Scale, Performance, and Fault Tolerance in a Filesystem Semi-Microkernel

By

Jing Liu

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

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2024

Date of final oral examination: August 26th, 2024

The dissertation is approved by the following members of the Final Oral Committee:

Andrea C. Arpaci-Dusseau, Professor, Computer Sciences
Remzi H. Arpaci-Dusseau, Professor, Computer Sciences
Michael M. Swift, Professor, Computer Sciences
Irene Zhang, Principal Researcher, Microsoft Research
Xiangyao Yu, Assistant Professor, Computer Sciences
Kassem M. Fawaz, Associate Professor, ECE

To my mother, for the begining chapter of education.
and
To my advisors, for the ultimate chapter of education.

Acknowledgments

The past several years in graduate school have profoundly reshaped me, both professionally and personally. The world has been different – COVID and GenAI have perhaps changed everything. I have also changed, from someone who was once clueless, barely spoke English, struggled to order a sandwich, and was uncomfortable with any cheese, to someone who delivered two hour-long oral defenses to a committee of world experts, and who now loves Wisconsin cheese and Madison. I want to express my deepest gratitude to the many people who have influenced, inspired, supported, and spent time with me.

First and foremost, I would like to thank my advisors, Andrea Arpaci-Dusseau and Remzi Arpaci-Dusseau. This PhD would not have been possible without their unconditional support, guidance, and patience. Words cannot express how lucky I am to have worked with them, and endless thanks are woven into my heart. From the moment Remzi shook my hand in his office when we first met, and from Andrea's elevator conversation about Erlang and Riak when we first talked, to the moment where I am able to conclude this journey with confidence to pursue a research career, all these memories are flooding back to me. I'm smiling with great happiness, and I know all of these memories will empower me in the future, always.

I am deeply grateful to Andrea and Remzi for their constant efforts

to "make me comfortable" in the earlier years, to find the "interface" to understand what I am doing, and to figure out the most effective way to teach me – generously allowing me to closely observe their art of doing things. I know my language skills were terrible when I started, but it's only in retrospect that I realize how much trouble that might have caused them, and how amazing it is that everything eventually worked out. As Andrea said, "we are always trying" (and yes, I also tried very hard because of this). I am grateful to them for giving me the opportunity to be an instructor for the undergraduate OS course, which fulfilled one of my lifelong wishes. The last year of graduate school was the toughest year of my life due to personal reasons, and without their support and help (even while they were on sabbatical), I would not have been able to get back on track.

Remzi has a magical ability to simplify things, helping me to make progress from a sensible starting point and approach problems from an easier angle. He also taught me how to communicate, design experiments, present results, and give great talks, all in his uniquely insightful and humorous way. I enjoyed working as a TA and helping to run course projects with Remzi, which was a lot of fun and taught me how to organize and identify research projects and interact with graduate students. I am also grateful for the textbook and zplot tool he created, both of which have brought me great joy.

Andrea is a person of rigorously high standards and kindness. I often referred to her as "the golden standard of truth," and it feels like a great achievement when my work aligns with her appreciation and comments. No matter how random or confusing my questions might be, Andrea somehow always carefully listens, provides answers, and often points out the way. I appreciate that she allowed me to audit her distributed systems lectures as a TA, so I could learn her approach to thinking about system research and writing flow from the perspective of a senior graduate stu-

dent. I thank her for teaching me how to ask the right research questions, organize my thoughts, converge on the point, and write papers. Andrea inspired me to listen carefully, be patient, and stay calm when facing extreme difficulties.

I'd like to thank my committee members for their valuable feedback on this dissertation. Michael Swift, Irene Zhang, Xiangyao Yu, and Kassem Fawaz have all provided insightful perspectives, comments, and thought-provoking questions during my defense. I especially thank Mike for spending time reading my dissertation and proposal in great detail and providing valuable feedback. I took CS-736 (advanced operating systems) with Mike, and I greatly admire his deep expertise in systems and sharp thinking. Irene has been a great mentor to me since I interned with her at Microsoft Research, and that wonderful summer in 2018 gave me the opportunity to work on core OS research, Demikernel, and learn SPDK, which laid the foundation for my dissertation work.

I am fortunate to have worked and interacted with many talented people in the Arpaci-Dusseau Group, including Ramnatthan Alagappan, Vinay Banakar, Tingjia Cao, Youmin Chen, Yifan Dai, Aishwarya Ganesan, Xiangpeng Hao, Tyler Harter, Jun He, Guanzhou Hu, Sudarsun Kannan, Eunji Lee, Kai Mast, Yuvraj Patel, Leo Prasath Arulraj, Suyuan Qu, Anthony Rebello, Sambhav Satija, Kaiwei Tu, Zev Weiss, Kan Wu, Suli Yang, Chenhao Ye, and Shawn Zhong. It was a great pleasure to work with Suli on the TAM project during my first two years, and I was amazed by her "taste for logic," clarity of thought, and brightness. I especially thank her for helping with my job search and job talk preparation. I thank Ramnatthan for sharing the office, engaging in fruitful discussions on technical details and other topics, occasionally comforting my worries, and offering advice as my senior fellow. Aishwarya has provided key suggestions and support during critical moments. I also thank Ram and Aishwarya for inviting me to departmental events, introducing me to visitors,

and kindly offering opportunities to get involved in projects and learn from them. I thank Shawn Zhong for generously inviting me to join his eBPF project, sharing fun knowledge, and happily modifying details according to my meticulous comments.

I would like to extend my special gratitude to my friends and colleagues who directly helped me with my dissertation project. The worst thing in graduate school is probably solving problems in isolation, so I am fortunate to have had them by my side. Anthony Rebello is the epitome of an ideal teammate – an extremely gifted programmer and the kindest person as a friend. I am grateful for the moments we spent discussing designs, debugging, writing code, talking about food, and countless other fun conversations. Yifan Dai has an exceptional sense of system performance, and his completely different way of thinking has made our collaboration so productive and fruitful.

I thank Chenhao Ye for helping me with the uFS experiments and preparing for artifact evaluation; without his help, I could not have finished the preparations for the artifact evaluation and my PhD preliminary talk during that crazy week. Xiangpeng Hao is also the best kind of colleague to work with. Sudarsun has been a great mentor to me, especially in the initial stages of building uFS, and I am grateful to him for meeting with me regularly during the challenging COVID period. I thank Tej Chajed for his help with the fault tolerance aspects of our filesystem.

My PhD journey has greatly benefited from internships at Microsoft Research, Google Madison, and Alibaba. I was fortunate to be part of the Demikernel roster with Amanda Raybuck. I would like to thank Irene once again for setting the highest example as a mentor and junior researcher. I was deeply impressed by her ambition and ability to organize systems research so effectively. I wanted to work with her because she looked so confident during her job talk at Wisconsin, and I ended up learning so much more from her. Beyond her research success, I am

grateful to her and Dan Ports for fun activities, dinners, and their constant support for young students. Dan even shared self-deprecating jokes to comfort me when I made mistakes. I hope to be at least half as good as them in the near future. I would also like to thank Marc de Kruijf, another excellent mentor, from whom I learned a great deal through the Snap project. Additionally, I am grateful to many others at the Google Madison office who are incredibly smart and enjoyable to talk with. Lastly, I thank Winsor Hsu for giving me the opportunity to work at Alibaba and for the memorable summer in California.

I have been involved and worked with many other smart and diligent graduate students in the fantastic Wisconsin system group, WACM, and SACM, including Anjali, Hayden Coffey, Sonia Cromp, Deepak Sirone Jegan, Konstantinos Kanellis, Mark Mansi, David Merrell, Ashwin Poduval, Isha Padmanaban, Hyojoon Park, Suchita Pati, Rajesh Shashi Kumar, Bijan Tabatabai, Sujay Yadalam, and Hsuan-Heng Wu. Mark Mansi is a wonderful friend, and I also thank him for his help when I started working for WACM. I appreciate Anjali and Suchita for bravely recommending me to run WACM and helping me throughout the entire year. Working for WACM has been a great learning experience, and I am grateful for the opportunity. I was also fortunate to work with David, Sonia, Isha, Rajesh, and Hyojoon to organize the first in-person CS student research symposium after COVID, which was a fantastic team effort. I have been collaborating with system students to organize numerous system reading group meetings and welcome weekends, and I thank all of them.

I would also like to thank the staff and faculty in the computer sciences department for their support. Angela Thorp has offered great support as the program coordinator, and I especially thank her for our trip to the Grace Hopper Conference in 2019. I am grateful to Jinyi Cai for our conversations and for inviting me to the Thanksgiving lunch. I also appreciate the support systems, especially the ESL and writing services

provided by the University of Wisconsin. I thank Terry Nuckolls for his English-speaking classes and his "see you mañana." I am also grateful to Dottie Mayne for her feedback and corrections on writing issues during the weekly writing group meetings, and for her encouraging words like "no worries, just do your science" in response to my concerns. Finally, I thank Babcock Ice Cream – because you can't ask for more than that!

I thank many other friends in Madison who have made my life more enjoyable, including Zhicheng Cai, Junda Chen, Jiefeng Chen, Zhihan Guo, Qian Li, Yunhe Liu, Xudong Sun, Yutian Tao, Qisi Wang, Yunhao Zhang, Ji Zhou, and Jinman Zhao. I especially thank Yanfang Le for the wonderful friendship and our many conversations about life, work, and the future. I am grateful to Zhihan Guo and Kan Wu for the dinners together and for taking me to the hospital after an accident injury. Zhicheng Cai once even drove me to Chicago so I could catch a flight to HotStorage. I thank Yifan Dai for our weekly grocery shopping trips, which have been a great source of relaxation and balance. I also thank Tianyin Xu for his help, advice, and mentorship through many phone calls.

I would like to thank the people who deeply impacted me during my undergraduate years at Nanjing University. I thank Daoxu Chen for stopping me from quitting college (I bet he didn't even know this), teaching us what a university truly is, and his ability to teach complex algorithms in a delightfully simple way. He is the first person I met who could teach better than textbooks and inspire my interest in computer science. I thank my undergraduate advisor, Tongwei Ren, for guiding me into research and teaching me how to make progress every day by encouraging me to write a daily report and providing tireless comments every morning at 7 am. I am also grateful to Xiaohong Wang for her generous help in improving my English during my first year of college; without her, I probably would not have been able to start my graduate school journey. Due to my deep worry about not passing the CET exam and failing to get a bachelor's de-

gree, I visited her office hours, and we essentially began a weekly routine of correcting my poorly written articles for an entire year, though without any explicit agreement.

I would like to thank my family for their unwavering support. I thank my parents for working hard when they were young so I could have the freedom to learn. It took me time to realize how profoundly my mother's happiness enriched my life and helped build my foundation. I am grateful for the flexibility she provided in my early schooling, fostering good habits, and avoiding unnecessary pressure. I also thank my aunt and uncle for their support, especially during difficult moments when my aunt stepped in, called me, and took over responsibilities, allowing me to focus on my studies in my final year. I deeply thank my cousin, Jiawei Chen, who is essentially my sister, for growing up with me. I am grateful for her regular phone calls and her offer to visit me during my last year, which made me feel the warmth of family.

Finally, thank you, my dear readers, for reading this dissertation. "It is my joy to share an epoch with you." Before the age of seventeen, I had never heard of anyone among the millions around me earning a PhD from a world-class institution. My life has been full of fortune, and I wish the same luck to be with you as well.

Contents

List of Tables

List of Figures

Abstract

The landscape of computing is evolving rapidly, with storage devices offering microsecond-level latency and substantial bandwidth. However, monolithic OS kernels like Linux struggle to keep up. These kernels incur significant overheads, particularly in the filesystem stack, and face scalability challenges on multi-core CPUs. Developing kernel code is difficult and slow, and upstreaming a kernel feature can take months or even years. Moreover, the increasing complexity of hardware and software heightens the risk of failures, as a single fault can crash the entire system.

To address these issues, this work explores a semi-microkernel architecture, where the I/O subsystem operates as a standalone user-space service, while the rest of the OS remains in the monolithic kernel. We focus on one critical I/O subsystem: filesystems. Several filesystem semi-microkernels, including uFS, Nebula, and uFS-Shadow, were built to investigate this approach, emphasizing performance, resource elasticity, and fault tolerance. uFS is a fully functional, high-performance, and crash-consistent user-space filesystem following the semi-microkernel approach. Nebula, based on uFS, provides fast, robust, and seamless recovery upon unexpected faults, as if no failure ever occurs. uFS-Shadow, when incorporated into Nebula, improves the reliability of uFS by recovering from both transient and deterministic errors.

In the first part of this dissertation, we focus on the architecture of uFS,

emphasizing its high performance. uFS leverages polling-based I/O and high-performance inter-process communication (IPC) for low latency. uFS achieves multi-core scalability through a "shared-nothing" design, where files are partitioned among threads, allowing each server thread to operate independently.

In the second part, we address resource elasticity in uFS by incorporating load management. In the semi-microkernel architecture, the filesystem server can scale CPU resources independently of applications because the application and server threads are decoupled. This feature allows uFS to balance performance and CPU efficiency while adapting to dynamic workloads.

In the third part, we examine the issue where filesystem applications cannot continue after a server crash, even though the entire system remains unaffected. The server buffers updates in memory, creating a state gap between what the application perceives and what is on disk. Simply restarting the server risks losing these updates, leading to potential data loss or silent errors. To address this, we introduce exit activation, a process recovery mechanism, which is code that runs after a server crash and uses the failed process's memory to safely recover the state gap before it is reclaimed by the OS.

In the final part, we introduce *robust alternative execution* (*RAE*), an approach to enhance the reliability of an existing high-performance filesystem via a shadow filesystem. This shadow system, which prioritizes correctness, takes over when the base filesystem encounters errors. By simplifying its design and omitting performance optimizations, the shadow filesystem is less prone to bugs, thereby improving overall reliability.

Overall, this approach is best suited for scenarios where the local filesystem needs to be specialized for hardware or where preventing filesystem faults from crashing the entire system is critical, with a sufficient user base to drive further customizations and filesystem innovation.

1 Introduction

Commonly used operating systems – monolithic OS kernels – are slow in performance when confronting modern hardware. For instance, the performance of storage devices evolves rapidly; emergent ultra-fast block devices achieve ultra-low latency and high bandwidth (i.e., several microseconds and over GB/sec). Unfortunately, monolithic operating systems like Linux fall short in delivering their full speed to applications. The kernel software stack, such as filesystems, becomes the bottleneck [19].

The monolithic OS kernel approach is also slow in terms of velocity and customization during development and deployment [9, 32, 134, 142, 143]. For instance, upstreaming a new feature to the Linux kernel can take months or years, and rolling out a new kernel version requires a machine reboot. However, today's extremely large scale of computing, the diverse types of applications and workloads, and the heterogeneous hardware pose pressing needs for rapid development and customization of system services [161].

The monolithic OS kernels suffer from numerous reliability issues due to poor fault isolation between subsystems – a fault in one subsystem can crash the entire machine, leading to service downtime and data loss [34, 81, 130]. Beneath the system call interface that applications interact with, tens of millions of lines of code run in the kernel space, comprising complex pieces of software like filesystems, network stacks, drivers, scheduling, and more, all interacting without isolation. Among them, filesystems

contribute a significant portion of the kernel codebase, complexity, and a notably large number of bugs [34, 130].

Drawing inspiration from the microkernel-based approach, building system services as user-space processes naturally alleviates two of these issues – velocity and reliability. Additionally, new performance opportunities arise from modern hardware (e.g., multi-core CPUs and IO devices) [86, 90, 172, 180], host-device data transfer protocols [62], and specialized performance development kits facilitating user-space direct access to devices without kernel involvement [58, 181].

As such, a practical approach to evolving OS architectures comes into play – an approach that combines the best of both kernel worlds: the compatibility and rich ecosystems of mature monolithic kernels, and the velocity and reliability of microkernels. Made possible by rising performance opportunities, new OS subsystems deliver high performance to end applications via a user-space multi-threaded service, hoisting an entire subsystem (e.g., a TCP stack or a filesystem) out of the kernel while leaving other OS functionality in the main monolithic OS (e.g., Linux). We call this kernel architecture a *semi-microkernel*.

The thesis of this dissertation is to fulfill the promises of the semi-microkernel approach for filesystems – achieving high *performance*, *scaling* resources up and down, and improving *fault tolerance*. Such an architecture provides both opportunities and challenges in three aspects, and we explore these aspects by building a vector of filesystem semi-microkernels – uFS, Nebula, and uFS-Shadow.

We address three main challenges. First, how can such a filesystem achieve high performance? Specifically, how can it deliver low latency and multi-core scalability? We investigate the performance extremes of such a filesystem by building uFS. Second, how can such a filesystem achieve resource elasticity by scaling the number of CPU cores according to dynamic application demands? uFS also strikes a balance between

absolute performance and CPU efficiency. Finally, how can such a filesystem achieve fault tolerance beyond fault isolation and the guarantees provided by existing disk-based full system crash recovery? We build Nebula and uFS-Shadow to improve application availability under this new process crash model concerning two fault models.

Our investigation is inspired by several semi-microkernels in the networking domain, such as IsoStack, Snap, TAS, and Shenango [99, 134, 155, 178]. Compared to kernel-bypass data-path library OSes [20, 92, 163, 171, 215], semi-microkernels, including ours, retain centralized control, policy management, and security enforcement for sharing hardware resources among multiple applications.

1.1 Functionality and High Performance

The first architectural opportunity and challenge we explore is high performance: a clean-slate design to achieve low latency and multi-core scalability. Our goals are:

G1. Make the most of ultra-fast devices and powerful CPUs.

G2. Achieve high performance across broader workloads [193, 196].

G3. Achieve strong single-threaded performance before scaling [140].

G4. Balance maximum scalability with system complexity [174].

We begin our exploration by building uFS, a fully functional and crash-consistent user-level filesystem. uFS supports commonly used filesystem calls.

The system consists of two main components: a uFS server and a uFS library. The uFS server is implemented as a multi-threaded process built atop the Storage Performance Development Kit (SPDK) [181]. Applications link with the uFS library to communicate with the server and

request file services via high-performance interprocess communication channels. The rest of the operating system remains as is, acting as an intermediary only in rare events (e.g., process startup) to provide authentication, but (importantly) does not participate in filesystem requests.

uFS relies on three base mechanisms as the foundation for low latency. Together, they overcome overheads throughout the Linux kernel stack, including system call, ring switch, interrupt handling, and deep stack overheads. The first and foremost mechanism is the *non-blocking filesystem thread* (*worker*), which adopts an event-driven programming model to process filesystem requests in a non-blocking manner, overlapping IO (submission to the device and polling for completion) with other computations (e.g., application interaction). Second, we design a fast App-filesystem IPC, incorporating a lock-free message ring buffer that leverages cache-to-cache transfer between cores and tailors the message format. Finally, we ensure that kernel interaction is kept off the common path, used only for initialization and security enforcement, and, importantly, not for filesystem requests.

The uFS library is also carefully designed for performance, including lease-based [72] caching of data and file descriptors to reduce client-server communication. The library also includes various other optimizations, mostly designed to reduce copying while maintaining security.

The multi-threaded uFS server contains a single primary and a variable number of worker threads, much like the original Google File System [67]. For simplicity, the primary handles the workloads that involve directory modifications (e.g., file creations and deletions), whereas workers handle the mainline data path (e.g., stats, reads, and writes to files).

uFS's design for multi-core scalability follows two key principles: avoid blocking-induced synchronization and separate designs for in-memory and on-disk data structures. uFS adopts a "shared-nothing" data-parallel architecture [184] across workers to avoid blocking; important data struc-

tures are designed to eliminate the need for synchronization across workers. The key unit of assignment across workers is the inode; at any given moment, a file inode is owned by a single worker, which handles all reads and writes to that file.

To provide scalable crash consistency, uFS employs a global journal that allows for a large number of concurrent sync transactions, requiring only a small critical section to synchronize the allocation of journal space (no blocking needed).

We evaluate the high-performance aspect of uFS with a series of microbenchmarks and macrobenchmarks. To isolate and understand different facets of filesystem performance, we create 32 single-operation microbenchmarks, covering factors like sharing, caching, and access patterns (e.g., random vs. sequential). We compare uFS to Linux ext4 [136], a traditional, time-tested, and optimized kernel-based filesystem.

Through microbenchmarks, we establish the baseline performance of uFS, showing that when utilizing only a single thread, it performs similarly to ext4 (sometimes better, sometimes slightly worse). We also show that uFS achieves excellent multi-core scalability, outperforming ext4 in half of the 32 microbenchmarks (by up to 3x) and performing similarly in most of the remaining workloads. In the cases where uFS performs better, when the workload is disk-bound, uFS performance scales well and quickly reaches its peak; and when the workload is memory-bound, uFS achieves higher throughput with multiple cores. Through macrobenchmarks of a web server and a file server [23, 196], we demonstrate the effectiveness of uFS's designs, such as caching and journaling.

1.2 Resource Elasticity

The second architectural opportunity and challenge we explore is resource elasticity: decoupled threads of application and the filesystem. Our goals

are:

G5. Improve CPU efficiency without sacrificing performance.

G6. Adapt dynamically to workload changes without overreacting.

In monolithic kernels such as Linux, an application launches its own threads. These threads serve as vessels to execute the storage stack code (e.g., filesystem, block scheduling, etc.) to interact with IO devices in privileged mode after trapping through system calls. In uFS, the uServer has the freedom to launch its own threads, independent of the number of application threads and their filesystem usage intensity.

As such, a semi-microkernel filesystem presents both the opportunity and the challenge of independently determining the amount of CPU resources dedicated to the server. This raises the questions: How many filesystem cores are needed for different application workloads to balance performance and CPU resource efficiency? And how should the number of cores be adjusted properly given that the workload changes over time?

We begin by augmenting uFS with mechanisms to monitor runtime CPU usage statistics, estimate resource demand across the spectrum of performance and CPU efficiency, and dynamically adjust the number of cores. We introduce a separate load management thread to make centralized observations and decisions by periodically gathering runtime statistics from each uFS server worker. The thread also controls the number of cores to use and directs workers to reassign inodes to balance the load.

One essential challenge for the load management mechanism is to identify performance metrics that can convey the load status (i.e., capturing both under-utilized and overloaded conditions) and are easy to obtain with negligible overhead. The non-blocking nature of the uFS server workers complicates this because each worker appears as 100% fully utilized to the host OS. We find that the combination of per-core effective CPU cycles (i.e., cycles spent on useful work rather than idly looping for

events) and the congestion (i.e., request queueing delay) is a good indicator of the load status.

We then design an algorithm to use the collected statistics and estimations to detect workload changes and determine the number of cores according to the configured policy for desired performance and CPU efficiency. We decompose the problem into two subproblems: (1) load balancing: how to balance the load with a fixed number of cores, and (2) core allocation: how many cores are needed given a relatively balanced load. Our algorithm thus predicts the load status after hypothetically performing load balance with the current number of cores or by increasing or decreasing only one core.

We evaluate the resource elasticity aspect of uFS with a series of microbenchmarks. We create 9 load-balancing microbenchmarks and 8 core allocation microbenchmarks, which carefully control the varying factors within each workload over time and cover the combination of various workloads.

We show that the adaptive load management works well, achieving nearly peak performance while minimizing the number of cores used by uFS. Finally, through a series of real application workloads, we demonstrate the overall performance benefits. Specifically, uFS improves the performance of LevelDB across eight different workloads (two writes and six YCSB [44]) from 1.3x to 4.6x.

1.3 Beyond Full System Crash Recovery: Process Crash Recovery

The third architectural opportunity we explore is fault tolerance, where the challenge is that restarting the filesystem is insufficient. Our goals are:

G7. Enable seamless recovery without losing performance benefits.

G8. Improve fault tolerance without altering the original filesystem behavior.

Semi-microkernels have significant benefits in fault isolation – a filesystem crash does not bring down the entire machine, so the host OS and unrelated applications remain unaffected. However, the applications using the filesystem cannot readily continue, and simply restarting the filesystem server is insufficient.

The reason why restarting the filesystem is insufficient is intuitive: at any given time, the filesystem buffers a large amount of updates in memory, creating a *state gap* between what has been perceived by the applications and what has been persisted to the disk.

Restarting the filesystem server while losing such a state gap is problematic for relevant applications. For instance, an application may find that data written just a minute ago has disappeared when it continues with a new filesystem server. As shown in our analysis of application reactions to a restarted filesystem server, the consequences can be severe and unpredictable; even durability-aware applications like databases (e.g., SQLite and LevelDB) can lose data. This occurs because the server may fail at any time under arbitrary combinations of operation sequences, leading to a wide range of state gaps being lost. However, this state gap problem has not been well understood in previous works [185].

We thus begin by formulating the crash model of monolithic (full system crash, s-crash) and microkernel filesystems (process crash, p-crash). Doing so allows us to rethink the opportunities and challenges that p-crashes bring – what are readily available (e.g., from the OS, from volatile memory, from devices, etc.)? Virtually all filesystems (including uFS) are designed with handling full system crashes in mind, employing techniques to ensure that on-disk states are consistent (e.g., journaling), but such full system crash recovery (s-crash recovery) falls short in addressing the state gap issue and does not guarantee no data loss.

We advocate for dedicated p-crash recovery mechanisms that handle any process crash except for power failures, which must be handled by s-crash recovery. With fast enough p-crash recovery, availability of the applications and the entire system that avoids more failover is increased.

Our key observation is that the failed filesystem server's memory state (including the state gap) is still present in volatile memory upon exit, before being reclaimed by the OS. This enables a new mechanism for recovering the state gap in a newly started process. We refer to this mechanism as *exit activation* – code that runs when a process crashes and is coordinated by the OS – to access the memory state before it is reclaimed. Exit activation tackles the state gap problem by allowing the failed server's memory state to also contribute to recovery.

We design and implement Nebula to realize exit activation, augmenting uFS with the machinery for fast and seamless recovery from unexpected faults, while maintaining high performance in the common case. Our key principles include: *clean restart* to discard problematic states and benefit from a clean address space; *limited trust of memory* to ensure that exit activation accesses only trusted memory; and *no force flush* to avoid extra overhead in the common path.

Central to recovery is an in-memory process log (p-log). During normal operation, Nebula records information about ongoing system calls into the p-log. After a p-crash, before the server process dies, the exit activation saves the p-log to a known location; then, when the restarted server begins execution, it can access the p-log (as well as traditional s-crash recovery logs) to restore filesystem state precisely. Applications continue running undisturbed, incurring only a momentary drop in performance.

The main issue with the p-log is that when a system call's effect is made durable, it should be removed from the state gap. However, the effects of system calls are often made durable in a non-sequential order. Our finding is that replaying the p-log requires ignoring some system

calls and, at times, modifying one call to another existing system call API (e.g., changing a `write` to an `lseek`) so that the original codebase can be used.

