_allocation* (CMA), is added to each node (i.e., function) in the call graph to speed up the ARCG generation. The CMA attribute is set on two occasions. First, when a memory allocation routine is encountered during the DFS. Second, a node has its CMA set if at least one of the node's children has its CMA attribute set.

At the end of the DFS, the functions that do not have the CMA attribute set are safely deleted from the call graph. The remaining nodes in the call graph constitute the ARCG. Figure 6.8 shows the ARCG for the ext2 file system.

Phase 2: Loops and Recursion

At this point, the tool has reduced the number of functions that must be examined. In this part of the analysis, we add logic to handle loops and recursions, and where possible, to help identify their termination conditions. The AMAlyzer searches for all `for`, `while`, and `goto`-based loops, and walks through each function within such a loop to find either direct calls to kernel memory allocators or indirect calls

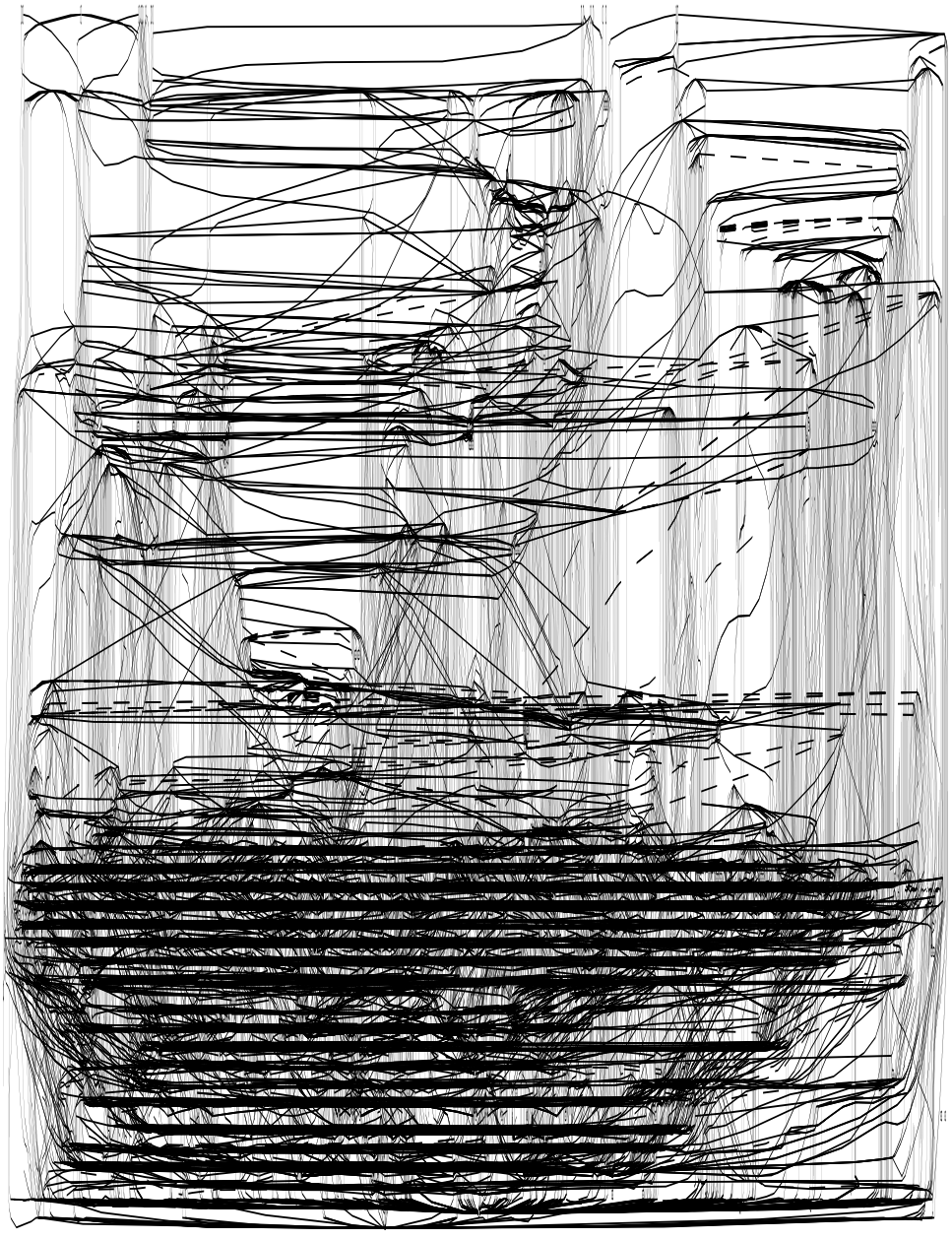

Figure 6.7: **The ext2 Call Graph.** *The figure plots the call graph for the ext2 file system. The gray nodes at the top represent entry points into the file system (such as* `sys_open()`*), and the black dots at the bottom are different memory allocators (such as* `kmalloc()`*). The dots in the middle represent functions in ext2 and the kernel, and lines between such dots (i.e., functions) indicate a function call.*

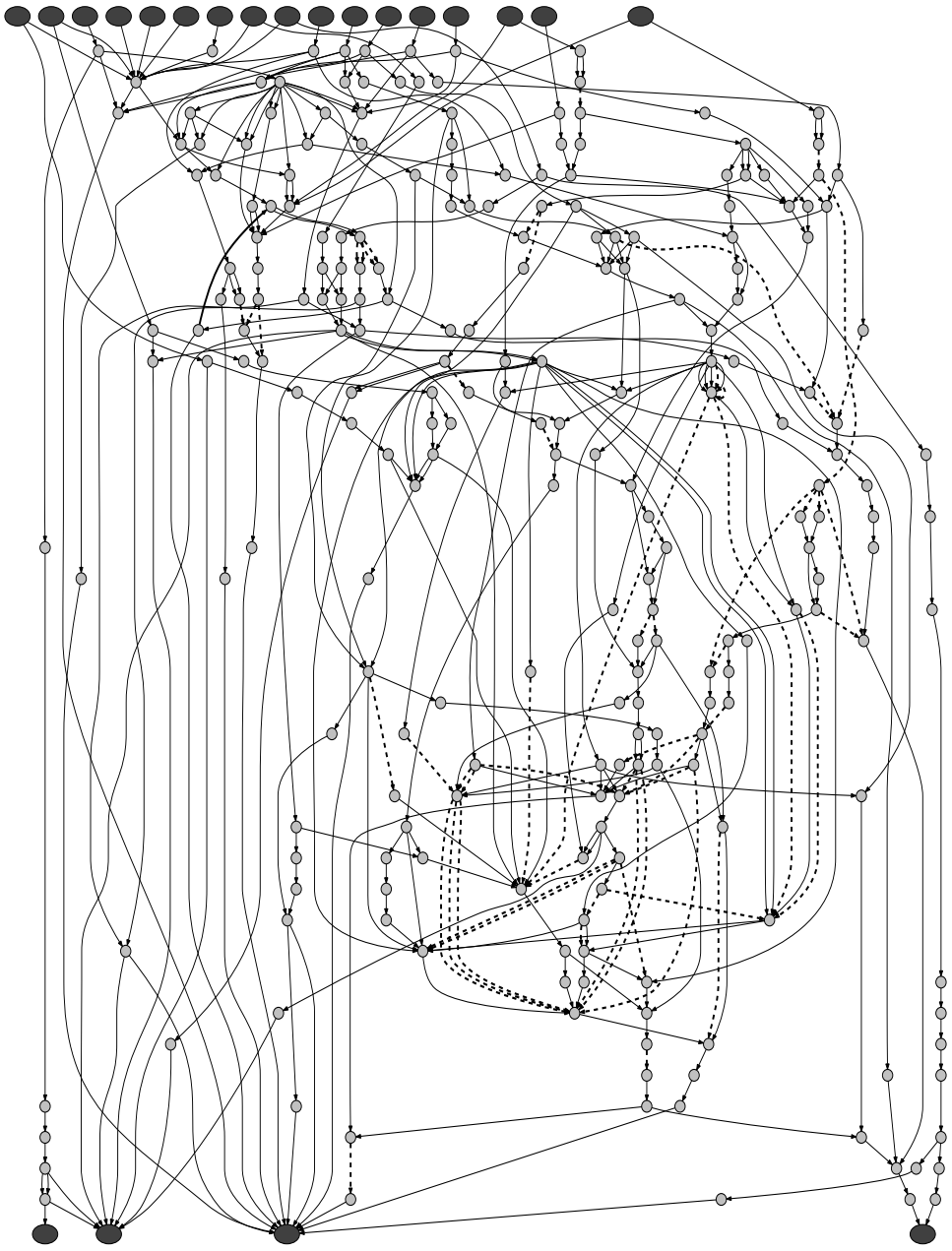

Figure 6.8: **The ext2 Allocation-Relevant Call Graph.** *The figure plots the ARCG for the ext2 file system. The larger gray nodes represent entry points into the file system, and the black nodes are different memory allocators called by ext2 either directly or indirectly. The smaller gray nodes represent functions in ext2 and the kernel, and dotted lines between such functions indicate the presence of a loop.*

through other routines. To identify goto-based loops, AMA uses the line numbers of the labels that the goto statements point to. To identify both recursions and function-call based loops, AMA performs a DFS on the ARCG and for every function encountered during the search, it checks if the function has been explored before. Once these loops are identified, the tool searches for and outputs the expressions that affect termination.

Phase 3: Slicing and Backtracking

The goal of this next step is to perform a bottom-up crawl of the graph, and produce a minimized call graph with only the memory-relevant code left therein. We use a form of backward slicing [190] to achieve this end.

In our current prototype, the AMAlyzer only performs a bottom-up crawl until the beginning of each function. In other words, the slicing is done at the function level and developer involvement is required to perform backtracking. To backtrack until the beginning of a system call, the developer has to manually use the output of slicing for each function (including the dependent input variables that affect the allocation size/count) and invoke the slicing routine on its caller functions. The caller functions are identified using the ARCG.

6.4.2 AMAlyzer Summary

As we mentioned above, the final output is a skeletal graph which can be used by the developer to arrive at the final pre-allocations with the help of slicing support in the AMAlyzer. For ext2-mfr, the reduction in code is dramatic: from nearly 200,000 lines of code across 2000 functions (7000 function calls) down to less than 9,000 lines across 300 functions (400 function calls), with all relevant variables highlighted. Arriving upon the final pre-allocation amounts then becomes a straightforward process.

