Shadow Filesystems: Recovering from Filesystem Runtime Errors via Robust Alternative Execution

Jing Liu, Xiangpeng Hao, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, Tej Chajed

University of Wisconsin – Madison

Abstract

We present *Robust Alternative Execution* (RAE), an approach to transparently mask runtime errors in performance-oriented filesystems via temporarily executing an alternative shadow filesystem. A shadow filesystem has the primary goal of robustness, achieved through a simple implementation without performance optimizations and concurrency while adhering to the same API and on-disk formats as the base filesystem it enhances. While the base performance-oriented filesystem may contain bugs, the shadow implementation is formally verified, leveraging advancements in the verification of low-level systems code. In the common case, the base filesystem executes and delivers high performance to applications; however, when a bug is triggered, the slow-but-correct shadow takes over, updates state correctly, and then resumes the base, thus providing high availability.

CCS Concepts: • **Software and its engineering** → **File systems management**.

Keywords: Filesystems, Reliability, Verification

1 Introduction

Modern filesystems are complex, with numerous bugs and security issues reported every year [34, 41]. Bugs can lead to crashes of the filesystem (and OS), harming system availability and user experience, and potentially leading to data loss or even – in a distributed setting – cascading failures [20, 25].

As a core-component of the OS, Torvalds has stated that the filesystem has a mission to "continue regardless" [35], i.e., to remain available to running applications if at all possible despite encountering a problem. For example, calling panic is discouraged in modern Linux kernel development, with a preference for issuing a warning message and continuing execution [27]. However, in-place failure handling, especially in the context of a complicated concurrent code base, is known to be challenging and error-prone [22, 53]. As such, in many cases, the best approach is simply to crash and recover from known on-disk state, and suffer the resulting loss of availability and related negative consequences.

A major impediment to continued execution after encountering a bug are complex, performance-optimized IO stacks. Filesystem implementations (e.g., ext4, btrfs, xfs) interact and evolve with performance-oriented components throughout the entire IO stack such as the inode cache, dentry cache, and block layer. Numerous performance enhancements have been introduced to those components in recent years, including block-mq, page folios, iomap, io_uring, and polling-mode IO [14, 15, 19, 47]. Moreover, filesystem implementations are highly concurrent, both when accessing in-memory and on-disk structures [43]. As such, bugs are likely to continue to exist in these systems, even in highly tested systems.

In this paper, we make the case for *Robust Alternative Execution* (RAE), a practical approach to improving the reliability of existing complex and performance-oriented filesystems via *shadow filesystems*. Most of the time, the shadow filesystem lies dormant, and the *base filesystem* runs and handles requests. However, when an error is detected in the base, the shadow is invoked. The shadow then performs recovery by (re)executing the problem-inducing operations, generating the necessary state updates, and returning control to the base filesystem.

A shadow filesystem, when paired with a base filesystem, represents a form of N-version programming [4]. However, its goals are different than the base. The shadow prioritizes correctness over performance to realize the simplest possible sequential implementation. Concurrency, caches, asynchronous execution, and other performance optimizations are omitted, as they are not essential.

A shadow filesystem improves the reliability of the filesystem by employing two techniques: *practical formal verification (static time)* and *extensive invariant checks (run time)*. By simplifying a given filesystem to its simplest (non-optimized) form, the implementation of the shadow can be readily verified using modern formal verification tools [33]. Since it is not necessary to reason about caching, multi-threading, and various performance tricks, formally verifying the shadow is practical, sustainable, and the verified code is easy to evolve with the base filesystem. Invariant checking at runtime complements formal verification by checking properties that are difficult to model formally, e.g., hardware failure. A shadow should thus employ extensive checks that are infeasible for performance-oriented base filesystems.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike International 4.0 License.