Nebula includes a range of techniques to achieve its goals. Fast, cache-aware per-core logging ensures that recording information to the p-log is fast and correct. An algorithm, called AIM for act/ignore/modify, transforms the p-log into the actions the server must take to recover the state gap. P-log and AIM capture the state gap such that no extra flush is needed in the common path. Checksums, and replication of the p-log, protect critical data structures; a careful protocol ensures they are maintained correctly. Finally, kernel-coordinated speculative restart that orchestrates control transitions efficiently and avoids the high overhead of device connection establishment, reducing restart time by seconds.

We create a new benchmark suite to evaluate the fault tolerance aspect of Nebula. The benchmark includes a large number of system call sequences, each exhibiting diverse combinations of system calls and varying over time. We include 24 synthetic system call sequences and 10 sequences from real-world applications (ranging in length from 77 to over 5,000 calls). We exhaustively injected p-crashes after each system call in the sequences, covering a wide range of diverse state gaps to recover from. This benchmark also allows us to understand the application's reaction to a restarted filesystem server, highlighting the importance of seamless recovery and addressing the state gap.

We perform a thorough empirical evaluation of Nebula, focusing on fault handling and performance. We show that Nebula achieves seamless recovery for applications, including utilities (sort, cp, zip) and durability-aware libraries (SQLite, LevelDB), with negligible impact on performance compared to uFS. Specifically, Nebula achieves transparent recovery over a large number (30,000+) of controlled and random fault-injection experiments, that emulate both fail-stop p-crashes and data corruption. Fur-

ther, Nebula incurs negligible performance (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, Nebula recovers in an acceptable amount of time ($\leqslant$500ms).

1.4 When Slow is Good

In the last part of this thesis, we explore fault tolerance further. Our goal is:

G9. Recover from both transient and deterministic errors.

Nebula cannot recover from deterministic bugs because its p-log replay leverages the original source code of uFS, which would fail again in the case of a deterministic bug. This motivates us to take a step back and ask why things go wrong (i.e., the bugs) in the first place, and how we can recover from almost all bugs so that the reliability of the system is fundamentally improved as if the bugs never existed.

Our key observation is that many bugs in filesystems stem from performance optimizations. While developing uFS and realizing performance benefits, numerous performance components such as caches are introduced. Worse, these components interact with each other in a multi-threaded setting, significantly increasing complexity, which is the main culprit for more bugs.

The same observation holds true for kernel filesystems. We studied the bug reports of the Linux kernel filesystem stack and found that both transient and deterministic bugs are prevalent. Similar to uFS, the implementation of core filesystem functionality (e.g., ext4, btrfs, and zfs) interacts and evolves with performance-oriented components throughout the entire IO stack, such as the inode cache, dentry cache, and block layer. Numerous performance enhancements have been introduced to these com-

ponents in recent years, including block-mq, page folios, iomap, io_uring, and polling-mode IO [47, 48, 60, 181].

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

A shadow filesystem, when paired with a base filesystem, represents a form of N-version programming [13]. However, its goals are different from the base. The shadow prioritizes correctness over performance to realize the simplest possible sequential implementation. Concurrency, caches, asynchronous execution, and other performance optimizations are omitted, as they are not essential.

We design and implement uFS-Shadow, a shadow filesystem for uFS. When incorporated into Nebula, uFS-Shadow enables Nebula to recover from both transient and deterministic bugs, thereby improving the reliability of the entire system. Together, the pair of Nebula and uFS-Shadow achieves high performance through uFS in the common path and robustness through uFS-Shadow in the alternative path.

We show that uFS-Shadow functions properly, as it can recover from over 30,000 transient p-crash fault injections. We demonstrate that uFS-Shadow can recover from deterministic bugs under 25 emulated deterministic errors via fault injection. The recovery time of uFS-Shadow is 1.6x slower than using uFS for replay, but the codebase of uFS-Shadow is only about 7% of uFS, representing a significant simplification while remaining fully functional in replaying the p-log.

Overall, our hypothesis for building filesystems in a semi-microkernel style is that high performance is possible (as opposed to both microkernels and monolithic kernels) and that better fault tolerance guarantees are feasible and beneficial (as opposed to monolithic kernels), which we have demonstrated.

However, a filesystem is a complex piece of software [34, 130, 195], requiring considerable effort to build, test, and maintain [9]. Compared to networking semi-microkernels, filesystems are more deeply coupled with the host monolithic kernel, particularly in process management (e.g., for permissions and `fork`), memory management (e.g., `mmap`), and the rich set of APIs they support. As a result, compatibility issues and general coordination with the host OS are more challenging. Resource elasticity is one example where the filesystem service inherently needs to take responsibility for CPU scheduling. Additional effort is needed to support a wide range of applications.

This approach would be beneficial in situations where storage access needs to be specialized (e.g., for specific hardware, energy consumption, heterogeneity, etc.) or where the consequences of a filesystem crash (i.e., a full system crash) are severe. Importantly, the cost of developing and maintaining the filesystem is better justified by a sufficient number of users. For instance, a filesystem not crashing the entire machine can be life-saving in vehicles [32]. Other scenarios include cloud VMs used by a large number of small or medium-scale applications that utilize local filesystems or a local filesystem serving as a backend for a distributed filesystem [9].

1.5 Contributions

We list the main contributions of this dissertation.

- **Semi-microkernel Approach in Filesystems.** We thoroughly ex-

plore the semi-microkernel approach in filesystems by building uFS, Nebula, and uFS-Shadow. We demonstrate that this approach can achieve high performance, benefit from resource elasticity, and improve fault tolerance. These semi-microkernels are implemented in over 41K lines of C++ code from scratch.

- **uFS.** We build uFS, a fully functional and crash-consistent user-space filesystem that supports commonly used filesystem calls. We show that uFS offers competitive single-threaded performance and achieves excellent multi-core scalability. uFS also dynamically adjusts the number of CPU cores according to workloads, balancing performance and CPU efficiency.

- **Exit Activation.** We propose exit activation, a new approach for process crash recovery. Exit activation is code that runs when a process crashes, accessing memory before it is reclaimed, and allowing the failed server's memory state to contribute to recovery.

- **State Gap Problem of Filesystem Process Crash.** We formulate and systematically analyze the crash model of microkernel filesystems, where buffered updates are lost due to a process crash. We identify the key problem – the state gap – which is the difference between the application's perceived state and the on-disk state. This state gap represents the effects of system call sequences executed in the past, which may not be in the sequential order perceived by the application. We also empirically analyze the application's reactions to such a crash model, covering five applications (cp, sort, unzip, SQLite, and LevelDB), ten operation sequences (ranging from 77 to over 5,000 system calls), and 30,000+ different state gaps. Our analysis shows that the consequences of the state gap are severe and unpredictable.

- **Nebula.** We build Nebula, augmenting uFS with the machinery for fast and seamless recovery from unexpected faults, with negligible common-path overhead. Nebula includes an in-memory process log (p-log), an exit activation data structure that is well-specified, well-protected, and the only memory region that needs to be accessed by exit activation code. The p-log accurately captures the source of the state gap (i.e., system calls). With the p-log and our novel algorithm, AIM (Act/Ignore/Modify), the state gap problem can be resolved by replaying the logged system calls (in the p-log) using the original uFS code, yet in a form varied from the original execution.

- **Robust Alternative Execution (RAE).** We propose Robust Alternative Execution (RAE), a practical approach to improve the reliability of existing complex and performance-oriented filesystems via shadow filesystems.

- **uFS-Shadow.** We build uFS-Shadow, a shadow filesystem for uFS, integrated into Nebula to realize RAE. Compared to Nebula, uFS-Shadow can recover from both transient and deterministic bugs. Through uFS-Shadow, we demonstrate how filesystem core functionality can be realized in a much simpler manner (7% of uFS codebase) that is less prone to bugs by eliminating performance optimizations as much as possible.

- **Benchmarks.** We create a series of benchmarks to evaluate multiple aspects of the semi-microkernels, including:

    1. 32 single-op microbenchmarks to understand performance, stressing various factors affecting filesystem performance.
    2. 9 load-balancing microbenchmarks and 8 core allocation microbenchmarks to evaluate resource elasticity, covering chang-

ing factors within each workload over time and the combination of various workloads.

3. A new methodology and benchmark to evaluate recoverability from the state gap, including 24 synthetic system call sequences and 10 sequences from real-world applications, covering a multitude of diverse state gaps (over 30,000).
4. A benchmark to emulate deterministic bugs.

1.6 Overview

We briefly describe the contents of the chapters of this dissertation.

- **Background.** In Chapter 2, we review the historically thriving microkernel operating systems, discussing their advantages, primary concerns, and modern relevance. We then discuss recent trends in hardware and applications and explain why, in light of these trends, mainstream monolithic kernels are problematic. We also cover the kernel-bypass approach for direct access to devices and the Storage Performance Development Kit upon which our work is based.

- **Semi-microkernel Approach.** In Chapter 3, we introduce and explain the semi-microkernel approach, comparing it with monolithic kernels and microkernels. We discuss the benefits of this approach.

- **Functionality and High Performance.** In Chapter 4, we present the design and implementation of uFS, focusing on functionality and high performance. We describe the single-threaded basic architecture, the designs for multi-core scalability, and the scalable crash consistency design. Finally, we evaluate uFS with a series of microbenchmarks and macrobenchmarks.

- **Resource Elasticity.** In Chapter 5, we present the load management features of uFS, including the mechanisms, policies, and algorithms to dynamically adjust the number of cores. We evaluate the resource elasticity of uFS with a series of microbenchmarks and real application workloads.

- **Beyond Full System Crash Recovery: Process Crash Recovery.** In Chapter 6, we improve fault tolerance by performing process crash recovery via exit activation. We design and implement Nebula, and evaluate its fault handling and performance.

- **When Slow is Good.** In Chapter 7, we introduce Robust Alternative Execution (RAE) and build uFS-Shadow to realize RAE in Nebula. We demonstrate that Nebula recovers from transient and deterministic errors. We evaluate the goodness (robustness and simplicity) and the slowness (recovery time) of uFS-Shadow.

- **Related Work.** In Chapter 8, we discuss related work, including modern explorations of OS architectures, filesystems, and multi-core systems. We present the literature on fault tolerance, covering the sources of faults, filesystem fault tolerance and reliability, and general fault tolerance techniques.

- **Conclusions and Future Work.** In Chapter 9, we summarize the dissertation and present a few high-level lessons learned. We also outline future work that can build upon this dissertation.

2 Background

In this chapter, we provide the background related to our work, presenting our motivations to revisit OS architectures and the foundations built from decades of research and practice. We start with the rise and fall of microkernels (§2.1). Then, we discuss the challenges faced by today's mainstream monolithic kernels due to the evolving landscape of hardware and applications (§2.2). Finally, we discuss recent efforts in rearchitecting OS architectures for I/O, including kernel-bypass approaches and the Storage Performance Development Kit (SPDK) (§2.3).

2.1 Rise and Fall of Microkernels

The mainstream, widely used operating systems today, such as Linux and Windows, are monolithic kernels. Monolithic kernels run the entire operating system in kernel space. In contrast, microkernels only include the most essential components in privileged mode, and the rest of the system services, such as device drivers, file systems, and network stacks, are run in user space.

Microkernels have long played a part in the discussion of how to best structure operating system service. One of the earliest microkernels, developed in the late 1960's, was Brinch Hansen's Nucleus [74], argued for a microkernel approach based on modularity.

A next generation of microkernels used similar arguments. For example, in the mid-1980s Mach [166, 213] stated: "[Mach] provides a small set of primitive functions designed to allow more complex services and resources to be represented as references to objects."

Microkernels were a hot topic in the 1980s [7, 166, 167, 213]. At that time, multiple operating systems were developed in response to multiprocessors and network-connected systems. Meanwhile, Unix had been integrated with new facilities such as System V streams, BSD sockets, various forms of semaphores, special files, making the software difficult to maintain.

However, performance was a major concern for the first generations of microkernels, perhaps reducing interest in microkernels for general-purpose usage. For example, performance studies revealed high caching costs [36] and overhead due to address space changes [59]. In later years, microkernels have seen a resurgence, both in research and in practice, across various domains [61, 76, 77, 191, 219]. For example, some variants of the L3 microkernel provide high performance [115]. Furthermore, the more compact nature of these systems has led to pioneering efforts in OS verification [101].

Nevertheless, microkernels did not become the mainstream architecture for general-purpose operating systems. By 2000, large-scale Mach kernel efforts had largely ended. Most microkernels primarily target specific domains, such as embedded systems, like L4 embedded in Qualcomm cellular modem chips [61], QNX in embedded systems [78], and Zircon (i.e., Fuchsia) in smartphones [70]. Linux, a monolithic kernel started in the 1990s, has become much more widely used and deployed on an extremely large scale in today's data centers and cloud.

Nowadays, monolithic kernels face similar challenges as Unix did in the 1980s to extend the kernel with much flexibility and velocity for a new purpose – performance. Hardware and applications have changed

significantly in the last decade, and monolithic kernels have become the performance bottleneck [19].

2.2 Monolithic Kernels Meet Modern Applications and Hardwares

Two trends motivate our work. The first is in hardware, including the emergence of low-latency and high-throughput block devices, and the need to leverage modern CPUs with multiple cores, especially in the wake of Moore's Law coming to an end [141, 194]. The second is in applications, which are rapidly increasing in volume and diversity, exhibiting dynamic workloads. We now describe these trends in more detail.

2.2.1 Hardware Trends

Hardware used by storage stacks has evolved, notably with the emergence of low-latency and high-bandwidth devices and multi-core CPUs.

2.2.1.1 Low-latency and High-bandwidth Block Devices

The evolution of storage devices continues apace. The most relevant to this thesis are the ultra-fast NVMe SSDs [104], which use high-speed PCIe buses for data transfer and the NVMe (Non-Volatile Memory Express) protocol to communicate with the host. NVMe is an open, logical-device interface specification designed to leverage the low latency and internal parallelism of solid-state storage drives [62]. Ultra-fast NVMe SSDs often come with some form of non-volatile memory, such as NAND flash or Optane memory.

Compared with traditional SATA SSDs, NVMe SSDs have much lower latency and higher bandwidth. For example, the Intel Optane SSD [90]

offers a read latency of around 6 microseconds, with maximal throughput (random read) up to 575K IOPS and bandwidth (sequential read) up to 2.6GB/s. Despite Intel's business decision to discontinue the Optane product, efforts to develop low-latency and high-throughput storage devices continue, such as Z-SSD [86]. The market for low-latency and high-bandwidth block devices continues to grow to support modern applications with high-performance requirements. Moreover, the development of the PCIe protocol also continues, targeting up to 16GT/s per lane, leading to 64GT/s for modern M.2 SSDs that use 4 lanes [85].

Persistent memory products, such as Intel Optane DC Persistent Memory, have encouraged the re-architecture of storage system design [211]. One main difference between the ultra-fast NVMe SSDs and persistent memory is that SSDs use the PCIe bus and appear as block devices in the system, whereas persistent memory is directly connected to the memory bus and is byte-addressable. Besides, persistent memory has one order of magnitude lower latency than NVMe SSDs (i.e., around hundred nanoseconds), but is not superior in bandwidth. Even though the commercial product of Intel Optane Persistent Memory has been put on hold, many persistent memory file systems [56, 207, 208] provide insights for designing high-performance file systems.

2.2.1.2 Multi-core CPUs

The ending of Moore's Law [141, 194] has emphasized the importance of leveraging multi-core CPUs and developing efficient concurrent software [53]. Compared to the limited number of cores in the early 2000s, modern CPUs can have up to 256 cores [159].

However, extracting performance benefits from modern CPUs is challenging, even for simple software, requiring a deep understanding of underlying hardware details. The CPU employs a range of powerful features to provide better performance, such as multiple levels of cache, out-of-

order execution, and speculative execution. The NUMA (Non-Uniform Memory Access) architecture, TLB shootdown [10], and CPU core frequency scaling [71] also affect software performance.

Software that makes poor use of these features can run significantly slower. For example, the Xeon Gold 6138 processor used in our evaluation has megabytes of L1 cache, 40MB L2 cache, and 50MB L3 cache, with the last level cache shared among all cores. The latency of accessing each level of cache varies significantly, from subnanosecond for L1 references to tens of nanoseconds for main memory references. On the upside, system software with careful design can gain large performance benefits and provide better primitives for applications.

However, powerful CPUs with a large number of cores and sophisticated features are less reliable when pushed to the limits of hardware and physical laws. Silent core corruption (i.e., mercurial cores [79]), where one core of a CPU produces incorrect results for certain computations, has caused silent data corruption and system crashes in data centers [17, 55, 204].

Furthermore, performance techniques such as out-of-order execution lead to vulnerabilities that, when exploited by attacks such as Meltdown and Spectre, can leak data, breaking assumptions of hardware isolation [103, 120]. To mitigate these vulnerabilities, monolithic kernels (e.g., Linux) enforce address space isolation, leading to large performance overheads [120].

2.2.2 Application Trends

Numerous new types of applications have been developed, which exhibit different access patterns and performance requirements. One interesting trend is the wide adoption of storage engines that can be embedded into applications, such as SQLite [182], LevelDB [68], and WiredTiger [150]. These storage engines interact with filesystems through the POSIX APIs, managing data structures like B-trees, Log-Structured Merge Trees, and

many others. Higher-level applications, such as web applications, stream processing, and databases, commonly use these storage engines and leverage the key-value interface to store and access data [28, 149].

Compared to the early days of microkernels, the ever-increasing number of modern applications, driven by the rapid development of the Internet, cloud computing, smartphones, and IoT devices, relies on a standard interface to interact with the OS – the Portable Operating System Interface (POSIX). Therefore, there is a clear need to bring hardware advances to existing applications with POSIX compatibility [197].

Moreover, applications are running in cloud environments, and many applications can run on one physical machine. These diverse applications exhibit dynamic and possibly bursty workloads [28, 177, 205], posing challenges to underlying systems.

2.2.3 Monolithic Kernels are Problematic

In the face of the hardware and application trends described above, monolithic kernels such as Linux fall short for three reasons: performance, velocity, and fault tolerance.

2.2.3.1 Performance

Monolithic kernels (e.g., Linux) have become performance bottlenecks due to overhead in the I/O stack and limitations in multi-core scalability.

The deep kernel I/O stack introduces significant overhead. For example, reading a 4KB block from ultra-fast NVMe SSDs through a widely-used kernel filesystem, Linux ext4, introduces several microseconds of overhead, doubling the latency (i.e., 14us vs. 6us) for applications. While such overhead was acceptable for previous storage devices with millisecond latencies, it becomes a bottleneck for ultra-fast NVMe SSDs with microsecond latencies.

The reasons for this overhead are manifold. First, a significant contributor to overhead is the interrupt-handling mechanism that supports blocked I/O. In contrast, polling offers better latency for high-performance devices [69, 190]. Second, the overhead for system calls and context switches is non-negligible. Indirect costs like cache pollution can also have a large impact [180]. Third, the Linux kernel is designed for generality, leading to multiple layers. For example, the block layer, the file system layer, and the VFS layer are all involved in the I/O path.

The scalability bottleneck of Linux impedes applications from achieving scalable performance when using multiple cores. Various scalability issues have been found by previous studies, such as the reference count in the dentry cache [25] and CPU schedulers [128].

Moreover, the Linux kernel filesystems are found to have several scalability limitations [144]. They introduce multiple locks, leading to high contention and scalability issues. For example, crash consistency mechanisms like journaling, copy-on-write, and log-structured writing are not scalable.

One issue of CPU scalability is that the Linux kernel, when designed, had little concern for multi-core scalability, leading to several coarse-grained locks. Therefore, scalability improvement is mostly achieved by breaking the bottlenecked locks into fine-grained ones, which may be time-consuming and introduce bugs [130].

2.2.3.2 Velocity and Customization

Another issue for the monolithic Linux kernel is that the pace of developing and upstreaming new features is constrained [134, 142, 143]. Linux needs to adapt to support emerging application workloads, patch security vulnerabilities, and incorporate new hardware. Extending the Linux kernel by modifying the kernel source code is a time-consuming and error-prone process.

One main reason that hinders velocity is the complexity of the Linux kernel code base. As a monolithic kernel developed over three decades, it now contains tens of millions of lines of code. Therefore, rigorous reviewing, testing, and validation are required to ensure the correctness of new features. In addition, kernel development itself is a challenging task, requiring a deep understanding of the kernel code base and experience.

For example, in the kernel source code of version 5.4 used in our evaluation, the total lines of code are around 16 million, with 12 million in the `drivers` directory, and 1.5 million in the `arch` directory. The rest of the codebase is also substantial, mainly in the `fs` directory (0.9 million), `net` directory (0.7 million), and `kernel` directory (0.2 million).

Consider the scale of computing today. Data centers and cloud providers are running over millions of machines, with an extremely large number of application workloads running at any time. A fast pace of deployment, with updates rolled out on a weekly cycle, is desired [134]. However, it takes months or years to upstream a new feature to the Linux kernel, and upgrading the machines with the new feature causes downtime, which is undesirable.

2.2.3.3 Fault Tolerance

Finally, the monolithic kernel is problematic for fault tolerance. One major drawback of running the entire system in kernel space is poor fault isolation – a fault in one subsystem can crash the entire machine, leading to service downtime and data loss. As the complexity of the kernel increases, the challenges in managing the reliability of the system also increase. This problem is exacerbated when discussing filesystems because they contribute significantly to kernel's complexity, are difficult to ensure correctness (thus more prone to bugs), and are a notable source (17%) of kernel vulnerabilities (e.g., buffer overflow attack) [34].

Such concerns regarding reliability have drawn increasing attention.

One such effort is the gradual adoption of Rust [137] in kernel development, such as drivers and filesystems [45], attempting to increase memory safety. However, for applications that rely on the OS, the kernel is too large to be free of bugs, too prone to failure on a large scale, and requires too much work (and time) to recover.

2.3 Rearchitecting OS Architectures

Given these challenges, researchers and industry practitioners have proposed various solutions, rearchitecting the OS architecture to better support modern applications and hardwares. We present the approaches of kernel-bypass and microkernel. We discuss storage performance development kit (SPDK), which is the main building block of our work.

2.3.1 Kernel-bypass and Microkernel

One approach to addressing the performance bottleneck of monolithic kernels in response to hardware trends is *kernel-bypass*, which involves building system services as libraries and linking them into applications. This allows the device's hardware interface to be directly mapped into the application address space, leading to superior performance [20, 92, 163, 171, 215].

The performance of kernel-bypass has led to the rise of user-level development kits. These libraries allow applications to directly access devices and bypass the kernel entirely. For example, the Data Plane Development Kit (DPDK) [58] consists of libraries to accelerate packet processing on a range of CPU architectures, enabling a variety of network-oriented applications to be readily implemented.

However, kernel-bypass requires extra consideration to manage the sharing of hardware resources among multiple applications. The centralized control of the kernel is lost, including the benefits of better re-

source management and isolation, scheduling decisions, and policy enforcement.

To obtain the benefits of high performance from direct device access while still retaining the benefits of centralized control, the microkernel approach seems to be a natural choice. However, realizing a complete microkernel is less practical, requiring significant effort to solve compatibility issues because modern applications assume POSIX as a default. The generalized microkernel, under real application workloads, also suffers from performance overhead due to the high frequency of IPCs [32]. These challenges are described in the seven-year effort to build a microkernel, HongMeng [32].

Therefore, in this work, we explore an approach that does not rebuild the entire OS as microkernels do, but instead hoists a subsystem into user space, leveraging the benefits of both direct access and centralized control [99, 134]. We call such an approach a *semi-microkernel*.

The emergence of storage development kits has now made it possible to explore the utility of a filesystem semi-microkernel; we discuss these kits next.

2.3.2 Storage Performance Development Kit

The Storage Performance Development Kit (SPDK) [181] is open-source software that enables the creation of high performance user-mode storage applications running on NVMe devices. It enables the construction of high-performance, user-space storage system services, which is otherwise difficult to achieve as the driver and device must run in privileged mode.

The SPDK is realized as a user-mode device driver, and, as such, the kernel is not involved in any interactions with the device (indeed, once active, the kernel no longer can access the device at all). SPDK provides an abstraction of a *queue pair* to submit requests and receive responses; within an application (in our case, the uFS server), each thread is as-

signed its own queue pair, and can submit requests to the device without coordination (i.e., locking).

In this work, we focus on the lowest level SPDK APIs, which enable direct submission of requests to the device. Specifically, reads and writes are submitted as NVMe commands to the device's queue pair via the calls of spdk_nvme_ns_cmd_read and spdk_nvme_ns_cmd_write, respectively. Higher level interfaces (such as the block stack) are provided by SPDK but not utilized herein.

SPDK provides a polling-based interface via a non-blocking call to spdk_nvme_qpair_process_completions. This approach works well for high-performance devices [146] and is more natural at user-level (where event handling can be clumsy or inefficient) but requires care, as excessive polling will waste CPU resources.

Memory management is also important when using the SPDK, as memory must be pinned to enable DMA to and from the NVMe device. To facilitate this, two calls (spdk_dma_malloc and spdk_dma_free) are provided. The current SPDK implementation uses Linux huge pages, and thus the calls to allocate memory must be used judiciously.

In general, the growing popularity of these types of toolkits gives rise to the question: can SPDK (or similar libraries) enable the construction of not only a specific high-performance application (which was perhaps the intended use-case), but a high-performance user-space filesystem? We believe the answer is yes, and develop an architecture to investigate this question more deeply.

2.4 Summary

In this chapter, we present the background, briefly reviewing the history of relevant OS architectures' exploration and evolution. We discuss the modern trends in hardware and applications that pose challenges to to-

day's monolithic OS kernels. Finally, we cover the basic building blocks of our work: a user-space NVMe device driver that enables direct access, allowing the filesystem subsystem of the OS to be hoisted into user space – an approach we take in this dissertation.

3 Semi-microkernel Approach

In this chapter, we present the core theme of this thesis: the semi-microkernel approach. We discuss the semi-microkernel architecture, comparing it with the traditional monolithic and microkernel architectures (§3.1). We then discuss the benefits (§3.2) and challenges (§3.3).

3.1 Semi-microkernels for IO Subsystems

In this work, we explore the semi-microkernel approach to building IO subsystems, specifically in the context of filesystems. A semi-microkernel works in tandem with the main (monolithic) operating system, but realizes a partial or even entire OS subsystem (e.g., the networking stack or filesystems) inside a user-level process. Applications wishing to use its services communicate with the semi-microkernel process via high-performance IPC channels. Most relevant to our work, recent efforts including IsoStack, Snap, TAS, and Shenango [99, 134, 155, 178] demonstrate the benefits in the networking domain.

