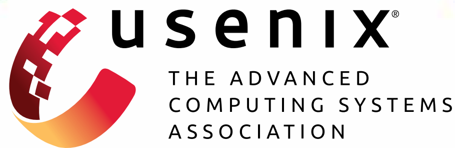


Fast, Transparent Filesystem Microkernel Recovery with Ananke

Jing Liu, *Microsoft Research;* Yifan Dai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, *University of Wisconsin–Madison*

https://www.usenix.org/conference/fast25/presentation/liu-jing

This paper is included in the Proceedings of the
23rd USENIX Conference on File and Storage Technologies.

February 25–27, 2025 • Santa Clara, CA, USA

ISBN 978-1-939133-45-8

Open access to the Proceedings of the 23rd USENIX Conference on File and Storage Technologies is sponsored by

Fast, Transparent Filesystem Microkernel Recovery with Ananke

Jing Liu*, Yifan Dai†, Andrea C. Arpaci-Dusseau†, Remzi H. Arpaci-Dusseau†
*Microsoft Research**, *University of Wisconsin–Madison*†

Abstract. We introduce Ananke, a high-performance filesystem microkernel service that provides transparent recovery from unexpected filesystem failures. Ananke does so by leveraging the unique opportunity of the microkernels, running a small amount of recovery code coordinated by the host OS at the moment of a process crash. Ananke can record key pieces of information not usually available during full-system crash recovery, enabling fast and transparent recovery for applications. Through over 30,000 fault-injection experiments, we demonstrate that Ananke achieves lossless recovery; we also show that Ananke recovers quickly, usually in a few hundred milliseconds. Through real application workloads, we show that Ananke delivers high performance in the common case; the extra work needed to detect faults and prepare for recovery incurs minimal overheads.

1 Introduction

Microkernels have existed for decades [11, 19, 20, 30, 53, 54, 72], finding widespread deployment in millions of devices [9] and usage across a range of domains, including real-time environments [21], mobile devices [34], secure systems [30], and desktop computer systems [74]. Recent research has also advocated partial microkernel-style architectures for high-performance datacenter environments, including networking [28,41] and storage [36] stacks. In these systems, a traditional kernel (i.e., Linux) serves as the base OS, but the subsystem of choice is placed within a separate user process, with total control over the relevant hardware components (e.g., the network card or storage device). These approaches deliver high performance and scale effectively with more cores [36, 41].

However, because server software – and underlying hardware – is imperfect, it can fail. Unlike a typical OS crash (a full *system crash*, or *s-crash* for short), which likely brings down the entire system (and any processes running upon it), a microkernel server failure can (likely) be contained to the process itself; client applications running on the system can (potentially) continue to run.

Ideally, after a server *process crash (p-crash)*, the server quickly restarts and resumes serving requests from client applications. However, in a filesystem service, doing so is challenging: the state of the filesystem, including buffered updates not yet persisted and ephemeral state such as open file descriptors must be exactly recovered. We refer to the difference between applications' expectation for filesystem state and the state recovered by s-crash recovery (i.e., persisted) as the *state gap*.

*Work partially done while at the University of Wisconsin–Madison.

Despite the literature on microkernel recovery [12,33] and filesystem recovery [10, 39, 63], how to fully realize the potential of a p-crash for microkernel filesystems remains unsatisfactorily answered. In this paper, we advocate a principled approach to p-crash recovery, which is complementary to s-crash recovery and provides a better guarantee of transparent recovery with a realistic fault model.

We do so by examining the challenges of p-crash recovery and leveraging the benefits of the entire machine being alive – the OS serves as an ideal coordinator, with information readily available in server memory that can contribute to recovery. We design and implement Ananke, a high-performance and robust filesystem microkernel service. Ananke is based on uFS [36], adding mechanisms for transparent recovery after an unexpected fault while maintaining high performance in the common case.

Central to recovery is an in-memory *p-crash log (p-log)*. During normal operation, Ananke records information about ongoing system calls into the p-log. Importantly, as the workload continues, the p-log always reflects the source of the state gap, even if the system calls can be persisted in a non-sequential order. After a p-crash, with OS coordination, a new process is started, and the p-log and client connections are properly transferred between the two processes. The new process can access the p-log – in addition to the on-disk s-crash recovery logs (e.g., journals) – to restore the filesystem state precisely. Applications continue running undisturbed, incurring only a momentary drop in performance.

Ananke includes a range of novel mechanisms. The p-log with the *AIM* (Act-Ignore-Modify) algorithm recovers state gaps without impacting common-path performance. A *kernel-coordinated speculative restart* robustly restarts a new process with clean state and avoids the overhead of device connection establishment, greatly reducing recovery time. Finally, lightweight checksums are used to promptly detect corruptions in the essential in-memory data structures.

We perform a thorough evaluation of Ananke, focusing on fault handling and performance. We show that Ananke achieves seamless recovery for applications, including utilities (sort, cp, zip) and durability-aware libraries (SQLite, LevelDB); Ananke achieves transparent recovery in a large number (30,000+) of controlled and random fault-injection experiments that emulate both fail-stop p-crashes and memory corruption. Further, Ananke incurs negligible performance overhead (less than 2% in most cases) and memory overhead (8 MB per core for a replicated p-log and less than 0.01% of workload memory for CRCs). Finally, Ananke recovers in an acceptable amount of time ($\leq$400ms).

This paper is organized as follows. We present background (§2), design (§3), implementation (§4), evaluation (§5), related work (§6), and conclude (§7).

2 Background and Extended Motivation

We present the crash models and implications of filesystem failures in monolithic and microkernel systems. We then highlight the need for separate process crash recovery in microkernel filesystems.

2.1 Monolithic Kernel Filesystem Failure ⇒ Full-system Crash

A filesystem may encounter an error that causes it to fail, due to reasons like software bugs [8,23,29,38] and hardware errors (e.g., CPU or memory) [2,14,22,31,49,59,61,70,73]. The crash model for a monolithic kernel filesystem failure is that when the filesystem crashes, the entire system, including the OS kernel, also crashes (i.e., a *full-system crash* or *s-crash* for short), resulting in the loss of all volatile memory states that have not been saved to disk.

One implication is that a filesystem failure shares the same crash model as other environmental causes of an s-crash, such as a power failure or a hard machine reset. Therefore, s-crash recovery commonly treats a filesystem crash the same as a power failure.

All modern monolithic kernel filesystems contain mechanisms to recover from an s-crash, relying solely on on-disk states. For example, Linux ext3/4 filesystems use journaling (a.k.a. write-ahead logging) to record relevant information (filesystem metadata and/or user data) about pending updates into what we term an *s-log* [67, 68]; if a later s-crash occurs, the filesystem utilizes the s-log to recover the filesystem to a consistent state. Of course, other techniques exist [37, 44, 56], but all share the goal of ensuring on-disk states are *consistent*, rather than guaranteeing no data loss.

Another implication is that when the filesystem fails, all applications running on the system also crash. Traditional s-crash recovery only ensures that the filesystem is returned to a consistent state, assuming that applications lose their progress. Since filesystems buffer updates in memory for performance [45], many data and metadata updates are not persisted at any instant, resulting in the loss of recent updates during an s-crash.

Traditional s-crash recovery can also be slow. Approaches based on full-disk scans (such as fsck [44]) are prohibitively slow [40], as they must scan all filesystem metadata to fix inconsistencies. More modern approaches (e.g., journaling [51,67]) are better, with performance proportional to the size of the log (rather than the entire filesystem disk space).

2.2 Microkernel Filesystem Failure ⇒ Process Crash

A microkernel filesystem exhibits a different crash model: the *process crash* (or p-crash for short). In this case, the filesystem server goes down after a failure, but the rest of the system remains up and running.

The first implication is better fault isolation, a well-known reliability benefit of microkernel design [9], where the OS, including other system services continue naturally.

The second implication is that applications interacting with the filesystem have the opportunity to continue, but not without challenges. The main challenge is the *state gap* between the application's perceived states (i.e., buffered in the crashed server's memory) and the on-disk states. Such state gap covers the semantic states of the filesystem, including on-disk data structures and file descriptors. Performing s-crash recovery is insufficient because it ensures that the on-disk states are consistent, but does not address the state gap.

Consider, for example, an application that issues ten *write*s to an empty file and then closes the file descriptor, at which point the filesystem server crashes. If we restart the filesystem server, and the application opens and reads the file again, it will see an empty file, since that is its on-disk state. However, the application's perceived state is that the file contains ten writes – a confusing outcome with severe data loss.

The state gap can be much more complex than this simple example, as discussed later (§3.6). The fundamental issue is that the in-memory states of the filesystem server can be updated (and buffered) by a rich set of filesystem APIs. The state gap may also be altered by partial flushes of the server's states to disk, as a result of the out-of-order durability employed by modern filesystems [42, 50]. As a result, after a p-crash, opened file descriptors, data writes, and metadata updates (e.g., creating, unlinking, and renaming files) will be unexpectedly lost.

2.3 Separate P-crash Recovery from S-crash Recovery

In this work, we advocate for a separate p-crash recovery mechanism in addition to traditional s-crash recovery. Such a recovery leverages the opportunity provided by p-crashes: if the server can quickly restart, recover, and resume serving requests, availability increases. With fast p-crash recovery, applications might not notice that the filesystem has crashed.

The benefits of p-crash recovery are manifold. First, applications can continue running with better correctness guarantees than those provided by s-crash recovery, achieving transparent recovery. A microkernel offers the advantage that filesystem applications do not lose their progress. For example, user applications do not need to restart their jobs or perform manual recoveries, which can be error-prone [50].