HOTSTORAGE '24, July 8–9, 2024, Santa Clara, CA, USA

© 2024 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0630-1/24/07

https://doi.org/10.1145/3655038.3665942

Determinism \ Consequence	No Crash	Crash	WARN	Unknown	Total
Deterministic	68	78	11	8	165
Non-Deterministic	31	26	19	7	83
Unknown	5	2	1	0	8

Table 1. Study of filesystem bugs (Linux ext4). Bugs that do not have reproducers, or are related to the interaction with IO (e.g., multiple inflight requests), or are related to threading, are classified as non-deterministic. Bugs are classified as Unknown in their consequence when the commit message does not contain clear clues of external symptoms. The columns present the numbers of bugs according to each consequence. WARN indicates the bug hits a WARN_*() path, the suggested substitute of BUG() in the Linux kernel. We collect the bugs by filtering the ext4's subtree's git log with the mentioning of "bugzilla" or "reported by" (256 bugs in total since 2013).

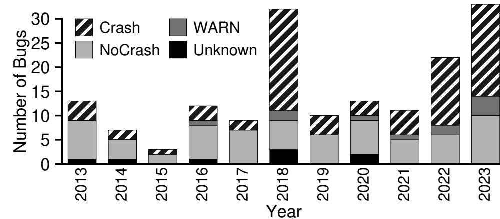

Figure 1. Number of deterministic bugs by the year. Examples of NoCrash consequences include data corruption, performance issue, permission issue, freeze, deadlock, etc.

With these techniques used in its construction, a shadow filesystem enables the base filesystem to tolerate a broad class of bugs, even (potentially) deterministic ones. As such, a shadow filesystem improves availability and robustness of the filesystem while maintaining high performance in the common case.

In the rest of this paper, we first present further background and motivation for shadow file systems. Next, we discuss the basic design of a shadow filesystem, including the fault model, basic approach, and necessary changes to the base. Finally, we discuss the roadmap towards realizing shadows for various modern filesystems, and then conclude.

2 Background and Approach

We present a mini-study of filesystem bugs that motivates the RAE approach for dealing with runtime errors in a unified manner, formulate the main problems, and lay out the key design perspectives and principles.

2.1 Recovering from Filesystem Runtime Errors

Modern high-performance filesystems suffer from hardware faults and software bugs, leading to runtime errors. As a core OS service, filesystem runtime errors may cause service downtime or data corruption.

Prior work shows how to handle (some) non-deterministic faults [7, 10, 17, 23, 48], however, in practice, many faults are deterministic. We study bugs in the Linux ext4 filesystem and categorize them. As shown in Table 1, deterministic bugs are prevalent (165/256), and a significant portion cause crashes or warnings that are detected as runtime errors (89/165). All errors that can be detected are handled by the shadow.

Figure 1 presents the number of deterministic bugs by the year of fixes. More bugs are fixed in recent years for two reasons. First, advances in testing reveal more vulnerabilities, especially in input sanity checks. Second, new kernel features such as blk-mq, page folios, and iomap [14, 15, 19] introduce new bugs. One notable type of deterministic bug occurs when a user mounts a crafted disk image and issues operations to trigger a null-pointer dereference or use-after-free in the kernel [13, 38, 52]; such images can bypass FSCK [26], leading to crashes from malicious attackers.

N-version programming [4–6] (NVP) is a classic approach that can handle deterministic bugs. NVP advocates the independent development of several versions of software with the same specification, running them simultaneously to generate output by combining the decision of each version (via voting). Despite its conceptual advantage of detecting and masking one version's fault, the assumption of statistically independent failures does not usually hold [30]. Further, maintaining and executing multiple versions (often, at least three) incurs excessive overhead.

We propose a different approach – *Robust Alternative Execution* – to handle arbitrary deterministic and non-deterministic runtime errors for a given base filesystem via a shadow filesystem. As shown in Figure 2, after detecting an error, a shadow is launched as the temporary substitute of the base to execute the problematic filesystem operation sequence and return correct metadata updates to the base. While the base is optimized for performance in the common path, the shadow strives for robustness. Such a solution must address the problems of contained reboot, state reconstruction, and error avoidance, as we discuss next.