Table 6.3 summarizes the results of our efforts. In the table, we present the parameterized memory amounts that must be pre-allocated for the 13 most-relevant entry points into the file system. From the table, we can see that number of allocated objects depend on the input paramenters (such as *NameLength*), format-time parameters (such as *Worst(Bitmap)*), and size of files or directories that are currently being accessed (such as *Size(ParentDir)*).

Entry point	**P**re-allocation required
truncate()	$(Worst(Bitmap) + Worst(Indirect)) \times (PageSize + BufferHead)$
lookup()	$(1 + Size(ParentDir)) \times (PageSize + BufferHead) + Inode + Dentry + NameLength + NamesCache$
lookuphash()	$(1 + Size(ParentDir)) \times (PageSize + BufferHead) + Inode + Dentry + NameLength + Filp$
sysopen()	$lookup() + lookuphash() + (4 + Depth(Inode) + Worst(Bitmap)) \times PageSize + (5 + Depth(Inode) + Worst(Bitmap)) \times BufferHead + Inode + truncate()$
sysread()	$(count + ReadAhead + Worst(Bitmap) + Worst(Indirect)) \times (PageSize + BufferHead)$
syswrite()	$(count + Worst(Bitmap)) \times (PageSize + BufferHead) + sizeof(ext2\_block\_allocinfo)$
mkdir()	$lookup() + lookuphash() + (Depth(ParentInode) + 4) \times PageSize + (Depth(Inode) + 8) \times BufferHead$
unlink()	$lookup() + lookuphash() + (1 + Depth(Inode)) \times (PageSize + BufferHead)$
rmdir()	$lookup() + lookuphash() + (3 + Depth(Inode)) \times (PageSize + BufferHead)$
access()	$lookup() + NamesCache$
chdir()	$lookup() + NamesCache$
chroot()	$lookup() + NamesCache$
statfs()	$lookup() + NamesCache$

Table 6.3: **Pre-Allocation Requirements for ext2-mfr.** *The table shows the worst-case memory requirements of the various system calls in terms of the kmem_cache, kmalloc, and page allocations. The following types of kmem_cache are used:* $NamesCache$ *(4096 bytes),* $BufferHead$ *(52 bytes),* $Inode$ *(476 bytes),* $Filp$ *(128 bytes), and* $Dentry$ *(132 bytes). The* $PageSize$ *is constant at 4096 bytes. The other terms used above include:* $Count$*: the number of blocks read/written,* $ReadAhead$*: the number of read-ahead blocks,* $Worst(Bitmap)$*: the number of bitmap blocks that needs to be read,* $Worst(Indirect)$*: the number of indirect blocks to be read for that particular block,* $Depth(inode)$*: the maximum number of indirect blocks to be read for that particular inode, and* $Size(inode)$*: the number of pages in the inode.*

6.4.3 Optimizations

As we transformed ext2 into ext2-mfr, we noticed a number of opportunities for optimization, in which we could reduce the amount of memory pre-allocated along some paths. We now describe two novel optimizations.

Cache Peeking

The first optimization, *cache peeking*, can greatly reduce the amount of pre-allocated memory. An example is found in code paths that access a file block (such as a `sys_read()`). To access a file block in a large file, it is possible that a triple-indirect, double-indirect, and indirect block, inode, and other blocks may need to be accessed to find the address of the desired block and read it from disk.

With repeated access to a file, such blocks are likely to be in the page cache. However, the pre-allocation code must account for the worst case, and thus in the normal case must pre-allocate memory to potentially read those blocks. This pre-allocation is often a waste, as the blocks will be allocated, remain unused during the call, and then finally be freed by AMA.

With cache peeking, the pre-allocation code performs a small amount of extra work to determine if the requisite pages are already in cache. If so, it pins them there and avoids the pre-allocation altogether; upon completion, the pages are unpinned.

The pin/unpin is required for this optimization to be safe. Without this step, it would be possible that a page gets evicted from the cache after the pre-allocation phase but before the use of the page, which would lead to an unexpected memory allocation request downstream. In this case, if the request then failed, AMA would not have served its function in ensuring that no downstream failures occur.

Cache peeking works well in many instances as the cached data is accessible at the beginning of a system call and does not require any new memory allocations. Even if cache peeking requires additional memory, the memory allocation calls needed for cache peeking can be easily performed as part of the pre-allocation phase.

Page Recycling

A second optimization we came upon was the notion of *page recycling*. The idea for the optimization arose when we discovered that ext2 often uses far more pages than needed for certain tasks (such as file/directory truncates, searches on free/allocated entries inside block bitmaps and large directories).

For example, consider truncate. In order to truncate a file, one must read every indirect block (and double indirect block, and so forth) into memory to know which blocks to free. In ext2, each indirect block is read into memory and given its own page; the page holding an indirect block is quickly discarded, after ext2 has freed the blocks pointed to by that indirect block.

To reduce this cost, we implement page recycling. With this approach, the pre-allocation phase allocates the minimal number of pages that need to be in memory during the operation. For a truncate, this number is proportional to the depth of the indirect-block tree, instead of the size of the entire tree. Instead of allocating thousands of blocks to truncate a file, we only allocate a few (for the triple-indirect, a double indirect, and an indirect block). When the code has finished freeing the current indirect block, we recycle that page for the next indirect block instead of adding the page back to the LRU page cache, and so forth. In this manner, substantial savings in memory is made possible.

6.4.4 Limitations and Discussion

We now discuss some of the limitations of our anticipatory approach. Not all pieces are yet automated; instead, the tool currently helps turn the intractable problem of examining 180,000 lines of code into a tractable one providing a lot of assistance in finding the correct pre-allocations. Further work is required in slicing and backtracking to streamline this process, but is not the focus of our current effort: rather our goal here is to demonstrate the feasibility of the anticipatory approach.

The anticipatory approach could fail requests in cases where normal execution would successfully complete. Normal execution need not always take the worst case (or longest) path. As a result, normal execution might be able to complete with fewer memory allocations than the anticipatory approach. In contrast, the anticipatory approach must always allocate memory for the worst case scenario, as it cannot afford to fail on a memory allocation call after the pre-allocation phase.

Cache peeking can only be used when sufficient information is available at the time of allocation to determine if the required data is in the cache. For file systems, sufficient information is available at the beginning of a system call, which allows cache peeking to avoid pre-allocation with little implementation effort. More implementation effort could be required in other systems to determine if the required data is in its cache.

6.5 The AMA Run-Time

The final piece of AMA is the runtime component. There are two major pieces to consider. First is the pre-allocation itself, which is inserted at every relevant entry point in the kernel subsystem of interest, and the subsequent cleanup of pre-allocated memory. Second is the use of the pre-allocated memory, in which the run-time must transparently redirect allocation requests (such as `kmalloc()`) to use the pre-allocated memory. We discuss these in turn, and then present the other run-time decision a file system such as Linux ext2-mfr must make: what to do when a pre-allocation request fails?

6.5.1 Pre-allocating and Freeing Memory

For pre-allocation, we require that file systems implement a single new VFS-level call, which we call `vfs_get_mem_requirements()`. This call takes as arguments information about which call is about to be made, any relevant arguments about the current operation (such as the file position) and state of the file system, and then returns a structure to the caller (in this case, the VFS layer) that describes all of the necessary allocations that must take place. The structure is referred to as the *anticipatory allocation description (AAD)*.

The VFS layer unpacks the AAD, allocates memory chunks (perhaps using different allocators) as need be, and links them into the task structure of the calling process for downstream use (described further below). With the pre-allocated memory in place, the VFS layer then calls the desired routine (such as `vfs_read()`), which then utilizes the pre-allocated memory during its execution. When the operation completes, a generic AMA cleanup routine is called to free any unused memory.

To give a better sense of this code flow, we provide a simplified example from the `read()` system call code path in Figure 6.9. Without the AMA additions, the code simply looks up the current file position (*i.e.*, where to read from next), calls into `vfs_read()` to do the file-system-specific read, updates the file offset, and returns. As described in the original VFS paper [100], this code is generic across all file systems.

With AMA, two extra steps are required, as shown in the figure. First, the VFS layer checks if the underlying file system is using AMA, and if so, calls the file system's `vfs_get_mem_requirements()` routine to determine the pending call's memory requirements. The VFS layer then allocates the needed memory using the AAD returned from the `vfs_get_mem_requirements()`. All of this work is neatly encapsulated by the `AMA_CHECK_AND_ALLOCATE()` call in the figure.

```
SYSCALL_DEFINE3(read, ...) {
   ...
   loff_t pos = file_pos_read(file);
   err = AMA_CHECK_AND_ALLOCATE(file, AMASYS_READ, pos, count);
   if (err)
        ...
   ret = vfs_read(file, buf, count, &pos);
   file_pos_write(file, pos);
   ...
   AMA_CLEANUP();
}

 int AMA_CHECK_AND_ALLOCATE(..., int syscallno, ...) {
     struct relevant_arguments *ra;
     struct anticipatory_allocation_description *aad;
     ...
      err = sb->vfs_get_mem_requirements(syscallno, ra, aad);
     if (err)
        return err;
     else
         return allocate_memory(aad);
}
```

Figure 6.9: **A VFS Read Example.** *This code snippet shows how pre-allocation happens during a read system call. Pre-allocation happens at the beginning of a system call and the call continues executing only if the pre-allocation (i.e., the AMA_CHECK_AND_ALLOCATE function) succeeds.*

Second, after the call is complete, a cleanup routine `AMA_CLEANUP()` is called. This call is required because the AMAlyzer provides us with a worst-case estimate of possible memory usage, and hence not all pre-allocated memory is used during the course of a typical call into the file system. In order to free this unused memory, the extra call to `AMA_CLEANUP()` is made.

6.5.2 Using Pre-allocated Memory

Central to our implementation is *transparency*; we do not change the specific file system (ext2) or other kernel code to explicitly use or free pre-allocated memory. File systems and the rest of the kernel thus continue to use regular memory-allocation routines and pre-allocated memory is returned back during such allocation calls.

To support this transparency, we modified each of the kernel allocation routines as follows. Specifically, when a process calls into ext2-mfr, the pre-allocation code (in `AMA_CHECK_AND_ALLOCATE()` above) sets a new flag within the per-task task structure. This *anticipatory flag* is then checked upon each entry into any kernel memory-allocation routine. If the flag is set, the routine attempts to use pre-allocated memory and returns one of the pre-allocated chunks; if the flag is not set, the normal allocation code is executed (and failure is a possibility). Calls to `kfree()` and other memory-releasing routines operate as normal, and thus we leave those unchanged.