Figure 3.1 illustrates the semi-microkernel architecture and how it compares to the traditional monolithic and microkernel architectures. The monolithic kernel, such as Linux, implements all OS services in the kernel, leading to a large and complex codebase and lacking isolation between the subsystems at runtime. In contrast, the microkernel architec-

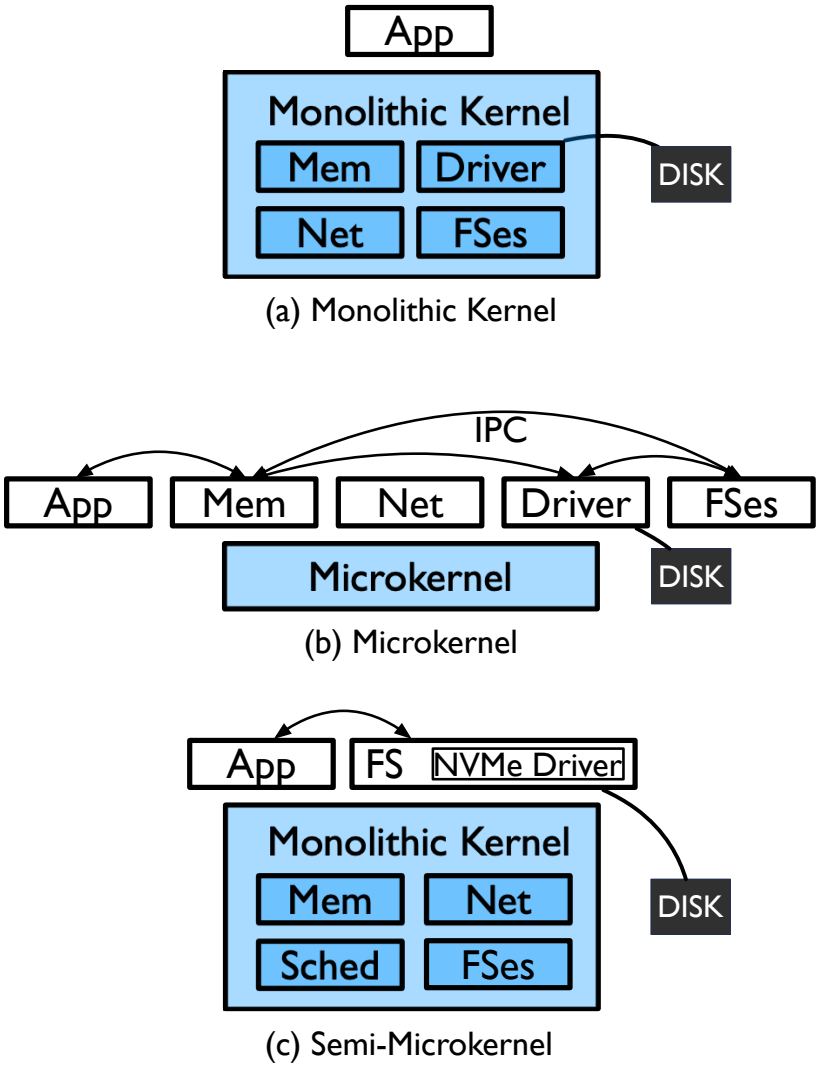


Figure 3.1: **Comparison of OS Architectures**. Monolithic kernels (e.g., Linux) run all OS services in kernel space. Microkernels move as many services as possible to user space, with strong isolation enforced by process boundaries. Semi-microkernels realize an OS subsystem in user space (e.g., a filesystem), working in tandem with the monolithic kernel.

ture favors modularity and extensibility, with the principle of minimality [61] guiding the division of kernel space and user space OS services. As many services as possible are moved to user space, with strong isolation enforced by process boundaries.

The semi-microkernels, sharing the same inspiration for modularity as microkernels, rely on the monolithic kernel for other services, such as scheduling and memory management. For the IO path, where hardware has made tremendous advances and where much performance potential can be fulfilled via direct access, the subsystems are built as user-space processes. The semi-microkernels thus co-exist with the host monolithic kernel.

In this thesis, we demonstrate the benefits and address challenges of such an architecture by building filesystem semi-microkernels, uFS, Nebula, and uFS-Shadow. We achieve high performance, resource elasticity, and realize the potential of fault tolerance.

3.2 Benefits

The semi-microkernel architecture has many benefits, including performance, velocity and customization, and fault tolerance, as we discuss below. We mainly focus on filesystems, though the benefits are general to other IO subsystems.

3.2.1 Performance

The semi-microkernel approach has the opportunity to achieve high performance. First, compared to the restrictions of in-kernel development, the user-space filesystem can get the full benefit of direct access and advanced hardware performance with a clean-slate and well-designed IO stack.

Second, the semi-microkernel can overcome the IPC overhead by designing the IPCs to be tailored for the filesystem's needs. The IPC frequency is also largely reduced in the semi-microkernel approach, as shown in Figure 3.1. The IPC is a form of indirect system call and the overhead concern is much more reduced in this age, because today's computers have a large number of CPU cores, allowing for the fast inter-core communication based on cache-to-cache transfer [172]. Further, the CPU cache locality of the filesystem process can be preserved, whereas today's system calls lose performance due to the cache pollution and ring switches.

3.2.2 Velocity and Customization

One critical advantage of the semi-microkernel approach is that it accelerates velocity and customization, i.e., the ability to quickly develop, modify, and deploy system software. Instead of the slow pace common in kernel development, the hoisted subsystem can be developed in a manner more similar to application code. Application-level tools and testing frameworks can also be utilized, further improving developer productivity.

Such an advantage is naturally inherited from microkernels. Many more policies can be implemented and experimented with in user space to specialize for hardware, filesystems, and application workloads, facilitating rapid innovation.

3.2.3 Fault Tolerance

When the IO subsystem is built as a user-space process, the fault isolation is greatly improved, a filesystem failure does not crash the kernel. As a result, the host OS kernel, and the irrelevant applications naturally continue to run.

3.2.4 Additional Benefits

By retaining the monolithic kernel for other services, the semi-microkernel avoids the need to rebuild the entire OS and the resulting compatibility issues, which take many years of effort to resolve [32]. The underlying host OS can always offer fallback support instead of a full substitution that is less reliable. Applications can also continue to leverage the rich Linux ecosystem.

Furthermore, the filesystem process provides centralized control for multiplexing to multiple applications. Hardware access is restricted to one system service, instead of delegating it to the application address space. This centralization simplifies resource management in terms of concurrency control, allocation, and security.

3.3 Challenges

Realizing the architectural benefits of the semi-microkernel approach is not without challenges. We describe three challenges that are unique to semi-microkernels. Other challenges for building high-performance filesystems are also important but are not the focus here.

3.3.1 High Performance: Low Latency and Multi-core Scalability

The filesystem process we built involves more functionality than a traditional kernel filesystem, such as Linux ext4, which primarily concerns the on-disk data structures and how to read/write them. We refer to these as the *core functionality*. Providing the core functionality correctly and efficiently is challenging, but it is not sufficient in a semi-microkernel filesystem process.

In the kernel filesystem stack, other layers are involved in processing filesystem requests, such as the interface layer that handles application interaction (e.g., system call mechanism, and VFS) and dispatches to a specific filesystem implementation. The page cache is integrated into the memory management subsystems [48, 119], and the block IO layer isolates the complexity of interacting with block devices [47] and relies on the interrupt handling mechanisms [118].

The unique challenge a semi-microkernel filesystem faces is that, without the historical burden and the convenience (and perhaps mature optimization) provided by these many other layers, we need to achieve competitive performance. Specifically, can the pieces of software achieve the low latency enabled by ultra-fast devices and the multi-core scalability enabled by modern CPUs? We answer these questions through uFS.

3.3.2 Resource Elasticity: Decoupled Threads of Applications and the Filesystem

Another main challenge unique to the architecture is resource elasticity, which is also an opportunity. The filesystem threads and application threads are decoupled (i.e., they run in different processes), allowing the filesystem to scale CPU resources independently of the number of application threads [134].

This raises questions: How can we achieve both high performance and CPU efficiency? In response to modern application trends where workloads are dynamic, how can the filesystem scale up and down to meet varying demands? We answer these questions through uFS.

3.3.3 Fault Tolerance: Restarting the Filesystem is Insufficient

The main challenge for fault tolerance is that applications using the filesystem do not naturally benefit from fault isolation, as simply restarting the filesystem is insufficient.

Therefore, what guarantees are required for applications to continue running after a filesystem crash? How can we improve the system's fault tolerance accordingly? Can fault tolerance mechanisms be incorporated without negating performance benefits? Finally, is it possible to recover from virtually any crash, as if the filesystem never fails? We address these questions through uFS and uFS-Shadow.

3.4 Summary

In this chapter, we introduce the semi-microkernel approach, which builds only the IO subsystems as user-space processes. We discuss the benefits of this approach. In the next four chapters of this thesis, we address the essential challenges to realize the benefits through the development of uFS, Nebula, and uFS-Shadow.

4

Functionality and High Performance

We start our exploration from building uFS. uFS is a multi-threaded POSIX-compatible user-space filesystem that directly access the storage devices. To explore the performance promise of the semi-microkernel approach for filesystems, uFS needs to address the architectural challenges of building such a filesystem that is free from the performance issues suffered by previous monolithic kernels and microkernels. To do so, uFS first establishes strong single-thread performance as the foundation. Additionally, uFS exploits the opportunity to design for multi-core scalability from scratch, scaling with the best single-threaded performance in mind.

The first performance axis is the latency of each supported filesystem system call. The performance of a filesystem is multifaceted, considering the rich set of interfaces (e.g., over tens of system calls), hierarchical abstractions to manage data (e.g., directories, names, and inodes), and diverse access patterns (e.g., cached or disk-bounded). As a fully functional filesystem, uFS supports commonly-used filesystem calls, aiming to be POSIX-compatible, such that modern applications can run atop it without modification.

We first describe the issues necessary for correctness and base-line efficiency given a single-threaded uFS server: managing interactions between the application, uFS server, and I/O device; scheduling requests; and library-side caching to avoid unnecessary interactions. Specifically, we describe how uFS overcomes the inter-process communication (IPC)

overhead between the user application and the filesystem server, how uFS reduces the I/O stack overhead by polling the device and processing requests in a non-blocking manner, and how uFS interacts with the host kernel off the common path for security, ensuring high performance without compromising correctness.

The second performance axis is the multi-core scalability of a multi-threaded uFS server. The question is, how can we achieve scalability with competent per-thread performance? The issues is known as the COST of scalability [140], meaning that achieving better scalability with multiple threads is easier with a relatively poor single-threaded base performance. Further, how can we strike the balance between maximal scalability and system complexity? Overall, our design is guided by the rule to allow each core to operate independently as much as possible, inspired by pioneering works like disjoint-access parallelism [91] and conflict-free operations [11]. Specifically, we follow two principles for multi-core scalability.

Our first principle for scalability is to avoid blocking-induced synchronization, achieved by partitioning on-disk and in-memory structures across threads, which is in synergy with the uFS server's non-blocking thread design. To constrain complexity, we partition the structures so that each inode is owned by a specific thread (i.e., worker) and directory operations are handled by a primary worker.

Another principle for scalability is to separate designs for in-memory and on-disk structures [23]. Specifically, the crash consistency design is critical for the on-disk write path. The scalability of in-memory data structures is achieved by avoiding sharing among workers, whereas all the workers share a single on-disk global journal to allow for a large number of concurrent sync transactions. The synchronization (only needed for allocating a journal transaction's disk space) is made non-blocking with a small critical section, such that the synchronization overhead when starting multiple journal transactions (e.g., tens of nanoseconds) is or-

ders of magnitude less than the disk latency (e.g., several microseconds). Therefore, we show that crash consistency can be added to uFS without harming performance, as each thread independently writes to a globally-ordered logical journal. The single global journal also reduces the complexity of the commit and crash recovery protocol, which is well-recognized as difficult to design, implement, and test for correctness [148, 158].

The chapter is organized as follows. We begin with the design of the single-threaded uFS server, describing the basic architecture for functionality, data structures, and techniques to achieve low latency for system calls (§4.1). Next, we present the design of the multi-threaded uFS server (§4.2) and crash consistency (§4.3). We then systematically evaluate uFS, comparing the single-thread and multi-thread performance with Linux ext4 (§4.4) under microbenchmarks and macrobenchmarks, covering a multitude of workloads, system calls, and performance aspects. Our evaluation demonstrates that uFS achieves strong single-thread performance, scales well with multi-cores, and that our design choices are effective.

4.1 Single-Threaded uServer

We introduce a single-threaded version of uFS to discuss the fundamental issues for a filesystem semi-microkernel: request scheduling, data structures, and caching.

Basic Architecture: As shown in Figure 4.1, uFS is composed of a filesystem process (uServer) and a library (uLib) linked with each application. The uServer is a user-level process within its own address space; we begin by assuming uServer is composed of a single thread, but this limitation is removed in the next section. We generally assume that each thread of uServer is pinned to a dedicated core. uServer interacts with the storage device with a hardware submission/completion queue pair containing NVMe commands; pinned memory is used to transfer data. uLib,

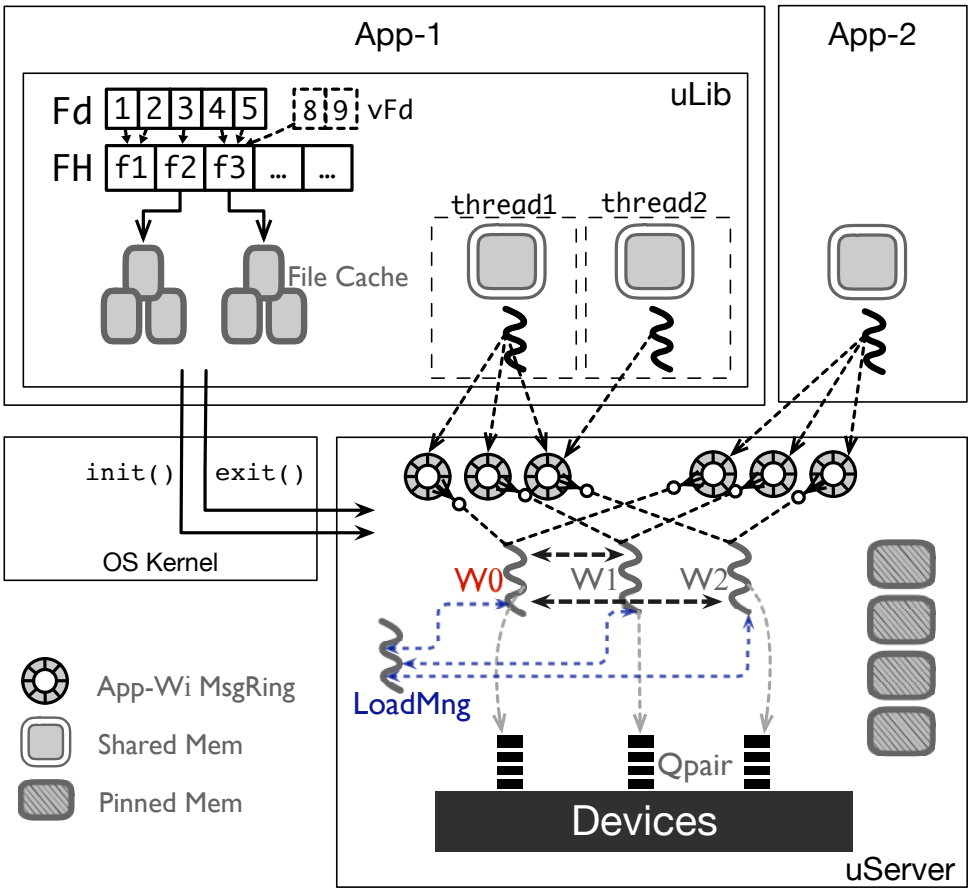


Figure 4.1: **Architecture of uFS, a Filesystem Semi-Microkernel**. *Multiple applications can share a single uFS. App-1 has a separate ring-buffer to communicate with each uServer worker; to minimize data transfer, App-1 contains fd and data caches and shares memory with uServer. uServer contains multiple workers pinned to cores, of which one acts as a primary; the load manager thread is not pinned. Only initialization must pass through the OS kernel.*

which is dynamically linked into each application, offers POSIX compatibility and coordinates the connection to uServer. Control and data transfers between uLib and uServer are separated. For control, uFS uses a per-application thread-safe lockless ring buffer. For data, each application I/O thread performs allocations from thread-private memory that is

shared with uServer.

The only time in which uFS interacts with the OS kernel is for initial authentication: when an application begins, uLib transparently invokes a new system call `uFS_init`. This system call assigns a key to each application and retrieves the application's credentials (i.e., pid, uid, and gid), which are then stored in uServer. This key is returned to the application and passed to uServer as part of any operation that requires permission checks.

uFS is POSIX-compliant with the exception of support for extended attributes, links, mmap, and chmod/chown. Many applications run seamlessly with uFS because of such compatibility [95, 197]. Supporting mmap could further improve the usability of uFS. One could leverage *userfaultfd* to dispatch an application's uFS-related page faults for mmap'd files to the uServer; we leave this to future work. uFS is still an initial implementation and is missing some optimizations found in mature filesystems such as read-ahead and delayed allocation.

Scheduling: To provide low latency and high throughput across clients, uFS balances attending to client requests with keeping the device utilized. The single-threaded server iterates through five tasks: receiving requests from clients in the message rings and placing them in a single internal ready queue; processing requests in the ready queue (currently in FIFO order); attending to background activity (e.g., flushing dirty blocks to the device and freeing blocks from deleted inodes); initiating device requests; polling the device for request completion; and notifying the client of results. Processing a client request may generate intermediate operations that are also placed in the ready queue (e.g., pathname lookup creates intermediate operations for reading and checking the permissions of each intermediate inode and directory). The server continues polling and serving requests while other I/O operations are underway.

Data Structures: uFS uses on-disk data structures similar to other

UNIX-based filesystems: superblocks, inodes, bitmaps, and directory entries. On-disk inodes are 512 bytes with standard information; in-memory inodes track related FDs and states (dirty, deleted, checkpointed). Bitmaps track blocks of different extent sizes.

Directory entries are simple mappings of name to inode number. The dentry cache is combined with a recursive permission map. For example, for the directory /a/b, the root map stores the pair <a, perms+map for /a>; the map of /a, stores <b, perms+map of /a/b>. As path are visited, they are cached in this permission map, with information obtained from inodes as needed.

Caching and Copy Elimination: uFS reduces data movement across the application, uServer, and device with three techniques: caching, leases, and shared memory. First, to avoid unnecessary I/O between the server and device, uServer contains a pinned user-level block buffer cache for inodes and data blocks. This simple LRU cache is accessed by physical block number; therefore, in-memory data structures contain pointers to the original on-disk representations.

Second, to avoid IPC round trips between the client and server, file descriptors and data are cached on clients with leases. Client caching of file descriptors (FDs) enables a subsequent open, close, or lseek (if it does not depend on current file size) to be handled locally by the client. The FD lease is invalidated if another client renames or unlinks this file, at which point the local objects are flushed to the server. FD caching improves the latency of an open from 5.5us down to 1.5us. The client cache of read data blocks is private to each process (but shared by threads). Multiple client processes can simultaneously hold a read lease; if a write request arrives at the server, the read lease will not be renewed and the writer must wait for two lease terms to expire. When there are no read leases, all reads are sent to the server. Read caching improves the latency of 16KB reads from 10us on the server down to 4.3-8us. We have also implemented a

prototype write cache, which is only enabled for the ScaleFS-Bench and LevelDB experiments. When cached, writes to newly created, private files are kept local until fsync is called on that file, at which point the dirty data is flushed to uServer.

Third, to avoid copying data between the application itself and uLib, applications can directly access a memory region shared between uLib and uServer. uFS introduces `uFS_malloc` to allocate from this shared space; the shared buffer can be exposed to the end application for maximum performance, or hidden within uLib for portability. For example, an application can use a buffer from `uFS_malloc` and then pass it to `uFS_allocated_write` to avoid any copies between the application, uLib, and uServer. Alternatively, if an application calls `uFS_write(buf)`, uLib calls `uFS_malloc`, copies the contents of buf to shared memory, and then calls `uFS_allocated_write`. Avoiding this extra copy significantly improves latency; for a 16KB append, copying data to the server requires 8.5us, sharing an allocated buffer requires 6.5us, and local caching requires only 2.3us.

4.2 Multi-Threaded uServer

A single-threaded semi-microkernel may not be able to deliver the full bandwidth of current I/O devices to applications. Monolithic kernel filesystems scale with additional cores with task parallelism: application threads running in privileged mode can concurrently access the same data on different cores. However, kernel filesystems have scalability bottlenecks from data dependencies and synchronization [43, 144].

In contrast, uFS adopts data parallelism for scalability, dividing filesystem data structures across different cores in a shared-nothing architecture [184]. Thus, the server process can be composed of multiple threads. A multi-threaded server must chose the granularity at which to divide

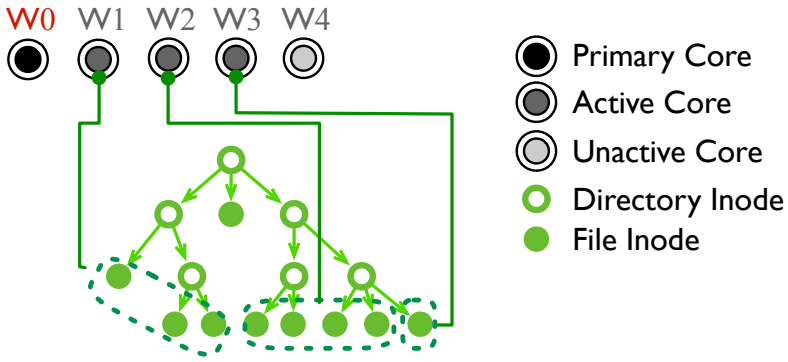


Figure 4.2: **Dynamic Inode Ownership**. *The dashed line indicates sets of inodes owned by each worker other than the primary; there is no relationship between the directory namespace and ownership. All inodes begin on the primary, but may be reassigned based on load. The primary owns all directory inodes.*

filesystem data across threads: the more fine-grained, the more parallelism, but also the higher the complexity and synchronization. In uFS, each server thread holds exclusive ownership of an individual file and each file can be mapped to any thread. Unlike other approaches that have statically partitioned files or data across nodes [84, 96], in uFS the mapping of inodes to threads is dynamic and independent of the directory hierarchy. The drawback of per-inode partitioning is that traffic to a single (or busy) large file cannot be split across server threads.

Basic Architecture: uServer is divided into multiple threads, with each thread pinned to a dedicated core. Each thread accesses the shared storage device directly with its own qpair; qpairs are not shared across threads, so no locking is needed. Similarly, each server thread has its own ring buffer to communicate with uLib in each application.

uServer is composed of one or more *worker* threads; one of the worker threads also acts as a *primary*. A ring buffer is added between the primary and other workers for communication within uServer. Each worker owns different file inodes and thus handles corresponding file operations. Be-

yond regular inode operations, the primary has two additional responsibilities. First, the primary owns all directory inodes; as a result, directory operations are serialized in the primary. By locating all directories in the primary, uFS avoids complex coordination for cross-directory operations. Second, the primary tracks the assignment of file inodes to threads. The primary possesses global knowledge of current inode assignments and serves as the central hub for the mechanism of reassigning inodes. All file inodes are initially assigned to the primary, but will be reassigned to other workers depending on load. A division of inodes across server threads is illustrated in Figure 4.2.

For directory operations, uLib contacts the primary thread. For file operations, uLib can contact any thread; if the contacted thread is not currently the file owner, uLib is notified and redirects requests accordingly.

Scheduling and Caching: With a multi-threaded server, each thread has its own ring buffer per-application, its own ready queue, and its own qpairs with the storage device. In its scheduling loop, each worker now also handles requests sent by the primary. A multi-threaded server changes caching only in that the server user-level buffer cache is now per-worker. In our prototype, each thread is allocated a fixed amount of pinned memory; dynamically sizing the buffer cache per worker remains future work.

Data Structures: Filesystem data structures are dynamically divided across server threads, with the inode as the unit of division. The worker that currently owns an inode is guaranteed to be the sole thread accessing the corresponding data, both in-memory and on-disk; thus, the ownership of an inode grants access to the data structures for handling operations involving only this inode (e.g., reading/writing/allocating data and reading inode metadata). With this separation, there is no lock or data contention for file operations. To enable independent writing of inodes across threads uFS ensures an inode fits in the atomic unit of the storage device (512B).

To manage the dynamic assignment of inodes to workers, the primary contains an inode map mapping inodes to workers; each worker tracks a list of inodes it owns. Given the primary owns all directory inodes, the primary modifies all dentries and performs all directory operations (e.g., creat); as a result, only the primary can allocate and deallocate inodes. Thus, the primary owns the imap and all dentries.

The on-disk representation of data bitmaps are more complex to handle since workers must allocate data blocks without synchronization; although bitmaps provide efficient allocation, they do entangle operations across threads given 512B atomic updates. The in-memory bitmap, in contrast, can be shared by several threads given atomic updates to a single cache line. Thus, the primary contains a *dbmap* block allocation table that maps data bitmap blocks to workers. Once a data bitmap block is used by a worker, that assignment is immutable; each worker allocates many dbmaps at a time to efficiently perform its own block allocations.

uServer also minimizes the sharing of in-memory data structures across cores. The dentry cache plays a critical role in performance, particularly for path resolution and permission checking [198]. The scalable lock-free concurrent dentry cache in uFS is based on an industry-quality hash map [3]. As described previously, each level of a pathname is a key in the map to retrieve the inode and the next level's map; the inode's permission bits are compared with the application's uid and gid. The dentry cache is single-writer (primary) and multi-reader (other workers). For most calls to open or stat, the paths are present in the dentry cache and readable by any worker. When an entry is not present, the primary finishes the lookup and inserts the items into the dentry cache. To guarantee atomicity, the primary handles some operations. For example, for atomic renames, no clients should see both filenames; thus, the primary deletes the relevant items from the dentry cache, forcing workers to redirect lookups to the primary.