2.2 Problem Statement

With RAE, our focus is on safe recovery after an error is detected. Figure 3 illustrates the challenges of recovering filesystem errors without affecting applications. The goal is for the shadow to safely re-execute an operation sequence that has triggered errors in the base. At runtime, a filesystem starts from an on-disk state (S_0) and executes a sequence of operations (i.e., Op0 to Op3) issued by applications. When Op3 completes, its effects become visible to the application, but the modifications to metadata and file descriptors may not be persisted due to caching. While executing each operation, other states not visible to the applications (e.g., intermediate

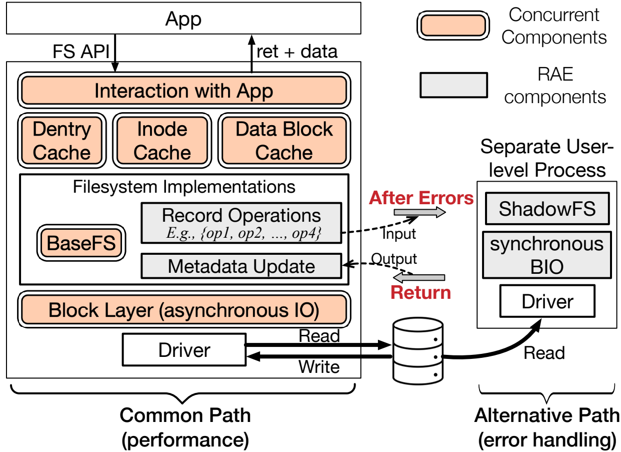

Figure 2. Filesystem architecture with shadow filesystems. The left side shows the constructs of modern concurrent and performance-oriented filesystems; the right side shows the shadow.

data structures like caches, etc.) are also changed. While executing Op4, an error is triggered, failing to reach the expected states of S_5.

We identify three main challenges of recovering from filesystem errors. First, *contained reboot*: the error of the base must not be propagated to user applications and not crash the entire machine. Such a reboot ensures that the erroneous in-memory states of the filesystem are cleaned and the subsequent recovery is based on a well-defined states that can be assumed correct (e.g., on-disk states of S_0). Once an error is detected, all the states in the base filesystem's memory is not trusted, so we need to reset them, including the metadata and file descriptors. For example, in some cases, the states at the point of S_2 could already be erroneous (not detected), though the return value is correct and the application is unaware of the error. The application must not be terminated and its own computation with an up-to-date view of op3's completion must be preserved. Techniques like OS kernel or kernel subsystem's microreboot [17, 48] facilitate contained reboot.

Second, *state reconstruction*: the recovery must ensure the resulting essential filesystem states (i.e., S_4) adhere to the API semantics (e.g., POSIX) for the completed operations, and allows in-flight operations (e.g., Op4) to complete. For instance, the inode number of a file and file descriptor numbers must be identical to the applications for completed operations. These essential states include all on-disk filesystem data structures and file descriptors; however, unessential states may differ (e.g., whether a clean inode is cached in memory). The recovery starts from the trusted-correct states (S_0) and reconstructs the states using the error-triggering operation sequence (from op0 to op4).

Third, *error avoidance*: during reconstruction, the original error and its manifestation path must be circumvented so that final states can be reached (S_5 and others). However, a

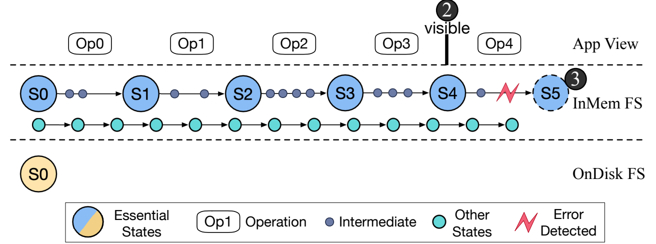

Figure 3. Main problems. ❶: Contained reboot. ❷: States reconstruction. ❸: Error Avoidance. Essential States include the metadata and file descriptors.

fundamental conflict exists between state reconstruction and error avoidance when dealing with deterministic errors [40]: the most straightforward approach for the base to reconstruct states is to re-execute the same sequence of operations, but this would trigger the same error. Returning an error code for the problematic operations is insufficient because the re-executed prefix sequence puts the system into an unknown state such that the base is unsafe to continue.

2.3 A Practical Approach: RAE

We advocate robust alternative execution (RAE), a practical approach to improve the reliability of existing high-performance filesystems [34] via shadow filesystems. The shadow filesystem is an alternative *simple* implementation of the base filesystem that executes operations in the error-handling path, reconstructing states and avoiding errors. Alternative executions for fault tolerance have been explored in device drivers [49, 50], CPU processors [3], and distributed sharding [1]; a shadow filesystem aims to tolerate a broader range of errors (including deterministic bugs) and handle stateful filesystems.