Allocation requests are matched with the pre-allocated objects using the parameters passed to the allocation call at runtime. The parameters passed to the allocation call are *size*, *order* (or *the cachep pointer*), and *the GFP flag*. The type of the desired memory object is inferred through the invocation of the allocation call at runtime. The size (for kmalloc and vmalloc) or order (for alloc_pages) helps to exactly match the allocation request with the pre-allocated object. For cache objects, the cachep pointer help identify the correct pre-allocated object.

One small complication arises during interrupt handling. Specifically, we do not wish to redirect memory allocation requests to use pre-allocated memory when requested by interrupt-handling code. Thus, when interrupted, we take care to save the anticipatory flag of the currently-running process and restore it when the interrupt handling is complete.

6.5.3 What If Pre-Allocation Fails?

Adding pre-allocation into the code raises a new policy question: how should the code handle the failure of the pre-allocation itself? We believe there are a number of different policy alternatives, which we now describe:

- **Fail-immediate.** This policy immediately returns an error to the caller (such as ENOMEM).

- **Retry-forever (with back-off).** This policy simply keeps retrying forever, perhaps inserting a delay of some kind (*e.g.*, exponential) between retry requests to reduce the load on the system and control better the load on the memory system. Also, such delays could help make progress in a heavily contended system.

- **Retry-alternate (with back-off).** This form of retry also requests memory again, but uses an alternate code path that uses less memory than the original

through page/memory recycling and thus is more likely to succeed. This retry can also back-off as need be.

Using AMA to implement these policies is superior to the existing approach, as it enables *shallow recovery*, immediately upon entry into the subsystem. For example, consider the fail-immediate option above. Clearly this policy *could* be implemented in the traditional system without AMA, but in our opinion doing so is prohibitively complex. To do so, one would have to ensure that the failure was propagated correctly all the way through the many layers of the file system code, which is difficult [71, 151]. Further, any locks acquired or other state changes made would have to be undone. Deep recovery is difficult and error-prone; shallow recovery is the opposite.

Another benefit that the shallow recovery of AMA permits is a unified policy [69]. The policy, whether failing immediately, retrying, or some combination, is specified in one or a few places in the code. Thus, the developer can easily decide how the system should handle such a failure and be confident that the implementation meets that desire.

A third benefit of our approach: file systems could expose some control over the policy to applications [7, 51]. Whereas most applications may not be prepared to handle such a failure, a more savvy application (such as a file server or database) could set the file system to fail-fast and thus enable better control over failure handling.

Pre-allocation failure is not a panacea, however. Depending on the installation and environment, the code that handles pre-allocation failures will possibly run quite rarely, and thus may not be as robust as normal-case code. Although we believe this to be less of a concern for pre-allocation recovery code (because it is small, simple, and usually correct "by inspection"), further efforts could be applied to harden this code. For example, some have suggested constant "fire drilling" [27] as a way to ensure operators are prepared to handle failures; similarly, one could regularly fail kernel subsystems (such as memory allocators) to ensure that this recovery code is run.

6.6 Analysis

We now analyze Linux ext2-mfr. We measure its robustness under memory-allocation failure, as well as its baseline performance. We further study its space overheads, exploring cases where our estimates of memory-allocation needs could be overly conservative, and whether the optimizations introduced earlier are effective in re-

	Process State		File-System State	
	Error	Abort	Unusable	Inconsistent
ext2-mfr$_{10}$	0	0	0	0
ext2-mfr$_{50}$	0	0	0	0
ext2-mfr$_{99}$	0	0	0	0

Table 6.4: **Fault Injection Results: Retry.** *The table shows the reaction of the Linux ext2-mfr file system to memory failures as the probability of a failure increases. The file system uses a "retry-forever" policy to handle each failure. A detailed description of the experiment is found in Table 6.2.*

ducing these overheads. All experiments were performed on a 2.2 GHz Opteron processor, with two 80GB WDC disks, 2GB of memory, running Linux 2.6.32.

6.6.1 Robustness

Our first experiment with ext2-mfr reprises our earlier fault injection study found in Table 6.2. In this experiment, we set the probability that the memory-allocation routines will fail to 10%, 50%, and 99%, and observe how ext2-mfr behaves both in terms of how processes were affected as well as the overall file-system state. For this experiment, the retry-forever (without any back-off) policy is used. Table 6.4 reports our results.

As one can see from the table, ext2-mfr is highly robust to memory allocation failure. Even when 99 out of 100 memory-allocation calls fail, ext2-mfr is able to retry and eventually make progress. This application never notices that the failures are occurring, and file system usability and state remain intact.

6.6.2 Performance

In our next experiment, we study the performance overheads of using AMA. We utilize both simple microbenchmarks as well as application-level tests to gauge the overheads incurred in ext2-mfr due to the extra work of memory pre-allocation and cleanup. Table 6.5 presents the results of our study.

From the table, we can see that the performance of our relatively-untuned prototype is excellent across both microbenchmarks as well as application-level workloads. In all cases, the extra work done by the AMA runtime to pre-allocate memory, redirect allocation requests transparently, and subsequently free unused

Workload	ext2 (secs)	ext2-mfr (secs)	Overhead (%)
Sequential Write	13.46	13.69	1.71
Sequential Read	9.04	9.05	0.11
Random Writes	11.58	11.67	0.78
Random Reads	146.33	151.03	3.21
Sort	129.64	136.50	5.29
OpenSSH	48.30	49.80	3.11
PostMark	55.90	59.60	6.62

Table 6.5: **Baseline Performance.** *The baseline performance of ext2 and ext2-mfr (without optimizations) are compared. The first four tests are microbenchmarks: sequential read and write either read or write 1-GB file in its entirety; random read and write read or write 100 MB of data over a 1-GB file. Note that random-write performance is good because the writes are buffered and thus can be scheduled when written to disk. The three application-level benchmarks: are a command-line sort of a 100MB text file; the OpenSSH benchmark which copies, untars, configures, and builds the OpenSSH 4.5.1 source code; and the PostMark benchmark run for 60,000 transactions over 3000 files (from 4KB to 4MB) with 50/50 read/append and create/delete biases. All times are reported in seconds, and are stable across repeated runs.*

memory has a minimal cost. With further streamlining, we feel confident that the overheads could be reduced even further.

6.6.3 Space Overheads and Cache Peeking

We now study the space overheads of ext2-mfr, both with and without our cache-peeking optimization. The largest concern we have about conservative pre-allocation is that excess memory may be allocated and then freed; although we have shown there is little time overhead involved (Table 6.5), the extra space requested could induce further memory pressure on the system, (ironically) making allocation failure more likely to occur. We run the same set of microbenchmarks and application-level workloads, and record information about how much memory was allocated for both ext2 and ext2-mfr; we also turn on and off cache-peeking for ext2-mfr. Table 6.6 presents our results.

From the table, we make a number of observations. First, our unoptimized ext2-mfr does indeed conservatively pre-allocate a noticeable amount more memory than needed in some cases. For example, during a sequential read of a 1 GB file, normal

Workload	ext2 (GB)	ext2-mfr (GB)	ext2-mfr (+peek) (GB)
Sequential Write	1.01	1.01 (1.00x)	1.01 (1.00x)
Sequential Read	1.00	6.89 (6.87x)	1.00 (1.00x)
Random Write	0.10	0.10 (1.05x)	0.10 (1.00x)
Random Read	0.26	0.63 (2.41x)	0.28 (1.08x)
Sort	0.10	0.10 (1.00x)	0.10 (1.00x)
OpenSSH	0.02	1.56 (63.29x)	0.07 (3.50x)
PostMark	3.15	5.88 (1.87x)	3.28 (1.04x)

Table 6.6: **Space Overheads.** *The total amount of memory allocated for both ext2 and ext2-mfr is shown. The workloads are identical to those described in the caption of Table 6.5.*

ext2 allocates roughly 1 GB (mostly to hold the data pages), whereas unoptimized ext2-mfr allocates nearly seven times that amount. The file is being read one 4-KB block at a time, which means on average, the normal scan allocates one block per read whereas ext2-mfr allocates seven. The reason for these excess pre-allocations is simple: when reading a block from a large file, it is *possible* that one would have to read in a double-indirect block, indirect block, and so forth. However, as those blocks are already in cache for these reads, the conservative pre-allocation performs a great deal of unnecessary work, allocating space for these blocks and then freeing them immediately after each read completes; the excess pages are not needed.

With cache peeking enabled, the pre-allocation space overheads improve significantly, as virtually all blocks that are in cache need not be allocated. Cache peeking clearly makes the pre-allocation quite space-effective. The only workload which does not approach the minimum is OpenSSH. OpenSSH, however, places small demand on the memory system in general and hence is not of great concern.

6.6.4 Page Recycling

We also study the benefits of page recycling. In this experiment, we investigate the memory overheads that arise during truncate. Figure 6.10 plots the results.

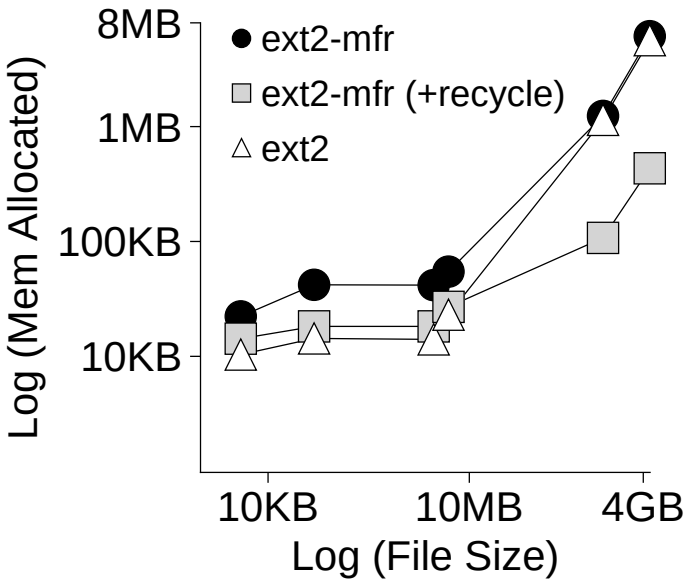

Figure 6.10: **Space Costs with Page Recycling.** *The figure shows the measured space overheads of page recycling during the truncate of a file. The file size is varied along the x-axis, and the space cost is plotted on the y-axis (both are log scales).*

In the figure, we compare the space overheads of standard ext2, ext2-mfr (without cache peeking or page recycling), and ext2-mfr with page recycling (without cache peeking). As one can see from the figure, as the file system grows, the space overheads of both ext2 and ext2-mfr converge, as numerous pages are allocated for indirect blocks. Page recycling obviates the need for these blocks, and thus uses many fewer pages than even standard ext2.