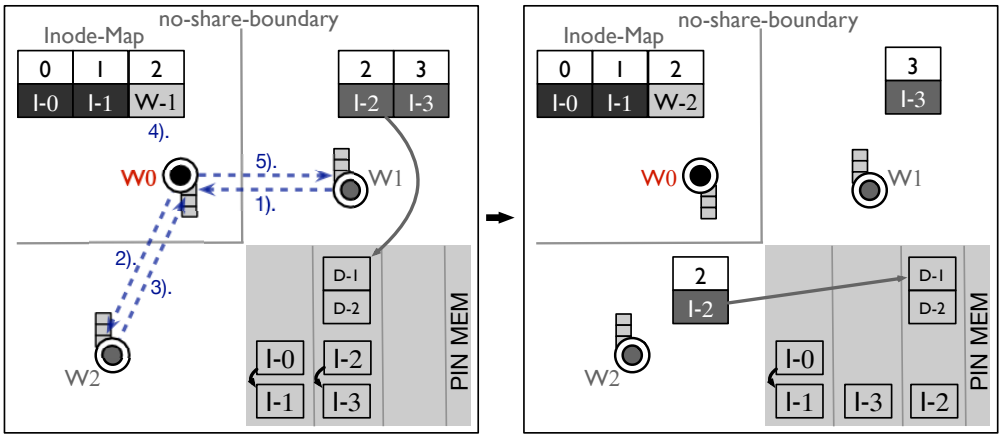


Figure 4.3: **Inode Reassignment**. *The left side shows initial state and 5 steps for inode 2 to be reassigned from w1 to w2. The right shows final state: the InodeMap on the primary is updated and w2 can use data associated with inode 2 in the buffer cache.*

Inode assignment mechanism: Figure 4.3 shows the mechanisms for reassigning a file inode to a worker; the policies for load balancing and determining the number of cores are described in Section 5.1. The assignment steps are as follows. **1**) The owning thread, w1, initiates the migration of inode I2 by removing I2 from its inode list and completing any related requests. The owner notifies the primary of all state associated with I2 (e.g., opened FDs and entries in the buffer cache). **2**) The primary marks the owner of I2 in its inode map as unknown and forwards this request to the new owner, w2. **3**) The new owner sets up I2's context by linking I2 into its own inode list and extracting the buffer cache entries it can use (no copying is performed); it sends an ack to the primary. **4**) The primary changes I2's owner to w2 in the inode map **5**) The primary notifies w1 that the reassignment is complete. Any requests that arrive at a non-owner are returned to the client to retry at the primary. Once the primary knows the owner, it informs the client to redirect requests to the new owner.

4.3 Crash Consistency

uFS is a crash-consistent filesystem based on ordered metadata journaling [162]. Like other ordered metadata journaling filesystems, uFS first writes user data blocks to their in-place locations on disk; then, within a transaction in the on-disk journal, it logs a description of the metadata changes; after the transaction is marked committed, the in-place metadata can be checkpointed and the transaction marked free. If a crash occurs after the transaction is committed but before it is freed, recovery replays the changes from the transaction. Without journaling, while running, uFS only flushes dirty data blocks for files and directories and on a graceful shutdown writes bitmaps and inodes.

Basic Architecture: uFS achieves highly-scalable performance with crash consistency by allowing each thread to write to a shared journal with minimum coordination. uFS achieves this by leveraging the property that each inode has one owner and, therefore, the owner can perform the transactions involving that inode. However, this ownership is complicated by the fact a migrated inode may contain blocks that were allocated on different workers. Physical journaling at the per-block level, as in ext4, would require writing block bitmaps that are not owned by the inode owner, requiring coordination.

uFS avoids coordination with logical journaling. Each in-memory inode tracks the associated updates to other metadata structures (e.g., the data bitmap) in its ilog, an in-memory per-inode logical log that moves with its inode if reassigned. Thus, when a worker writes a transaction with an inode, it owns everything needed to apply the logical changes. When an inode is reassigned, it leaves no residual state with the previous thread [57]; as a result, an fsync on a reassigned inode requires no coordination with other threads. The primary performs similar operations for all directories with a logical dirlog.

uFS uses a global journal; the global journal simplifies the task of ap-

plying transactions in order, while still allowing threads to write concurrently. Because each thread knows the number of journal blocks for a transaction, it can atomically reserve a contiguous range of blocks; threads writing later simply reserve the next range. Journal recovery handles the case where entries that appear earlier in the journal are not committed.

Commits: In the common case of an fsync of a file, the owning thread commits the single ilog. For batched transactions, multiple ilog entries from the same worker can be placed in the same journal entry. For a full system sync, each worker fsyncs its own inodes. Directory operations (such as rename) that require atomicity across inodes imply that those inodes have the same owner; in uFS, all directories are owned by the primary fulfilling this requirement. Like ext4, fsync on a dirty directory will fsync all dirty directories; the primary commits the dirlog and ilogs of all dirty directory inodes. uFS avoids orphaned inodes and correctly handles directories that may be committed before the new inodes they reference.

While most data structures across threads are independent of one another, some exceptions exist. First, an unlink of an inode owned by a worker other than the primary (which owns the directory) requires reassigning the inode to the primary. Second, dependencies can occur across file inodes due to re-allocated data blocks. For example, unlinking an inode X may deallocate a block B which is then allocated to inode Y. To prevent the incorrect ordering of Y, X in the journal, deallocated blocks can be reallocated only after they are committed (similar to reuse after notification [40]).

Checkpoints: A checkpoint, triggered by low free space in the journal, writes committed metadata (i.e., inodes, bitmaps, and directory data) to their in-place locations. Since the current in-memory metadata may be dirty and not yet be committed, uFS maintains a stable in-memory copy of all committed metadata that is used for checkpoints. uFS uses message passing to update stable versions of data block bitmaps on other workers.

Recovery: After a crash, committed transactions are replayed. The primary challenge in uFS is to replay all committed transactions even if some appear after uncommitted entries; this can occur since threads write concurrently to the journal. If recovery were to stop at an incomplete transaction, committed transactions would be lost.

Incomplete transactions can be skipped for the following reasons. First, multiple fsyncs to the same inode are handled serially by the owner (or workers, in the case of inode reassignment); thus, a later fsync to the same inode will not complete if the previous fsync to that inode did not. Similarly, if an incomplete entry contains multiple inodes, it is guaranteed that none of those inodes are in later transactions. Second, similar to other filesystems such as ext4, XFS, and Btrfs [168], on an fsync failure, uFS will accept no more writes; thus, if recovery encounters an incomplete entry, no subsequent journal entries will involve the same uServer thread.

Recovery finds the end of the journal by reading its superblock. Since uFS updates this on-disk superblock only periodically, the contents may be stale by N blocks; thus recovery reads N blocks past the end to find valid entries.

4.4 Evaluation

uFS is implemented in about 35K lines of C++ code and is publicly available [6]. We have also developed tools to simplify development and to check correctness. Our command line tool, cli, supports operations like listdir, stat, and mkdir; cli also transfers files to and from the host filesystem, explicitly manages inode ownership, and dumps metadata for checking.

We run all of our experiments on an Intel Xeon Scalable Gold 6138 SkyLake 2.9GHZ processor with 20 physical cores. We use one machine with 60GB (for microbenchmarks) and another with 120GB of RAM (for

	Random Sequential	Memory Bounded Disk Bounded	Private Share
read	x	x	x
write	x	x	x
append	-	x	x
stat1	-	-	x
statAll	-	-	x
listdir	-	-	x
creat	-	-	x
unlink	-	-	x
rename	-	-	x

Table 4.1: **32 Single Op Microbenchmarks**. An x indicates the specified parameter is varied; - indicates it is not. Data operations are 4KB; writes are non-allocating.

Filebench, and ScaleFS-Bench). Each filesystem runs on an Intel Optane 905P Series (960GB) SSD. The OS is Linux 5.4.0 and uFS uses SPDK 18.04.

We evaluate uFS by answering the following questions:

- How good is the baseline of single-threaded uFS compared to ext4?
- Is uFS scalable with multiple server cores?
- Is client-side caching effective?
- Does crash consistency add significant overhead?

4.4.1 Correctness

We ensure uFS passes the test cases in Linux's LTP project [129], adding inode reassignment across workers at controlled points. We use LevelDB extensively to validate data integrity since it stresses the filesystem and checksums all operations.

We have experimentally verified that uFS is crash consistent for a range of scenarios. Using an approach similar to others [148, 158],[1] we emulate crashes by systematically corrupting blocks in the on-disk journal; we recover with those corrupted images and verify that the recovered filesystem matches expectations. We use workloads with multiple applications that perform allocations and commit to the journal. After recovery, all files had the expected size and data, and all bitmaps were consistent. We also test creat, rename, mkdir, and unlink; all directories and files are as expected and uFS is consistent after recovery.

4.4.2 Benchmarks and Methodology

We evaluate single-threaded uServer by crafting 32 *single op microbenchmarks* to understand how well each basic primitive (i.e., filesystem operation) perform in uFS. We then evaluate the multiple-threaded uServer with the single op microbenchmarks, comparing its performance to the single-threaded uServer for each workload. Workloads from Filebench [193] are also used to evaluate the scaled performance of uServer. Finally, we evaluate the uServer with ScaleFS-Bench for scalability comparison with a state-of-the-art scalable filesystem – ScaleFS [23].

We commonly compare with a monolithic kernel filesystem, ext4. We use ext4 as our standard because it is a widely-used highly-optimized kernel filesystem that scales well under many different workloads[144]; uFS uses similar mechanisms and data structures to those in ext4 (as opposed to the B-Trees in XFS [187] and Btrfs [135]).

To evaluate base performance and scalability, we have created 32 single op microbenchmarks for data and metadata primitives, as described in Table 4.1. The benchmark contains data operations (read, write, and append) and metadata operations (stat1, statAll, listdir, creat, unlink, and

[1] We cannot use CrashMonkey because the tool replays bio_requests not present in SPDK.

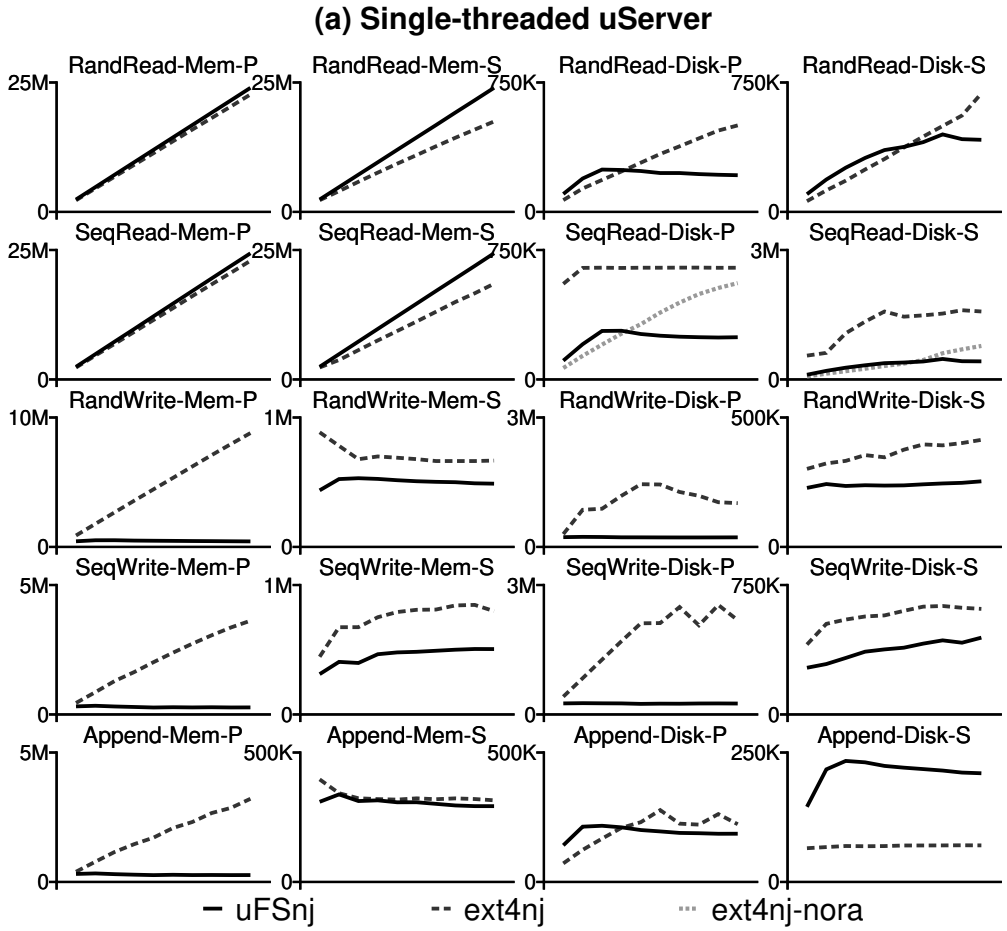


Figure 4.4: **Data Operation Performance (a): Single-Threaded**. *The x-axis shows the number of clients (up to 10), and the y-axis shows the throughput. The number of uServer cores is fixed at 1. In "*-Mem-*" workloads, client read-caches and the server cache are warmed for uFS; the buffer cache is warmed for ext4; writing cache in uFS is not enabled and we ensure no disk access happen in "*Write-Mem-*" cases. Results with ext4 no-readahead (i.e., nora) are shown for sequential reads from disk. "nj" indicates the journaling is disabled.*

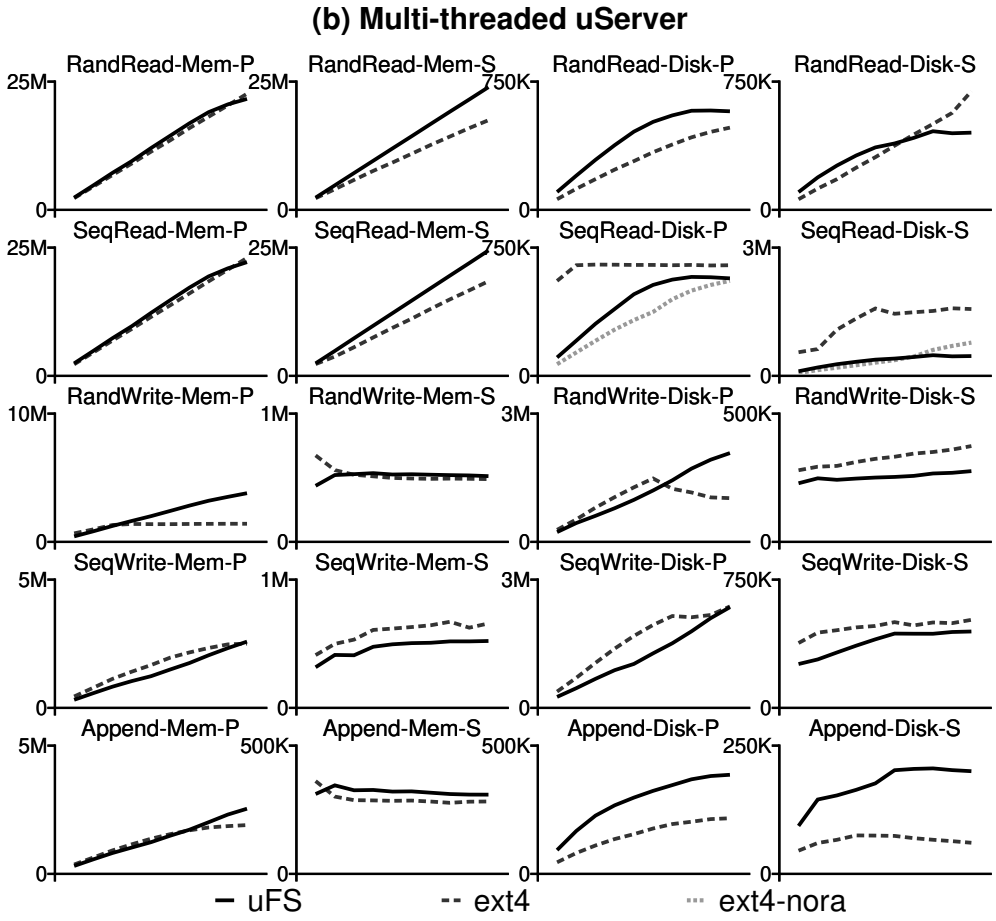


Figure 4.5: **Data Operation Performance (b): Multi-Threaded**. *The x-axis shows the number of clients (up to 10), and the y-axis shows the throughput. The number of uServer cores is scaled to match the number of clients (up to 10). In "*-Mem-*" workloads, client read-caches and the server cache are warmed for uFS; the buffer cache is warmed for ext4; writing cache in uFS is not enabled and we ensure no disk access happen in "*Write-Mem-*" cases. Results with ext4 no-readahead (i.e., nora) are shown for sequential reads from disk. Both ext4 and uFS use journaling.*

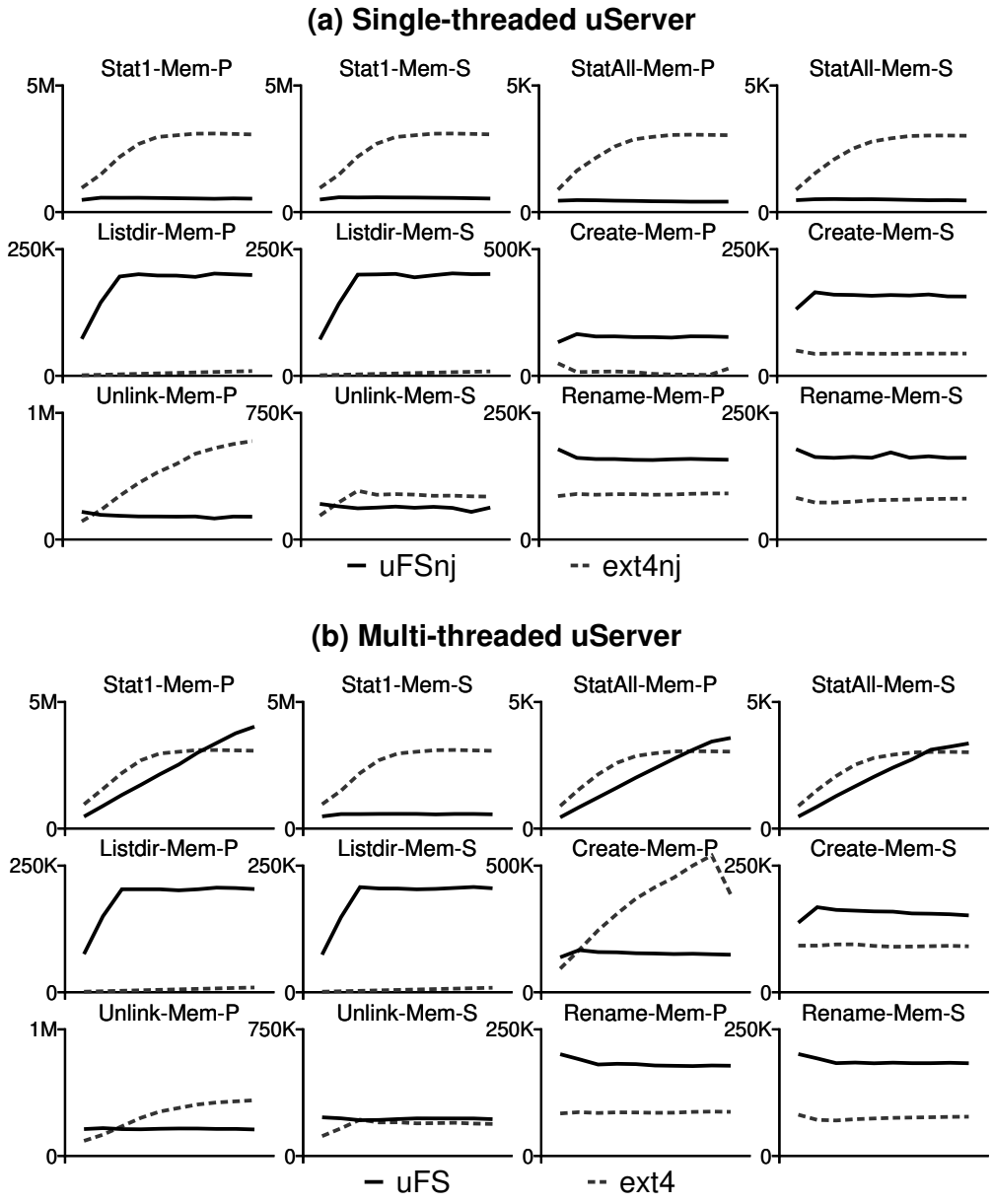


Figure 4.6: **Metadata Operation Performance: Single-Threaded vs. Multi-Threaded**. *The x-axis shows the number of clients (up to 10), and the y-axis shows the throughput. In (a), the number of uServer cores is fixed at 1; in (b), the number of uServer cores is scaled to match the number of clients (up to 10). In all the experiments, the benchmark suite performs warmup round for both systems. "nj" indicates the journaling is disabled and both ext4 and uFS use journaling in (b).*

rename). The combination of each operation and its parameters generates 32 workloads, stressing uFS in different aspects.

For data operations, the access pattern could be either sequential or random, and the target data could be completely in-memory (i.e., after warmup) or on-disk (i.e., accessed once with a cold cache). Metadata operations are always memory-bound as the amount of metadata is relatively small. For all operations, our microbenchmarks cover cases where the accessed metadata/data is shared by multiple clients and where the accessed metadata/data is private to each client.

We present the base performance of single-threaded uFS and ext4 in Figure 4.4 (data operations) and Figure 4.6 (metadata operations), both in (a) for later comparison with the scaled uServer in (b) (§4.4.4). We describe the details of the workloads in below sections.

4.4.3 Single-Threaded uFS

We have two goals in evaluating single-threaded uFS. First, we demonstrate that with a single client, uFS delivers reasonable performance relative to that of a monolithic kernel filesystem, ext4. Second, we show that as the number of clients increases, a single uServer core is a bottleneck for I/O-intensive workloads.

For base performance, we examine only the left-most point in each graph (1 client). For many in-memory data operations, specifically read (sequential and random) and append, ext4 and uFS perform similarly. The exceptions are that ext4 performs better on in-memory overwrites (sequential and random, shared and private files); one client performs particularly well on ext4 because data is not shared (nor invalidated) across CPU caches.

For on-disk workloads, uFS performs better for append and random reads; with a single client, uFS outperforms ext4 by 1.5x for random reads due to the efficiency of its device-access path. Ext4 performs better for

sequential reads because read-ahead is not yet implemented in uFS; disabling read-ahead in ext4 removes this advantage. For on-disk overwrite workloads, we lower the kernel's *dirty_flush_ratio* to ensure that ext4 writes a similar amount of data to disk as uFS; however, overwrites still perform worse on uFS because it does not yet perform sophisticated batching for background flushes. Finally, uFS performs notably better for synchronous journal-intensive workloads (e.g., sequential appends to disk) due to its fast device access.

For metadata operations from a single client (Figure 4.6), uFS performs better than ext4 on listdir, create, and rename, and similarly on stat and unlink; uFS performs especially well on listdir by pre-fetching dentries during opendir. Overall, uFS is sufficiently well-optimized relative to ext4 for uFS to be a reasonable semi-microkernel building block.

As illustrated by the full set of data points in Figure 4.4 and Figure 4.6 (a), the scalability of single-threaded uFS and ext4 with additional clients is dramatically different. Although many write and append workloads had comparable performance on a single core, ext4 scales with the number of clients, whereas single-threaded uFS does not. For many of the metadata operations, scalability is flat for both systems; one exception is stat, for which ext4 scales but single-threaded uFS does not. For many read workloads, while both ext4 and single-threaded uFS scale somewhat, ext4 scales better.

We explore single-threaded uFS for random on-disk reads in more detail. Figure 4.7 shows the CPU utilization of the uServer as a function of the bandwidth it is able to deliver; although increasing the size of reads (4KB-64KB, across lines) and the number of clients (within a line) improves bandwidth, the single core is 100% utilized with just 2 or 3 clients and thus never obtains the peak device bandwidth of 2.5GB/s. These results show that multiple server cores are required for scalable performance.

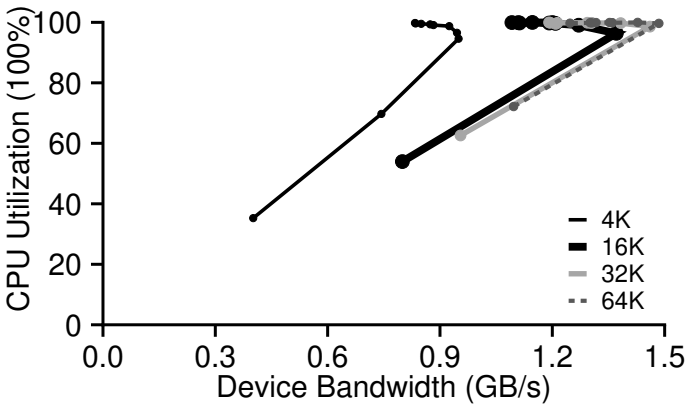


Figure 4.7: **Single-threaded Server Bottlenecks**. *CPU utilization as a function of delivered bandwidth for different random read sizes and numbers of clients with 1 uServer core.*

4.4.4 Multi-Threaded uFS

Given a more intense I/O workload, the multi-threaded uServer can effectively utilize additional cores. We demonstrate this scalability for the single op microbenchmarks, for Filebench's Varmail and Webserver [193], and for ScaleFS-Bench [23]. We show that journaling does not harm the scalability of uFS, and uFS benefits significantly from client-side caching.

4.4.4.1 Single Operations

Figure 4.5 and Figure 4.6 (b) show the performance of uFS and ext4, both with ordered metadata journaling, on the single op microbenchmarks. The server is allocated as many cores as there are clients; this represents the best-case performance for uFS when no sharing of workers or load balancing is needed.

Comparing uFS's performance to that in Figure 4.4, we see that many operations benefit significantly from additional server cores. In particular, the throughput of reads to private on-disk files increases significantly

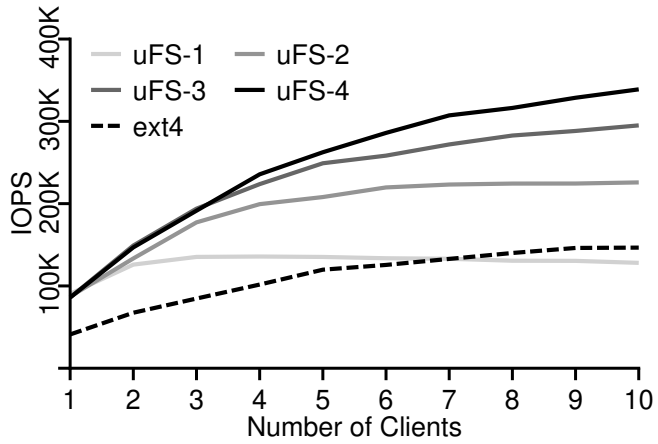


Figure 4.8: **Multi-threaded Varmail**. *Different lines represent different numbers of uServer threads.*

since each worker can perform independent I/O and quickly reach the device throughput limit; uFS now scales slightly better than ext4 with read-ahead disabled. Similarly, writes (both random and sequential) and appends to private files, whether in-memory or on-disk, scale since each worker can be used effectively. The scalability of writes and appends to shared files does not improve because the load is directed to a single worker. uFS makes this trade-off based on the assumption that even though file sharing is common, intensive shared-file access within a small time interval is rare.