As shown in Figure 2, RAE records the operation sequence that produces the buffered update (of metadata and file descriptors) in the base [48]. After an error is detected, the in-memory metadata and file descriptors in the base are reset during a contained reboot because they may be erroneous. A separate shadow filesystem process is launched to execute the recorded operations (reading any data required from the disk), and the shadow filesystem produces new (and correct) metadata structures that are directly used by a rebooted base. As such, the rebooted base starts from the recovered metadata and file descriptors without needing to re-execute the error-triggering operation sequence. The data pages are shared between the base and the shadow because only applications can detect their corruption.

The shadow filesystem has three essential properties: it is simple, leverages advances in formal verification, and employs extensive runtime checks.

Simple yet equivalent implementation: Modern filesystems are complex, primarily due to optimizations for concurrency, caching, and asynchronous interaction with storage devices. The software architecture (Figure 2) of modern

filesystems includes an interface layer for (un)marshaling and user-interaction (e.g., VFS), a cache for inode, data blocks, and directory entries, and numerous concurrency-related optimizations. A block layer (e.g., blk-mq) interacts with storage devices in an asynchronous fashion, employing various mechanisms and policies for performance. In between, various filesystem implementations (e.g., ext4, xfs, etc.) share the same caching and block layers: a filesystem implementation's complexity and thus likelihood of bugs is exacerbated due to the interactions with these performance components.

To improve robustness, a shadow eliminates all performance optimizations, focusing instead on correctness. It thus aims to be the simplest possible yet equivalent implementation of the base filesystem. To reduce complexity and the likelihood of bugs, a shadow is not interactive to users, does not have concurrency, does not have sophisticated caching structures and policies (e.g., LRU 2Q [36]), performs IO synchronously, and does not write to devices, as shown on the right side of Figure 2. A shadow does not need logic for on-disk durability, because it never writes to the disk: completed sync operations are already on disk (and serve as the shadow's input) and incomplete sync operations are delegated back to the base filesystem.

Practical formal verification: To achieve robustness, a shadow aims to be absolutely correct, as inspired by prior verified filesystems such as FSCQ [11, 12]. Note that the simplifications made in the shadow also simplify the formal verification. Prior works verifying filesystems focus on complex aspects like concurrency and crash safety, which are eliminated in the shadow; therefore, a formally-verified shadow filesystem is more feasible for non-experts in formal methods.

The time is ripe for shadow filesystems due to the availability of better tools for verifying low-level system code. Verus [33], an open-source, semi-automated verification tool for Rust, is designed to verify low-level systems with a reduced proof burden. Given Rust's capability to interact with hardware, we believe that a formally verified shadow filesystem is practical [9]. Given the adoption of Rust in system software like Chromium and the Linux kernel, the shadow's implementation can evolve alongside existing complex filesystems, similar to how specifications using lightweight formal methods are integrated into AWS S3's storage systems [8].

Extensive runtime checks: In addition to formal verification at compile time, the shadow employs extensive runtime checks to ensure liveness. Runtime checks defend against transient hardware faults that are outside of the specification, e.g., the silent data corruption of CPU cores [18, 24, 51]. These runtime checks include validating input operations and disk images, as well as other sanity checks. Due to performance concerns, runtime checks are commonly disabled in the base, but the shadow can enable all possible checks to survive dynamic errors without performance concerns.

3 Shadow Filesystems

We describe shadow filesystems, covering the fault model, enhancements to the base filesystem, control flow between base and shadow, and the shadow's recovery process.

3.1 Fault Model

The shadow filesystem aims to handle all software bugs (transient and deterministic) and transient hardware faults. Deterministic hardware faults requires additional techniques [45], for instance, avoiding a specific memory region [46]. We assume that errors are detected before being persisted to disk, which can be achieved by techniques like validating upon sync [21, 32]. Such detection enhancement helps identify the problematic operation sequence. Finally, we trust device drivers are not malicious.