6.6.5 Conservative Pre-allocation

We also were interested in whether, despite our best efforts, ext2-mfr ever under-allocated memory in the pre-allocation phase. Thus, we ran our same set of workloads (i.e., all performance benchmarks) and checked for the same. To identify under-allocated memory, we add a check inside each memory allocation function (such as kmalloc); the check reports an error when objects are not found in the pre-allocated pool during a file-system request. In no run during these experiments

and other stress-tests did we ever encounter an under-allocation, giving us further confidence that our static transformation of ext2 was properly done.

6.6.6 Policy Alternatives

We also were interested in seeing how hard it is to use a different policy to react to allocation failures. Table 6.7 shows the results of our fault-injection experiment, but this time with a "fail-fast" policy which immediately returns to the user should the pre-allocation attempt fail.

	Process State		**File-System State**	
	Error	Abort	Unusable	Inconsistent
ext2-mfr$_{10}$	15	0	0	0
ext2-mfr$_{50}$	15	0	0	0
ext2-mfr$_{99}$	15	0	0	0

Table 6.7: **Fault Injection Results: Fail-Fast.** *The table shows the reaction of Linux ext2-mfr using a fail-fast policy file system. A detailed description of the experiment is found in Table 6.2.*

The results show the expected outcome. In this case, the process running the workload immediately returns the `ENOMEM` error code; the file system remains consistent and usable. By changing only a few lines of code, an entirely different failure-handling behavior can be realized.

6.7 Summary

It is common sense in the world of programming that code that is rarely run rarely works. Unfortunately, some of the most important code in systems falls into this category, including any code that is run during a "recovery". If the problem that leads to the recovery code being enacted is rare enough, the recovery code itself is unlikely to be battle tested, and is thus prone to failure.

In this chapter, we presented Anticipatory Memory Allocation (AMA), a new approach to avoiding memory-allocation failures deep within the kernel. By pre-allocating the worst-case allocation immediately upon entry into the kernel, AMA ensures that requests further downstream will never fail, in those places within the code where handling failure has proven difficult over the years. The small bits of recovery code that are scattered throughout the code need never run, and system robustness is improved by design.

As we build increasingly complex systems, we should consider new methods and approaches that help build robustness into the system by design. AMA presents one method (early resource allocation) to handle one problem (memory-allocation failure), but we believe that the approach could be applied more generally. We believe that the only true manner in which to have working recovery code is to have none at all.

Chapter 7

Related Work

Reliability has been a major focus of computer systems designers since the early days of computers. Initially, hardware failures were orders of magnitude more frequent then they are today [46, 185]. Due to the advances in hardware reliability mechanisms, software has now become the dominant source of system failures [66].

In this thesis, we developed recovery techniques to improve the reliability of file systems through restartability and resource reservation. We now look at previous work that has similar goals of restartability and resource reservation and discuss how our techniques differ from them.

The rest of the chapter is organized as follows. First, in Section 7.1, we review previous work on improving reliability by restarting components on failures. We then look at previous work on improving reliability through failure avoidance via pre-allocation of resources in Section 7.2.

7.1 Reliability through Restartability

Restarting components on failures has been a popular method to survive software failure. We now look at solutions designed to restart kernel- and user-level components on failures.

7.1.1 Restartable OS Components

We now discuss previous systems designed to increasing operating system fault resilience via a restartable approach. We classify previous approaches along two axes: *overhead* and *statefulness*.

We classify fault isolation techniques that incur little overhead as *lightweight* and more costly mechanisms as *heavyweight*. Heavyweight mechanisms are not

likely to be adopted by file systems, which have been tuned for high performance and scalability [23, 82, 170], 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 are referred to as *stateless*; most device driver recovery techniques are stateless. Techniques that can handle components with in-memory and persistent storage are *stateful*; when recovering from file-system failure, stateful techniques are required. We now discuss the previous systems in detail.

The renaissance in building isolated OS subsystems is found in Swift *et al.*'s work on Nooks and subsequently shadow drivers [171, 172]. In Nooks, the authors use memory-management hardware to build an isolation boundary around device drivers; not surprisingly, such techniques incur high overheads [171]. The 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; such an approach cannot be directly applied to file systems.

Device driver isolation can also be obtained through Virtual Machine Monitors (VMM) [52, 104]. In Xen, Fraser et al. show that it is possible to isolate and share device drivers across operating systems by running device drivers in separate virtual machines [52]. In L4, LeVasseur et al. went a step further and show that it is possible to isolate and still run unmodified drivers in their original operating systems in a virtual machine [104]. In both approaches, communication to the device driver happens through the VMM. In the event of an error, only the virtual machine running the buggy device driver is affected. To restart a failed driver, the VMM starts a new virtual machine and initializes the failed device driver to a predefined state. The disadvantages of a VMM-based approach are that one needs to run separate instances of virtual machines for each device driver, the restart mechanism is stateless, and isolation and data copying costs are high.

Isolation and restart of buggy device drivers have also been performed using microkernel-based approaches [79, 80, 88, 103, 194]. In Minix, the device driver is run as a user-mode process by encapsulating it in a private address space that is protected by the MMU hardware. Faults in a device driver do not impact other operating system components, as the driver is run in a separate address space. Upon a driver failure, Minix simply reincarnates the failed driver to service subsequent requests. Moreover, in a subsequent work, Herder et al. also implemented parameterized policy scripts to provide flexibility in the way drivers are restarted after a failure [80].

Nexus is another microkernel-based approach, where drivers are executed as user-level processes [194]. Nexus uses a global, trusted reference validation mechanism [4], device safety specification, and device-specific reference monitor. In Nexus, the safety specification is compiled with the reference monitor. Upon a transition from (or to) the driver, the reference monitor leverages the safety specifications to check driver interactions for permissible and normal behavior.

CuriOS, a recent microkernel-based operating system, also aims to be resilient to subsystem failure [42]. CuriOS achieves this end through classic microkernel techniques (*i.e.*, address-space boundaries between servers) with an additional twist: instead of storing session state inside a service, CuriOS 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 [141]. 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. 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.

In summary, the advantage of micro-kernel-based approaches is that the shared state between drivers and the operating system is small; such small state simplifies recovery by a large extent. The drawback of the micro-kernel-based approaches is that they work well only for stateless systems such as network, block, and character drivers and cannot be applied to stateful components such as file systems. Moreover, most of the commodity operating systems are monolithic kernels and not microkernels [28, 171].

SafeDrive takes a different approach to fault resilience [200]. 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. 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 also cannot be applied directly to file systems.

EROS is a capability-based operating system designed to support security and reliability needs of active systems [156]. EROS provides restartability through

	Heavyweight	Lightweight
Stateless	Nooks/Shadow[171, 172]* Xen[52], Minix[79, 80] L4[104], Nexus[194]	SafeDrive[200]* Singularity[103]
Stateful	CuriOS[42] EROS[156]	Membrane*

Table 7.1: **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.*

three principles: no kernel allocation, atomicity of operations, and a stateless kernel. No kernel allocations ensures that the OS does not explicitly allocate or free resources. Atomicity of operations guarantees that all operations are either completed in bounded time or not executed at all. Finally, the stateless kernel, ensures that all of the kernel state resides in user-allocated storage. In EROS, checkpoint of the entire system is taken periodically, which can be used in the event of an failure. The advantage of this approach is that the entire system can be restored back upon failures. This approach has two major drawbacks: first, one needs to rewrite the entire operating system and other components to adhere to the design principles of EROS; second, the overheads of checkpointing the entire operating-system state are prohibitively expensive to be deployed in commodity systems.

The results of our classification are presented in Table 7.1. 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.

7.1.2 Restartable User-level Components

In the context of restarting user-level components, there has been some significant advances in the past few decades. We now discuss the prior work done in restarting

user-level components and their relevance and applicability to stateful components such as user-level file systems.

Simple, whole program restart was proposed as a first attempt to handle software failures [65, 164]. Restart of an entire program helps programs to survive failures caused due to non-deterministic bugs. The advantage of these solutions is that the restart mechanism is simple and straight forward. The drawback of simple restart is that it is stateless and lossy; requests that arrive between a crash and a restart are discarded in these systems.

Software rejuvenation, an alternative solution to whole program restart, was proposed to reduce the down-time periods of services (or applications) [21, 60, 101]. Software rejuvenation is a proactive approach instead of the commonly used reactive approach. The key idea of software rejuvenation is to periodically rejuvenate (or restart) to a fresh state even if there are no failures, thus eliminating any residual or corrupt in-memory state. The goal of software rejuvenation is to handle corrupt states that could eventually result in resource leaks or deadlocks. Software rejuvenation is not applicable for user-level file systems as the restart mechanism could result in lost updates and have noticeable downtime.

Microreboot was proposed to avoid entire program restarts [30, 31, 129]. The idea of microreboot is a fine-grain restart approach, where individual application components are selectively restarted on a failure. Microreboot accomplishes selective restart by separating process recovery from data recovery. The advantage of microreboot is that the restart times are significantly lower than an entire program restart. The microreboot mechanism has a few drawbacks: first, microreboot works only for stateless components where the application state needs to be recorded in specialized state stores; second, requests need to be idempotent without any side effects; finally, frequently-used resources should be leased. In other words, applications and services must be redesigned and rewritten to work with microreboot mechanisms.

Many general checkpoint and restart mechanisms were proposed to survive failures in the past [24, 47, 144]. The common approach taken in these solution is to checkpoint the program state, rollback the program state on failure, and then re-execute the program after recovery. The checkpoint refers to a consistent state that the system can trust, and can be recorded on disk [34, 91, 105, 187], non-volatile memory [108], or in remote memory [3, 134, 201]. Additional support (such as logging) is needed to deal with messages and in-flight operations [20, 91, 109, 110]. The drawback of these approaches is that they are primarily designed to work with distributed systems, require application rewrite, or both; hence, these approaches are not applicable for stand-alone user-level file systems.