Second, filesystem failures and p-crashes occur frequently [38], whereas power failures are relatively rare [16, 24]. While modern cloud environments are designed to handle power failures, these solutions are complex, involving failover with sophisticated protocols [48]. Delegating a p-crash to a global failure scenario is not only unnecessary but also increases the risks of severe issues from failover invoked by an s-crash [24]. Separating p-crash recovery from s-crash recovery reduces these complexities and potential risks.

Third, p-crash recovery can take advantage of another microkernel architecture's benefit: restarting the filesystem

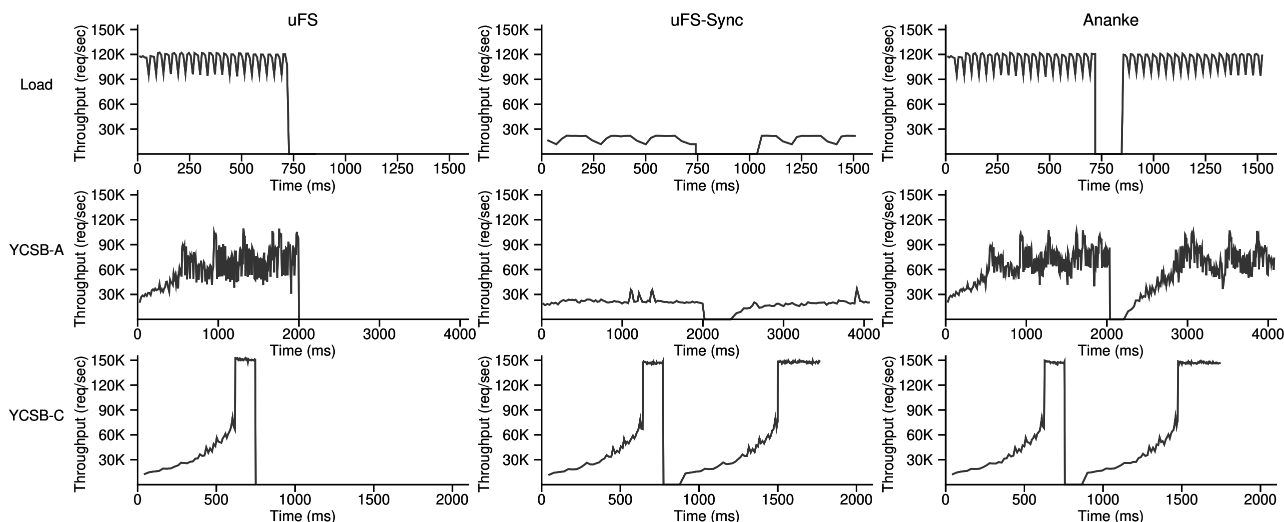


Figure 1: **Recovery with uFS, uFS-Sync, Ananke for LevelDB**. *First row (Load): After the p-crash at time 700ms, LevelDB on uFS is not able to continue without manual intervention. With uFS-Sync, LevelDB continues after the p-crash, but performance suffers because dirty pages are persisted after every operation. Ananke achieves the best of both: transparent recovery and high performance. Recovery time with uFS-Sync is 285ms; with Ananke 114ms. Second row (YCSB-A): Recovery with uFS-Sync 310ms; Ananke 171ms. Third row (YCSB-C): Recovery with uFS-Sync 108ms; Ananke 102ms.*

	Ananke	uFS	uFS-Sync	Membrane [63]	Rio [10]	Otherworld [13]	TxIPC [33]
Recover State Gap	■	□	■	■	■	■	■
Low Perf. Impact	■	■	□	▒	■	■	■
Memory Corruption	■		■	▒	□	□	□
Robust Restart	■		■	▒	■	■	□
Prompt Detection	▓		□	□	□	□	□

Table 1: **Qualitative Comparison with Relevant Systems.** *Black indicates the best and white the worst. The last three rows are not applicable to uFS.*

server requires less effort, can be fast, and can rely on all the services coordinated by the host OS (e.g., fork, tmpfs, pseudo file systems, etc.). In contrast, restarting a kernel filesystem is challenging and less robust when reusing the same kernel [63] or slow if using a new kernel instance [13].

Unfortunately, current applications are not able to continue operating transparently after microkernel filesystem p-crashes, even when the filesystem, uFS [36], has been extended to automatically restart and reconnect with active clients. The first column of Figure 1 shows that when LevelDB (with three different workloads) runs on a restartable/reconnecting version of uFS, a p-crash of the fileserver causes LevelDB to terminate and not be able to continue without manual intervention to repair data. The fundamental problem is that a state gap exists between the filesystem state that LevelDB expects and what is provided by the microkernel filesystem when it resumes.

2.4 Alternatives for P-Crash Recovery

There are several methods to workaround the challenges of state gap [12, 58] for p-crash recovery, all with their costs. One straightforward method to handle a p-crash is to transform it into an s-crash. Specifically, when the fileserver goes down, either bring down the entire system or the related applications using the filesystem. However, such an approach misses the opportunities and benefits of p-crash recovery.

Second, one might consider a client-side recovery mechanism, where the client is in charge of the state gap (e.g., by retrying). However, this requires the application to track the state gap, which is complex because the client must be notified whenever the server persists any of its state (e.g., from a background sync). Furthermore, retry during recovery is a fundamentally difficult problem of distributed coordination [6] in the presence of multiple clients.

A final alternative is to eliminate the state gap by avoiding server-side write buffering. By forcing all updates to be persistent before replying to applications, the server ensures that the on-disk state is always up-to-date. Thus, s-crash recovery is sufficient to restore the system to its most recent state. Even though such an approach can provide failure transparency to applications, forcing updates to disk results in unacceptably poor common-case performance. The second column of Figure 1 shows the poor performance when adopting such an approach in uFS (modified as uFS-Sync); in this case, LevelDB is able to continue transparently after the p-crash, but its throughput is unacceptably low on any workloads with significant writes (i.e., Load and YCSB-A).

By contrast, Ananke is able to deliver the full potential of separating p-crashes from s-crashes for applications. The third column of Figure 1 shows that when LevelDB is run on Ananke, it achieves the same common-case throughput as it did on the original uFS system before the p-crash, and continues operating transparently after the p-crash occurs.

2.5 Challenges in Principled P-Crash Recovery

Previous filesystems have made significant steps towards providing transparent filesystem recovery to applications, but none have yet delivered the full potential of p-crash

recovery. We examine the limitations of some existing restartable kernel filesystems (such as Membrane [63], Rio [10], and Otherworld [13]) and previous microkernel filesystems (such as TxIPC [33]). Table 1 summarizes how well these systems, including uFS and uFS-Sync, address the following challenges for ideal recovery.

C1 Recover state gap. Recovering the exact state gap is the major challenge for filesystem p-crash recovery and ensures that applications can continue without knowing the filesystem failed. All of the systems under comparison (except uFS) address this fundamental challenge.

C2 Low impact on common-path performance. Given that recoveries are rare, filesystems must incur low memory and performance overhead in the common-case. While Rio, Otherworld, and TxIPC have sufficiently prioritized recovering the state gap while maintaining common-case performance, Membrane has some similar performance problems as uFS-Sync, since Membrane must flush all state to disk on any fsync or create [63]; we measure this overhead in our evaluation. Ananke recovers the state gap with little impact on performance by ensuring that updates are not unnecessarily forced to disk.

C3 Handle memory corruption. The comparison systems are not robust to faults that cause memory corruption, since they reuse either the entire kernel or process address space (as in Membrane [63], Otherworld [13], and TxIPC [33]), or filesystem metadata (as in Rio [10] and Otherworld [13]). This reuse overlooks the risk that memory may have been corrupted before the error was detected, potentially causing further errors during and after recovery. Ananke accesses well-defined and protected memory regions after an error is detected, limiting its trust in memory.

C4 Robust restart. Once p-crash recovery begins, it should be robust and always able to restart the filesystem. However, if the restarted system reuses some resources from the failed system, recovery may inadvertently depend on those failed resources. Membrane [63] and TxIPC [33] perform non-trivial cleanup through unwinding and then reuse the failed thread for recovery and to handle future requests. In contrast, Ananke advocates robust restart with a new process and a clean address space.

C5 Prompt error detection. Previous systems may not detect errors promptly, potentially causing incorrect results to be returned to applications or corrupt data to be persisted to disk. Ananke enhances error detection by selectively checking that important subsets of state – specifically, semantic states – are not corrupted; these checks are lightweight and always enabled in the common path.

3 Ananke Design

We discuss the design of Ananke, beginning with goals and design principles, and then delve into Ananke's mechanisms, focusing on the core data structure (the p-crash log, or p-log) and p-log replay algorithm (AIM).

3.1 Goals for P-Crash Recovery

Ananke is designed with the following goals:

Transparent p-crash recovery. P-crash recovery should be lossless; applications should not perceive that Ananke has p-crashed and restarted, should not lose updates, and should not receive any confusing or incorrect results.

Fast common-path performance. Since faults are relatively rare, any improvements of fault recovery should not degrade common-path performance.

Realistic fault model. Although prior work often assumes faults are fail-stop and do not lead to memory corruption [10,63], Ananke handles a wider range of realistic faults including *transient* faults that may cause memory corruption (e.g., due to hardware or software faults).

Fast p-crash recovery. Since applications may be awaiting results during p-crash recovery, the system should restart quickly to minimize disruption.

Few filesystem codebase modifications. Filesystems are complex pieces of code containing many performance optimizations, so recovery should not require onerous changes to the original codebase.

3.2 Base Architecture

Ananke derives from a high-performance open-source filesystem microkernel called uFS [36]. uFS is a fully-functional, POSIX-compliant, multi-threaded user-space filesystem that directly interacts with high-performance storage devices (via SPDK drivers [60]) and requires minimal kernel involvement. uFS employs a thread-per-core architecture for scalability [27, 47], where each thread is called a worker. The centralized filesystem service is shared among application processes.