3.2 Design

Overview: To run the shadow for failure recovery, the base filesystem must record the operation sequence that tracks the gap between the applications' view and the on-disk state. Essentially, this is an execution trace that records the order that operations were handled, reducing the non-determinism of the shadow. The recorded operation sequence also reflects the outcome of the operations, such as the return value, new file descriptors, and new inode numbers. When a file descriptor is closed and the buffered updates are flushed to disk, the corresponding recorded operations can be discarded.

When an error is detected, control is transferred to a recovery procedure that addresses the three problems formulated previously (§2). The recovery procedure discards the in-memory states of the base and reboots it without affecting the OS or applications. The reboot incorporates the base's crash recovery mechanism, such as journal replay. The recovery procedure then launches the shadow, delivers the operation sequence to it, waits for the outcome, and returns the results to the rebooted base. During recovery, new application operations are not admitted.

Recovery: The shadow filesystem is launched as a separate userspace process to ensure the strong isolation of faults and a clean interface between the base and shadow; the shadow does not interact with applications. It has two operation modes: *constrained* and *autonomous*. When the base is executing, the shadow is dormant; its constrained and autonomous modes handle completed and in-progress operations, respectively.

In constrained mode, the shadow executes completed operations, possibly performs disk reads, and produces output along with a set of modified data structures. Constrained mode also cross-checks with the output of the original execution. Discrepancies in output are reported; whether or not to continue can be configured. For inode number and file descriptor allocation, the shadow validates if the value produced by the base filesystem is usable, rather than performing its

own allocation (which could lead to a different value). The shadow omits operations that returned an error by the base.

In autonomous mode, the shadow executes the in-progress operations whose return values have not been seen by the client. In this mode, the shadow must make policy decisions such as allocating new inode numbers (as opposed to simply validating the decisions made by the base). The shadow does not handle fsync, following the rule of not writing to disk.

Hand-off back to the base: The base filesystem must support *metadata downloading* by providing extensively-tested interfaces to absorb the output of the shadow: a set of file descriptors and on-disk metadata structures. To implement the interfaces, the base reads these structures and reuses its existing logic to place them into its cache, marked as dirty. After the hand-off, the base resumes execution and admits new operations, at which point all state within the base filesystem is correct and up to date.

3.3 Contrasting Base and Shadow

The base and shadow filesystems are different in four ways.

Performance optimizations: As shown in Figure 2, the base filesystem interacts with components that are omitted from the shadow. Specifically, the shadow does not use a dentry cache, and instead always performs path lookup from the root inode and scans the directory entries. The shadow does not utilize the concurrent inode and data block caches; instead, it uses a simple data structure to manage filesystem structures read from disk during recovery. The shadow is strictly single-threaded and uses synchronous IO; it does not deal with locking, concurrency, or block-level IO scheduling.

Robustness properties: The shadow has a strong correctness guarantee, relying on static formal verification and extensive runtime checks. Thus, the shadow devotes a large amount of code to assertions and validations for given procedures.

API support: The shadow filesystem supports the same set of filesystem operations as the base, excluding those that persist data to disk, such as `fsync`. We omit the sync family API for simplicity, to avoid interacting with the crash consistency protocol. The contained reboot enables such simplification because it recovers buffered updates and does not need a protocol to make states durable. If the base fails in the middle of fsync, our current design relies on the shadow for the prefix operations and the base to perform fsync again after the hand-off.

Core functionality: For a given operation sequence, the output at the API level and the effects to on-disk structures must be equivalent between the base and the shadow. While some policy decisions might differ, the two must agree on essential invariants. For example, allocating ten data blocks for a 4K write can be valid behavior for a specific base filesystem, but the specific blocks allocated might differ, leading to different data bitmaps. Allowing such difference while ensuring equivalence is one key challenge in developing the shadow.

4 Roadmap and Challenges

We posit that shadowing is a generic approach that works for modern performance-oriented filesystems regardless of the architecture of the OS. Linux kernel filesystems have been the *de facto* for decades and these complex kernel filesystems are not likely to be replaced in the near-future [2]. However, future trends for simplifying development and fault isolation are hinted at by the recent revisiting of microkernels in industry [42] and research [37, 42], and by the eBPF framework [44]. Therefore, we plan to explore case studies along two paths: kernel filesystems and microkernel filesystems. We now describe the main challenges in each approach.