Rx, a checkpoint restart mechanism, was proposed to tolerate application failures statefully [140]. Rx has similar goals as Re-FUSE but is different in its implementation. Rx checkpoints the process state using COW-based techniques and records the application-specific state (such as file handles). On an error, Rx restarts the crashed application and changes the environment (such as allocated memory) in the hope that the failure does not happen again. Though Rx statefully restores the in-memory state of user-level processes, it does not have any mechanisms to restore the on-disk state of processes and is left as future work.

N-version programming is another popular approach to tolerating failures [9, 10, 15, 147]. The idea is very simple; different instances of the software are concurrently run within the same system. The diversity in the software implementation helps to avoid the same failure in all instances. As long as there are sufficient running instances to determine the majority, the system can continue operating even in the presence of failures. The advantage of this approach is that one need not build any checkpoint-restart or any other heavy-weight mechanisms. Recovery blocks is a variant of N-version programming where multiple versions of the same block exist [86, 143]. A block is a unit of execution, and a variant of a block would only be executed if the original blocks encounters an error during its execution. Another related approach to N-version programming is the multi-process model, where the same application instance in run multiple times [175]. Unfortunately, N-version systems and its variants are too expensive to be deployed in the real world. Moreover, such systems have to pay significant performance and storage costs.

In summary, there has been a great deal of work on restarting user-level processes on failures. Unfortunately, these solutions cannot be directly applied to user-level file systems, as recovery mechanisms in file systems need to be stateful and lightweight.

7.2 Reliability Through Reservation

Eager reservation of resources helps prevent allocation failures in systems. We now look at the related work that deals with improving reliability of systems through pre-allocation or reservation of resources. We also compare and contrast the related work with our anticipatory memory allocation approach.

A large body of related work is found in the programming languages community on heap usage analysis, wherein researchers have developed static analyses to determine how much heap (or stack) space a program will use [1, 26, 35, 36, 84, 85, 180, 184]. Determination of heap space enables one to preallocate memory, thus avoiding memory allocation failures during later stages of program execution. The

general use-case suggested for said analyses is in the embedded domain, where memory and time resources are generally quite constrained [35]. Whereas many of the analyses focus on functional or garbage-collected languages, and thus are not directly applicable to our problem domain (i.e., languages that are procedural and require explicit memory management), we do believe that some of the more recent work in this space could be applicable to anticipatory memory allocation. In particular, Chin *et al.*'s work on analyzing low-level code [35] and the live heap analysis implemented by Albert *et al.* [1] are promising candidates for further automating the AMA transformation process.

The more general problem of handling memory bugs has also been investigated in great detail [8, 17, 44, 140, 146]. Berger and Zorn provide an excellent discussion of the range of common problems, including dangling pointers, double frees, and buffer overruns [17]. Many interesting and novel solutions have been proposed, including rolling back and trying again with a small change to the environment (*e.g.*, more padding) [140], using multiple randomized heaps and voting to determine correctness [17], and even returning "made up" values when out-of-bounds memory is accessed [146]. The problem we tackle is both narrower and broader at once: narrower in that one could view the poor handling of an allocation failure as just one class of memory bug; broader in that true recovery from such a failure in a complex code base is quite intricate and reaches beyond the scope of typical solutions to these classic memory bugs.

Our approach of using static analysis to predict memory requirements is similar in spirit to that taken by Garbervetsky et al [59]. Their approach helps to come up with estimates of memory allocation within a given region. Whereas, AMA helps to come up with the estimate of memory allocation for the entire file-system operation. Moreover, their system does not consider the allocations done by native methods or internal allocation performed by the runtime system, and does not handle recursive calls. In contrast, AMA estimates the allocations done by the kernel along with handling recursive calls inside file systems.

Finally, the AMA approach to avoiding memory-allocation failure is reminiscent of the banker's algorithm [45] and other deadlock-avoidance techniques. Indeed, with AMA, one could build a sort of "memory scheduler" that avoids memory over-commitment by delaying some requests until other frees have taken place.

Chapter 8

Future Work and Conclusions

"Program testing can be used to show the presence of bugs, but never to show their absence!" – Edsger Dijkstra

A great deal of research has been done in the design and implementation of local file systems to improve various aspects of it. For example, performance [57, 114, 117, 149], scalability [113, 170], consistency management [53, 56, 155, 182], and indexing support [62] are a few areas that have had significant amount of innovation in the recent past. However, some of the critical aspects of file system design and implementation have not been improved at all. In particular, *recovery* in file systems has been ignored to a large extent.

Recovery is a critical component in file systems, as it is the component that deals with faults within the file system or in other components that the file system interacts with. Unfortunately, the recovery component in file systems is not robust due to the presence of numerous bugs. Researchers have developed many tools in the last decade that use language-based support [71, 92], software engineering [48, 106], model checking [50, 197], static analysis [49, 195], fault injection [13, 15, 69, 138], and symbolic execution [196] to identify bugs in the file system code; most of the bugs identified by these tools are in the recovery component of such systems.

Even though many tools can detect bugs in the file system code they cannot guarantee that file systems are free from them [49]. Moreover, previous works have also shown that even when file system developers are aware of the problems, they do not know how to reproduce, react, or fix them [71, 72, 138]. Hence, we believe that the right approach is to accept the fact that failures are inevitable in file systems; we must learn to cope with failures and not just hope to avoid them.

In this dissertation, we explored two different approaches to improving the reliability of commodity file systems: restartability and reservation. These two ap-

proaches help file systems to survive faults and hence, failures within itself and in other components that they interact with. First, we introduced Membrane, an operating system framework to support restartable kernel-level file systems (Chapter 4). Then, we introduced Re-FUSE, a framework built inside the operating system and FUSE to restart user-level file systems on crashes (Chapter 5). Finally, we presented AMA, a mechanism that combined static analysis, dynamic analysis, and domain knowledge, to simplify recovery code that dealt with memory allocation failures in file systems (Chapter 6).

In this chapter, we first summarize our solutions and results (Section 8.1). We then list a set of lessons learned from years of researching file system reliability (Section 8.2). Finally, we outline future directions where our work can possibly be extended (Section 8.3).

8.1 Summary

This dissertation focused on developing recovery techniques to improve file system reliability and is mainly divided in two parts: reliability through restartability and reliability through reservation. We focus on local file systems due to their ubiquitous presence and the new challenges they present. We now summarize the results in both parts.

8.1.1 Reliability Through Restartability

The first part of this dissertation is about improving file system reliability through restartability. In this work, we focus on reliability of kernel-level and user-level file systems and developed frameworks to statefully restart them upon failures.

Kernel-level File Systems

For kernel-level file systems, we designed and implemented a generic framework (Membrane) inside operating systems to support restartability. Membrane enabled kernel-level file systems to tolerate a wide range of fail-stop faults by selectively restarting the failed file system in a transparent and stateful way. The transparent and stateful restart of kernel-level file systems allows applications to continue executing requests in file systems even in the presence of failures.

Lightweight and stateful restart of kernel-level file systems was difficult to implement with existing techniques. To solve this problem, we came up with three novel techniques: Generic COW-based checkpointing, page stealing, and a

skip/trust unwind protocol. Our generic COW-based checkpointing mechanism enabled low-cost snapshots of file system-state that served as recovery points after a crash with minimal support from existing file systems. The page stealing technique greatly reduced logging overheads of write operations, which would otherwise have increased the time and space overheads. Finally, the skip/trust unwind protocol prevented file-system-induced damage to itself and other kernel components on failures through careful unwind of in-kernel threads via both the crashed file system and kernel proper.

We evaluated Membrane with the ext2, VFAT, and ext3 file systems. Through experimentation,we showed that Membrane enabled existing file systems to crash and recover from a wide range of fault scenarios. We also showed that Membrane has less than 5% overhead across a set of file system benchmarks. Moreover, Membrane achieved 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 improved robustness with complete application transparency; even though the underlying file system had crashed, applications continued to run.

User-level File Systems

For user-level file systems, we designed and implemented a generic framework (Re-FUSE) inside the operating system and FUSE to support restartability. Re-FUSE enabled user-level file systems to tolerate a wide range of fail-stop and transient faults through statefully restart of the entire user-level file system on failures. Re-FUSE also ensured that the applications were oblivious to file-system failures and could continue executing requests in user-level file systems even during recovery.

In Re-FUSE, we added three new techniques to statefully restart user-level file systems. The first was request tagging, which differentiated activities that were being performed on the behalf of concurrent requests; the second was system-call logging, which carefully tracked the system calls issued by a user-level file system and cached their results; the third was non-interruptible system calls, which ensured that no user-level file-system thread was terminated in the midst of a system call. Together, these three techniques enabled Re-FUSE to recover correctly from a crash of a user-level file system by simply re-issuing the calls that the FUSE file system was processing when the crash took place. Additional performance optimizations, including page versioning and socket buffering, were employed to lower the performance overheads.

We evaluated Re-FUSE with three popular file systems, NTFS-3g, SSHFS, and AVFS, which differ in their data-access mechanisms, on-disk structures, and features. Less than ten lines of code were added to each of these file systems to

make them restartable, showing that the modifications required to use Re-FUSE are minimal. We tested these file systems with both micro- and macro-benchmarks and found that performance overheads during normal operations are minimal. The average performance overhead was less than 2% and the worst-case performance overhead was 13%. Moreover, recovery time after a crash is small, on the order of a few hundred milliseconds in our tests.

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

8.1.2 Reliability Through Reservation

The second part of this dissertation is about improving file system reliability through reservation. The reservation was done for in-memory objects that could be allocated while executing file system requests. We focused on kernel-level file systems and developed Anticipatory Memory Allocation (AMA), a mechanism to eliminate the scattered recovery code for memory-allocation failures inside operating systems. As part of AMA, we added few lines of recovery code (around 200) inside a single function to deal with pre-allocation failures. In other words, we showed that it is possible to perform shallow recovery for memory-allocation failures during file system requests. We also added flexible, unified recovery policies (retry forever and fail-fast) on top of it.

To identify and reserve all the in-memory objects required to satisfy file-system requests, we use a combination of static analysis, dynamic analysis, and domain knowledge in file systems. We also added two novel optimizations, *cache peeking* and *page recycling* to reduce the space overheads of AMA. Cache peeking avoids pre-allocation of cached objects by peeking into the in-memory cache before execution and page recycling reuses one or more of the pre-allocated pages during a request, and thus reduces the space overheads.