Applications connect to the filesystem server and issue filesystem calls (i.e., operations) through message ring buffers in shared memory (i.e., App-Worker MsgRing in Figure 2). uFS has the typical in-memory data structures of a modern filesystem [4, 42, 62]: a page cache, directory-entry cache, inode table, and various bitmaps. uFS uses journaling for s-crash recovery (i.e., s-log).

3.3 Design Principles and Overview

Ananke follows several high-level design principles to achieve its goals; we label the related challenges that each principle addresses.

No force flush (C1,C2). In order to ensure that common-case performance remains fast, Ananke avoids forcing buffered updates to disk. Ananke uses a P-Crash Log (p-log) to accurately captures the source of state gap.

Limited trust in memory (C3). Ananke limits the memory regions used after a failure to only the p-log. To ensure the p-log can be trusted, it is protected with mechanisms such as checksums and replication.

Clean restart (C4). To avoid using the failed filesystem process resources, Ananke launches a new process with a clean address space to execute the recovery logic and handle new

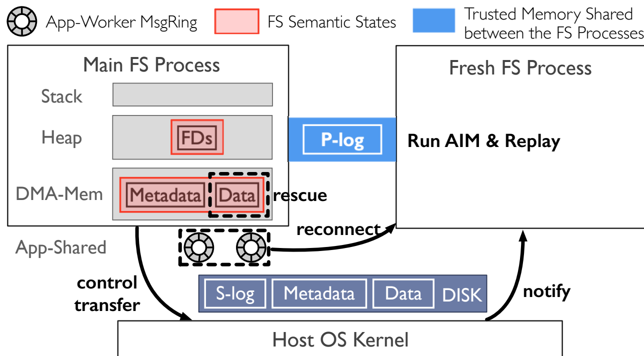


Figure 2: **Illustration of P-Crash Recovery in Ananke.** *The difference between in-memory semantic states (red boxes) and the on-disk states after completing a sequence of operations is the* ***state gap****. After an error is detected in main process, recovery is initialized by the host OS; the OS also notifies the fresh process to start and consume the p-log.*

requests from applications.

Fail fast (C5). To avoid returning incorrect results to applications or persisting corruption to disk, Ananke detects errors and memory corruption as early as possible with assertions and checksums over important data structures.

Using these principles, Ananke operates as follows (as shown in Figure 2).

(1). During normal execution, the Ananke (main) filesystem process services application requests via shared-memory message ring-buffers, similar to uFS. Ananke differs from uFS in that, upon completion of each request, Ananke records the operation into a trusted p-log (replicated and checksummed). After operations have been persisted (after either an application calls `fsync` or Ananke performs an internal background sync) or when file descriptors are closed, the relevant entries in the p-log are updated to indicate they will not need to be exactly replayed during recovery. Ananke ensures there is sufficient space in the p-log by garbage collecting out-dated entries as needed.

(2). When an error is triggered in the Ananke filesystem process (e.g., by a software exception from a checksum failure over important semantic state), the host OS kernel obtains control and invokes a short procedure to rescue the data pages referenced by the p-log. After a fresh Ananke filesystem process is created with clean state, the p-log and shared-memory message ring-buffers are shared with it, and the failed filesystem process is simply discarded.

(3). The fresh filesystem process completes recovery by re-initializing the pinned memory regions with the storage device, performing s-log recovery, transforming entries in the p-log with the AIM algorithm so operations can be correctly replayed, and re-attaching message ring-buffers with applications. During replay, Ananke uses only the explicitly rescued memory and the on-disk state of the file system. Finally, the old p-log memory region is released.

(4). Ananke is now ready to resume serving old and new applications.

3.4 P-Crash Recovery Mechanisms

We now describe the mechanisms Ananke uses to address the aforementioned challenges of p-crash recovery.

Trusted P-Crash Log and AIM (C1,C2,C3). Ananke's core mechanisms for transparently recovering the state gap between applications' views of the filesystem and its actual durable state are the *P-Crash Log (p-log)* data structure and the AIM algorithm. Among memory not shared with clients, the p-log is the only trusted region after a failure. The p-log resides in a dedicated memory region that is shared between the failed process (with write access) and the fresh process (read-only access). To protect against corruption, the p-log is safeguarded by checksums and replication. The p-log records the source of the state gap – the system calls and arguments that, when replayed properly, can reconstruct the state gap. Details of the p-log are in Section 3.5.

Recovering the state gap is challenging for two reasons. First, the updates produced by some (but not all) of the completed operations may have been made durable, and in a non-sequential order. Second, other important state, such as file descriptors, may be affected by later operations such as write and close, complicating replay. Naively replaying operations is problematic since operations that do not update state do not need to be replayed, and subsequent operations may alter the preconditions of previous operations.

To address these complications, Ananke employs the Act-Ignore-Modify (AIM) algorithm. The input to this algorithm is the precise set of system calls and their arguments executed by the filesystem, as recorded in the p-log. The output is a new set of system calls that the original filesystem code can execute to reconstruct the in-memory state perceived by the applications. In AIM, Act indicates the operation should be directly replayed; Ignore indicates it can be ignored; and, Modify means an update must occur that is a modification of the original operation. AIM is used for both garbage collection (in the common path) and replay (during recovery). AIM ensures that the p-log is complementary to the s-log, and together they constitute the up-to-date view of the filesystem perceived by applications. AIM is described fully in Section 3.6.

Kernel-coordinated Speculative Restart (C4). We introduce *kernel-coordinated speculative restart* to enable efficient and robust recovery, following a *clean restart* of the Ananke filesystem process. This mechanism addresses two issues. First, any exit of a failed Ananke filesystem process is promptly detected by the host OS. If the filesystem process exits for any reason (e.g., a hardware exception, a software exception due to a checksum or assertion failure, or a segmentation fault) the OS is notified and takes control; the OS then coordinates the transition between the failed and fresh filesystem processes and orchestrates the execution of the recovery logic across the two. The coordination ensures that resources shared between the main filesystem process and applications (i.e., IPC connections) and the host OS (i.e., per-

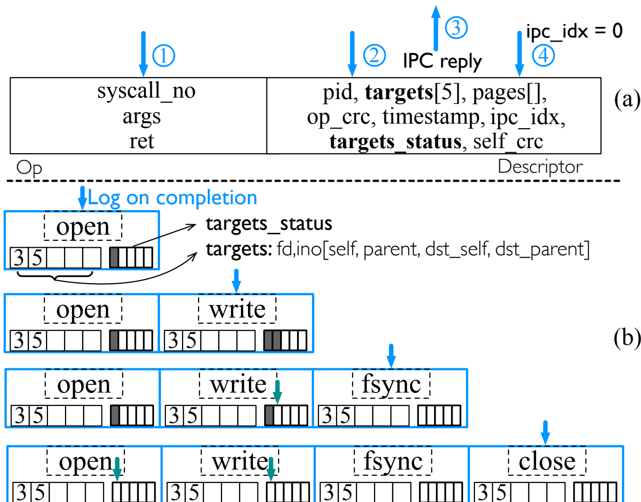


Figure 3: **P-log Design.** *(a) A P-log entry. (b) P-log entries while serving a sequence of open, write, fsync, and close. Blue arrows: the initial logging of an operation; circled numbers: the order of writes for atomicity. Green arrows: clearing the targets_status field upon fsync and close.*

missions and driver connections), are properly established by the fresh filesystem process.

Second, passing control from the failed process to the fresh process is fast and efficient. Ananke hides the latency of starting a fresh filesystem process by creating and initializing this process speculatively, before it is needed. Specifically, when an Ananke filesystem process starts, a secondary passive process is also created, which then blocks on notification from the OS that the first has failed. Importantly, the passive process performs costly initialization of device access (i.e., several seconds) in parallel with the main process.

Lightweight detection of corruption (C5). To prevent incorrect results from being returned to applications or persisted to disk, Ananke fails fast when memory is corrupted. Ananke integrates a lightweight mechanism to enhance fault detection by adding checksums to all in-memory semantic structures, such as file descriptors on the heap and metadata and data in DMA-able memory (i.e., red boxes in Figure 2).

The checksums are verified on each read or write access and updated on each write. As shown in the evaluation, this protection is lightweight, adding only a 0.2% memory overhead and 2.85% performance overhead. Note that these checksums are not used to determine which data structures can be trusted during recovery – no memory other than the replicated p-log is used during recovery; the checksums are used only to detect that memory has been corrupted and thus a fault has occurred.

3.5 P-Crash Log (p-log)

The purpose of the p-log is to track client operations whose effects contribute to the state gap. The p-log is organized as a circular buffer. Each core has its private p-log, written only by one worker thread. To be more robust to memory corruption, the p-log is replicated and pointer-less.

Each p-log entry, as shown in Figure 3(a), contains an operation (system call, arguments, and return value) and a p-log-entry descriptor. The descriptor contains the process identifier (pid), logical pointers to written data pages (pages[]), a checksum of the operation (op_crc), a completion timestamp, and a checksum for the descriptor (self_crc). Importantly, the descriptor contains information for tracking this operation's contribution to the state gap: the *targets* array which records the involved file descriptor (the first element) and inodes (the remaining four elements, in order), and the bitmap *targets_status* where each bit corresponds to a particular target, indicating whether a target's change induced by the operation is part of the state gap. Such a p-log design, combined with AIM (§3.6), allows Ananke to accurately track the state gap while meeting the stringent low-overhead requirement in the common path.