4.1 Linux Kernel Filesystems

The Linux kernel file IO stack follows the architecture in Figure 2. Commonly-known filesystems like ext4, btrfs, and xfs are a variety of base filesystem implementations sharing the same infrastructure. Therefore, all base filesystems can share the fault-anticipation mechanisms for recording operation sequences and transferring control. However, each base must provide its own interface for metadata downloading from its shadow since the on-disk formats differ.

Contained reboot: Contained reboots are challenging in kernel filesystems, but earlier work [17, 48] demonstrates how this can be done without crashing the entire machine. However, kernel infrastructure has evolved significantly during the last decade, with io_uring and block-mq in the block layer, complex policies for the page cache [16], and new mechanisms for managing pages (e.g., folios and iomap) with modern storage devices [15, 19]. The shadow must reset the interactions with these components; one plausible solution is to rely on existing code paths to do so (e.g., code that drops caches when memory pressure is high, and unmount).

States reconstruction: The on-disk formats of kernel filesystem implementations are not well-documented [31], although an explicit ABI is in demand [26]. This lack enables crafted images to bypass the checks of e2fsck and crash the entire machine with a sequence of filesystem operations [26]. We hope that the implementation of a formally-verified shadow filesystem can serve as an ABI. Another technical challenge is that the shadow's synchronous IO mechanism must also be verified, requiring direct reading from the block device, bypassing the kernel's BIO layer whose implementation is challenging to verify. However, the kernel does not expose such an interface to invoke device driver code that bypasses its BIO layer. One approach is to unmount the device and then read the device through a user-space NVMe driver [47].

4.2 Microkernel Filesystems

Several microkernels are available [23, 29, 37], but, to the best of our knowledge, only uFS [37] is optimized for filesystem performance. Microkernel filesystems are interesting given their natural fault isolation and thus effortless delivery of a

contained reboot. It will be interesting to compare the effort needed to support kernel and microkernel filesystems.

4.3 Cross-Cutting Challenges

One interesting and challenging question, regardless of the path, is how to implement the simplest version of a filesystem via Rust that can also be easily verified. Using Rust to interact with hardware poses new challenges due to its compilation-time restrictions, and our requirement of "simplest for verification" makes the question more interesting.

How formal verification is used in shadow filesystems has novel angles. Compared to prior formally-verified filesystems [11, 12, 54], verification of a shadow is greatly simplified due to its reduced complexity, yet it necessitates stronger liveness guarantees. For instance, to ensure the shadow is robust against crashes given a crafted filesystem image and call sequence, the input image must be guaranteed to be valid, essentially requiring a verified version of the filesystem checker (FSCK). Another issue is how to add the constraints of completed operations.

Another interesting issue is the time required for recovery. Even though recovery performance is not a primary concern for the shadow filesystem, recovery time does impact the expected response time observed by applications with in-flight operations.

From a correctness aspect, the interaction between the shadow and the base, especially metadata downloading, requires a lean, well-defined, and thoroughly tested interface. Our current plan is to reuse the code from the base's implementation to read the metadata from the device and fill the base's cache (e.g., page cache, inode cache). We expect to quantify the code we trust (i.e., reused).

Finally, the system must ensure the base and shadow filesystems produce equivalent output for a sequence of operations. Verification alone is insufficient for this property, therefore, testing is necessary before using the shadow [8]. The testing phase uses the base as a reference filesystem to test the shadow by running a large volume of workloads and monitoring for discrepancies. Disagreements between the base and shadow indicate bugs in the base or missing conditions in the shadow. In the former case, running the shadow is an effective way to stress the bug in the base, as the sequence and outputs are recorded (input to the shadow), making the shadow filesystem a valuable post-error testing tool, especially for inputs often missed by testing frameworks [28, 39]. In the latter case, such discrepancies help to improve the shadow to cover the missing conditions. Either way, reporting the discrepancies is necessary, thereby enhancing system reliability.

5 Conclusion

Building a high-performance concurrent filesystem is challenging; formally verifying a complex filesystem with performance optimization and concurrency is similarly daunting. With RAE, we introduce the idea of two filesystems working in tandem to achieve a singular goal: high performance in the common case and correctness and high-availability despite bugs and errors in rare cases. With separate concerns and largely-reduced complexity, the formally-verified shadow is practical to develop and evolve together with the base.