We demonstrated the benefits of AMA by applying it to the Linux ext2 file system and built a memory-failure robust version of ext2 called ext2-mfr. Through experimentation, we showed that ext2-mfr is robust to memory-allocation failure; even for a memory-allocation failure probability of .99, the ext2-mfr is able to either retry and eventually make progress or fail-quickly and return error depending on the specified failure policy. In all cases, AMA ensured that file system and the operating system are consistent and usable. We also showed that ext2-mfr has less with 7% performance overheads and 8% space overheads for commonly-used

micro- and macro-benchmarks. Further, little code change is required, thus demonstrating the ease of transforming a significant subsystem.

Overall, we showed that AMA achieves its goal of tolerating memory-allocation failure, and thus altogether avoids of one important class of recovery bug commonly found in kernel code.

8.2 Lessons Learned

In this section, we present a list of general lessons we learned while working on this dissertation.

- **Recovery as a first class citizen.** Traditionally, systems have been designed and built with performance as the main goal. As a result, recovery features are not designed carefully and are added as an after thought. For example, we have shown that, in btrfs, a newly evolving file system, all necessary recovery components needed to handle memory-allocation failure have not yet been built (Section 6.2). We have also observed that, in the file systems, recovery features are scattered and buried deep within the code. The complex designs combined with the bias for performance make both the intent and the realization of recovery in file systems difficult to evolve.

- **Hardened operating-system interfaces.** Operating systems consist of many components, and these components need to interact with one another to complete application requests. Unfortunately, many of these components blindly trust each other, and as a result, the interface between these layers is not hard as it should be. For example, in Membrane, we showed that the operating system does not check parameters and return values from file systems at many places; this resulted in corruption of operating-system state.

  To build robust operating systems, we need to clearly specify the intent (or semantics) of each operation, its input and output parameters, and expected return values [88]. The operating-system proper or its components should use such information to check and validate the parameter during its interaction with other components to prevent bugs from silently corrupting its state.

- **Interfaces to support fault injection.** For easier reliability testing, systems should provide suitable interfaces that enable a variety of fault-injection scenarios. For our reliability experiments, such interfaces would have helped greatly. In our experience, to perform our fault-injection experiments, we had to change a considerable amount of operating system and file system

code. More specifically, we had to modify different file system components (such as namespace management), virtual file system layer, memory management layer, block driver layer, and the journaling layer to return failures during our fault injection experiments.

8.3 Future Work

In terms of future work, our vision is to build highly-reliable and highly-available systems. This section outlines various directions for this vision.

8.3.1 Reliability through Automation

Reliability research in the last decade has shown time and again that recovery code in systems is insufficient, missing, or incorrect [37, 49, 68, 71, 138, 151, 171, 195–197]. From our own experience in building recovery techniques in the operating system, we believe that the right approach to improving operating system reliability is to automate the recovery process to the largest possible extent. The automation would help eliminate the need for manual implementation of recovery code in operating systems. More importantly, automation could be independent of components (such as file systems, device drivers, etc.) and resources (such as memory, I/O, etc.).

Previous research that had similar goals built transactional support inside operating systems [75, 124, 135, 136]. In these solutions, the transactional support tracks the changes performed during request execution; on an error, the transactional mechanism automatically reverts all recent changes. There are a few drawbacks of these approaches that make them less attractive in commodity systems. First, these system are heavyweight. Second, they still rely on programmers to correctly invoke and implement all of the transactional code. Third, they do not handle all possible errors; for example, memory-allocation failures or I/O errors during transactions are not handled in TxOS [136]. Finally, they require wide-scale changes to the operating-system code.

Unlike previous approaches, we would like to explore the possibility of building transactional support outside the operating system. Specifically, we would like to explore the possibility of leveraging the support available in hardware transactional memory to automate the recovery process in operating systems [118]. Our vision of hardware-assisted recovery is as follows. First, during regular operations, the hardware could automatically log the changes done in the context of a request [81, 118, 142]; to prevent corruption, the hardware could restrict the operat-

ing system from accessing the memory locations of the logs. Second, for atomicity and isolation, the hardware could eagerly detect and resolve conflicts on updates to shared operating system data structures [118]. Third, on an error, the hardware would automatically revert the actions of the request using its log and either retry the request or directly return an error back to the application with little support from the operating system [150]. Finally, to optimize the recovery time overheads, we would like to explore the possibility of adding support for implicit (at the function boundary) and explicit checkpoints (requested by the operating system).

There are many advantages of automating recovery using a hardware-based mechanism. First, a large portion of the recovery code no longer needs to be implemented by operating system developers and would be automated in the hardware layer. Second, the recovery process is transparent to the applications and the operating system. Third, it is possible to get near-native performance, as fault anticipation and recovery can be performed at the hardware layer. Fourth, it is easy for the operating system to implement flexible recovery policies on top a hardware layer. Finally, detection of errors at the operating-system level could be an optimization and not an requirement for many error scenarios (e.g., null pointer exception).

8.3.2 Other Data Management Systems

While we limit the focus of this dissertation to local file systems, several of the issues we discuss here such as fault-tolerance and recovery are applicable to other data management systems such as distributed file systems and database management systems. The solutions developed in this dissertation could be applied to other systems to improve their reliability. First, the skip/trust unwind protocol could help recover and cleanup systems that have intricate interactions between components (or layers). Second, non-interruptible system calls could be used in multi-threaded systems that require stronger atomicity guarantees. Third, the static analysis technique and runtime support developed in AMA could be used to estimate and pre-allocate memory in systems (such as databases) that perform many dynamic memory allocations.

8.4 Closing Words

"Simplicity is prerequisite for reliability."

– Edsger Dijkstra

Availability is important in all systems and data availability is of utmost importance. File systems will become more powerful and complex in the future, and

reliability of file systems will directly determine data availability. In this dissertation, we have adhered to three important principles that helped us face the colossal challenge of building practical recovery techniques to improve file system reliability.

First, *simplicity should be the pursuit in achieving reliable systems*. Recovery code in operating systems is complex, difficult to get right, and is in the order of thousands of lines of code written in a low-level language such as C. To make things worse, the recovery code is scattered all over the operating system and file system, making it hard to reason or verify either manually or using sophisticated techniques such as static analysis. As Dijkstra said (quoted above), we aimed at designing simple recovery mechanisms such that they are easy to reason about and verify.

Second, *reliability need not come at the cost of performance*. File systems have been tuned for high performance and scalability [23, 82, 170], hence, heavyweight reliability techniques are not likely to be adopted by them. In all our solutions, we made a conscious effort to minimize the performance overheads through our design and added optimizations, when possible, to reduce the overheads further.

Third, *favor generality over particularity*. There are a gamut of user-level and kernel-level file systems that exist today. Solutions that are tailored to a particular system tend to be limited in their applicability and require wide scale design and code changes; hence, are not widely adopted in commodity systems. In all our solutions, we favor generality and backward compatibility, in the hope that developers will adopt our recovery techniques to improve the reliability of commodity operating systems and file systems.

Bibliography

[1] Elvira Albert, Samir Genaim, and Miguel Gomez-Zamalloa. Live Heap Space Analysis for Languages for Garbage Collection. In *International Symposium on Memory Management (ISMM '09)*, Dublin, Ireland, June 2009.

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

[3] Cristiana Amza, Alan L. Cox, and Willy Zwaenepoel. Data replication strategies for fault tolerance and availability on commodity clusters. In *In Proceedings of the International Conference on Dependable Systems and Networks*, 2000.

[4] J. Anderson. Computer security technology planning study volume ii. *USAF*, 1972.

[5] T.E. Anderson, M.D. Dahlin, J.M. Neefe, D.A. Patterson, and R.Y. Wang. Serverless Network File Systems. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 109–26, Copper Mountain Resort, Colorado, December 1995.

[6] Apple. Technical Note TN1150. http://developer.apple.com/technotes/tn/tn1150.html, March 2004.

[7] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Nathan C. Burnett, Timothy E. Denehy, Thomas J. Engle, Haryadi S. Gunawi, James Nugent, and Florentina I. Popovici. Transforming Policies into Mechanisms with Infokernel. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 90–105, Bolton Landing, New York, October 2003.

[8] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient Detection of All Pointer and Array Access Errors. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI '04)*, pages 290–301, Washington, DC, June 2004.

[9] A. Avizienis. The n-version approach to fault-tolerant software. *IEEE Trans. Softw. Eng.*, 11:1491–1501, December 1985.

[10] A. Avizienis and L. Chen. On the implementation of n-version programming for software fault tolerance during execution. In *Proceedings of the 1st International Computer Software and Applications Conference*, 1977.

[11] Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Dependability Analysis of Virtual Memory Systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '06)*, Philadelphia, Pennsylvania, June 2006.

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

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

[14] Lakshmi N. Bairavasundaram, Meenali Rungta, Nitin Agrawal, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Systematically Benchmarking the Effects of Disk Pointer Corruption. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '08)*, Anchorage, Alaska, June 2008.

[15] Lakshmi N. Bairavasundaram, Swaminathan Sundararaman, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Tolerating File-System Mistakes with EnvyFS. In *Proceedings of the USENIX Annual Technical Conference (USENIX '09)*, San Diego, California, June 2009.

[16] Mary Baker, John Hartman, Martin Kupfer, Ken Shirriff, and John Ousterhout. Measurements of a Distributed File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 198–212, Pacific Grove, California, October 1991.

[17] Emery D. Berger and Benjamin G. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI '06)*, Ottawa, Canada, June 2006.

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

[19] Eric J. Bina and Perry A. Emrath. A Faster fsck for BSD Unix. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '89)*, San Diego, California, January 1989.

[20] Kenneth P. Birman. *Building secure and reliable network applications, chapter 19.* Manning Publications Co., Greenwich, CT, USA, 1997.

[21] Andrea Bobbio and Matteo Sereno. Fine grained software rejuvenation models. *Computer Performance and Dependability Symposium, International*, 0:4, 1998.

[22] Jeff Bonwick. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *Proceedings of the USENIX Summer Technical Conference (USENIX Summer '94)*, Boston, Massachusetts, June 1994.

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

[24] Anita Borg, Jim Baumbach, and Sam Glazer. A message system supporting fault tolerance. In *Proceedings of the ninth ACM symposium on Operating systems principles*, SOSP '83, pages 90–99, New York, NY, USA, 1983. ACM.

[25] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, 2006.

[26] V. Braberman, F. Fernandez, D. Garbervetsky, and S. Yovine. Parametric Prediction of Heap Memory Requirements. In *International Symposium on Memory Management (ISMM '08)*, Tucson, Arizona, June 2008.