The scalability of reads to memory remains similar to that with a single core since client caching was effective to begin with. Finally, metadata operations involving directories are still handled by the primary, and thus do not scale; stat on private files and statall on private and shared directories all scale well since groups of files can be handled by different workers.

Comparing ext4 with journaling in Figure 4.5 to ext4 without journaling in Figure 4.4, we see that random writes to private in-memory files perform much worse with ext4 journaling, because ext4 starts a journal transaction (and suffers from spinlock contention) even though an over-

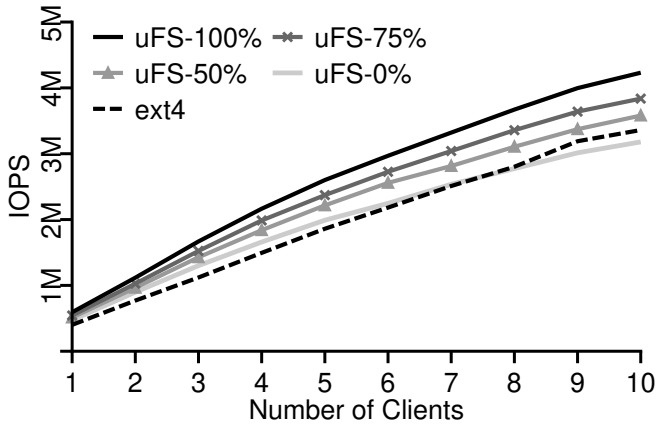


Figure 4.9: **Multi-threaded Webserver**. *Different lines represent different percentages of the workload fitting in the client cache.*

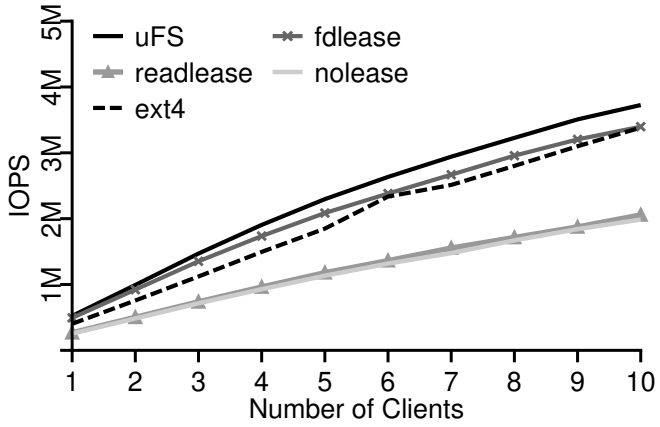


Figure 4.10: **Multi-threaded Webserver (Readonly Workload)**. *Lines show the impact of leases in uFS for a 50% client cache hit rate.*

write operation doesn't require a new journal transaction [5]. The improved performance of ext4 with journaling on create is a known anomaly [144].

Journaling in uFS does not have as strong of an impact because each worker thread participates in writing to the journal. With more detailed experiments of uFS journaling (not shown), we have verified that as the frequency of fsync increases, performance of journaling decreases, as ex-

pected, and this decrease is due to writing more data to the device and not synchronization. Writing to the global journal involves a small critical section to reserve the contiguous blocks, but eliminating this synchronization does not improve performance (validated by writing to per-worker journals). We have also verified that journaling in uFS does not impose overheads on write-intensive in-memory workloads which must track logical changes to in-memory inodes in ilogs. Journaling and no-journaling uFS obtain equivalent throughput (graph not shown): about 900kops/s with 64 byte writes and 350kops/s with 4K writes. All of our subsequent experiments use journaling in both ext4 and uFS.

4.4.4.2 Varmail: Scaling uServer

uFS obtains good base performance and scalability on I/O-intensive workloads beyond single operations. The Varmail benchmark in Filebench [193] performs reads and writes to many 16 KB files; we modify Varmail to perform periodic fsyncs so that data is written to disk during the benchmark. Varmail stresses file allocation and deletion, and is characterized by many small writes to separate files followed by fsyncs. In uFS, the file creates are all performed on the primary.

We compare uFS and ext4 scaling the number of clients, closely examining the benefits of additional workers.[2] As shown in the first graph of Figure 4.8, uFS is much more scalable than ext4 on Varmail. Ext4 does not scale well with additional clients because the one jbd2 journaling thread becomes a bottleneck performing the many fsync operations. uFS is well-suited to the Varmail workload because each client reads and writes independent files, which can be distributed efficiently across workers.

Even for the base case of a single client and worker, uFS performs better than ext4 due largely to the difference in fsync time (30us vs. 100us).

[2]Since Varmail is relatively static, uFS performs static inode balancing such that the primary handles no file inodes given many other workers ($\geqslant 3$), and only a percentage of file inodes with 1 or 2 others.

When the number of clients is scaled but uFS is limited to a single worker, uFS performs better than ext4 up through 7 clients. Increasing the number of uServer workers to 2, 3, and 4 continues to increase throughput as each worker initiates more I/O requests; increasing beyond 4 workers does not improve performance because the primary is the bottleneck (CPU $>$ 75%). These results motivate the need to dynamically choose the number of workers to not waste resources.

4.4.4.3 Webserver: Caching in uLib

We evaluate how well uFS handles read-intensive in-memory workloads using the Webserver in Filebench [193]. Each client opens, reads, and closes 10,000 16KB private files; a small write is performed to a log file after 10 reads. Both ext4 and the uFS server easily cache the working sets for all clients in main memory and thus neither triggers substantial I/O read traffic. The Webserver stresses the ability of a single worker to handle many appends to a single file and for clients to efficiently cache recently-accessed in-memory data.

We isolate uFS client cache performance by configuring each client's read cache to contain from 0 to 100% of the client's working set; each client's FD cache fits all its opened files (requiring only 64B/FD). The second graph of Figure 4.8 shows uFS outperforms ext4 when the client cache hit rate is above 25%: handling reads within the uLib client is extremely efficient. Furthermore, uFS outperforms ext4 with only a few clients when their append rate to a single file can be handled by a single worker.

The third graph of Figure 4.8 shows the effectiveness of FD and read leases for a 50% read-cache hit rate (patterns for other hit rates, not shown, are similar). In this workload, read leases without FD leases are not beneficial because every read is preceded by an open. FD leases on their own are effective since the benefit of an FD lease is much higher than a read

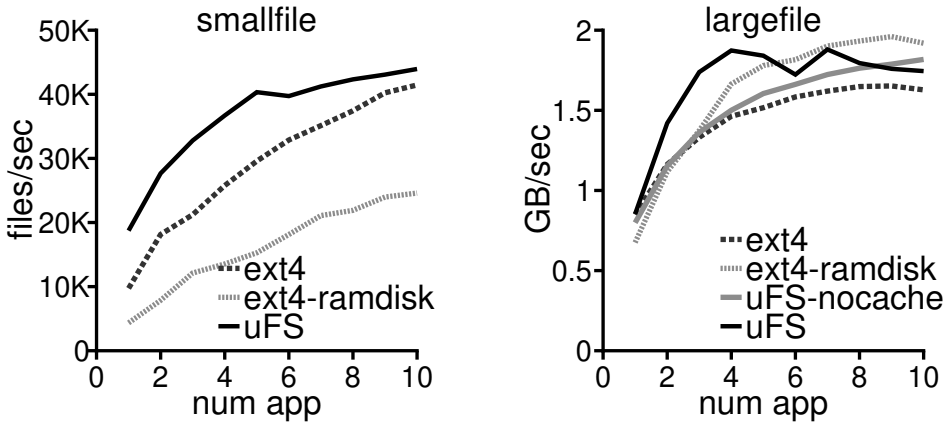


Figure 4.11: **ScaleFS-Bench Performance**. *The throughput of smallfile and largefile workloads. ext4-ramdisk indicates ext4 is using ramdisk. In the second graph, uFS enables the write cache to handle 256K continuous 4KB append before fsync().*

lease (open: 5.5us on server vs. 1.5us local; 16KB read: 10us on server vs. 4.3-8us local). As shown by the final uFS performance, given an FD lease, a read lease provides additional benefits.

Since the cost of a client opening a file and reading from the buffer cache in ext4 (2.5us and 6.5us) is less than the cost of a client transferring data from the server, uFS performs client caching with read and FD leases. While both leases are needed for uFS to outperform ext4 for read-intensive in-memory workloads, FD leases are especially valuable given their low memory overhead.

4.4.4.4 ScaleFS-Bench

We evaluate uFS with two more workloads to better understand how uFS compares to ScaleFS, a scalable kernel filesystem developed in xv6 [23]. We port their smallfile and largefile benchmarks (with minimal modifications) and follow their methodology of using ext4 on ramdisk as the baseline. We cannot compare ScaleFS directly to uFS due to a lack of hard-

ware support for NVMe in xv6.

In the first graph of Figure 4.11, each application creates 10,000 1KB files, calls sync once, reads each file, and unlinks each file. uFS performs better than ext4 at each data point in the graph, yet ext4 scales better due to the burst unlink phase that stresses uFS's primary worker. If we eliminate the unlink phase, uFS has 1.4x performance on the right-most data point, indicating an optimization for bulk primary-only operations (such as unlink) would be useful.

In the second graph, each benchmark instance creates one private file, issues 100MB of writes (4KB at a time), and finally calls fsync. uFS achieves device bandwidth (2GB/sec) much faster than any others, yet shows some fluctuation when increasing the number of applications. We believe that more careful scheduling of device IO is required to regulate bursts from multiple concurrent uServer threads and provide consistently high performance. Enabling the write cache avoids unnecessary IPC and thus better utilizes the device.

One surprising finding is that comparisons should be performed on actual devices – not ramdisk – even when focusing on CPU scalability. As seen from the graphs, ext4 on the fast SSD has similar or better performance than ext4 on ramdisk. Upon further investigation (with the *RandRead-Disk-P* workload in Figure 4.5), we found that the kernel spends a large amount of time waiting on ramdisk IO after yielding at *io_schedule*. Thus, the performance of a filesystem run on ramdisk may be limited by the less-optimized block layer.

4.5 Summary and Conclusions

In this chapter, we show that uFS, a filesystem semi-microkernel, achieves strong single-thread performance and excellent multi-core scalability.

To overcome the performance issues of monolithic kernels (costly sys-

tem call and I/O stacks) and microkernels (IPC overhead), uFS incorporates techniques to exploit modern hardware and customize for filesystems to capitalize on the architectural potential: fast app-filesystem IPC, non-blocking filesystem threads, and kernel interaction off the common path.

As opposed to the multi-core scalability bottlenecks found in the Linux kernels [23, 43, 144], uFS allows each core to operate independently as much as possible, avoiding synchronizations among workers for in-memory data structures by dynamically partitioning file inodes across threads, and enabling multiple concurrent sync transactions by a globally-shared logical journal.

Furthermore, uFS addresses other challenges needed for realizing a fully functional and high-performance filesystem, such as correctness of crash consistency, a scalable dentry cache, copy elimination, and client-side caching. As we have shown in our evaluation, each technique is necessary for the performance of a given operation and the workloads that stress a particular request handling path of the filesystem.

We use a variety of microbenchmarks and macrobenchmarks to evaluate uFS. Our 32 single-operation microbenchmarks show that uFS achieves low latency and high throughput, and it scales well, isolating the performance of each operation, and the impact of each filesystem component and design choice. The benchmarks and results are also helpful to make performance estimations for more complex workloads. uFS, while using multiple threads, offers high performance under Filebench's varmail and webserver benchmarks and demonstrates its scalability under the ScaleFS benchmark.

So far, we have focused on the absolute best performance uFS can achieve while retaining reasonable complexity, without much consideration for resource efficiency. However, as an OS system service, uFS needs to inherit and fulfill an OS-level role for resource management, the topic of the next chapter. Understanding and improving the upper bound of

performance is crucial for designing and measuring the costs and benefits of resource efficiency.

5 Resource Elasticity

One major architectural difference innate to semi-microkernels is the decoupled threads of applications and the OS service [99, 134], allowing to scale the OS service independently from the applications using them [134].

In a monolithic kernel, the CPU resources used by applications and filesystems are coupled because they use the same threads to execute code, where the application portion is in user space and the filesystem portion, written by kernel developers, is in kernel space.

With a filesystem semi-microkernel, the applications and filesystems no longer share the same thread entity, allowing the filesystem to scale resources independently of the number of application threads. Intuitively, during peak load, more resources, especially CPU cores, are needed to handle the application workloads. The filesystem needs to scale up to meet these demands, and, equally importantly, scale down when the load decreases to avoid wasting cores.

In this chapter, we discuss the resource elasticity of uFS, focusing on CPU resources. We present the design of the load management feature of uFS, starting with the mechanism to effectively collect runtime statistics and enforce load management decisions. uFS incorporates a centralized thread that periodically gathers runtime statistics from the worker threads and makes decisions based on our load management policies.

We extract simple statistics that enable high-quality decision-making. Our algorithm detects CPU resource mismatches with the current load,

determines the appropriate number of cores, and ensures proper load distribution across workers. It addresses core allocation by estimating and comparing the resulting performance and CPU efficiency across different configurations: adding a core, removing a core, or retaining the current number of cores. This comparison is facilitated by the algorithm, which orthogonally solves the load balancing problem by estimating the ideal load status for each worker given a particular number of cores.

The chapter is organized as follows. We first present the design of the load management (§5.1), including the basic architecture, policies, and algorithms. We then evaluate the CPU resource elasticity of uFS using microbenchmarks with controlled changes of various parameters such as data size, intensity, and hotness (§5.2.2 and §5.2.3). Finally, we demonstrate that uFS with load management meets or exceeds Linux ext4 performance, in some cases by a large margin under high application demand (§5.2.5).

5.1 Load Management

The load management feature of uFS adapts the number of cores dedicated to the server and balances the allocation of inodes across those cores as a function of the current workload. Determining the number of cores is both a challenge and an opportunity that does not exist for traditional kernel filesystems. One option is to statically set the number of uServer threads equal to the number of I/O-intensive application threads; this enables each application thread to send most of its work to a dedicated server thread. However, for many workloads, there is a mismatch between the ideal number of application and server threads: if a few server threads saturate the I/O device, there is no benefit to adding more; if a single client generates significant I/O, additional threads may be useful. Therefore, the option we explore is to dynamically choose the number of

server threads to obtain both high I/O throughput and a low core count.

For a given number of cores, uFS determines an assignment of inodes to workers that balances the load with a minimum of inode reassignments. This inode assignment must take into account a number of factors. First, the amount of work for each inode is different, depending on the rate of requests, the types of requests (e.g., reads vs. writes or size), and current system state (e.g., whether data is cached and operations will be in-memory). Second, the amount of work associated with an inode can change substantially over time (e.g., accesses to a particular file can be bursty or only occur in one phase of an application [75, 179]). Finally, co-locating inodes from the same client can improve performance, due to queueing delays.

Basic Architecture: uFS adds a low-overhead *load manager* thread to the server (not pinned to a dedicated core); the manager wakes periodically to gather load statistics from each worker, decide on the number of cores to use in the next window, and to direct the workers to perform load balancing. The manager has minimal responsibilities: it tells each overloaded worker only the goal it must achieve in terms of how much load to redistribute; the manager does not tell workers how to achieve this goal (e.g., which inodes to redistribute). Thus, the overhead of identifying inodes is distributed across the workers; each worker contains detailed knowledge of the load caused by each inode and can accurately determine which inodes should be moved. The primary, handling all directory operations, has extra work compared to other workers; this load is included naturally in this approach and thus fewer file inodes may be allocated to the primary.

Goal and Statistics: Though different goals are possible, uFS tries to minimize both the number of cores and the queuing time of each request, by keeping each below a configurable threshold. Thus, each worker collects the CPU cycles spent on useful work within its scheduling loop, the

CPU cycles spent on work for each client, and *congestion*, the average number of independent requests in the queue ahead of each request.[1] Statistics are smoothed across collection windows; the manager does not need a globally-consistent view and may read worker statistics at slightly different times.

The manager translates between per-client congestion and per-client load; both metrics are needed because clients care about their observed congestion, whereas the system can more easily manage load. The conversion takes into account dependencies across synchronous requests from the same client and non-linear effects at high loads.

Algorithm: Periodically (every 2ms), the manager determines whether cores should be added/removed and/or load should be redistributed. The current N workers are split into source and destination sets based on whether their congestion falls above or below a threshold. If there are no workers with high congestion, the manager predicts that the workers can be reduced to $N-1$ if a set of workers can accept all the load from the least-busy worker while maintaining low congestion. Otherwise, the manager determines if better load balancing on the current N workers would reduce congestion below the threshold. Because keeping requests from one client on the same worker reduces queueing delays for synchronous requests, the manager first attempts to move all load associated with an entire client; if this is not sufficient, the manager determines percentages of client load to move. Finally, if no amount of load can be moved to meet the congestion goals, these steps are repeated with $N+1$ cores. To increase stability, the predicted congestion must match measurements for several windows before the number of cores will be changed or load shifted.

At the end of each balancing window, the manager has determined the amount of per-client load to shift from each over-loaded worker to each

[1] Requests to the same inode are not independent because reassigning that inode will not reduce the waiting time for related requests.

under-loaded worker. This goal is shared with each over-loaded worker, which uses per-inode load statistics to determine a set of inodes with appropriate load. Workers avoid reassigning inodes with low (or unknown) activity, since moving those inodes incurs overhead without substantially shifting load. The worker uses the inode reassignment mechanism described previously.

5.2 Evaluation

uFS balances load across available workers and adjusts the number of workers to achieve low client congestion (performance) and a reasonable core count (CPU efficiency). We evaluate the load balancing and core allocation strategies using well-controlled, dynamic workloads. We also evaluate uFS with load management using LevelDB [68]. We use the same machines with 60GB of RAM for microbenchmarks and 120GB of RAM for LevelDB, as in §4.4. Each experiment is repeated 5 times.

Our evaluation in this section answers the following questions:

- Can uFS adapt to workload changes?
- How well does load management in uFS improve CPU efficiency? And how does it affect performance?
- How well does uFS handle an I/O-intensive application such as LevelDB running the YCSB workloads?

5.2.1 Benchmarks and Methodology

We again create two microbenchmark suites: one for load balancing and another for core allocation evaluation. The workloads are designed to exercise uFS with load management under various degrees of heterogeneity and by inducing different types of load changes, respectively.

We construct 9 *load balancing microbenchmarks*, each varying one parameter as shown in Table 5.1. Each base workload (i.e., a, b, c, e, f, or g) only varies one parameter between half of the clients, including in-memory vs. on-disk, data size (4KB vs. 16KB), hot vs. cold data, and access pattern (overwrite vs. append). Therefore, with each base workload, we independently stress one type of workload variation. Then, the combinations of reading workloads (i.e., abc), writing workloads (i.e., efg), and all workloads (i.e., all-abcdef) gradually increase the heterogeneity of workloads.

We also create 8 *core allocation microbenchmarks*, each of which varies the load placed on the filesystem in a single dimension, as shown in Table 5.2. Core-a changes the percentage of in-memory workload versus on-disk workload by altering the frequency of fsyncs among writes; core-b changes the load intensity of the clients by altering the think time between each application's requests; core-c changes the number of clients stressing the system, and therefore the volume of data they access changes as well; core-d changes the size of in-memory writes, altering the amount of work needed in the in-memory writing path.

We compare uFS with the baseline of uFS_max, where each client is matched with a dedicated worker; uFS_max favors performance without constraints on core usage. Finally, we evaluate uFS with a production-quality data-intensive application, LevelDB [68], using the two load and six YCSB workloads, comparing with Linux ext4. We also report the number of cores used by uFS for each workload.

5.2.2 Load Balancing

We first demonstrate that uFS can balance inodes with different costs across a *fixed* number of cores. We compare uFS to two alternatives: uFS_RR (round-robin inode allocation on the same number of cores as uFS) and uFS_max (each client is matched with a dedicated worker). We stress dif-

	Parameter
read-a	in-mem / on-disk (4KB)
read-b	4KB / 16KB (on-disk)
read-c	hot / cold (4KB, in-mem)
read-abc	2 read-a, 2 read-b, 2 read-c
write-e	4KB / 16KB (with fsync)
write-f	overwrite / append (in-mem)
write-g	hot / cold (overwrite, in-mem)
write-efg	2 write-e, 2 write-f, 2 write-g
all-abcefg	read-abc, write-efg

Table 5.1: **9 Load-Balancing Microbenchmarks**. Each base workload contains 6 clients generating work that varies per inode. The combination workloads contain 6 clients from the base workloads. Each client accesses between 50 and 200 different inodes.

	Parameter	Workload	Range	# of Steps
core-a	On-disk / in-mem	N*write(4K) + flush	N: (1, ∞)	19 7
core-b	Think time	In-mem read + think(T)	T (us): (15, 2)	20 6
core-c	# files	In-mem read	clients: (1, 6)	12 4
core-d	Write size	write(N) + flush	N (KB): (64, 4K)	17 5

Table 5.2: **8 Core Allocation Microbenchmarks**. Each workload varies over time a specific parameter: on-disk vs. in-memory, think time, number of files, and the size of operations. One version varies the parameter gradually (e.g., in 19 discrete steps) while a second more abruptly (e.g., in 7 steps). Each workload contains up to 6 clients each accessing 40 files.

ferent costs per inode by the load balancing microbenchmarks (Table 5.1). For six clients, uFS and uFS_RR are allocated only four workers whereas uFS_max uses six.

Figure 5.1 compares the throughput of uFS and uFS_RR, scaled to uFS_max. For all workloads, uFS achieves between 88% and 100% of the

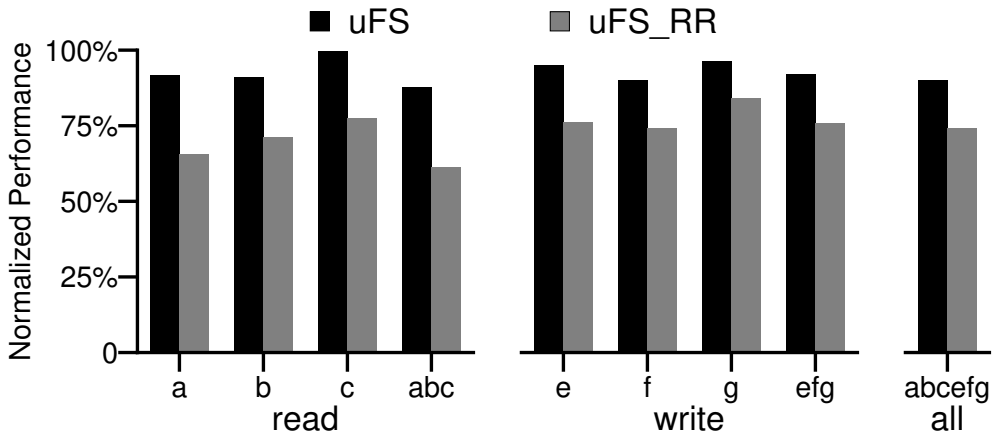


Figure 5.1: **Load Balancing Performance with Fewer Cores**. *The throughput of uFS and uFS_RR running on only 4 workers are each normalized to the throughput where each of the 6 clients has its own dedicated worker. Each experiment is repeated 5 times.*

uFS_max's throughput, but on 4 cores as instead of 6; uFS_RR achieves throughput only between 61% and 84%. Across workloads, the more significant the difference across operation costs (e.g., workloads read-abc and all-abcdef), the more important it is to quickly find a suitable placement of inodes. For all workloads, the median time for uFS to find a stable placement is low, between 25 and 75ms.

5.2.3 Core Allocation

We show that uFS dynamically adapt to a changing workload and adjust the number of cores by the core allocation microbenchmarks (Table 5.2). We again consider a maximum of 6 client threads. In these experiments, uFS determines a minimal number of cores that provides sufficiently low congestion and then balances inodes across them. We compare to uFS_max where each client has a dedicated core. Figure 5.2 shows that uFS delivers between 91% and 98% of the throughput of uFS_max with only 60% of the cores.

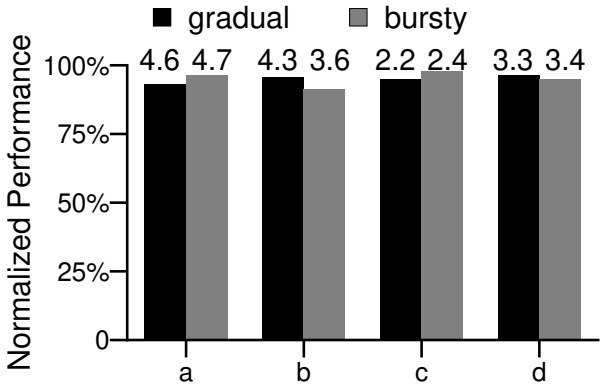


Figure 5.2: **Core Allocation Performance**. *Each bar shows the performance of uFS normalized to that of uFS_max, where each of the 6 clients has its own dedicated worker. The numbers on top of each bar are the average number of cores used by uFS.*

5.2.4 Dynamic Behavior under a Challenging Workload

We illustrate the adjustments of uFS over time with a challenging workload: 8 different I/O-intensive clients enter and exit the system and change their offered load (described in the caption of Figure 5.3). Figure 5.3 shows the CPU utilization on uFS_max given 8 dedicated cores. Even with 8 I/O-bound clients, 8 server cores leads to many wasted cycles: each core is below 50% utilization and some are below 20%.[2] Due to polling by the server thread, the OS scheduler believes each thread is using 100% of the CPU once the worker is active (i.e., has non-zero utilization in the graph); for utilization, we show the percentage of CPU cycles effectively performing uFS work. Using an average of 4.73 active (up to 8) cores, clients on uFS_max achieve 695Kops/s (not shown),

The two graphs in Figure 5.4 and Figure 5.5 show the throughput and CPU utilization of uFS; uFS is configured to start on 1 core but can grow to 8 cores. The CPU graph shows that one core handles the load of the first

[2]The periodic CPU spikes are due to long tail latencies when polling the device and occur on cores with more on-disk work.