While we explore this RAE approach via shadows in the filesystem context, we believe it may also be interesting to pursue the more general two-pronged approach in other systems as well. The adoption of fully-verified code into existing mature systems, as well as using it as a building block to improve overall reliability, has great potential.

Acknowledgement

We thank Raju Rangaswami (our shepherd), anonymous reviewers, and the ADSL group for their comments and suggestions. This material was supported by funding from NSF CNS-1838733. Jing Liu was supported by a Meta PhD Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF or other institutions.

References

[1] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, Roberto Peon, Larry Kai, Alexander Shraer, Arif Merchant, and Kfir Lev-Ari. 2016. Slicer: Auto-Sharding for Datacenter Applications. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*. Savannah, GA.

[2] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory Ganger, and George Amvrosiadis. 2019. File Systems Unfit as Distributed Storage Backends: Lessons from 10 Years of Ceph Evolution. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '19)*. Ontario, Canada.

[3] Todd M. Austin. 1999. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'99)*. Haifa, Israel.

[4] Algirdas A. Avižienis. 1995. The Methodology of N-Version Programming. In *Software Fault Tolerance*, Michael R. Lyu (Ed.). John Wiley & Sons Ltd., Chapter 2.

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

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

[7] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. 2020. Theseus: an Experiment in Operating System Structure and State Management. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI '20)*. Virtual Conference.

[8] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. 2021. Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '21)*. Virtual Event, Germany.

[9] Matthias Brun, Reto Achermann, Tej Chajed, Jon Howell, Gerd Zellweger, and Andrea Lattuada. 2023. Beyond isolation: OS verification as

a foundation for correct applications. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HOTOS '23)*. Providence, Rhode Island.

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

[11] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay undefinedleri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2017. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*. Shangai, China.

[12] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare logic for certifying the FSCQ file system. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*. Monterey, California.

[13] Zhihao Cheng. 2018. ext4_handle_inode_extension: i_size_read(inode) < EXT4_I(inode)->i_disksize. https://bugzilla.kernel.org/show_bug.cgi?id=217159.

[14] Jonathan Corbet. 2013. The multiqueue block layer. https://lwn.net/Articles/552904/.

[15] Jonathan Corbet. 2021. Clarifying memory management with page folios. https://lwn.net/Articles/849538/.

[16] Jonathan Corbet. 2021. Multi-generational LRU: the next generation. https://lwn.net/Articles/856931/.

[17] Alex Depoutovitch and Michael Stumm. 2010. Otherworld: Giving Applications a Chance to Survive OS Kernel Crashes. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys '10)*. Paris, France.

[18] Harish Dattatraya Dixit, Sneha Pendharkar, Matt Beadon, Chris Mason, Tejasvi Chakravarthy, Bharath Muthiah, and Sriram Sankar. 2021. Silent Data Corruptions at Scale. *CoRR* abs/2102.11245 (2021). https://arxiv.org/abs/2102.11245

[19] Jake Edge. 2023. Converting filesystems to iomap. https://lwn.net/Articles/935934/.

[20] Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. 2010. Availability in Globally Distributed Storage Systems. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI '10)*. Vancouver, Canada.

[21] Daniel Fryer, Kuei Sun, Rahat Mahmood, Tinghao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown. 2012. Recon: Verifying File System Consistency at Runtime. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*. San Jose, CA.

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

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

[24] Peter H. Hochschild, Paul Turner, Jeffrey C. Mogul, Rama Govindaraju, Parthasarathy Ranganathan, David E. Culler, and Amin Vahdat. 2021. Cores That Don't Count. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HOTOS '21)*. Ann Arbor, Michigan.

[25] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. 2022. Metastable Failures in the Wild. In *Proceedings of the 16th USENIX Conference on Operating Systems Design and Implementation (OSDI '22)*. Carlsbad, CA.

[26] Jonathan Corbet. 2020. The ABI status of filesystem formats. https://lwn.net/Articles/833696/.

[27] The kernel development community. 2024. Linux kernel coding style: Do not crash the kernel. https://www.kernel.org/doc/html/latest/process/coding-style.html#do-not-crash-the-kernel.