A p-log entry's operation is logged upon its completion with the descriptor filled, as shown in the example in Figure 3(b). The first operation, open, involves two targets: a file descriptor (=3) and a file inode (ino=5), as recorded in targets; the open adds a new file descriptor to the state gap, so the first bit in targets_status is set. The second operation, write, changes the offset of the file descriptor (=3) and the contents of the file inode (ino=5), so the first two bits in targets_status are set. The remaining elements in the targets correspond to the parent, destination, and destination's parent inodes (e.g., for creat, rename). The bits will be cleared when subsequent operations cause the client's view of that target to match its persistent state – either the file descriptor is closed or the inode is made durable (e.g., after an fsync or a background flush). In the example, after the inode is synced to disk, Ananke finds all the p-log entries referring to this inode number and clears their corresponding bits in targets_status (the 3rd row); after the close, the fd bit of write and open are cleared (the 4th row).

Thus, upon a p-crash, a given operation in the p-log may have some bits cleared while others remain set, indicating that some targets are still part of the state gap while others are not. For example, if a p-crash occurs right after the fsync in Figure 3(b), only the first bit is still set in the p-log entry of write, indicating that the fd(=3)'s offset is part of the state gap, while the write to the inode (ino=5) is not. AIM leverages this information to decide how to reconstruct the state gap during recovery.

Note that the p-log does not contain a copy of any file data to be written, but instead stores logical pointers to those pages in the page cache (i.e., offsets of the pages in the DMA-able region). Given the end-to-end argument, we believe that applications should handle user data corruption, while the filesystem protects its own data [57].

3.6 AIM (Act-Ignore-Modify) Algorithm

We now discuss AIM, the core mechanism that leverages the p-log to track and reconstruct the state gap.

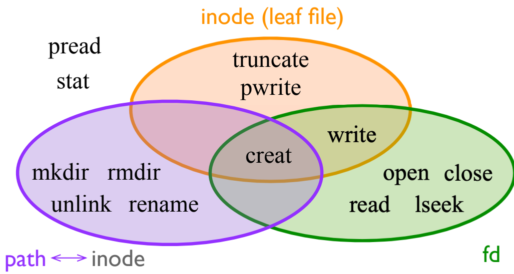


Figure 4: **Intuition behind P-Log and AIM** *We enumerate POSIX APIs and categorize them according to how filesystem abstractions are updated. Representative ones are shown for brevity; mmap, chown, and chmod are excluded.*

Intuition. Applications change the runtime states of a filesystem through the lens of APIs and abstractions provided by the filesystems. We thus categorize the APIs according to the abstractions they change in Figure 4: file descriptors (fd), file inodes (inode), and mappings from pathnames to inodes (path↔inode). Essentially, changes to these abstractions compose the state gap; among them, only fd is ephemeral while other changes are persistent. Our p-log captures all of these changes effectively, as we discuss next.

Operations in the green circle of Figure 4 can change an fd, including its existence (e.g., open and close) and its values (e.g., offset for read and write). Operations in the orange circle can change an inode by changing its data (e.g., write and creat which initialize inodes). Operations in the purple circle affect path↔inode(e.g., creat and rename). Other operations, such as pread and stat, do not introduce changes. The challenge of recovery is that operations cannot be naively replayed, since subsequent operations could remove some (or all) of the changes from the state gap made by earlier operations, or could alter preconditions, such as the existence of an fd or a path↔inode.

Our p-log design tracks the state gap via the targets and targets_status, which covers all three abstractions: fd is the first element, inode is the second element, and path↔inode changes are captured by the parent inodes **in the later elements**. For example, a rename maximally involves a source inode, and a destination inode, and two parent inodes whose data contents are path↔inodes. As a result, whether or not an operation contributes to the state gap (i.e., the conditions to *Ignore* an operation) can be checked quickly with those bits, which is crucial for efficient garbage collection of p-log entries in the common path.

If an operation does contribute to the state gap, the distinction between *Act* and *Modify* as well as the resulting operations of *Modify* is determined during recovery. Figure 4 helps describe the intuition for Modified operations: if some of an operation's effects are no longer part of the state gap, it degrades to operations that can exert the remaining effects, visually moving towards a less overlapped region in the diagram (e.g., creat to open, write to lseek). In this way, we leverage existing APIs (and their implementation) to reconstruct the state gap.

#	System Call Sequence followed by a P-Crash	Consequence AIM outcome
1	open(f) read(f) open(f) lseek(f)	BadFd AM
2	open(f) write(f) close(f) open(f) write(f) close(f)	DLoss AAA
3	open(f) write(f) sync(f) open(f) lseek(f) sync(f)	BadFd AMI
4	open(f) write(f) sync(f) write(f) open(f) lseek(f) sync(f) write(f)	BadFd,DLoss AMIA
5	open(f) write(f) sync(f) close(f) open(f) write(f) sync(f) close(f)	✓ IIII
6	creat(f) write(f) creat(f) write(f)	BadFd,FLoss AA
7	creat(f) write(f) close(f) creat(f) write(f) close(f)	FLoss AAA
8	creat(D/f) write(D/f) sync(all) open(D/f) lseek(D/f) sync(all)	BadFd MMI
9	creat(D/f) write(D/f) close(D/f) sync(all) creat(D/f) write(D/f) close(D/f) sync(all)	✓ IIII
10	open(f) write(f) close(f) rname(f,f1) open(f) write(f) close(f) rname(f,f1) #nlink=0	NRevoke,DLoss IIIA
11	open(D/f) write(D/f) close(D/f) rname(D/f,D/f1) sync(all) open(D/f) write(D/f) close(D/f) rname(D/f,D/f1) sync(all)	✓ IIIII
12	open(D/f) write(D/f) close(D/f) rname(D/f,D/f1) sync(D) open(D/f1) write(D/f1) close(D/f1) rname(D/f,D/f1) sync(D)	DLoss MAAII
13	creat(D/f1) write(D/f1) sync(D/f1) CreatReuseDInode(D/f1) lseek(D/f1) sync(D/f1)	BadFd,FLoss MMI
14	creat(D/f1) write(D/f1) sync(D) open(D/f1) write(D/f1) sync(D)	BadFd,Garbage MAI
15	creat(D/f1) write(D/f1) close(D/f1) sync(D/f1) CreatReuseDInode(D/f1) write(D/f1) close(D/f1) sync(D/f1)	FLoss MIAI
16	creat(D/f1) write(D/f1) close(D/f1) sync(D) open(D/f1) write(D/f1) close(D/f1) sync(D)	Garbage MAAI

Figure 5: **Example System-Call Sequences and output of AIM**. *The first row of each group shows the system call sequence (rname: rename) and the consequence of a restarted uFS that drops the state gap (DLoss: DataLoss; FLoss: FileLoss; NRevoke: Rename Revoked; ✓: Successful) after a single p-crash. The second row shows the system calls performed by Ananke during replay (green operations are modified; gray operations are ignored) and the AIM outcome.*

Finally, to ensure the determined operation type can be executed successfully, we must use proper arguments regardless of whether the original form (i.e., *Act*) is employed. Specifically, we need to ensure that an fd resolves to the designated inode; we may need to modify the pathname argument so that it resolves to the proper inode, even if the specific path↔inode has been changed (and persisted).

We believe our p-log and AIM mechanisms for p-crash recovery are applicable to POSIX-compliant filesystems and filesystems whose API are based on these three abstractions. The specific number of bits needed for targets_status can vary in each filesystem depending on the number of targets that an API can change.

Algorithm. We now describe how Act/Ignore/Modify is determined.

Ignore: an operation can be safely ignored if and only if all associated file descriptors are closed, and all inode-related state updates have been made durable. That is, all the bits in the targets_status are cleared.

Act: the operation is replayed in its original form if and only if the following three conditions are true: 1) If a file descriptor was updated, either the file descriptor is still open or it is closed but the corresponding inode has not been synced.

2) None of the referenced inodes have been synced. 3) For any path-based operation, the dependent path↔inode that was changed by subsequent operations must not have been persisted; that is, the number of bits set for inodes in the targets_status matches the inode targets (in targets[1:]), and any dependent path↔inode is still valid on disk.

Modify: if neither Act nor Ignore applies, then a modified form of the operation must be replayed.

AIM finalizes the pathname argument for a path-based operation during recovery, with the assistance of additional data structures. AIM scans the p-log entries in order and maintains a map of path to operations that depend on them. When a pathname is changed by subsequent operations (e.g., rename), AIM can thus find the previous operations and update the pathname accordingly.

Examples. Figure 5 shows examples of p-log sequences where a p-crash occurs after the last operation; the first row for each example shows the outcome if state gap is not handled after recovery; the second row shows the transformations performed by AIM. Note that the sync operations can either be explicit calls from the application or internal background flushes performed by the filesystem.

Example 1 shows that if recovery does not replay this sequence, then subsequent accesses by the application lead to BadFd errors; AIM replays the open but modifies the read to lseek since the read data was already returned to the application. Example 2 illustrates that operations whose effects have not been persisted, must be replayed, or data would be lost; example 3 shows that if a write has been persisted, the write must be changed to an lseek to still update the file offset while not overwriting (potentially changed) existing data. Example 12 shows that a sync of the directory, including the effects of a rename, requires the new pathname for open during replay. The more complex examples 13 and 15 show how `CreatReuseDInode` is used when s-crash recovery does not sync the directory inode upon an fsync of a newly created file inode; for a filesystem that proactively syncs the directory inode (e.g., ext4), AIM works properly since the sync of the directory is reflected in the log entry, allowing all operations to be Ignored.

Physical Dependencies. One might wonder how AIM handles physical dependencies across logically independent operations [17, 45]: these dependencies are handled automatically by the p-log's mechanism for tracking updates to inodes in combination with the durability protocol of the filesystem. For instance, consider the dependencies due to a coupled imap, where logically independent inodes are stored in the same physical block. When creat(D0/f0) and creat(D1/f1) depend on the same imap, and sync is called on D0, the file and directory inodes for both D0 and D1 are synced in one transaction to maintain on-disk consistency (i.e., in s-log). When sync(D0) is completed, the p-log entries for both create operations are updated to reflect that all four inodes have been synced. With these p-log entries, as desired, AIM will not replay the create operations; if the file descriptors have been closed, AIM will ignore the operations, also as desired. In summary, tracking updates based on actual changes at the file descriptor and inode level makes it straight-forward to follow complex dependencies across operations.