[27] Aaron B. Brown and David A. Patterson. To Err is Human. In *EASY '01*, 2001.

[28] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 143–156, Saint-Malo, France, October 1997.

[29] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS '06)*, Alexandria, Virginia, November 2006.

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

[31] 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.

[32] Remy Card, Theodore Ts'o, and Stephen Tweedie. Design and Implementation of the Second Extended Filesystem. In Proceedings of the First Dutch International Symposium on Linux, 1994.

[33] 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.

[34] Yuqun Chen, Kai Li, and James S. Plank. Clip: A checkpointing tool for message passing parallel programs. *SC Conference*, 0:33, 1997.

[35] Wei-Ngan Chin, Huu Hai Nguyen, Corneliu Popeea, and Shengchao Qin. Analysing Memory Resource Bounds for Low-Level Programs. In *International Symposium on Memory Management (ISMM '08)*, Tucson, Arizona, June 2008.

[36] Wei-Ngan Chin, Huu Hai Nguyen, Shengchao Qin, and Martin Rinard. Memory Usage Verification for OO Programs. In *Static Analysis Symposium (SAS '05)*, 2005.

[37] 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.

[38] Wu chun Feng. Making a case for efficient supercomputing. *Queue*, 1:54–64, October 2003.

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

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

[41] Flavin Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34:56–78, February 1991.

[42] 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.

[43] Brian Demsky and Martin Rinard. Automatic Detection and Repair of Errors in Data Structures. In *The 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '03)*, Anaheim, California, October 2003.

[44] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory Safety Without Runtime Checks Or Garbage Collection. In *LCTES '03*, 2003.

[45] E. W. Dijkstra. EWD623: The Mathematics Behind The Banker's Algorithm. Selected Writings on Computing: A Personal Perspective (Springer-Verlag), 1977.

[46] R. H. Doyle, R. A. Meyer, and R. P. Pedowitz. Automatic failure recovery in a digital data-processing system. In *Papers presented at the the March 3-5, 1959, western joint computer conference*, IRE-AIEE-ACM '59 (Western), pages 159–168, New York, NY, USA, 1959. ACM.

[47] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34:375–408, September 2002.

[48] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions . In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI '00)*, San Diego, California, October 2000.

[49] 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.

[50] Dawson Engler and Madanlal Musuvathi. Static Analysis versus Software Model Checking for Bug Finding. In *5th International Conference Verification, Model Checking and Abstract Interpretation (VMCAI '04)*, Venice, Italy, January 2004.

[51] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 251–266, Copper Mountain Resort, Colorado, December 1995.

[52] 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.

[53] Christopher Frost, Mike Mammarella, Eddie Kohler, Andrew de los Reyes, Shant Hovsepian, Andrew Matsuoka, and Lei Zhang. Generalized File System Dependencies. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, Stevenson, Washington, October 2007.

[54] Jason Gait. Phoenix: a safe-in-memory file system. *Communications of the ACM*, 33(1):81–86, January 1990.

[55] Gregory R. Ganger. Blurring the Line Between Oses and Storage Devices. Technical Report CMU-CS-01-166, Carnegie Mellon University, December 2001.

[56] Gregory R. Ganger, Marshall Kirk McKusick, Craig A. N. Soules, and Yale N. Patt. Soft updates: a solution to the metadata update problem in file systems. *ACM Transaction on Computer Systems*, 18:127–153, May 2000.

[57] Gregory R. Ganger and Yale N. Patt. Metadata Update Performance in File Systems. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI '94)*, pages 49–60, Monterey, California, November 1994.

[58] John F. Gantz, Christopher Chute, Alex Manfrediz, Stephen Minton, David Reinsel, Wolfgang Schlichting, and Anna Toncheva. The Diverse and Exploding Digital Universe. www.emc.com/collateral/analyst-reports/diverse-exploding-digitaluniverse.pdf, 2007.

[59] Diego Garbervetsky, Sergio Yovine, Víctor Braberman, Martín Rouaux, and Alejandro Taboada. On transforming java-like programs into memory-predictable code. In *JTRES '09: Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 140–149, New York, NY, USA, 2009. ACM.

[60] Sachin Garg, Antonio Puliafito, Miklos Telek, and Kishor Trivedi. On the analysis of software rejuvenation policies. *Proceedings of the Annual Conference on Computer Assurance*, 1997.

[61] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 29–43, Bolton Landing, New York, October 2003.

[62] Dominic Giampaolo. *Practical File System Design with the Be File System*. Morgan Kaufmann, 1999.

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

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

[65] Jim Gray. Why do computers stop and what can be done about it? In *Symposium on Reliable Distributed Systems*, pages 3–12, 1986.

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

[67] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[68] Weining Gu, Z. Kalbarczyk, Ravishankar K. Iyer, and Zhenyu Yang. Characterization of Linux Kernel Behavior Under Errors. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '03)*, pages 459–468, San Francisco, California, June 2003.

[69] Haryadi S. Gunawi, Vijayan Prabhakaran, Swetha Krishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving File System Reliability with I/O Shepherding. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, pages 283–296, Stevenson, Washington, October 2007.

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

[71] 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.

[72] Philip J. Guo and Dawson Engler. Linux Kernel Developer Responses to Static Analysis Bug Reports. In *Proceedings of the USENIX Annual Technical Conference (USENIX '09)*, San Diego, California, June 2009.

[73] 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.

[74] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson R. Engler. A System and Language for Building System-Specific, Static Analyses. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '02)*, Berlin, Germany, June 2002.

[75] Roger Haskin, Yoni Malachi, Wayne Sawdon, and Gregory Chan. Recovery management in quicksilver. *ACM Transactions on Computer Systems*, 6:82–108, 1988.

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

[77] Val Henson, Zach Brown, Theodore Ts'o, and Arjan van de Ven. Reducing fsck time for ext2 file systems. In *Ottawa Linux Symposium (OLS '06)*, Ottawa, Canada, July 2006.

[78] Val Henson, Arjan van de Ven, Amit Gud, and Zach Brown. Chunkfs: Using divide-and-conquer to improve file system reliability and repair. In *IEEE 2nd Workshop on Hot Topics in System Dependability (HotDep '06)*, Seattle, Washington, November 2006.

[79] 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.

[80] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Failure Resilience for Device Drivers. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '07)*, pages 41–50, Edinburgh, UK, June 2007.

[81] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 253–262, 2006.

[82] 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.

[83] Dave Hitz, James Lau, and Michael Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the 1994 USENIX Winter Technical Conference*, Berkeley, CA, January 1994.

[84] M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First Order Functional Languages. In *The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '03)*, New Orleans, Louisiana, January 2003.

[85] Martin Hofmann and Steffen Jost. Type-based amortised heap-space analysis. In *In ESOP 2006, LNCS 3924*, pages 22–37. Springer, 2006.

[86] James J. Horning, Hugh C. Lauer, P. M. Melliar-Smith, and Brian Randell. A program structure for error detection and recovery. In *Operating Systems, Proceedings of an International Symposium*, pages 171–187, London, UK, 1974. Springer-Verlag.

[87] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[88] Galen C. Hunt, James R. Larus, Martin Abadi, Paul Barham, Manuel Fahndrich, Chris Hawblitzel Orion Hodson, Steven Levi, Nick Murphy, Bjarne Steensgaard, David Tarditi, Ted Wobber, and Brian Zill. An Overview of the Singularity Project. Technical Report 2005-135, Microsoft Research, 2005.

[89] Sitaram Iyer and Peter Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 117–130, Banff, Canada, October 2001.

[90] Weihang Jiang, Chongfeng Hu, Yuanyuan Zhou, and Arkady Kanevsky. Are disks the dominant contributor for storage failures?: A comprehensive study of storage subsystem failure characteristics. *Trans. Storage*, 4:7:1–7:25, November 2008.

[91] David B. Johnson and Willy Zwaenepoel. Recovery in distributed systems using optimistic message logging and check-pointing. *J. Algorithms*, 11:462–491, September 1990.

[92] Rob Johnson and David Wagner. Finding User/Kernel Pointer Bugs With Type Inference. In *Proceedings of the 13th USENIX Security Symposium (Sec '04)*, San Diego, California, August 2004.

[93] Michael B. Jones, Daniela Roşu, and Marcel-Cătălin Roşu. Cpu reservations and time constraints: efficient, predictable scheduling of independent activities. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 198–211, Saint-Malo, France, October 1997.

[94] N. Joukov, A. M. Krishnakumar, C. Patti, A. Rai, S. Satnur, A. Traeger, and E. Zadok. RAIF: Redundant Array of Independent Filesystems. In *Proceedings of 24th IEEE Conference on Mass Storage Systems and Technologies (MSST 2007)*, pages 199–212, San Diego, CA, September 2007. IEEE.

[95] Hannu H. Kari, H. Saikkonen, and F. Lombardi. Detection of Defective Media in Disks. In *The IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems*, pages 49–55, Venice, Italy, October 1993.

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

[97] Kimberly Keeton, Cipriano Santos, Dirk Beyer, Jeffrey Chase, and John Wilkes. Designing for disasters. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, San Francisco, California, April 2004.

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

[99] Gene H. Kim and Eugene H. Spafford. The Design and Implementation of Tripwire: A File System Integrity Checker. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security (CCS '93)*, Fairfax, Virginia, November 1994.

[100] 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.

[101] Nick Kolettis and N. Dudley Fulton. Software rejuvenation: Analysis, module and applications. In *Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, FTCS '95, pages 381–, Washington, DC, USA, 1995. IEEE Computer Society.

[102] 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.

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

[104] J. LeVasseur, V. Uhlig, J. Stoess, and S. Gotz. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, Washington, November 2006.

[105] K. Li, J. F. Naughton, and J. S. Plank. Real-ti concurrent checkpoint for parallel programs. In *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, PPOPP '90, pages 79–88, New York, NY, USA, 1990. ACM.

[106] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, California, December 2004.

[107] David Lie, Andy Chou, Dawson Engler, and David L. Dill. A Simple Method for Extracting Models from Protocol Code. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA '01)*, Goteborg, Sweden, June 2001.

[108] D. E. Lowell and P. M. Chen. Discount Checking: Transparent, Low-Overhead Recovery for General Applications. Technical Report CSE-TR-410-99, University of Michigan, 1999.

[109] David E. Lowell, Subhachandra Chandra, and Peter M. Chen. Exploring failure transparency and the limits of generic recovery. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 20–20, Berkeley, CA, USA, 2000. USENIX Association.