[28] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. 2019. Finding Semantic Bugs in File Systems with an Extensible Fuzzing Framework. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '19)*. Ontario, Canada.

[29] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Michael Norrish, Rafal Kolanski, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*. Big Sky, Montana.

[30] John C. Knight and Nancy G. Leveson. 1986. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering* SE-12, 1 (1986), 96–109. https://doi.org/10.1109/TSE.1986.6312924

[31] Michalis Kokologiannakis, Ilya Kaysin, Azalea Raad, and Viktor Vafeiadis. 2021. PerSeVerE: Persistency semantics for verification under ext4. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–29.

[32] Harendra Kumar, Yuvraj Patel, Ram Kesavan, and Sumith Makam. 2017. High Performance Metadata Integrity Protection in the WAFL Copy-on-Write File System. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*. Santa Clara, CA.

[33] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying rust programs using linear ghost types. *Proceedings of the ACM on Programming Languages* OOPSLA1 (2023), 286–315.

[34] Jialin Li, Samantha Miller, Danyang Zhuo, Ang Chen, Jon Howell, and Thomas Anderson. 2021. An incremental path towards a safer OS kernel. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HOTOS '21)*. Ann Arbor, Michigan.

[35] Linus Torvalds. 2002. [BK PATCH] USB changes for 2.5.34. https://yarchive.net/comp/linux/BUG.html.

[36] Inc. Linux Kernel Organization. [n.d.]. Linux Page Replacement Policy. https://www.kernel.org/doc/gorman/html/understand/understand013.html.

[37] Jing Liu, Anthony Rebello, Yifan Dai, Chenhao Ye, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2021. Scale and Performance in a Filesystem Semi-Microkernel. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '21)*. Virtual Event, Germany.

[38] Wenqing Liu. 2018. array-index-out-of-bounds in fs/f2fs/segment.c. https://bugzilla.kernel.org/show_bug.cgi?id=215657.

[39] Yifei Liu, Manish Adkar, Gerard Holzmann, Geoff Kuenning, Pei Liu, Scott A. Smolka, Wei Su, and Erez Zadok. 2024. Metis: File System Model Checking via Versatile Input and State Exploration. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST '24)*. Santa Clara, CA.

[40] David E Lowell, Subhachandra Chandra, and Peter Chen. 2000. Exploring Failure Transparency and the Limits of Generic Recovery. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI '00)*. San Diego, CA.

[41] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. 2013. A Study of Linux File System Evolution. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST '13)*. San Jose, CA.

[42] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Mike Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Mike Ryan, Erik Rubow, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. 2019. Snap: a Microkernel Approach to Host Networking. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '19)*. Ontario, Canada.

[43] Daejun Park and Dongkun Shin. 2017. iJournaling: Fine-Grained Journaling for Improving the Latency of Fsync System Call. In *Proceedings of the USENIX Annual Technical Conference (USENIX '17)*. Santa Clara, CA.

[44] Cilium Project (post in Hacker News). 2020. EBPF is turning the Linux kernel into a microkernel. https://news.ycombinator.com/item?id=22953730.

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

[46] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. 2009. DRAM Errors in the Wild: A Large-scale Field Study. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '09)*. Seattle, WA, USA.

[47] SPDK Open-source Team. 2021. The Storage Performance Development Kit. https://spdk.io/doc.

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

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

[50] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. 2004. Recovering Device Drivers. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*. San Francisco, CA, 1–16.

[51] Shaobu Wang, Guangyan Zhang, Junyu Wei, Yang Wang, Jiesheng Wu, and Qingchao Luo. 2023. Understanding Silent Data Corruptions in a Large Production CPU Population. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP '23)*. Koblenz, Germany.

[52] Wen Xu. 2018. use-after-free in ext4_put_super(). https://bugzilla.kernel.org/show_bug.cgi?id=200931.

[53] Junfeng Yang, Can Sar, and Dawson Engler. 2006. 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, WA.

[54] Mo Zou, Haoran Ding, Dong Du, Ming Fu, Ronghui Gu, and Haibo Chen. 2019. Using concurrent relational logic with helpers for verifying the AtomFS file system. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '19)*. Ontario, Canada.