P-log Concurrency. The original filesystem decides which core handles an operation; Ananke logs an operation on the core that completes it. Upon a sync that involves multiple inodes, logged operations related to each inode must clear the status in the targets_status. The completion of a sync is asynchronously propagated to cores via messages, after which each core updates its private p-log.[1] Ananke borrows the idea of using an operation's timestamp as the linearizability point to form a global order such that each private per-core log avoids extra inter-core communication in the critical path [4].

P-log Replay. After the p-log is transformed by AIM, the fresh process replays the p-log actions to fully restore the filesystem. The replay actions reuse as much of the existing filesystem machinery as possible.

Ananke includes several new internal APIs. In the internal API, create/mkdir can specify an inode number, because the inode number is visable to applications, as specified in the POSIX standard. Similarly, open can specify a file descriptor number. `CreatReuseDInode` is introduced to accommodate the situation where, as shown in Figure 4, the effect of creat initializing an inode is not needed (as the subsequent writes have been persisted), but the changes to path↔inode and fd need to be redone.

P-log Garbage Collection. Garbage collection of p-log entries is important so that the p-log does not consume too much memory. Garbage collection of the p-log is based simply on AIM: when an operation is safe to Ignore, the log-entry can be garbage collected. Unlink and rename are preserved until no file descriptor relies on the involved inodes.

4 Implementation

Ananke is based on the open-source repository of uFS [36], to which we add p-crash detection and recovery mechanisms; we add approximately 4K LoC.

Basic Flow of Control. In uFS, each operation comes from an application via IPC in a shared per-client message ring. The command stays in the message ring until completed, at which point it is marked; a client can poll on the completion bit to know the operation is finished and obtain results.

The p-log is realized as a set of per-core logs. For high performance, each worker thread in Ananke has its own private p-log and thus need not worry about concurrent updates [4]. Entries are added to the p-log after an operation completes, at which point Ananke allocates an entry and copies the mes-

[1] uFS handles directory inodes on a single core (primary), and operations on a given inode are handled exclusively by one core most of the time for better CPU-core locality and scalability, which helps reduce the complexity of such protocols.

sage to the p-log. If the filesystem fails while an unlogged operation is executing, the message ring contains the information needed to re-execute the operation. System calls that do not change in-memory state (e.g., stat) are not logged.

Exactly-Once Semantics. To ensure correct recovery, each request must update system state exactly once [55]. After restart, Ananke will replay operations in the p-log and the remaining in-flight operations in the message ring.

Ananke updates the p-log before replying to ensure that it does not lose an operation. If Ananke first marked the reply complete and then logged the operation, it would risk losing the operation if a p-crash occurred immediately after setting the completion status. However, updating the p-log first risks double execution, if the filesystem crashes immediately after logging and but before setting the status bit.

To avoid double execution, Ananke follows a careful update protocol, as shown in Figure 3 (b). In step 1, the system call information is written to the p-log; in step 2, other information about the entry is updated, including a reference to the message ring entry (i.e., ipc_idx). A (low-cost) compiler barrier is inserted between these two updates to avoid compiler reordering [25]; no memory fence is required because each p-log has a single writer and possible reading of the p-log during recovery is handled by a single recovery thread [43]. In step 3, Ananke sets the status bit in the message ring, and finally, in step 4, Ananke clears the p-log entry's ipc_idx (and atomically updates the self_crc to match). During recovery, when Ananke finds a p-log entry containing a valid logical offset of ipc_idx, it compares it with the corresponding message (pointed to by ipc_idx) to determine if the on-ring message needs to be discarded.

P-Log Replication. Two copies of the p-log are maintained to enable recovery given a single p-log corruption. For each update, Ananke writes first to the primary, and then to the secondary, with a compiler barrier between. When updating the status of a p-log entry, a CRC validation is first performed for the primary. If it fails, the replica CRC is validated and copied into the primary. During recovery, if the primary is corrupted, the replica is used (if its CRC validation succeeds). If both are corrupted, recovery aborts.

Checksum. We add checksums (CRC) to each core metadata structure to help detect corruption. For inodes, the on-disk representation has reserved space for padding to the disk block size, so we embed a one-byte checksum into the existing representation. For the datablock bitmaps, inode bitmaps, and dentry blocks, we add a one-byte checksum for every 32 bytes; calculating checksums for this smaller chunk (instead of 4KB) reduces the amount of memory touched per checksum and can leverage modern CPU hardware-accelerated instructions [26].

Kernel-coordinated Speculative Restart. In our current implementation, the kernel invokes a signal handler of the main process, which saves the data pages related to p-log entries. This procedure is invoked in the main process but not on the worker threads' stacks (by `sigaltstack`). The notification to the fresh process is implemented using a mutex shared between the main and fresh processes with the robustness attribute set to `PTHREAD_MUTEX_ROBUST` [1]. This attribute synchronizes the termination of the mutex holder to the fresh process.

Limitations and Assumptions. We assume the logging code is correct and the stack is intact during a recovery. Another assumption is that saving data pages to a known location (currently in tmpfs) can be done successfully. We assume that data page corruption is handled by applications; as such, we do not add protection to, or use redundancy for, data pages.

Ananke has not been specifically designed to handle memory corruption of the shared message ring (i.e., App-Worker MsgRing in Figure 2). Our rationale is that the MessageRing is part of the shared state between the applications and the filesystem, regulated by their security contract. For example, a misbehaving application can also corrupt the message ring. A corruption protection mechanism ensuring security would be interesting for future work.

5 Evaluation

We demonstrate: 1) Ananke delivers transparent recovery for applications without sacrificing common-case performance. 2) Ananke recovers from a large number (30,000+) of controlled and random fault-injections emulating both fail-stop p-crashes and data corruption 3) Ananke incurs negligible performance and memory overhead. 4) Ananke recovers in an acceptable amount of time ($\leq$ 400ms), based on the number of logged operations.

5.1 Methodology and Comparison Systems

We compare Ananke to the two baseline systems we introduced in Section 2: uFS and uFS-Sync, both of which incorporate the restarting mechanism designed for Ananke. uFS relies solely on s-crash recovery, while uFS-Sync forces updates to be directly flushed to disk, eliminating the state gap. We also implement Membrane-style replay [63], which only logs the operations and requires a full sync for each fsync and operations like create/mkdir to simplify the state gap.

To emulate faults that cause fail-stop p-crashes, we insert null pointer dereferences which cause a SIGSEGV. To stress the exactly-once semantics ensured by the p-log, we also inject p-crashes during client operations. We assume that faults are transient and that the same fault does not reoccur during replay; handling deterministic errors requires additional techniques [35, 52]. To emulate faults that lead to memory corruption, we inject targeted memory corruption into the filesystem process address space, as the literature shows that random bitflips often do not manifest as detectable errors [5,13]. We exhaustively corrupt each memory region (e.g., stack, heap, etc.) by uniformly filling a particular chunk of memory with a specific value.

Our evaluations are performed on a machine with 128GB

Workload	close	create	fstat	fsync(dir)	fsync(file)	lseek	mkdir	open	opendir	pread	pwrite	read	rename	stat	unlink	write
Sort	✓	✓	✓			✓		✓				✓		✓	✓	✓
CpDir	✓	✓	✓				✓	✓	✓			✓		✓		✓
Unzip	✓	✓				✓	✓	✓				✓		✓		✓
SQLite	✓	✓	✓	✓	✓			✓		✓	✓			✓	✓	
LevelDB	✓	✓		✓	✓			✓	✓			✓	✓	✓	✓	✓

Figure 6: **Operations in Applications.** *The five applications use a range of system calls as shown with ✓. Sort is an external gnu sort over 10M data. CpDir and Unzip operate on a 5-directory tree with depth of 3 and 4 files of sizes 100KB, 110KB, 200KB, and 210KB. SQLite performs a sequential load with 400 keys, 2 transactions. LevelDB performs a sequential load of 2000 keys.*

RAM, AMD 2.80GHz CPU, and an NVMe SSD (Samsung PM173X) with raw device latency around 70us [15]. The garbage collection threshold for the p-log in Ananke is 4MB.

5.2 Transparent Recovery vs. Performance

We briefly summarize the results from the experiments originally shown in Figure 1, where we run LevelDB with three continuous workloads (Load, YCSB-A, YCSB-C), inject a single p-crash, and report the throughput delivered by LevelDB over time for uFS, uFS-Sync, and Ananke.

With uFS, throughput is high until the p-crash occurs, at which point, uFS restarts, replays the on-disk journal for s-crash consistency, and recovers its connections with the LevelDB client. However, on its next interaction with uFS, LevelDB encounters an error and exits, requiring manual intervention to repair the database. In contrast, uFS-Sync transparently recovers from the p-crash, but common-case performance suffers significantly on non-read-only workloads; in particular, YCSB-Load (sequential writes) and YCSB-A (1:1 read:write mix) are approximately 5x and 3x slower.

Finally, as desired, Ananke transparently recovers from the p-crash while still delivering high throughput before and after the p-crash. We make several observations. First, for YCSB-Load, the periodic drops in throughput occur due to compaction threads in LevelDB performing additional I/O; performance is similar in uFS and Ananke. Second, for write-intensive workloads (Load), throughput does not drop after recovery since Ananke re-uses dirty data pages; for read-only workloads (YCSB-C), performance drops but increases again, because clean pages are discarded and refetched to rewarm the page cache. Finally, the recovery time of Ananke is sometimes slightly faster than that of uFS-Sync; recovery time is explored in detail below (§5.6).