[110] David E. Lowell and Peter M. Chen. Free transactions with rio vista. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, SOSP '97, pages 92–101, New York, NY, USA, 1997. ACM.

[111] 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.

[112] Martin Streicher. Monitor file system activity with inotify. http://www.ibm.com/developerworks/linux/library/l-ubuntu-inotify/index.html, 2008.

[113] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, Laurent Vivier, and Bull S.A.S. The New Ext4 Filesystem: Current Status and Future Plans. In *Ottawa Linux Symposium (OLS '07)*, Ottawa, Canada, July 2007.

[114] Marshall K. McKusick, William N. Joy, Sam J. Leffler, and Robert S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[115] Marshall Kirk McKusick, Willian N. Joy, Samuel J. Leffler, and Robert S. Fabry. Fsck - The UNIX File System Check Program. Unix System Manager's Manual - 4.3 BSD Virtual VAX-11 Version, April 1986.

[116] Microsoft Corporation. http://www.microsoft.com/hwdev/, December 2000.

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

[118] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. Logtm: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pages 254–265, Feb 2006.

[119] Andrew Morton. Re: [patch] jbd slab cleanups. kerneltrap.org/mailarchive/linux-fsdevel/2007/9/19/322280/thread#mid-322280, September 2007.

[120] Madanlal Musuvathi, David Y.W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[121] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: Type-Safe Retrofitting of Legacy Software. *ACM Transactions on Programming Languages and Systems*, 27(3), May 2005.

[122] George C. Necula, Scott McPeak, S. P. Rahul, and Westley Weimer. Cil: An infrastructure for c program analysis and transformation. In *International Conference on Compiler Construction (CC '02)*, pages 213–228, April 2002.

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

[124] Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. Speculative execution in a distributed file system. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, pages 191–205, 2005.

[125] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M Chen, and Jason Flinn. Rethink the Sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 1–16, Seattle, Washington, November 2006.

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

[127] John K. Ousterhout, Herve Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proceedings of the 10th ACM Symposium on Operating System Principles (SOSP '85)*, pages 15–24, Orcas Island, Washington, December 1985.

[128] Nicolas Palix, Gael Thomas, Suman Saha, Christophe Calves, Julia L. Lawall, and Gilles Muller. Faults in linux: ten years later. In *Proceedings of the 45th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*, pages 305–318, Newport Beach, CA, March 2011.

[129] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kiciman, Matthew Merzbacher, David Oppenheimer, Naveen Sastry, William Tetzlaff, Jonathan Traupman, and Noah Treuhaft. Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. Technical Report CSD-02-1175, U.C. Berkeley, March 2002.

[130] David Patterson, Garth Gibson, and Randy Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD Conference on the Management of Data (SIGMOD '88)*, pages 109–116, Chicago, Illinois, June 1988.

[131] David A. Patterson. Availability and maintainability performance: New focus for a new century. In *USENIX Conference on File and Storage Technologies*, 2002.

[132] R. Hugo Patterson, Stephen Manley, Mike Federwisch, Dave Hitz, Steve Kleiman, and Shane Owara. Snapmirror: File-system-based asynchronous mirroring for disaster recovery. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, Berkeley, CA, USA, 2002. USENIX Association.

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

[134] J. S. Plank, K. Li, and M. A. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, October 1998.

[135] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel. LOCUS: A Network Transparent, High Reliability Distributed System. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP '81)*, pages 169–177, Pacific Grove, California, December 1981.

[136] Donald E. Porter, Owen S. Hofmann, Christopher J. Rossbach, Alexander Benn, and Emmett Witchel. Operating Systems Transactions. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI '08)*, San Diego, California, December 2008.

[137] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '05)*, pages 105–120, Anaheim, California, April 2005.

[138] 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.

[139] Martin De Prycker. *Asynchronous Transfer Mode: Solution for the Broadband ISDN*. Prentice Hall PTR, 1994.

[140] 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.

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

[142] Hany E. Ramadan, Christopher J. Rossbach, Donald E. Porter, Owen S. Hofmann, Aditya Bhandari, and Emmett Witchel. Metatm/txlinux: transactional memory for an operating system. In *Proceedings of the 34nd Annual International Symposium on Computer Architecture (ISCA '07)*, pages 92–103, San Diego, California, USA, June 2007.

[143] B. Randell. System structure for software fault tolerance. In *Proceedings of the international conference on Reliable software*, pages 437–449, New York, NY, USA, 1975. ACM.

[144] B. Randell, P. Lee, and P. C. Treleaven. Reliability issues in computing system design. *ACM Comput. Surv.*, 10:123–165, June 1978.

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

[146] Martin Rinard, Christian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and Jr. William S. Beebe. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, California, December 2004.

[147] Rodrigo Rodrigues, Miguel Castro, and Barbara Liskov. Base: using abstraction to improve fault tolerance. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 15–28, New York, NY, USA, 2001. ACM.

[148] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A comparison of file system workloads. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '00, pages 4–4, Berkeley, CA, USA, 2000. USENIX Association.

[149] 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.

[150] Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Aditya Bh, and Emmett Witchel. Txlinux: Using and managing hardware transactional memory in an operating system. In *In Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*, pages 87–102. ACM, 2007.

[151] Cindy Rubio-Gonzalez, Haryadi S. Gunawi, Ben Liblit, Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. Error Propagation Analysis for File Systems. In *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation (PLDI '09)*, Dublin, Ireland, June 2009.

[152] Fred B. Schneider. Implementing Fault-Tolerant Services Using The State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[153] Bianca Schroeder and Garth Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST '07)*, pages 1–16, San Jose, California, February 2007.

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

[155] Margo I. Seltzer, Gregory R. Ganger, M. Kirk McKusick, Keith A. Smith, Craig A. N. Soules, and Christopher A. Stein. Journaling Versus Soft Updates: Asynchronous Meta-data Protection in File Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '00)*, pages 71–84, San Diego, California, June 2000.

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

[157] Gopalan Sivathanu, Swaminathan Sundararaman, and Erez Zadok. Type-Safe Disks. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, Washington, November 2006.

[158] David A. Solomon. *Inside Windows NT*. Microsoft Programming Series. Microsoft Press, 2nd edition, May 1998.

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

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

[161] Sourceforge. HTTPFS. http://httpfs.sourceforge.net/, 2010.

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

[163] Sourceforge. SSH Filesystem. http://fuse.sourceforge.net/sshfs.html, 2010.

[164] Mark Sullivan and Ram Chillarege. Software defects and their impact on system availability: A study of field failures in operating systems. In *Symposium on Fault-Tolerant Computing*, pages 2–9, 1991.

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

[166] Swaminathan Sundararaman, Sriram Subramanian, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Membrane: Operating System Support for Restartable File Systems. *ACM Transactions on Storage (TOS)*, 6(3):133–170, Sep 2010.

[167] Swaminathan Sundararaman, Laxman Visampalli, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Refuse to Crash with Re-FUSE. In *Proceedings of the EuroSys Conference (EuroSys '11)*, Salzburg, Austria, April 2011.

[168] Swaminathan Sundararaman, Yupu Zhang, Sriram Subramanian, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Making the Common Case the Only Case with Anticipatory Memory Allocation. In *Proceedings of the 9th USENIX Symposium on File and Storage Technologies (FAST '11)*, San Jose, California, February 2011.

[169] Swaminathan Sundararaman, Sriram Subramanian, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Michael M. Swift. Why panic()? Improving Reliability with Restartable File Systems. In *Workshop on Hot Topics in Storage and File Systems (HotStorage '09)*, Big Sky, Montana, October 2009.

[170] 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.

[171] 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.

[172] 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.

[173] 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.

[174] Andrew S. Tanenbaum, Jorrit N. Herder, and Herbert Bos. Can we make operating systems reliable and secure? *Computer*, 39:44–51, 2006.

[175] The Apache Software Foundation. Apache http server project. http://httpd.apache.org, 2010.

[176] The Data Clinic. Hard Disk Failure. http://www.dataclinic.co.uk/hard-disk-failures.htm, 2004.

[177] The Open Source Community. FUSE: File System in Userspace. http://fuse.sourceforge.net/, 2009.

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

[179] 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.

[180] TUGS. StackAnalyzer Stack Usage Analysis. http://www.absint.com/stackanalyzer/, September 2010.

[181] Tuxera. NTFS-3g. http://www.tuxera.com/, 2010.

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

[183] Stephen C. Tweedie. EXT3, Journaling File System. olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html, July 2000.

[184] Leena Unnikrishnan and Scott D. Stoller. Parametric Heap Usage Analysis for Functional Programs. In *International Symposium on Memory Management (ISMM '09)*, Dublin, Ireland, June 2009.

[185] J. von Neumann. Probabilistic logics and synthesis of reliable organisms from unreliable components. In C. Shannon and J. McCarthy, editors, *Automata Studies*, pages 43–98. Princeton University Press, 1956.

[186] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[187] Yi-Min Wang, Yennun Huang, Kiem-Phong Vo, Pe-Yu Chung, and C. Kintala. Checkpointing and its applications. In *Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, FTCS '95, pages 22–, Washington, DC, USA, 1995. IEEE Computer Society.

[188] S. Webber and J. Beirne. The stratus architecture. In *Proceedings of the 21st International Symposium on Fault-Tolerant Computing (FTCS-21)*, pages 79–87, Montreal, Canada, June 1991.

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

[190] Mark Weiser. Program Slicing. In *International Conference on Software Engineering (ICSE '81)*, pages 439–449, San Diego, California, May 1981.

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

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

[193] Torres Wilfredo. Software fault tolerance: A tutorial. Technical Report TM-2000-210616, NASA Langley Technical Report Server, 2000.

[194] 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 Symposium on Operating Systems Design and Implementation (OSDI '08)*, San Diego, California, December 2008.

[195] 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.

[196] Junfeng Yang, Can Sar, Paul Twohey, Cristian Cadar, and Dawson Engler. Automatically Generating Malicious Disks using Symbolic Execution. In *IEEE Security and Privacy (SP '06)*, Berkeley, California, May 2006.

[197] 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.

[198] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. End-to-end Data Integrity for File Systems: A ZFS Case Study. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST '10)*, San Jose, California, February 2010.

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

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

[201] Yuanyuan Zhou, Peter M. Chen, and Kai Li. Fast cluster failover using virtual memory-mapped communication. In *Proceedings of the 13th international conference on Supercomputing*, ICS '99, pages 373–382, New York, NY, USA, 1999. ACM.