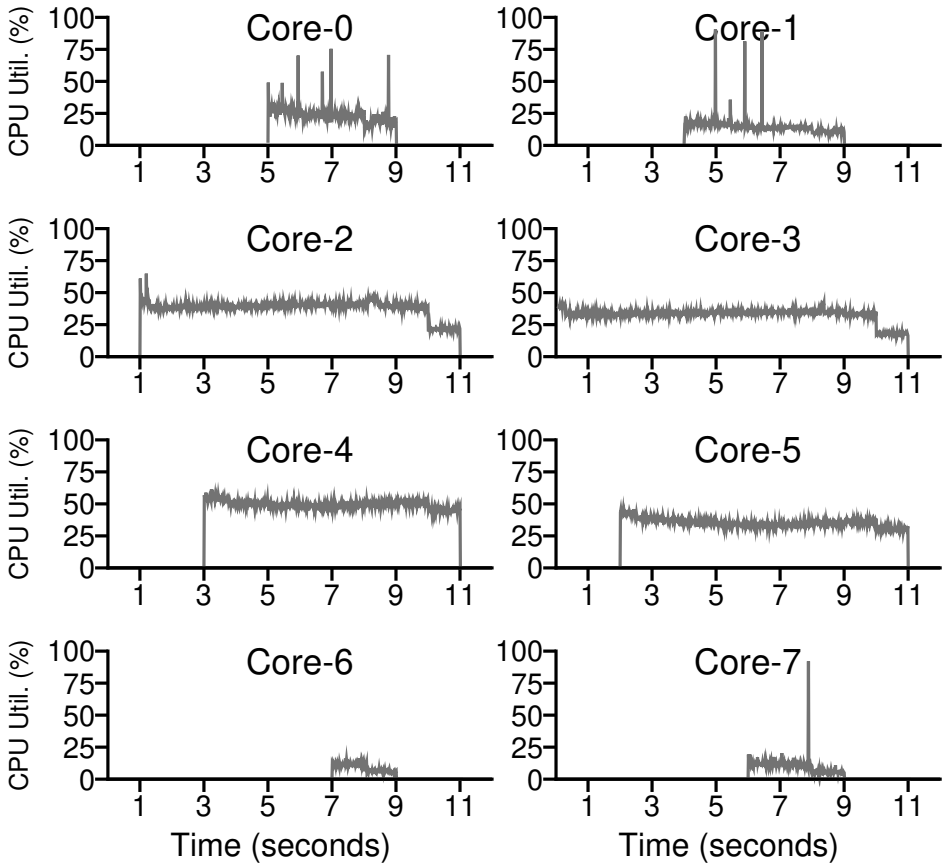


Figure 5.3: **Dynamic Behavior under 8 App Workloads (1): CPU Utilization where Core Usage is Not Restricted**. *The number of uServer cores are set at 8. Workloads: a-0: large on-disk read, a-1: small on-disk read, b-0: cold in-memory read, b-1: hot in-memory read, c-0: write+sync large, c-1: write+sync small, d-0: append, d-1: overwrite. Seconds 0-7: one app joins each second (b,c,a,d); sec 8: a,d increase thinktime; 9: a,d exit; 10: b,c increase thinktime; 11: b,c exit.*

two clients; as more clients join through time 8s, uFS activates new cores when congestion is high and rebalances inodes. At time 8, uFS observes that core 0 is congested, so adds another core and shifts work from both core 0 and 4 to core 5. When the workload decreases after time 9, uFS

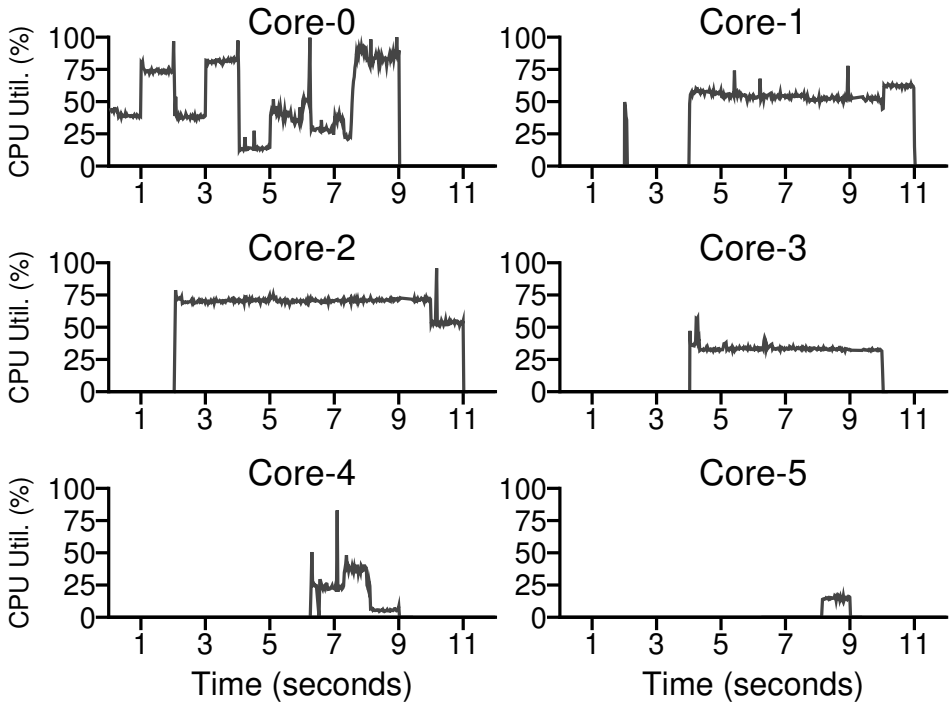


Figure 5.4: **Dynamic Behavior under 8 App Workloads (2): CPU Utilization with Load Management**. *Workloads are the same as in Figure 5.3.*

removes cores and rebalances inodes. Due to its rebalancing and core allocation policies, uFS achieves similar throughput with a smaller number of cores; uFS delivers 609Kops/sec on an average of 3.4 (up to 6) cores, or 88% of the throughput with 72% of the CPU resources.

5.2.5 LevelDB

We lastly show that uFS performs and scales well for LevelDB [68] with YCSB workloads. We measure LevelDB with two ways to load the database and six YCSB workloads [44]. Figure 5.6 shows that on all workloads, uFS has better base performance than ext4, and much better scalability. For I/O-intensive workloads, ext4 becomes a bottleneck due to its single-

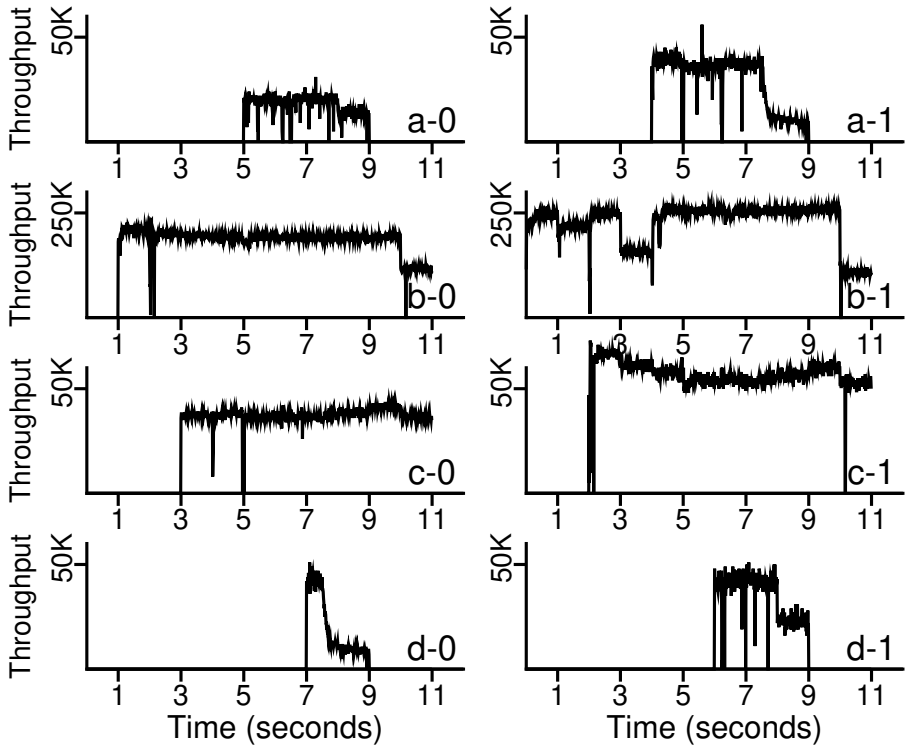


Figure 5.5: **Dynamic Behavior under 8 App Workloads (3): Application Throughput with Load Management**. *Workloads are the same as in Figure 5.3.*

threaded journaling, and adding more load to the system does not lead to an increase in ext4 throughput. uFS scales very well with increasing load. Due to the many private writes performed by LevelDB clients, the write cache is especially beneficial in uFS. With additional load, the uFS load manager determines that additional cores are beneficial and thus allocates an average of 6 server cores for the 10 clients across the eight workloads. Thus, the throughput of uFS scales well with the number of clients; for example, on YCSB-F with 10 clients, uFS delivers 1.88x the throughput of ext4.

In Figure 5.6, the system with uFS (uServer) uses between 4 and 8

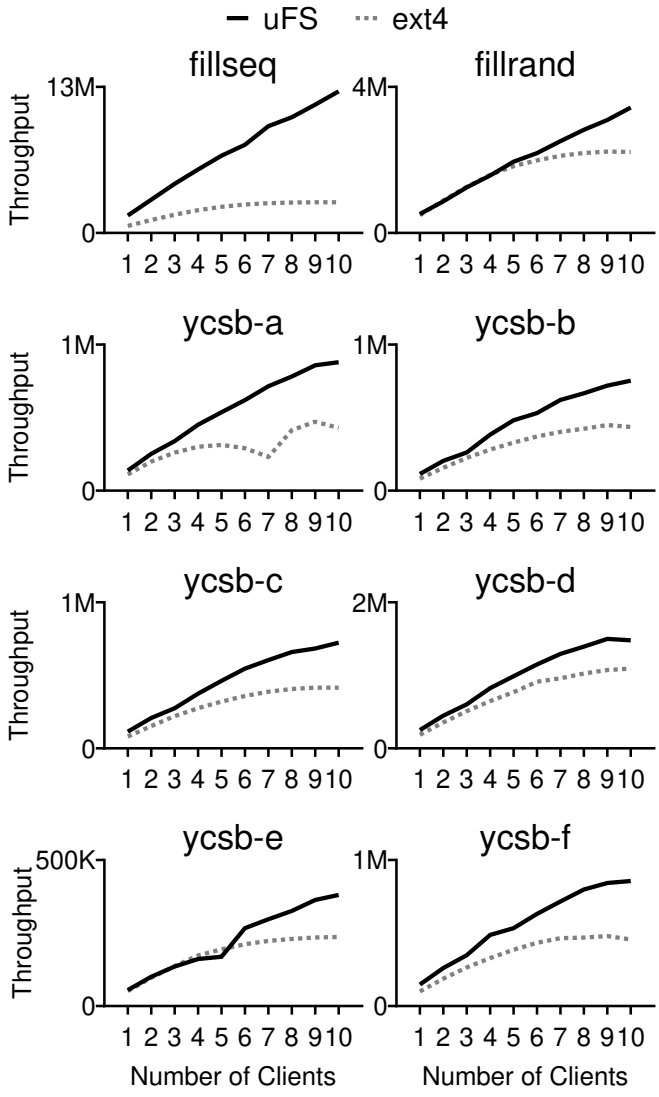


Figure 5.6: **Performance of LevelDB on YCSB.** *The number of clients is increased along the x-axis. The workloads are: Sequential Load, Random Load, A (write-heavy, w:50%, r:50%), B (read-heavy, w:5%, r:95%), C (read-only), D (read latest, w:5%, r:95%), E (range-heavy, w:5%, range:95%), and F (read-modify-write:50%, r:50%). We use 16B keys and 80B values with 10M entries, for 1GB per client. YCSB runs 100K operations. Across the 8 workloads uFS allocates 4, 7, 4, 8, 7, 6, 5, and 5 cores for 10 clients.*

more cores than that with ext4. We conduct another experiment (not shown) in which the number of LevelDB clients running on ext4 is increased to use the same number of cores as uFS. However, since the additional cores cannot be used effectively by ext4, more clients and cores do not result in any significant performance gain (a maximum of 7% improvement on ycsb-e and some performance degradation on other workloads).

5.3 Summary and Conclusions

In this chapter, we present the load management feature of uFS, which adapts the number of cores dedicated to the server and balances the load across cores. Because semi-microkernels scale independently of applications, they can benefit from additional cores and provide scalable performance. This is demonstrated by our LevelDB experiments, which compare uFS to the monolithic kernel filesystem, Linux ext4.

We have shown that uFS can dynamically control the number of cores through well-designed microbenchmarks and real application workloads, achieving CPU resource efficiency while offering high performance. Furthermore, uFS adapts well to workload changes, and such resource elasticity is essential due to the dynamic nature of modern data-intensive application workloads [28].

6

Beyond Full System Crash Recovery: Process Crash Recovery

We explore another major architectural advantage of semi-microkernel architecture: fault tolerance [32]. Microkernels have been complimented for better fault tolerance. However, such benefit is precisely fault containment [174] – a filesystem crash does not bring down the entire system.

The benefit of fault containment is significant, as other system services, the underlying OS, and unrelated applications naturally continue. The question then is: can applications of the filesystem seamlessly continue with a restarted filesystem server?

The answer to this question is an unfortunate no, yet not surprising for stateful filesystems – restarting the filesystem server is insufficient. Because the filesystem server buffers updates in memory for performance, there is a *state gap* between the application's perceived states and the on-disk states. However, a simply restarted filesystem server can only rely on the outcome of full-system crash (i.e., s-crash) recovery, ensuring that the on-disk states are consistent, but not addressing the state gap. To avoid confusion and data loss to applications, the state gap must be exactly recovered, i.e., seamless recovery is desired.

In this chapter, we realize seamless recovery in Nebula, equipping uFS with a range of mechanisms to perform filesystem process recovery. Nebula answers essential questions: How do microkernel filesystems allow

seamless recovery such that applications can continue without noticing that the server has failed? What additional opportunities does the naturally continued host OS provide? And how can such enhancements be incorporated into uFS without sacrificing its performance?

We introduce *exit activation*, a novel mechanism for process crash recovery. Exit activation is code that runs when a process crashes, accessing the memory states before it is reclaimed by the OS. This allows the failed server's memory state to contribute to the recovery of state gap. Combined with other mechanisms for a clean restart and lightweight protection of in-memory states, exit activation enables Nebula to achieve fast and seamless recovery with negligible overhead in the common path.

The first challenge is a robust and efficient restart mechanism as the foundation. One principle of Nebula is a *clean restart*: to discard problematic states from the failed server and start a fresh filesystem process for future request serving. The restart mechanism first ensures that the exit of the filesysem server is timely monitored and the restart is performed as soon as possible. The restart mechanism also re-establishes the connection between the fresh filesystem server and the applications, as well as the connection between the filesystem server and the devices.

The second challenge is the robustness to memory corruption. How can we trust the memory of a just crashed process? We follow the principle of *limited trust of memory*. The idea is that, exit activation only accesses a piece of memory that is known to be safe during a p-crash recovery. Techniques such as checksums and replication are employed to protect the memory and data structures positioned in the trusted memory.

The final and important challenge is to recover the state gap in the fresh filesystem. We introduce an exit activation data structure: p-log, which is the only memory region that is trusted by exit activation, that captures the state gap by recording the sources (i.e., operations) and relevant information. Our principle is *no force flush* to avoid the performance

slowdown in the common-path. The intuition is that the state gap, depending on when and what to flush, should be decided by the original filesystem, representing the design rationales for durability correctness and performance optimization policies. The main issue with the p-log is that when an operation's effect is made durable, the effect should be removed from the state gap. However, the effects of operations are likely to be made durable in a non-sequential-order. Our finding is that replaying the p-log requires ignoring some of the operations, and sometimes, modifying one operation to another existing system call API (e.g., write to lseek), such that the original code base can be used.

The chapter is organized as follows. We first present the crash model of monolithic and microkernel filesystems, discussing the implications and opportunities (§6.1). We also discuss the benefits and necessities of a separate p-crash recovery machinery (§6.1.3 and §6.1.4). We then present the design of Nebula (§6.3), including the fault model, the goals, and the challenges. Next, we present the core data structure for exit activation, p-log, and how it addresses the state gap problem via AIM algorithm (§6.3.5). Fianlly, we demonstrate that Nebula meets its goals thorough a systematic evaluation (§6.6), including extensive fault injection (over 30,000+ cases) under complex application workloads, and performance analysis of the normal execution and the recovery. We also perform a comprehensive study of application's reaction to the state gap problem (§6.6.2).

6.1 Crash Model

We present the crash models of filesystem failures in monolithic kernels (full-system crash, s-crash) and microkernels (process crash, p-crash), discussing the consequences and implications, and noting the opportunities and challenges arising from the p-crash model. We describe the

traditional s-crash recovery for monolithic kernel filesystems and why it falls short in fulfilling the opportunities provided by p-crashes. We propose having a separate p-crash recovery mechanism in addition to s-crash recovery. We also discuss alternatives for performing transparent p-crash recovery.

6.1.1 Monolithic Kernel Filesystem Failure ⇒ Full-system Crash

A filesystem may encounter an error that causes it to fail, due to a multitude of reasons like software bugs [31, 81, 100, 130] and hardware errors (e.g., CPU or memory) [17, 55, 79, 106, 153, 175, 183, 204, 214]. In a monolithic kernel, a filesystem failure typically leads to a full-system crash (or *s-crash* for short). In other words, the crash model of a monolithic kernel filesystem failure is that when the filesystem crashes, the entire system crashes, including the OS kernel, resulting in lossing all the states in the volatile memory that has not been persisted to on-disk states.

One implication is that a filesystem failure shares the same crash model as other environmental (and perhaps more disruptive) causes of a full-system crash, such as a power failure or a hard machine reset. Therefore, s-crash recovery that treats a filesystem crash the same as a power failure is the de facto focus when discussing filesystem crash recovery in monolithic kernels.

All modern monolithic 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* [199, 200]; if a later s-crash occurs, upon system restart, the filesystem utilizes the s-log to recover the filesystem to a consistent state. Of course, other techniques exist [125, 139, 173], 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 the applications running on the system also crash. Traditional s-crash recovery only ensures that the filesystem is returned to a consistent state because it fundamentally assumes that applications lose their perceived progress with the filesystem. Because filesystems buffer updates in memory for performance [147], at any given instant in time, many data and metadata updates have not been persisted; as such, most recent updates are lost upon a s-crash.

Traditional s-crash recovery can also be slow. Approaches based on full-disk scans (such as fsck [139]) are prohibitively slow [132], as they must scan all filesystem metadata to find and fix inconsistencies. More modern approaches, such as journaling [162, 199], are better, with performance proportional to the size of the log (rather than the entire filesystem disk space).

6.1.2 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 of such a p-crash model is better fault containment, a well-recognized reliability benefit of microkernel design [32]. Consider the number of other OS services (e.g., networking, memory management, scheduling, etc.) and applications that do not interact with the filesystems; all of these continue naturally.

The second implication is that applications interacting with the filesystem have the opportunity to continue, but not without challenges. The main challenge in allowing the applications to continue seamlessly 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 concerns 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 issues ten *write*s to an empty file and close the file descriptor; then the filesystem server crashes. Assume we restart the filesystem server, and the application opens the file again and attempts to read it. Because the on-disk state of the file is empty, the filesystem server reports that the file is empty to the application. However, the application's perceived state is that the file contains ten writes – a confusing outcome and severe data loss. The state gap in this case contains the filesystem metadata and data changes due to the ten writes.

The state gap can be much more complex than this simple example, as discussed later (§6.3.5 and §6.6.3.2). The fundamental issue is that the in-memory states of the filesystem server are updated (and buffered) by a rich set of filesystem APIs in various ways. The state gap can also be altered by partial flushings of the server's states to the disk, as a result of the out-of-order durability commonly employed by modern filesystems [136, 158]. After a p-crash, opened file descriptors are lost, and other data writes or metadata updates (e.g., creating, unlinking, and renaming files) will also be unexpectedly lost. As we will show in Section 6.6.3.2, even durability-aware libraries like SQLite and LevelDB do not always continue transparently when the fileserver p-crashes and they may lose committed data.

6.1.3 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 enough p-crash recovery, applications might hardly notice that the filesystem has crashed. Moreover, recovering the state gap is essential to avoid confusion for applications.

The benefits of p-crash recovery are manifold. First, applications can continue running with better correctness guarantees than those provided by s-crash recovery. The opportunity offered by a microkernel is that filesystem applications are not forced to lose their progress. For example, there is no need for user applications to restart their jobs or perform manual recoveries. S-crash recovery can add extra complications for applications because their durability protocols are often error-prone, as described by Pillai et al. [158]. Although s-crash recovery ensures the consistency of filesystem data, applications still find it challenging to determine how much of their work has been completed. A p-crash recovery mechanism can transparently avoid such complications, as the application's progress can be preserved without excessive cost, again because it is not a power failure; preservation in volatile memory is sufficient.

Second, filesystem failures and p-crashes occur frequently, whereas power failures are relatively rare. Modern cloud environments are designed to handle power failures (often with correlated failures [63]), but with a great deal of complexity. For example, another data center might be used for failover, with sophisticated protocols [152]. Delegating p-crash to a global failure scenario is not only unnecessary but also increases the likelihood of severe issues found in failover invoked by an s-crash, like metastable failures [82]. Separating p-crash recovery from s-crash recovery thus reduces such complications and potential issues.

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

nel filesystem server is challenging and less robust when reusing the same kernel [185] or slow if using a new kernel instance [54].

6.1.4 Alternatives for P-Crash Recovery

One concern in designing dedicated p-crash recovery mechanisms is the complexity of handling the state gap. Can we somehow handle p-crashes with a simply restarted filesystem server?

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 fails to capitalize on the opportunities and benefits provided by p-crash recovery.

Another alternative, which does not deal with the state gap, 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 the failure transparency to applications, it does not really perform p-crash recovery. Unfortunately, forcing updates to disk results in poor common-case performance. As we show in Section 6.6.5, the impact is dramatic, up to 6x slower than buffering updates in memory.

Furthermore, one might also consider a client-side recovery mechanism, where the client is in charge of retrying its own job. However, such an approach requires the application to track the state gap, more complex than done in the server-side because a notification is needed from the server to the client when the state gap changed by the internal filesystem behaviors (e.g., background sync). And the retry during recovery is a fundamental difficult problem of distributed coordination [26] in the presence of multiple applications.

6.2 Exit Activation

We make further observations on what is available in the presence of p-crashes and note an opportunity for p-crash recovery mechanisms. In addition to on-disk states, the in-memory states of the failed filesystem server remain present in volatile memory upon exit, and such exit events are readily recognized by the continuing host OS.

As such, one compelling approach is to enable filesystem recovery to access server memory state *before* it is reclaimed by the OS. Doing so gives access to the most recent updates, and thus provides a path to completely recover the filesystem without losing updates. The intuitions are: (1) the instantly buffered states represent the application-perceived states and the effects of the inflight operations; (2) the state gap between application-perceived states and on-disk states can be derived from the memory of the filesystem server.

An *exit activation* is code that runs when a process crashes. At this moment in time, all (possibly corrupted) memory state is visible and can be accessed by the exit handler. The exit activation can make persistent any necessary information about pending updates, perform other relevant actions, and restart the filesystem process. The restarted process can then fully recover, re-establish connections with active clients, and transparently resume serving requests.

Exit activation allows both the memory state (and persisted state) to contribute to recovery, ensuring that no updates are lost. Exit activation is the powerful mechanism for p-crash recovery to correctly reconstruct the state gap in the newly started process. This approach also allows more recovery work to be done after the crash occurs, thereby incurring far less overhead in the common path compared to approaches that simplify state gaps by forcing updates to disk.

Of course, p-crash recovery via exit activations is not without challenges. How can the recovery process trust memory contents of a recently

failed server? How to ensure the correctness of reconstructed state gap in the newly started process via these memory contents? And finally, how can p-crash recovery be performed quickly, so as to minimize disruption to running applications?

6.3 Nebula Design

We discuss the design of Nebula, another filesystem semi-microkernel based on uFS, that provides transparent p-crash recovery via a range of p-crash recovery mechanisms including exit activations. We discuss the fault model, goals, and challenges. We then delve into a range of mechanisms that Nebula employs to address these challenges, especially the p-log data structure and how it effectively tackles the state gap via the core, AIM algorithm. Finally, we present the workflow of exit activation and p-crash recovery in Nebula.

6.3.1 Fault Model

6.3.1.1 Potential Faults and Errors

Filesystems experience a range of errors caused by hardware and software faults. The underlying hardware or software cause of an error is a *fault* [16]; a fault is active (e.g., executing buggy code or using a flipped bit) when it causes an error where the system's state deviates from the correct state.

Many examples of faults causing filesystem errors exist. For instance, DRAM and SRAM transient and permanent faults not caught by hardware (e.g., ECC) lead to data corruptions that are visible at run-time [106, 153, 175, 183, 214]. CPU core faults lead to computational errors and silent data corruption [17, 55, 79, 204]. Filesystem code contains semantic and concurrency bugs causing corruptions and crashes [31, 81, 100, 126, 130]

taking years to fix [42, 93]. Even enterprise systems encounter errors (scribbles and wild writes) in production [106].

6.3.1.2 Targeted Faults to Tolerate

The design of Nebula aims to recover from *transient* faults that may include *memory corruption*. Nebula focuses on transient faults, so that a restart/recovery will not re-trigger the same fault repeatedly; additional techniques will be required to handle permanent faults [18, 98]. Some faults simply cause a p-crash to occur without any memory corruption; Nebula should also handle this simpler class of faults.

It is only when a fault manifests as an error (i.e., is detected) that any recovery can be performed. Our fault model is more realistic than prior works that assume fail-stop conditions, as adding checks or assertions immediately after a fault is not practical (e.g., in cases of wild writes, hardware corruption, and incorrect values due to software bugs). For example, if a value is silently corrupted in memory, the process may continue until the value is accessed again, at which point the corruption may propagate to other data structures.

We believe that corrupted semantic states pose significant problems for filesystem correctness. Despite the virtual impossibility of achieving fail-stop behavior for every fault, Nebula strives to fail as early as possible when essential states (i.e., semantic states) are corrupted, before incorrect data is returned to a user or persisted to disk [65, 106]. Nebula validates the integrity of in-memory semantic states with checksums upon every access.

6.3.2 Goals for P-Crash Recovery

Seamless p-crash recovery: In Nebula, p-crash recovery should be lossless; applications should not perceive that Nebula has p-crashed and restarted,

should not lose updates, and should not receive any confusing or incorrect results (e.g., a file should not revert to an older form).

Fast p-crash recovery: In Nebula, p-crash recovery should be fast. During p-crash recovery, running applications are awaiting results; therefore, the system should p-crash and restart quickly to minimize disruption to applications.