5.3 Transparent Recovery: Recovering the State Gap

To stress the recovery of the uFS variants on p-crashes, we inject 30,000+ faults in workloads of real applications and random p-crashes while running multiple applications.

5.3.1 Real Application Behavior

We evaluate how different system utilities (gnu-sort, cp, unzip) and data-intensive applications (SQLite and LevelDB) react to simple p-crashes and restarts with uFS and Ananke.

Workload	#of ops	Fail after each op					Fail during an op
		uFS				Ananke	Ananke
		S_OK	F_OK	F_BAD	S_BAD	S_OK	S_OK
Sort	5327	4	0	5322	1	5327	5327
Sort (w/s)	5327	4	0	5322	1	5327	5327
CpDir	82	4	0	77	1	82	82
Cpdir (w/s)	82	6	0	71	5	82	82
Unzip	77	11	6	47	13	77	77
Unzip (w/s)	77	18	6	27	26	77	77
SQLite	154	3	89	62	0	154	154
SQLite (w/s)	154	3	54	97	0	154	154
LevelDB	1997	5	26	1966	0	1997	1997
LevelDB (w/s)	1997	5	28	1964	0	1997	1997

Figure 7: **Transparent Recovery of Applications.** *A single p-crash is inserted after (or during) each system call of the five benchmark applications. With uFS, applications may return the wrong error code (F_OK and F_BAD) or have the wrong data (S_BAD and F_BAD); with Ananke, all applications are correct.*

Figure 6 shows these applications exercise a range of file system calls. For each workload, we inject a simple p-crash both after and during each of the 15,000+ filesystem operations; such injection attempts to cover more diverse state gap. Since behavior is dependent on when data is persisted to disk, we control the timing of flushes performed by uFS in the background (if an application directly calls fsync, the flush is performed immediately).

The desired result is for each application to exit with the same return code and produce the same content as when there is no p-crash. We characterize the actual results for each fault-injection experiment as S_OK, S_BAD, F_OK, and F_BAD, where S/F indicates the return code (success or failure) by the application and OK/BAD indicates whether the data is identical to an execution with no failures; thus, S_OK is ideal and S_BAD is strictly incorrect; F_OK and F_BAD may be acceptable, since they indicate an error the application could not handle, but require manual intervention.

Figure 7 shows that with uFS, applications are not robust to p-crashes with many instances of return code failures (F_OK/F_BAD) and bad data (S_BAD/F_BAD). A single background sync can change the consequences drastically. For the utilities (gnu-sort, cp, unzip), most of the cases are F_BAD because the utilities depend on opened file descriptors that are lost. More problematic cases of S_BAD occur because applications are not careful with fsync, ignore some failed return codes, or simply print error messages.

While simple utilities may not be expected to correctly retry operations when a filesystem returns an error, production-quality durability-aware libraries, such as SQLite and LevelDB, are designed to handle these cases with write-ahead-logging. With uFS, as desired, SQLite and LevelDB never return success if data was lost or corrupted (S_BAD=0): if there is a problem with the data, SQLite and LevelDB correctly return an error. However, in many cases, SQLite and LevelDB exit prematurely with error codes. In these cases, we reopen the database and try to read back the inserted keys: F_OK indicates the reopening succeeds and no data is lost; F_BAD signifies the reopening fails, and the database reports corruption or data loss. We have

Region	# of Cases	Successful Restart	Correct FS Metadata	Correct FS Data
Stack	15	15 (100%)	15 (100%)	11 (73%)
Heap	2547	2547 (100%)	2547 (100%)	2542 (99.8%)
DMA-mem: Metadata	375	375 (100%)	375 (100%)	375 (100%)
P-log	436	436 (100%)	436 (100%)	436 (100%)

Table 2: **Ananke Recovery with Memory Corruption**. *The corruption experiments fully enumerate each memory region and we report the number of cases as where the fault manifests to detected errors (e.g., filesystem or application errors). The percentages of cases with detected errors for each memory region are: 0.71%, 20.4%, 73.5%, and 100%.*

verified that offline tools can manually repair a corrupted database, but not recover lost data; in addition, requiring offline tools reduces system availability and burdens administrators [50, 69]. Thus, even durability-aware applications need more support than uFS.

To provide transparent filesystem availability, Ananke must go beyond merely restarting and rebuilding connections and instead ensure that all states are recovered properly. Figure 7 shows Ananke meets this goal: all five applications proceed successfully for all 30,000+ p-crash points, regardless of whether the p-crash occurred after or during a system call and whether or not a background sync occurred: as desired, the applications return S_OK and have identical data as when no p-crash occurs.

5.3.2 Multiple Processes

Ananke is a shared filesystem service that handles multiple client applications running concurrently. To demonstrate that Ananke transparently recovers multiple processes, we simultaneously run three applications (LevelDB, Sort, and CpDir) and inject p-crashes at 300 random points. In all cases, with Ananke the three applications continue executing correctly and return S_OK.

5.4 Transparent Recovery: Memory Corruption

We demonstrate that Ananke provides transparent p-crash recovery in the presence of memory corruption, showing the benefits of Ananke building from on-disk state and relying on the well-protected p-log; any corrupted state in memory is simply discarded.

We inject memory corruption into the four major memory regions of the filesystem process address space (as depicted by Figure 3): stack, heap, filesystem metadata (within DMA-able memory), and the p-log. After completing the first half of the workload (i.e., creating directories of files), the fault injection derives the runtime memory layout from `/proc/self/maps`, injects memory corruption in a region (64B in the stack; 4KB elsewhere), and then Ananke continues with the rest of the workload. At the end, the correctness of all filesystem metadata and data is verified. Our experiments exhaustively cover the corruption of all memory regions, with a total of 18,100 experiments.

Table 2 reports the 3,373 cases where the faults manifest into an error; corrupting memory that is not accessed by the workload does not lead to an error. In all 3,373 cases that

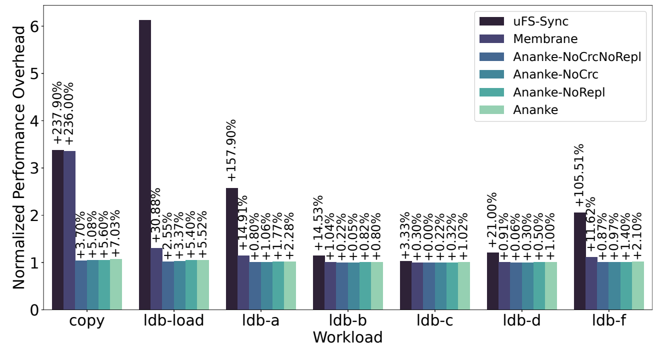


Figure 8: **Performance Overhead of uFS-Sync, Membrane, and Ananke Variants**. *Ananke-repl writes to 2 CRC'ed p-logs; Ananke-CRC adds CRCs to filesystem in-memory structures (updated on all writes; checked on all reads); Ananke-repl-CRC combines replication and CRCs.*

lead to errors and subsequent p-crashes, Ananke restarts successfully (column three) and has correct metadata (column four). In a few cases (4 corruptions of the stack and 5 of the heap), filesystem data is not recovered properly due to limitations in our current implementation: after the OS kernel monitors the exit of the main filesystem process, the main process runs a signal handler to rescue the data pages; thus corrupting these regions in the stack and heap halts the procedure. We expect to address this limitation by moving the rescue procedure to read-only kernel space (e.g., with eBPF).

5.5 Common-Case Overheads

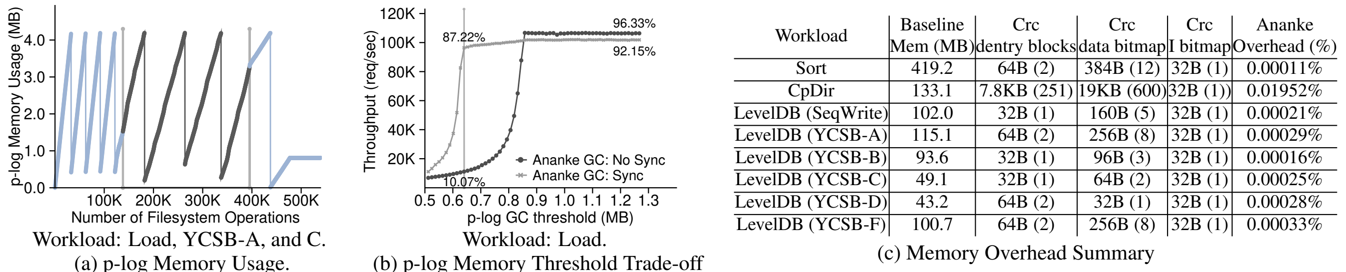
While both Ananke and uFS-Sync provide transparent recovery, uFS-Sync provides this with a costly method: persisting dirty data before returning to the client. In contrast, Ananke provides transparency efficiently by saving only a small p-log. We show that Ananke incurs negligible performance and memory overhead compared to the baseline uFS.

5.5.1 Performance Overhead

Figure 8 shows the performance overheads of uFS-Sync and Ananke normalized to uFS for a range of data-intensive applications: copy and LevelDB for Load and five YCSB workloads. We use sufficiently long workloads to trigger garbage collection in Ananke. We separate the sources of overhead in Ananke by examining variants that do not protect essential data structures with CRC and/or do not replicate the p-log.

uFS-Sync causes significant slowdown to the applications (up to 6x slower than uFS); the slowdowns are directly related to the amount of meta-data and data writes in the workload (e.g., LevelDB Load is the most write-intensive and has the most slowdown; YCSB-B, C, D are read-dominated and have low overhead; copy and YCSB-A and F have more balanced read/write ratios and intermediate overheads.

We have also implemented Membrane-style replay which requires an extra full sync of all dirty data and meta-data in order to handle fsync operations and directory updates like creat/mkdir [63]. For the copy workload that involves many directory/file creations, Membrane causes a significant slowdown (3.4x); it also incurs large overheads for LevelDB



Workload: Load, YCSB-A, and C.
(a) p-log Memory Usage.

Workload: Load.
(b) p-log Memory Threshold Trade-off

Workload	Baseline Mem (MB)	Crc dentry blocks	Crc data bitmap	Crc I bitmap	Ananke Overhead (%)
Sort	419.2	64B (2)	384B (12)	32B (1)	0.00011%
CpDir	133.1	7.8KB (251)	19KB (600)	32B (1))	0.01952%
LevelDB (SeqWrite)	102.0	32B (1)	160B (5)	32B (1)	0.00021%
LevelDB (YCSB-A)	115.1	64B (2)	256B (8)	32B (1)	0.00029%
LevelDB (YCSB-B)	93.6	32B (1)	96B (3)	32B (1)	0.00016%
LevelDB (YCSB-C)	49.1	32B (1)	64B (2)	32B (1)	0.00025%
LevelDB (YCSB-D)	43.2	64B (2)	32B (1)	32B (1)	0.00028%
LevelDB (YCSB-F)	100.7	64B (2)	256B (8)	32B (1)	0.00033%

(c) Memory Overhead Summary

Figure 9: **Memory Usage (LevelDB).** *In (c), the amount of Baseline Mem includes: one is the difference between the amount of memory in the data segment (heap) before and after running each workload; the second is the in-memory file system structures (corresponding to disk, including data blocks and metadata blocks); Overhead is calculated using the CRC memory added.*

Load, YCSB-A, and YCSB-F. Since Membrane only logs the operations, its overhead is smaller than that of Ananke for read-intensive workloads (e.g., YCSB-C), but still higher than Ananke without replication and CRC. The comparison with Membrane demonstrates the necessity of "no force flush" for high common-path performance.

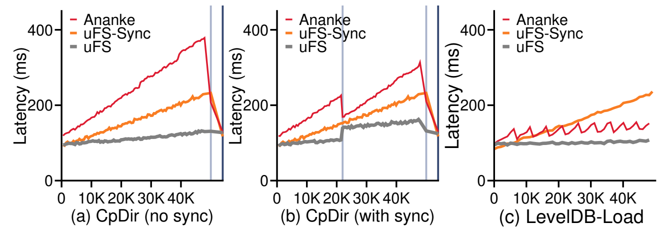
For all workloads, the performance overhead of Ananke is low: for very write-intensive workloads (copy and LevelDB-Load) the performance overhead of Ananke with no memory protection is less than 4%, while adding both p-log replication and CRC protection to essential data structures raises it to 7%; for all other workloads, the overhead of Ananke with full memory protection is below 2%.

5.5.2 Memory Overhead

Ananke adds a small amount of memory overhead to uFS in two ways: for the p-log (and its replica) and for CRC checksums. We demonstrate the memory usage of the p-log over time as well as the trade-off between the maximum size of the p-log and performance; we then quantify the memory overhead for CRC checksums.

The memory usage of the p-log is proportional to the number of requests that have not been garbage collected. Figure 9(a) presents the memory used by the p-log when LevelDB consecutively runs the Load, YCSB-A, and YCSB-C workloads. When the size of the p-log reaches a configurable threshold (4MB), Ananke performs garbage collection, reclaiming operations that will not be replayed on a p-crash; the p-log cannot be shrunk to 0 because some file descriptors remain open as LevelDB runs. As shown, the write-intensive Load workload fills the p-log faster than read-write YCSB-A, which fills the p-log faster than read-only YCSB-C.

Figure 9(b) illustrates the trade-off between the maximum size allocated to the p-log and performance for LevelDB-Load. We consider two forms of garbage collection, both of which are triggered when the p-log reaches a threshold: GC-NoSync which returns if no p-log entries can be reclaimed; GC-Sync, which triggers a background sync if no p-log entries were reclaimed. For both types of garbage collection, LevelDB throughput is unacceptably low if the p-log threshold is too small. GC-Sync enables a smaller p-log to be used with acceptable performance compared to GC-NoSync (e.g., 0.65MB instead of 0.85MB), but slightly reduces LevelDB



(a) CpDir (no sync) (b) CpDir (with sync) (c) LevelDB-Load

Figure 10: **Restart time with uFS, uFS-Sync, and Ananke.** *The light blue vertical lines indicate the time where a background sync occurs; the deep blue (last) vertical lines indicate a checkpoint. X-axis is the timing (# of ops) of a p-crash.*

performance when the p-log is sufficiently large (92% of the baseline instead of 96%) due to more frequent sync operations. Thus, we use GC-NoSync with a p-log threshold (4MB) that is more than sufficient for even the most write-intensive workloads.

Figure 9(c) summarizes CRC memory overhead relative to the in-memory size of each workload. Each in-memory 4K block of inode bitmaps, data bitmaps, and directory entries uses 32 one-byte CRCs; the one-byte CRC for each inode is embedded within the existing inode structure and therefore does not require extra memory. Thus, CRC adds minimal memory overhead (at most 0.02%).

5.6 Recovery Performance

In our final experiments, we show recovery time for uFS, uFS-Sync, and Ananke in Figure 10. Since recovery time depends on the state gap, we consider two write-intensive workloads: CpDir and LevelDB-Load. We inject a p-crash after every 500 system calls (shown along the x-axis), with each data point corresponding to a single p-crash (i.e., one experiment); the y-axis is the time for recovery after that p-crash point. All approaches recover in less than 400ms, since kernel-coordinated speculative restart overlaps the 2.9 seconds to rebuild the device connection with running the workload.

For uFS, recovery time is relatively constant. Note that even though uFS restarts, the currently-running application may not be able to usefully continue. uFS sees a jump in recovery time after a sync is performed (CpDir w/ sync at 20K ops) because more operations exist in the s-log. The very small increase in recovery time for other workloads as they make more progress occurs because the kernel must de-

stroy a larger uFS address space. For uFS-Sync, recovery time increases as the applications run longer because every operation is persisted to the s-log; the time for recovery is minimal at the end after uFS-Sync performs an s-log checkpoint (marked by the last vertical line).

Finally, the recovery time of Ananke is directly related to the number of operations in the p-log. The size of the p-log is reduced when the application calls fsync or a background sync is performed, as illustrated in CpDir when a background sync is performed at the first vertical line and in LevelDB-Load periodically with foreground syncs. While recovery may be faster in uFS or uFS-Sync, recovery time in Ananke is bounded by the maximum size of the p-log and remains less than 400ms even for write-intensive workload with no fsyncs.

6 Related Work

Hardware and Software Faults. Real-world faults and studies motivate transparent filesystem recovery and Ananke. Hardware corruption is real. The study of DRAM and SRAM faults [61] revealed that hardware-based resilience techniques (e.g., ECC) cannot perfectly detect and repair the hardware faults, resulting in unpredictable undetected errors (e.g., corruptions) for software systems to handle. Unfortunately, users may not adopt DRAM with strong hardware protections due to cost [73]. More recently, Alibaba [70], Google [22] and Meta [14] reported that CPU core faults can lead to silent data corruption.

Software bugs are also prevalent, often causing crashes or memory corruption. Lu *et al.* [38] found that around 20% of bugs lead to machine crashes in their study of Linux kernel filesystem patches. Data corruption accounts for the largest percentage (40%) of bug consequences, although it is unclear how many of these bugs cause in-memory corruption. Bugs are also common in the Linux memory management subsystem, resulting in a significant number of crashes [23].

Fault Tolerant Systems. Filesystem fault tolerance can be improved by enhancing detection to fail fast. Recon [18] and WAFL [31] both check in-memory filesystem structures to detect semantic violations when committing updates to disk; they are complementary to Ananke.

Previous works also attempt to recover from kernel filesystem crashes. Membrane [63] introduces a restarting framework for kernel filesystems that uses sophisticated techniques to unwind the threads, unmount the filesystems, and replay logged operations. Yet, it requires additional flushing in the common path; it also does not fully leverage a clean address-space and is vulnerable to memory corruption.

Rio file cache [10] preserves the kernel file cache and enables an automatic warm reboot when the OS kernel crashes; Rio cannot recover when semantic states in the cache are corrupted before the crash. Nova-Fortis [71] is a kernel NVM filesystem that tolerates the corruption of NVM devices, highlighting the problem of memory corruption when filesystem data is in persistent memory. Sinnce Rio's file cache essentially functions as persistent memory used by the restarted OS, memory corruption cannot be overlooked.

Ananke is also similar to efforts such as microreboot [7] and Rx [52]. However, microrebooting [7] requires applications to be designed in a *crash-only* fashion, where important state must be saved in a separate data store (e.g., a transactional database). A crash-only filesystem will incur significant overhead with large amount of data. Rx [52] makes several retrying attempts from a checkpoint by altering various environmental factors while replaying; whole-system checkpointing with multiple versions is costly for a filesystem.

Efforts have been made to make operating system kernels more robust to failures. Nooks [64] and Shadow Driver [65] were designed to recover the kernel from a component (i.e., driver) crash. Recovery Domain [32] provides request-oriented (e.g., a system call) recovery for the OS kernel, but assumes that a fault's influence is limited to a single request without affecting the rest of the kernel. Otherworld [13] microreboots the monolithic OS kernel and reuses the application's address space in the new kernel; it risks reusing corrupted memory. Ananke only reuses the trusted p-log.