Fast common-case performance: Providing fast and seamless fault recovery is important, but faults are rare; as such, any enhancements to improve fault recovery should not degrade common-case performance. The primary implication is that any additional work needed to prepare for recovery should be minimal.

Few filesystem codebase modifications: Filesystems are complex pieces of code containing many performance optimizations (e.g., avoiding synchronous operations and processing requests concurrently). Recovery should not require onerous changes to the original codebase.

6.3.3 Challenges

Realizing exit activation in Nebula and achieving our goals require addressing the following challenges.

6.3.3.1 Robust and efficient restart mechanism

A robust restart mechanism is the foundation for any kind of p-crash recovery. Our principle is *clean restart* [73] – to ensure that problematic states in the failed process are discarded. Therefore, a *fresh filesystem process* is utilized to service future application requests. We refer to the failed process as the *main filesystem process*.

The first issue for restart is ensuring that the exit of the main filesystem process is timely monitored by the host OS, which also orchestrates the execution of the exit activation code and coordinates the transition

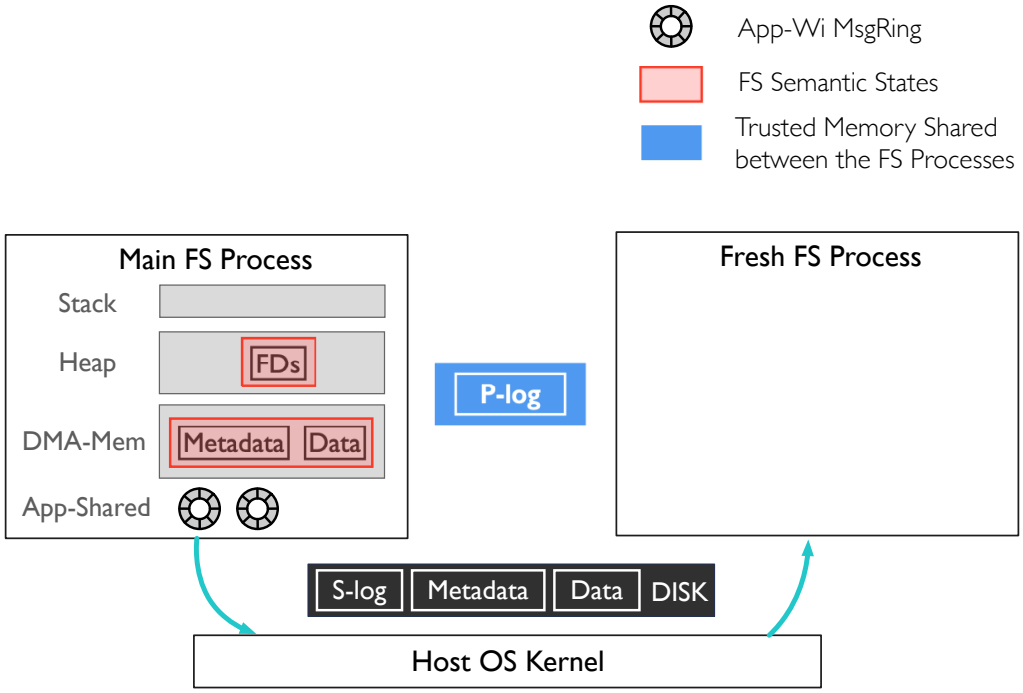


Figure 6.1: **Illustration of Exit Activation in Nebula.** *The difference between in-memory semantic states (highlighted by red bouned boxes) and the on-disk states after completing a sequence of operations is the* ***state gap****. After an error is detected in main process, exit activation started by the host OS kernel (left green arrow); exit activation also notifies the failover process (right green arrow) to start and consume the p-log.*

between the two processes. Second, the restart mechanism, including the initialization of the fresh process, must be fast to accelerate recovery performance. Third, the restart ensures that resources shared between the main filesystem process and applications (i.e., IPC connections), as well as those the main filesystem process shares with the host OS (i.e., permissions and driver connections), are properly established by the fresh filesystem process.

6.3.3.2 Memory corruption in failed process

Inspecting the memory of the failed process is compelling for p-crash recovery because the more states extracted after the exit, the less work is needed to preserve the states in the common no-failure path. For example, if we can undo the changes of all the in-flight operations and directly duplicate all the semantic states to the fresh process, little extra work is needed in the common path, under the assumption that all faults are fail-stop. However, the fail-stop assumption is virtually impossible, and Nebula follows the principle of *limited trust in memory*, favoring robustness.

Therefore, extra care must be taken to avoid reusing any corrupted semantic states or the failure of exit activation due to corruption (e.g., a corrupted pointer). The key is to ensure that the trusted memory region is well-defined and safe to access.

The issue is that the filesystem's in-memory semantic states (including file descriptors, metadata, and data) are scattered across the address space and mingled with other data structures (e.g., hashmaps, lists, vectors), as shown in Figure 6.1. How can we have strong confidence in the memory we trust? How can we make the trusted memory robust to memory corruption? And how can such trust be assured with less overhead in the common path? Furthermore, can we strives to fail as early as possible when these more essential states (i.e., semantic states) are corrupted to get more benefit from exit activation? (§6.3.4)

6.3.3.3 Capture state gap without affecting common-path performance

As described earlier (§6.1.2), the state gap between the application's perceived states (i.e., buffered in the crashed server's memory) and the on-disk states – arising from the p-crash model – is a main impediment for applications to continue.

The main issue is accurately capturing the state gap, which is made challenging due to the out-of-order durability that flushes part of the buffered states to disk. These states may not necessarily be flushed in the order they were updated in memory. Capturing such a state gap is thus non-trivial and requires interposition in the common path with negligible overhead. Our principle is *no force flush* – to avoid forcing buffered updates to be flushed to disk, which would degrade performance.

As mentioned above, one straightforward approach satisfying the performance requirement is to undo the changes of all the in-flight operations and directly reuses all the semantic states after a failure [114]. However, directly reusing the buffered states is vulnerable to corruption and cannot satisfy limited trust of memory.

6.3.4 P-crash Recovery Mechanisms

We now describe the mechanisms that Nebula uses to address the challenges: kernel-coordinated speculative restart, lightweight protection of in-memory semantic states, and exit activation data structure to capture the state gap. Figure 6.1 illustrates exit activation in Nebula.

6.3.4.1 Kernel-coordinated Speculative Restart

We introduce a technique, *kernel-coordinated speculative restart*, to enable efficient and robust restarts. The idea is that, when the main filesystem process first starts, a fresh process is also started, but put to passive mode, awaiting for notification from the OS. Upon a failure in the main process, its exit event is monitored by the OS, which then notifies the fresh process.

The IPC connections between the main process and applications are authenticated shared memory known by the OS, which is passed into the fresh process. Importantly, the fresh process performs costly initialization

of device access (which takes up to seconds) in parallel with the main process before blocked on the OS notification, thereby allowing fast recovery.

6.3.4.2 Lightweight protection of in-memory semantic states

To ensure that the filesystem process fail fast when semantic states are corrupted, Nebula integrates a lightweight mechanisms by adding checksums to all in-memory semantic structures. These are highlighted in the red box in Figure 6.1 and include file descriptors on the heap, as well as metadata and data in DMA-able memory. Given the large address space, selectively protecting these components is more cost-effective in preventing the filesystem from producing incorrect results for applications or persisting corrupted data to disk.

Our checksum is updated on each write access and verified during each access (both read and write). As we will show in the evaluation, this protection is lightweight, adding only 0.2% to memory overhead and 2.85% to performance overhead in the common path. The protection (by checksums) is used solely to enhance detection and avoid errors propagating to the applications or disk, whose effects are very challenging to recover from.

Note that these in-memory semantic states are not trusted by the exit activation, and the only trusted memory region is the P-log, as we describe next.

6.3.4.3 Exit activation data structure: capture the state gap

Our final and crucial mechanism is the exit activation data structure, the *P-Crash Log* (*p-log*), which captures the state gap and enables lossless recovery. Instead of reusing the results of executed operations (i.e., buffered states) that is vulnerable to corruption , the p-log records the source of the state gap. P-log enables a clean design that only one well-designed data

structure is modified in the common-path and accessed after the failure for recovery.

The p-log resides in a dedicated memory region that is shared between the main process (with write access) and the failed process (read-only access). It records the source of the state gap – the system calls and arguments that, when replayed properly, can reconstruct the state gap. To protect against corruption, the p-log is safeguarded by checksums and replication. Furthermore, because it is in-memory and does not require costly flush operations, the p-log does not alter the common path, incurring negligible overhead.

As we will discuss next, capturing the state gap between the in-memory semantic states and the on-disk states is challenging, given that only some of the p-log's effects may have been made durable, and in a non-sequential order.

6.3.5 Exit Activation Data Structure: P-log

We now delve into the design of the p-log, which is core to Nebula for capturing the state gap. We describe the data structure, including when and what information is recorded in the p-log. We introduce the AIM (Act-Ignore-Modify) transformation, the core algorithm designed to address the challenges posed by the state gap. Naively replaying the operations is insufficient because the state gap does not directly correspond to a subsequence of operations completed in history.

6.3.5.1 Basic Data Structure

To enable p-crash recovery, Nebula records information about recent file system activity, representing operations that the server has executed since the last persisted state. The relevant information is stored in the *p-crash log* (*p-log*), a new in-memory structure. The goal for p-log is to accurately

capture the source of state gap, which can be replayed by the original filesystem code to reconstruct the most up-to-date in-memory states in fresh process.

The p-log is organized as a circular buffer. Each p-log entry contains an event source (system call, arguments, and return value) and log-entry descriptor that records essential information such as the process identifier (pid), file descriptor (fd), relevant inode numbers, and a timestamp based on completion time to form a globally-ordered sequence. Each core has its private p-log, only written by the given worker thread. To be more robust to memory corruption, the p-log is designed to be pointer-less, protected by checksums and can leverage replication.

The log-entry descriptor also tracks the status of the corresponding changed file descriptor and changed inodes, referred to as *updated_targets*. Upon a subsequent close or sync/fsync, the status of each target in this log-entry descriptor is also updated (i.e., the changes are cleared). For example, when an inode is made durable (after a sync/fsync), we find all the log entries related to this inode number and update the status in the log-entry descriptor, indicating that the change imposed by this log entry has been made durable and thus does not need to be reconstructed.

6.3.5.2 P-log Transformation

The primary issue with recording operations in the p-log is that naively replaying all operations can be problematic for two reasons. First, only operations contributing to changes of in-memory semantic states need to take actions. For example, consider a sequence of `open(f)`, `read(f)`, and `close(f)`. If the filesystem crashes after the `read(f)`, the state gap contains a file descriptor, and the `open` needs to be replayed. However, if the filesystem crashes after the `close`, the entire sequence does not contribute to the state gap (i.e., the file descriptor is destroyed), and the three operations should be ignored. Furthermore, operations that contribute to

the dirty updates of semantic states like inodes, directories, and bitmaps should be replayed, unless their effects have been made durable.

Second, subsequent operations may alter the preconditions of previous operations. For example, consider a sequence of `open(D/f)`, `write(D/f)`, `close(D/f)`, `rename(D/f, D/f1)`, and `sync(D)`. If the filesystem crashes after `sync(D)`, the precondition of `open(D/f)` (i.e., the pathname D/f corresponds to the inode) has changed because the effect of the `rename` is persisted into s-log; thus, replaying `open(D/f)` does not work because the path is invalid. However, if the filesystem crashes before `sync(D)`, replaying `open(D/f)` is needed.

To address these issues, Nebula 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 directly to reconstruct the most up-to-date in-memory semantic states (as perceived by the applications) in the fresh process. Act indicates the operation will directly be replayed; Ignore indicates it can be ignored; and, Modify means some state change must occur but not by replaying the original operation.

We now describe the AIM algorithm in detail and how it is used for garbage collection (in the common path) and replay (during recovery).
AIM-based Transformation: Principally, the first question to answer for each operation in the p-log is whether it can be ignored (i.e., Ignore). Each operation potentially changes two sets of states: states of file descriptors and states of inodes. For instance, a read changes a file descriptor's offset, while a write changes both a file descriptor's offset and an inode's size. An operation can change the states of up to three inodes and one file descriptor at most.

The condition for safely ignoring an operation is that any associated file descriptor must be closed (if applicable), and all inode-related state

changes must have been made durable. For example, an append operation results in state changes to a file inode and to a file descriptor. A create changes the states of two inodes (i.e., a directory and a file). A rename can change the states of two or three inodes (i.e., two directories and one file when the destination is deleted). A close operation can be ignored if the corresponding file descriptor is no longer needed – the related dirty inode states have been synced. Moreover, an open operation can only be ignored if the file descriptor is closed and dirty writes to the inode have been persisted.

For example, with the sequence of `open`, `read`, and `close`, If the crash occurs after `close`, all the operations should be ignored. Otherwise, the prefix sequence should be replayed. Similarly, with the sequence of `open(f)`, `write(f)`, `close(f)`, and `sync(f)`, if the crash occurs after `sync(f)`, all the operations should be ignored.

If the operation cannot be ignored, the next question is whether the operation needs to be replayed as it is (i.e., Act)? The condition for replaying the operation in its original form is that all of the changed inodes's states have not been synced, and if an file descriptor was changed, either the file descriptor is still open or the file descriptor is closed but the inode is not synced. For example, with the sequence of `open(f)`, `write(f)`, and `close(f)`, if the filesystem crashes after every operation, the prefix sequence should be acted.

Finally, if neither ignoring nor acting upon the operation is appropriate, such log entry indicates that part of the state changes exerted by the operation needs to be reconstructed, but the original operation should not be replayed. We find that the partial effects can be generated by a *Modified* operation form (e.g., other operation type or modified argument), such that the redo of modified operation can be done mostly by the original code base.

Consider the example sequence mentioned above, `open(D/f)`, `write(D/f)`,

`close(D/f)`, `rename(D/f,D/f1)`, and `sync(D)`. If the filesystem crash occurs after `sync(D)`, the `open(D/f)` is modified to `open(D/f1)`, and the `rename` is ignored because the changed inode (only D) has been synced. In this case, the rename only changes the directory, but not changing states of the file. Another example sequence is `open(f)`, `write(f)`, and `sync(f)`, if the filesystem crash occurs after `sync(f)`, the `open(f)` is acted, and the `write(f)` is modified to `lseek`. The modification to lseek is correct because the inode's states change have been synched.

Our modifications include: `read` to `lseek`, `write` to `lseek`, `read` to `pread`, `write` to `pwrite`, `create` to `open`, and change pathname.

Other Considerations: One might wonder if including file descriptors and inodes in the updated_targets sufficiently captures changes to semantic states. The main semantic states not directly recorded in the p-log include the inode bitmap and data bitmap. However, changes to inode bitmaps are logically reflected in the inode states. For example, for a create operation, the newly created inode's number is recorded in the p-log. Data bitmaps, used to track data block allocations, do not need to be recorded in the p-log because the fresh process can safely make different decisions regarding data block allocations.

Another concern is the dependencies between operations [64, 147]. However, the p-log does not require extra consideration for such issues with our method of tracking changes to inodes during a given operation. Dependencies between operations are tracked and handled by the durability protocol of the underlying filesystem. For instance, consider the dependencies due to a coupled imap. When `create(D0/f0)` and `create(D1/f1)` depend on the same imap, and `sync` is called on D0, both the file and directory inodes for D0 and D1 should be synced by the original filesystem within one durable transaction to maintain the consistency of the on-disk states (i.e., in s-log). When `sync(D0)` is completed, the log entries for the two create operations are updated to reflect that the changes to all four

inodes have been synced. With these log-entries and AIM, the replay will not act the create opearations, and if the file descriptors have been closed, the replay will ignore the operations.

P-log Garbage Collection: Garbage collection (GC) of p-log entries is important to Nebula, because without active GC, the log will grow to occupy too much memory. The GC of p-log is based on AIM, using the Ignore subset: when operation is safe to ignore, the log-entry can be garbage collected.

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.

Nebula includes a few new internal APIs. One issue is that inode number should be the same when a create/mkdir is replayed, because the inode number can be used by applications, as specified in the POSIX standard. Similar constraints apply to file descriptors from open. Thus new internal APIs are added to specify the inode number and file descriptor.

6.3.6 Workflow of Nebula with Exit Activation

A successful exit activation involves the following steps:

(1). During normal execution, the main filesystem process proceeds and services applications via the IPC connections (i.e., the shared-memory message ring-buffer). What differs from uFS is that upon completion of each operation, Nebula records the operation and its relevant information (e.g., updated_targets) into the p-log. Upon `close`, the previous log entries related to the same file descriptor are updated in their entry descriptor. Upon `sync` (or an internal background sync), the previous log entries related to the same inode are updated in their entry descriptor to indicate durability.

(2). When an error is triggered in the main filesystem process (e.g., hardware or software exceptions), the OS obtains the control and invokes

a procedure that rescue the related data pages (pointed by the p-log) to the host OS. And then, the main process resource is reclaimed. As a result, failed process's private in-memory states are discarded and the pinned memory region is also reset. The exceptions are the p-log and the shared-memory message ring-buffer, which are shared memory with the fresh process and applications.

(3). The OS notifies the fresh filesystem process to complete the exit activation. The fresh filesystem re-initializes the pinned memory region, scans the p-log, performs the AIM transformation, replys the resulting actions, and re-attachs the IPC connections with the application. During the replay, the on-disk states, the rescued data pages (stored in tmpfs), and the AIM-transformed operations are used. Next, the p-log memory region is released.

(4). The fresh filesystem process is now ready to serve the applications. It becomes a new main filesystem process, and another fresh filesystem process is also initialized.

6.4 Implementation

We discuss the implementation of Nebula. We begin with the open-source repository of uFS [122] and add p-crash detection and recovery mechanisms. The uFS code base consisted of roughly 35K lines of code; we add approximately 4K LoC for Nebula.

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 server thread in Nebula has its own private p-log and thus need not

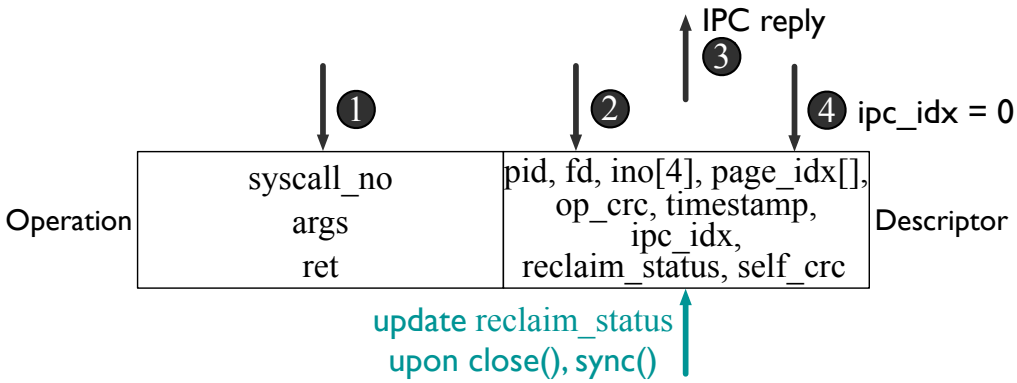


Figure 6.2: **A p-log entry**. The circled numbers show the order of updates to the entry. The black arrow indicates the initial logging of one operation. The green arrow indicates updating the reclaim_status field upon close and sync. Each box represents one cacheline.

worry about concurrent updates [23]. Figure 6.2 shows the information in each entry. Entries are added to the p-log after an operation completes, at which point Nebula allocates an entry and copies the message 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, as they do not need to be replayed.

Exactly-Once Semantics: To ensure correct recovery, each request must update system state exactly once [170]. After restart, Nebula will replay completed (but not persisted) operations from the p-log and the remaining in-flight operations from the message ring.

Nebula updates the p-log before replying to ensure that it does not lose an operation. If Nebula 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, Nebula follows a careful update protocol,

as shown in Figure 6.2. 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 [1]; 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 [138]. In step 3, Nebula sets the status bit in the message ring, and finally, in step 4, Nebula clears the p-log entry's ipc_idx (and atomically updates the self_crc to match, as it is within a cacheline).

During recovery, when Nebula 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); if the message is still marked in progress, the logged operation is discarded. If the p-crash occurs after the message status is marked complete, the logged operation will be used and the on-ring message will not be re-executed. Compiler barriers are inserted where needed to avoid write reordering.

Garbage Collection: To safely remove an entry from the log, Nebula must carefully track the status of each entry. This information is embedded in the descriptor's reclaim_status bitmap and inode array. This bitmap tracks if inodes (and corresponding data/metadata) related to this operation have been persisted; the relevant inode numbers are kept in the ino array. It also tracks whether the file descriptor (if there is one) is yet closed. When all inodes are persisted and the file descriptor is closed, the entry can be reclaimed.

To prevent a scenario where a large number of writes on a file descriptor without closing it consumes too much memory, after an inode sync, the entries containing both writes and offset changes are compacted into a single entry to only preserve the offset change. We discuss the memory limit for the p-log and garbage collection later (§6.6.5.2).

P-Log Replication: Two copies of the p-log are maintained to enable recovery given a single p-log corruption. For each update, Nebula 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 the primary and the replica are corrupted, recovery aborts.

Checksums: 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 [89].

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*. 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 while executing exit activation. Another assumption is that saving data pages to a known location 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.

6.5 Qualitative Comparison

In this section, we qualitatively compare Nebula with related systems (as shown in Table 6.1), including: Membrane [185], a restarting framework for kernel filesystems; Rio [37], which directly reuses the kernel's buffer cache after a s-crash; Otherworld [54], which microreboots the OS kernel without affecting applications; and TxIPC [114], the state-of-the-art recovery support for microkernels.

We later perform a quantitative comparison with the Membrane approach to address the state gap problem in §6.6, examining the cost incurred in the common case that violates the no force flush principle. We do not compare with approaches that address the state gap by fundamentally assuming that the filesystem metadata is not corrupted, such as in Rio, Otherworld, and TxIPC. Membrane is also vulnerable to memory corruption, but it is vulnerable to corruption in the rest of the monolithic kernel space, not because of reusing the filesystem states, which is less of a concern with a clean restart like ours. It is relatively less sensible to quantitatively compare the restarting mechanisms due to differences in kernel architecture.

We also introduce two baseline systems for comparison: uFS^ and uFS-Sync^. Neither system systematically performs p-crash recovery nor tackles state gap, as described in §6.1.4. uFS^ augments the original open-source crash-consistent version of uFS with only restarting (i.e, reconnect with applications and devices). uFS-Sync^ is a variant of uFS^ that goes one step further for correctness; it provides failure transparency to applications by relying on s-crash recovery. uFS-Sync^ eliminates the state gap by synchronously writing all updates to the disk to ensure that all operations are durable before making results visible to clients.

	Nebula	uFS^	uFS-Sync^	Mem-brane [185]	Rio [37]	Other-world [54]	TxIPC [114]
Robust Restart	■	■	■	□	■	■	□
Robust to Memory Corruption	■	■	■	□	□	□	□
Handling State Gap	■	▫	▫	▪	■	■	■
Negligible Performance Impact	■	■	▫	▪	■	■	■
Practical Detection Assumption	■	■	▪	▪	▫	▫	▪

Table 6.1: **Qualitative Comparison with Other Systems (a).** *Black indicates the best and white the worst.*

	Nebula	uFS^	uFS-Sync^	Mem-brane [185]	Rio [37]	Other-world [54]	TxIPC [114]
Robust Restart	new process	new process	new process	unwind, remount	new os kernel	new os kernel	unwind
Robust Memory Corruption	reuse p-log	no reuse	no reuse	reuse kernel space	reuse metadata	reuse kernel space	reuse process address space
Handling State Gap	p-log	not handle	full sync upon each	full sync upon many	reuse in-memory states	reuse in-memory states	reuse in-memory states
Negligible Performance Impact	p-log, checksum	no extra work	full sync upon each	full sync upon many	atomic metadata updates	no extra work	instruction undo-log
Practial Detection Assumption	on-disk states	on-disk states	on-disk states	on-disk check-points	in-memory states (fail-stop)	in-memory states (fail-stop)	in-memory states (gap $\leqslant$ one op)

Table 6.2: **Qualitative Comparison with Other Systems (b).** *Brief explanation: restart mechanism, memory vulnerable to corruption, methods to handle state gaps, work added to normal execution, and assumed correct states.*

The other four systems we compare with, instead, incorporate well-thought design and fair amount of work in the recovery path after the failure occurs. Their techniques can be used for exit activation, but unfortunately, cannot meet all our goals in Nebula.

The comparison is based on five crucial properties: robust restart, robust to memory corruption, handling state gap, negligible performance impact, and practical detection assumption. As shown in Table 6.1, Nebula excels across all five properties. We discuss each property in detail below. Table 6.2 provides a brief explanation of the comparison.

6.5.1 Robust Restart

The robustness of a restart reflects the likelihood that the restarting mechanism will succeed; successful recovery fundamentally depends on this robustness.

Systems that use a new address space – either through a new process (Nebula, uFS^, and uFS-Sync^) or a new OS kernel (i.e., Rio and Otherworld) – are less error-prone.

Systems that run code by the original threads (e.g., the same sets of stacks) for restart are considered less robust. TxIPC rolls back the effects of ongoing IPC, reusing the failed process, whose address space might contain erroneous states that impede the restart. Membrane unwinds the threads executing the kernel filesystem code, unmounts, and remounts. The restart is less robust because it is *soft*, only running the unwind and unmount procedures and freeing filesystem objects within the same monolithic kernel's address space, instead of performing a hard reset that completely drops the kernel space.

6.5.2 Robust to Memory Corruption

Robustness to memory corruption in the failed system depicts the health of the initial state for a successfully restarted system, reflecting how effectively the recovery cleans up problematic states from the failed system. Systems that reuse more unprotected memory from the failed system are less robust.

A hard reset (i.e., starting a new process or a kernel) has the benefit of discarding all memory from the failed system, as employed by Nebula, uFS^, and uFS-Sync^.

However, Rio and Otherworld, despite using a new OS kernel, are vulnerable to memory corruption because they intentionally reuse parts of the failed system's memory, like the buffer cache containing metadata in Rio. Otherworld copies the application's address space into the new kernel.

In contrast, systems that utilize a soft reset for restart are naturally susceptible to memory corruption. Membrane does not reuse the filesystem metadata, but it reuses the kernel's address space, which contains numerous states interacting with the kernel filesystem. As noted by the authors, "some filesystem bugs can still corrupt kernel state outside the filesystem, and recovery will not succeed" [185]. TxIPC suffers from the same issue becuase of reusing the failed process's address space, which might contain corrupted states before handling the current IPC.

Nebula indeed reuses p-log, but it is well-specified and well-protected, making it safe to reuse. Therefore, Nebula exhibits better tolerance for memory corruption.