Some previous work on microkernels adds fault-tolerance machinery [3, 33, 66]. CuriOS [12] stores service states in clients' address spaces with virtual memory based protection to survive a service crash; however, the memory permissions and process isolation incur significant overhead. OSIRIS [3] and TxIPC [33] incorporate instruction-level undo logs to roll back problematic in-flight requests but cannot handle bad states (e.g., corruption) that occurred earlier than the in-flight request. Rust-based microkernels such as Redleaf [46] and Theseus [5] provide fault recovery, but they are not yet mature enough to support POSIX-compliant filesystems.

7 Conclusion

We presented the design, implementation, and evaluation of Ananke. Through thorough experimentation, we demonstrated that Ananke handles a wide range of faults and recovers transparently underneath running applications. Ananke does so while providing high performance in the common case, thus showing that fault tolerance does not need to incur excessive costs.

8 Acknowledgement

We thank Angela Demke Brown (shepherd) and the anonymous reviewers for their comments, which greatly improved our paper. We thank Swaminathan Sundararaman and Sriram Subramanian for discussion on Membrane. This material was supported by funding from NSF grant CNS-2402859, Microsoft, and Influxdata. Jing Liu was partially 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] pthread_mutexattr_setrobust(3). , 2024.

[2] David F. Bacon. Detection and Prevention of Silent Data Corruption in an Exabyte-scale Database System. In *The 18th IEEE Workshop on Silicon Errors in Logic System Effects*, 2022.

[3] Koustubha Bhat, Dirk Vogt, Erik van der Kouwe, Ben Gras, Lionel Sambuc, Andrew S. Tanenbaum, Herbert Bos, and Cristiano Giuffrida. Osiris: Efficient and consistent recovery of compartmentalized operating systems. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 25–36, 2016.

[4] Srivatsa S. Bhat, Rasha Eqbal, Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Scaling a File System to Many Cores Using an Operation Log. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*, Shangai, China, October 2017.

[5] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. 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, November 2020.

[6] Mike Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, WA, November 2006.

[7] 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)*, San Francisco, CA, December 2004.

[8] Dongjie Chen, Yanyan Jiang, Chang Xu, Xiaoxing Ma, and Jian Lu. Testing file system implementations on layered models. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, Seoul, South Korea, 2020.

[9] Haibo Chen, Xie Miao, Ning Jia, Nan Wang, Yu Li, Nian Liu, Yutao Liu, Fei Wang, Qiang Huang, Kun Li, Hongyang Yang, Hui Wang, Jie Yin, Yu Peng, and Fengwei Xu. Microkernel goes general: Performance and compatibility in the HongMeng production microkernel. In *Proceedings of the 16th USENIX Conference on Operating Systems Design and Implementation (OSDI '24)*, Santa Clara, CA, July 2024.

[10] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher Aycock, Gurushankar Rajamani, and David Lowell. The Rio File Cache: Surviving Operating System Crashes. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, Cambridge, MA, October 1996.

[11] D. R. Cheriton and W. Zwaenepoel. The distributed V kernel and its performance for diskless workstations. In *Proceedings of the 9th ACM Symposium on Operating System Principles (SOSP '83)*, pages 129–140, Bretton Woods, New Hampshire, October 1983.

[12] 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, CA, December 2008.

[13] Alex Depoutovitch and Michael Stumm. Otherworld: Giving applications a chance to survive os kernel crashes. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys '10)*, Paris, France, April 2010.

[14] Harish Dattatraya Dixit, Sneha Pendharkar, Matt Beadon, Chris Mason, Tejasvi Chakravarthy, Bharath Muthiah, and Sriram Sankar. Silent Data Corruptions at Scale. *CoRR*, abs/2102.11245, 2021.

[15] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (USENIX '19)*, Renton, WA, July 2019.

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

[17] 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)*, pages 307–320, Stevenson, WA, October 2007.

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

[19] Per Brinch Hansen. The Nucleus of a Multiprogramming System. *Communications of the ACM*, 13(4):238–241, April 1970.

[20] 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.

[21] Dan Hildebrand. An architectural overview of qnx. In *USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 113–126. Citeseer, 1992.

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

[23] Jian Huang, Moinuddin K. Qureshi, and Karsten Schwan. An Evolutionary Study of Linux Memory Management for Fun and Profit. In *Proceedings of the USENIX Annual Technical Conference (USENIX '16)*, Denver, CO, June 2016.

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

[25] Google Inc. Google Benchmark: Preventing Optimization. https://github.com/google/benchmark/blob/main/docs/user_guide.md#preventing-optimization.

[26] Intel. Intelligent Storage Acceleration Library. https://github.com/intel/isa-l, 2023.

[27] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Stonebraker Michael, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: A high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499, aug 2008.

[28] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP Acceleration as an OS Service. In *Proceedings of the EuroSys Conference (EuroSys '19)*, Dresden, Germany, March 2019.

[29] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. 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, October 2019.

[30] 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. seL4: Formal Verification of an OS Kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, Big Sky, Montana, October 2009.

[31] Harendra Kumar, Yuvraj Patel, Ram Kesavan, and Sumith Makam. 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, February 2017.

[32] Andrew Lenharth, Vikram S. Adve, and Samuel T. King. Recovery Domains: An Organizing Principle for Recoverable Operating Systems. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*, WA, DC, March 2009.

[33] Wentai Li, Jinyu Gu, Nian Liu, and Binyu Zang. Efficiently recovering stateful system components of multi-server microkernels. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 494–505, 2021.

[34] Jochen Liedtke. Improving ipc by kernel design. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*, Asheville, North Carolina, December 1993.

[35] Jing Liu, Xiangpeng Hao, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, and Tej Chajed. Shadow filesystems: Recovering from filesystem runtime errors via robust alternative execution. In *Proceedings of the 16th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '24, pages 15–22, 2024.

[36] Jing Liu, Anthony Rebello, Yifan Dai, Chenhao Ye, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Scale and Performance in

a Filesystem Semi-Microkernel. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '21)*, Virtual Event, Germany, October 2021.

[37] R. Lorie. Physical Integrity in a Large Segmented Database. *ACM Transactions on Databases*, 2(1):91–104, 1977.

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

[39] Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Physical Disentanglement in a Container-Based File System. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.

[40] Ao Ma, Chris Dragga, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. ffsck: The Fast File System Checker. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST '13)*, San Jose, CA, February 2013.

[41] 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. Snap: a Microkernel Approach to Host Networking. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '19)*, Ontario, Canada, October 2019.

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

[43] Paul E. McKenney. Is parallel programming hard, and, if so, what can you do about it? (release v2023.06.11a), 2023.

[44] 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.

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

[46] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. RedLeaf: Isolation and Communication in a Safe Operating System. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI '20)*, Virtual Conference, November 2020.

[47] Neha Narula, Cody Cutler, Eddie Kohler, and Robert Morris. Phase reconciliation for contended In-Memory transactions. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.

[48] Andrew Newell, Dimitrios Skarlatos, Jingyuan Fan, Pavan Kumar, Maxim Khutornenko, Mayank Pundir, Yirui Zhang, Mingjun Zhang, Yuanlai Liu, Linh Le, Brendon Daugherty, Apurva Samudra, Prashasti Baid, James Kneeland, Igor Kabiljo, Dmitry Shchukin, Andre Rodrigues, Scott Michelson, Ben Christensen, Kaushik Veeraraghavan, and Chunqiang Tang. RAS: Continuously Optimized Region-Wide Datacenter Resource Allocation. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '21)*, Virtual Event, Germany, October 2021.

[49] Edmund B. Nightingale, John R. Douceur, and Vince Orgovan. Cycles, Cells and Platters: An Empirical Analysis of Hardware Failures on a Million Consumer PCs. In *Proceedings of the EuroSys Conference (EuroSys '11)*, Salzburg, Austria, April 2011.

[50] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.

[51] 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, CA, April 2005.

[52] 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, UK, October 2005.

[53] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew. Machine-Independent

Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 31–39, Palo Alto, CA, 1987.

[54] Richard F. Rashid and George G. Robertson. Accent: A communication oriented network operating system kernel. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP '81)*, Pacific Grove, CA, December 1981.

[55] Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter Sewell. SibylFS: Formal Specification and Oracle-Based Testing for POSIX and Real-World File Systems. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, Monterey, California, October 2015.

[56] 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.

[57] Jerome H Saltzer, David P Reed, and David D Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS)*, 2(4):277–288, 1984.

[58] Russel Sandberg. The Design and Implementation of the Sun Network File System. In *Proceedings of the 1985 USENIX Summer Technical Conference*, pages 119–130, Berkeley, CA, June 1985.

[59] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. 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, 2009.

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

[61] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B. Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. Memory Errors in Modern Systems: The Good, The Bad, and The Ugly. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*, Istanbul, Turkey, March 2015.

[62] Sun Microsystems. ZFS: The last word in file systems. www.sun.com/2004-0914/feature/, 2006.

[63] 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):1–30, 2010.

[64] 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.

[65] 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, CA, December 2004.

[66] Andrew S Tanenbaum, Jorrit N Herder, and Herbert Bos. Can we make operating systems reliable and secure? *Computer*, 39(5):44–51, 2006.

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

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

[69] User Requests Help to Restore Corrupted Data due to WiredTiger Panic. Corrupt Collections WT Panic ERROR. https://jira.mongodb.org/browse/SERVER-32795?jql=text%20~%20%22WT_PANIC%22, 2018.

[70] Shaobu Wang, Guangyan Zhang, Junyu Wei, Yang Wang, Jiesheng Wu, and Qingchao Luo. 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, October 2023.

[71] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*, Shangai, China, October 2017.

[72] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In

Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP '87), pages 63–76, Austin, Texas, November 1987.

[73] 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, CA, February 2010.

[74] Asmae Ziani. Understanding the Differences between Popular Operating Systems. https://medium.com/@asmaeziani47/understanding-the-differences-between-popular-operating-systems-77aec4bdcddc, 2023.