6.5.3 Handling State Gap

Handling state gap is especially important for stateful systems like filesystems to ensure that the recovery does not lose any updates. If a system can only

handle less realistic state gap, it needs to induce extra behavior during normal execution (usually costly) to enforce the state gap to meet certain constraints.

Systems that directly reuse the in-memory states of the failed system (Rio, Otherworld, and TxIPC), despite more vulnerable to memory corruption, readily handle state gaps, as no states are dropped.

In contrast, systems that relies on a checkpoint (e.g., on-disk states) needs to consider the state gap between the last checkpoint and the failed system's in-memory states. uFS^ does not handle the state gap, leading to potential data loss. uFS-Sync^ avoids the state gap issue by flushing every update. Membrane is an improved version of uFS-Sync^, which simplifies the state gap. Membrane turns an fsync into a full sync of the entire buffer; it also requires an extra sync after directory-related operations (e.g., create, unlink, etc.) to ensure full transparency (e.g., same inode number).

Nebula goes to great lengths to accommodate the realistic state gap produced by the original filesystem under applications' workloads. The p-log records the source of the difference between in-memory states and on-disk states, accurately conveying the state gap without altering the filesystem's behavior during normal execution.

6.5.4 Negligible Performance Impact

Any recovery machinery should incur negligible overhead to the normal execution when no-failures occur; such overhead directly relates to the methods for handling state gaps and enhancing error detection (e.g., SFI [203]).

Therefore, comparison according to this property often shows results similar to handling state gaps. Systems that simplifies the state gap commonly incurs significant overhead during normal execution, such as uFS-Sync^ and Membrane.

Systems that better handle realistic state gap, instead, perform well during normal execution. Rio needs to ensure the updates to in-memory metadata are atomic. TxIPC uses an instruction-level undo log to rollback the IPC effects. Nebula relies on the p-log to capture the state gap, which is designed to be efficient.

6.5.5 Practical Detection Assumption

All recovery mechanisms necessarily assume a version of the filesystem states to be correct, serving as the starting point for recovery. They are also aware of potential corruption in the failed system's memory, leading to assumptions for error detection that establish guarantees for recovery correctness.

Systems that directly reuse the in-memory states typically assume faults are fail-stop, presuming the in-memory states are correct. In contrast, systems using a checkpoint and replay method assume that errors are detected before the next checkpoint; for example, Nebula, uFS^, uFS-Sync^, and Membrane assume the on-disk states are correct. Assuming on-disk states are correct is more practical than assuming faults are completely fail-stop.

6.6 Evaluation

Our evaluation in this section answers the following questions:

- Does Nebula achieve seamless recovery for applications?
- How problematic is the state gap for real-world applications in the case where it is not handled?
- Is Nebula robust to memory corruption?

- How much performance overhead and memory overhead does Nebula incur in the common-path when no failure occurs?
- Does Nebula achieve fast recovery? How long does it take for each mechanism during recovery?

6.6.1 Benchmarks and Methodology

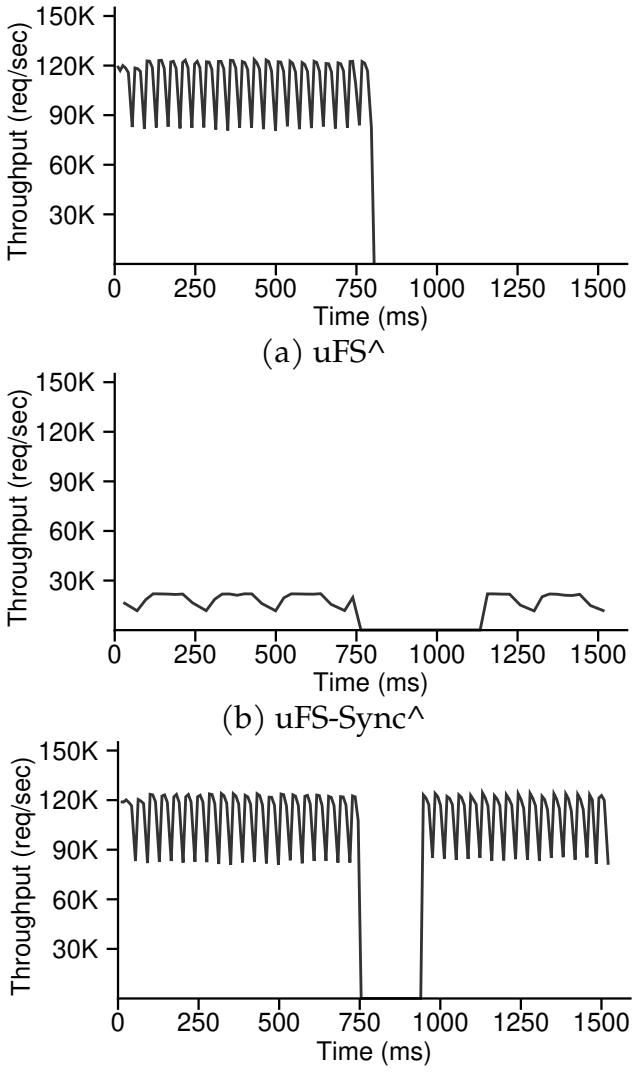
We also create benchmark suites for evaluating the recovery correctness, common-path performance overhead, and recovery time of Nebula.

Baselines Systems: We compare Nebula to two baseline systems: uFS^ and uFS-Sync^, both of which are integrated with the restarting mechanism designed for Nebula.

We demonstrate the necessity of handling state gap by analyzing applications' reaction to uFS^, showing the benefits of lossless recovery provide by Nebula. We demonstrate that exit activation, designed for p-crash recovery, achieves the goal of negligible performance overhead in common path by comparing with uFS-Sync^, which provides failure transparency thorough s-crash recovery. We also show that Nebula incurs far less common-path overhead than an approach of using Membrane-style reply (e.g., 0.15% vs. 3.43x).

Recovery Correctness: Our benchmarks perform intensive fault injection to show the effectiveness in addressing the state gap and the robustness against memory corruption.

We run several workloads, including 24 controlled system-call sequences and ten workloads for five real-world applications (gnu-sort, cp, unzip, SQLite, and LevelDB). The length of operation sequence is long enough to cover a wide range of system calls and their combinations. Importantly, we inject fail-stop p-crashes in the filesystem server immediately after and in the middle of every single operation during the workloads. By enumerating all crash points during a workload sequence, our benchmarks

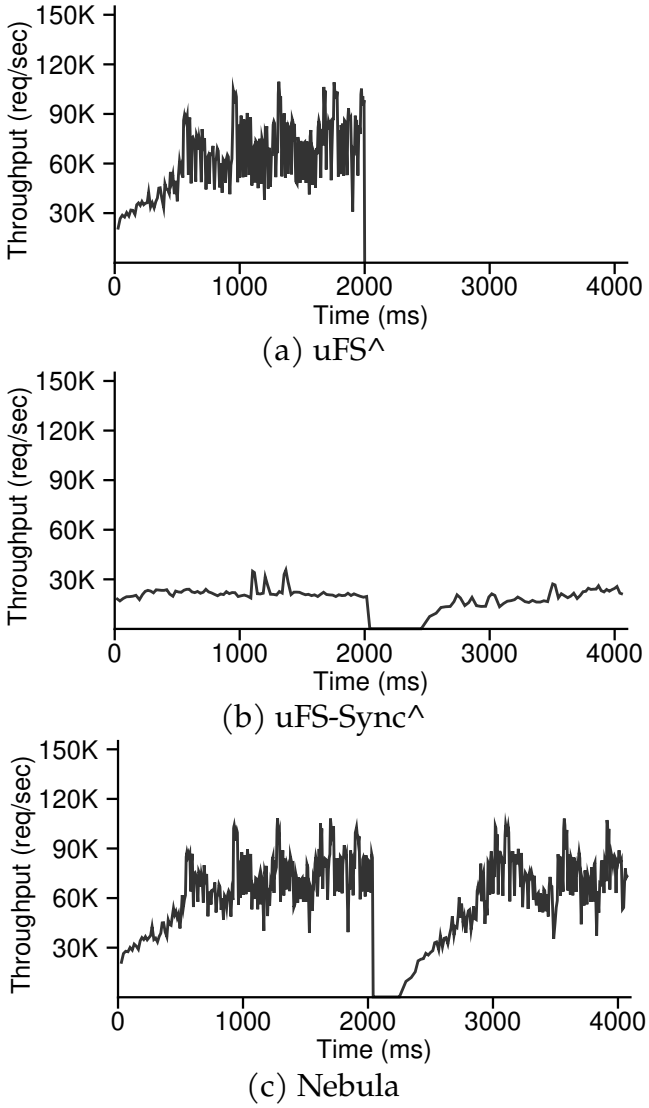


(a) uFS^

(b) uFS-Sync^

(c) Nebula

Figure 6.3: **Recovery with uFS^, uFS-Sync^, Nebula on LevelDB Load**. After the p-crash at time 800ms, 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. Nebula achieves the best of both: transparent recovery and high performance. Recovery time with uFS-Sync^ is 346ms; with Nebula it is 178ms.

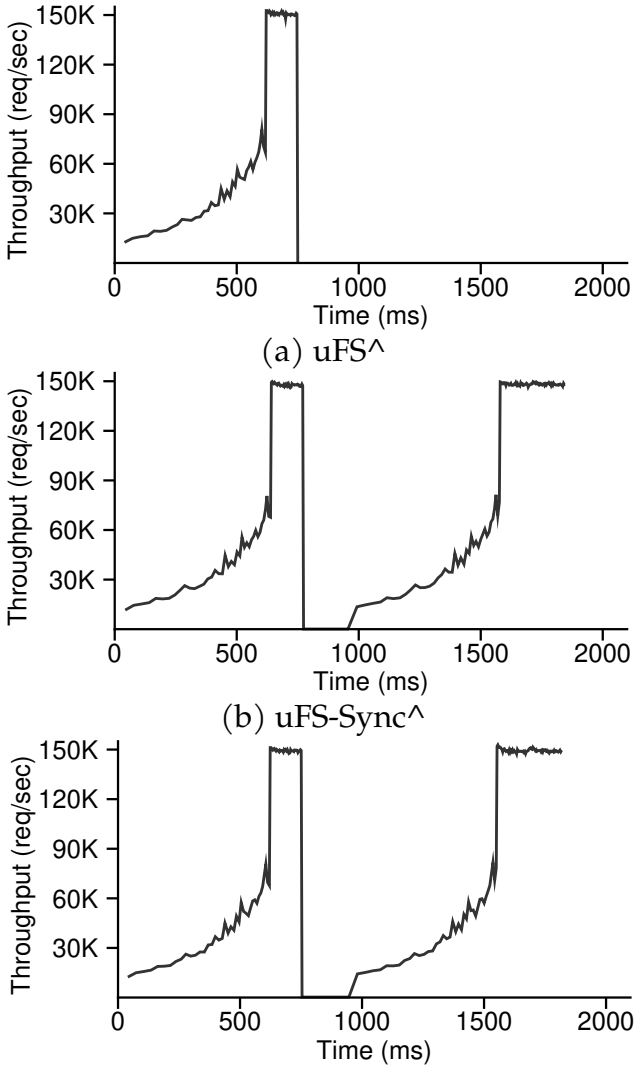


(a) uFS^

(b) uFS-Sync^

(c) Nebula

Figure 6.4: **Recovery with uFS^, uFS-Sync^, Nebula on LevelDB YCSB-A**. On a read-write mixed workload, the common case performance of uFS-Sync^ suffers because dirty pages are persisted after every operation. Recovery time with uFS-Sync^ is 337ms; with Nebula it is 240ms.



(a) uFS^

(b) uFS-Sync^

(c) Nebula

Figure 6.5: **Recovery with uFS^, uFS-Sync^, Nebula on LevelDB YCSB-C**. After a restart, read-only workloads must rewarm the page cache. Recovery time with uFS-Sync^ is 165ms; with Nebula it is 180ms.

evaluate the correctness of handling each single prefix of the operation sequence and the resulting different state gaps. To the best of our knowledge, this is the first systematic evaluation and analysis of the state gap problem in filesystem recovery, and their consequences to real-world applications.

A p-crash during an operation stresses the exactly-once semantics ensured by the p-log entry design. The fail-stop p-crash is emulated by a null pointer dereference, causing a SIGSEGV.

We inject memory corruption across the main memory regions: stack, heap, DMA-able memory, and p-log. According to the literature [24, 54], random bitflips often do not manifest to detected errors; therefore we employ targeted memory corruption. We exhaustively corrupt all the memory regions, by setting a particular trunk of memory to be uniformly filled with a specific value.

Common-path Performance Overhead and Recovery Performance: We run copy, sort, and LevelDB with six workloads to evaluate the common-path overhead, comparing it with uFS-Sync^. We evaluate the recovery time of Nebula by injecting p-crashes after every 500 operations in the workload, showing the impact of restarting, s-crash recovery, and exit activation on recovering the state gap.

6.6.2 Transparent Recovery vs. Performance

Our first experiments highlight that after a p-crash, Nebula provides transparent recovery to unmodified applications and does not impact common-case performance. In contrast, uFS^ is not able to recover from a simple p-crash; uFS-Sync^ recovers from the p-crash, but has significantly worse performance for non-read-only workloads.

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 Nebula, as shown in Figure 6.3, Figure 6.4, and Figure 6.5.

The first graph of each workload set shows LevelDB throughput with uFS^. When each workload begins, 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. The reasons LevelDB and other applications have poor recovery with uFS^ are explored later (§6.6.3).

The second graph of each set shows throughput with uFS-Sync^. As desired, uFS-Sync^ transparently recovers from the p-crash allowing LevelDB to continue operating without manual intervention: there is simply a pause in throughput for a few hundred milliseconds. However, common-case performance of uFS-Sync^ suffers significantly compared to that of uFS^ for non-read-only workloads; in particular, the YCSB-Load workload which consists of sequential writes and YCSB-A which has a 1:1 read:write mix, are approximately 5x and 3x slower. The performance of uFS-Sync^ is compared to Nebula in detail later (§6.6.5).

Finally, the third graphs show throughput with Nebula. As desired, Nebula transparently recovers from the p-crash while still delivering high throughput before and after the p-crash. We make several observations from these graphs. 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 Nebula. Second, for write-intensive workloads (e.g., YCSB-Load in Figure 6.3), throughput does not drop after recovery since Nebula re-uses dirty data pages; for read-only workloads (e.g., YCSB-C in Figure 6.5), performance drops but then increases again, because clean pages are discarded and refetched to rewarm the page cache. Finally, the recovery time of Nebula is sometimes slightly

#	System Call Sequence Followed by P-Crash	uFS^ uFS
1	open(f) read(f) open(f) lseek(f)	BadFd ✓
2	open(f) read(f) close(f) open(f) read(f) close(f)	✓ ✓
3	open(f) write(f) close(f) open(f) write(f) close(f)	DLoss ✓
4	open(f) pwrite(f) close(f) open(f) pwrite(f) close(f)	DLoss ✓
5	open(f) write(f) sync(f) open(f) lseek(f) sync(f)	BadFd ✓
6	open(f) write(f) sync(f) write(f) open(f) lseek(f) sync(f) write(f)	BadFd,DLoss ✓
7	open(f) pwrite(f) sync(f) open(f) pwrite(f) sync(f)	BadFd ✓
8	open(f) write(f) sync(f) close(f) open(f) write(f) sync(f) close(f)	✓ ✓

Figure 6.6: **Recovery of uFS^ and uFS on 24 System-Call Sequences: (a) Include Simple Operations Unrelated to Directories**. The first row of each group shows the system call sequence in the test and the uFS^ result (DLoss: DataLoss) after a single p-crash. The second row shows the system calls performed by uFS on restart (green operations are modified; gray operations are ignored) and its result (successful: ✓).

faster than that of uFS-Sync^. Recovery time is explored in detail below (§6.6.6).

We now explore the strengths and weaknesses of uFS^, uFS-Sync^, and Nebula for performing transparent recovery and delivering high common-case performance.

6.6.3 Transparent P-crash Recovery: Handling State Gap

To stress the recovery of the uFS variants on p-crashes, in experiments below we inject more than 30,000 faults in workloads containing both controlled system-call sequences and real applications. We show that both UNIX utilities (sort, cp, and unzip) and production-level data libraries

#	System Call Sequence Followed by P-Crash	uFS^ uFS
9	creat(f) close(f) creat(f) close(f)	FLoss ✓
10	creat(f) write(f) creat(f) write(f)	BadFd,FLoss ✓
11	creat(f) write(f) close(f) creat(f) write(f) close(f)	FLoss ✓
12	creat(D/f) write(D/f) sync(all) open(D/f) lseek(D/f) sync(all)	BadFd ✓
13	creat(D/f) write(D/f) close(D/f) sync(all) creat(D/f) write(D/f) close(D/f) sync(all)	✓ ✓
14	open(f) write(f) unlink(f) open(f) write(f) unlink(f)	BadFd,DLoss,DFE ✓
15	open(f) write(f) close(f) unlink(f) open(f) write(f) close(f) unlink(f) #nlink=0	DFE ✓
16	open(f) write(f) close(f) rname(f,f1) open(f) write(f) close(f) rname(f,f1) #nlink=0	NRevoke,DLoss ✓
17	open(f) write(f) unlink(f) sync(all) creat(f) lseek(f) unlink(f) sync(all)	BadFd ✓
18	open(f) write(f) close(f) unlink(f) sync(all) open(f) write(f) close(f) unlink(f) sync(all)	✓ ✓
19	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)	✓ ✓
20	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 ✓
21	creat(D/f1) write(D/f1) sync(D/f1) CreatReuseDInode(D/f1) lseek(D/f1) sync(D/f1)	BadFd,FLoss ✓
22	creat(D/f1) write(D/f1) sync(D) open(D/f1) write(D/f1) sync(D)	BadFd,Garbage ✓
23	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 ✓
24	creat(D/f1) write(D/f1) close(D/f1) sync(D) open(D/f1) write(D/f1) close(D/f1) sync(D)	Garbage ✓

Figure 6.7: **Recovery of uFS^ and uFS on 24 System-Call Sequences: (b) Include Operations Related to Directories**. The first row of each group shows the system call sequence in the test (rname represents rename) and the uFS^ result (DLoss: DataLoss; DFE: Deleted File Exists; FLoss: FileLoss; NRevoke: Rename Revoked) after a single p-crash. The second row shows the system calls performed by uFS on restart (green operations are modified; gray operations are ignored) and its result (successful: ✓).

(SQLite and LevelDB) are not robust to uFS^ p-crashes, exhibiting failure for return codes, data loss, and corruption. In contrast, Nebula recovers from all p-crashes such that applications continue successfully.

6.6.3.1 Controlled System-Call Sequences

To ensure that Nebula correctly recovers a range of system calls, we build a test suite containing 24 sequences of system calls with a p-crash at the end. Figure 6.6 and Figure 6.7 show the original system call sequence, the result when uFS^ is used, and the operations Nebula replays on restart, modified or ignored by AIM as indicated.

The results show that Nebula correctly handles all cases, in contrast to uFS^. Specifically, if any file descriptors are not closed before the p-crash, uFS^ returns EBADFD when fd is used since uFS^ has no record of it (e.g., in 1, 5, 6, 7, 12, 14, 21, and 22). If any file writes are not synced, uFS^ loses data (e.g., in 3, 4, 6, 14, 16, and 20). If a create's change is lost, uFS^ loses entire files (9, 10, 11, 21, and 23). Finally, unlink and rename operations can be lost (14, 15, 16).

6.6.3.2 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 both Nebula and uFS^. Figure 6.8 gives workload details and 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 and examine the application's resulting behavior. 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 filesystem content as when there is no p-crash.

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 6.8: **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.

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 incorrect; F_OK and F_BAD may be acceptable, since they indicate an error the application could not handle, but require manual intervention.

Figure 6.3 shows that applications are not robust to p-crashes with uFS^, with many instances of return code failures (F_OK/F_BAD) and bad data (S_BAD/F_BAD). The application behavior with uFS^ is described further in Figure 6.9, illustrating how a p-crash at different points in a realistic filesystem call sequence can result in various unexpected and challenging outcomes for applications. A single background sync can change the consequences drastically. Moreover, given the complex combinations of system call sequences exhibited by real-world applications and their unpreparedness for p-crashes, a systematic solution like Neb-

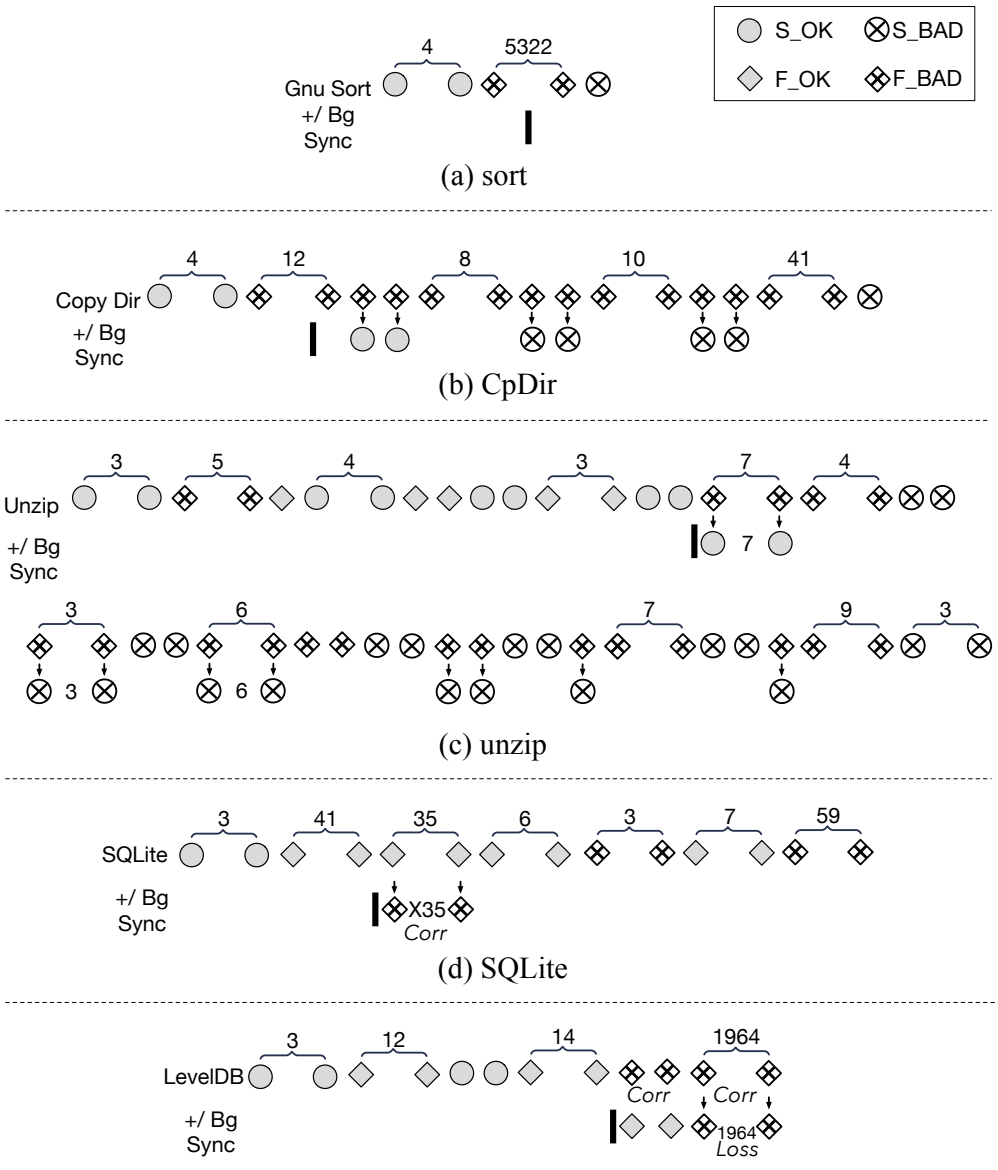


Figure 6.9: **Application Reaction to p-crash of uFS^.** Each symbol indicates the impact of a p-crash and restart with uFS^ after each system call. The first line shows the results with no background sync; the second line shows with a background sync, where the thick bar shows where the sync occurs. The arrows between lines indicate cases that differ across no sync and sync.

Workload	#of ops	Fail after each op					Fail during an op
		uFS^				Nebula	Nebula
		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

Table 6.3: **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 Nebula, all applications are correct.

ula is necessary to ensure transparent recovery, especially for handling the state gap.

For Sort and CpDir with uFS^, most cases are F_BAD (5322/5327 and 77/82) because the utilities depend on an opened file descriptor that is lost, causing both to terminate with an error code. Sort does not `fsync` after its last operation, causing a problematic case where the exit code indicates success, but data is lost (S_BAD). For CpDir, a background sync flushes a newly-created destination directory, causing some later operations to succeed and confusing the utility into returning success even though data is lost (5/82).

Unzip with uFS^ usually results in F_BAD (47/77), but has more complicated behavior. First, unzip retries some failed system calls, but does not correctly identify the root cause and retries irrelevant operations. Second, unzip sometimes prints warning messages, but continues executing and incorrectly exits with a success return code. As a result, many cases with a background sync report S_BAD (26/77).

While simple utilities may not be expected to correctly retry operations when a filesystem returns an error, production-quality libraries that care about durability, such as SQLite and LevelDB, contain recovery mechanisms such as write-ahead-logging. To study their reactions to p-crashes, we run a simple load workload. 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, the database reports corruption (labeled Corr), or there is data loss (labeled Loss). We have verified that offline tools can manually repair the database when it cannot be opened or is corrupted, but not when data is lost; however, offline tools reduce system availability and burden administrators [158, 201]. Thus, even applications with sophisticated durability techniques need more support than uFS^.

These results indicate it is unrealistic to expect current applications to correctly handle p-crashes: to provide transparent filesystem availability, a reliable filesystem must go beyond merely restarting and rebuilding connections and instead ensure that all states are recovered properly as if no error occurred. Figure 6.3 shows Nebula 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.

6.6.3.3 Multiple Processes

As a shared filesystem service, handling multiple applications is essential. To demonstrate that Nebula transparently recovers multiple processes,

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 6.4: **Nebula Recovery after 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 errors or application errors). The percentages of cases with detected errors for each memory region are: 0.71%, 20.4%, 73.5%, and 100%.*

we simultaneously run three applications (LevelDB, Sort, and CpDir) and inject p-crashes at 300 random points. In all cases, with Nebula the three applications continue executing correctly and return S_OK.

6.6.4 Transparent P-crash Recovery: Memory Corruption

We demonstrate Nebula provides transparent p-crash recovery in the persence of memory corruption. These experiments show that Nebula recovers naturally from memory corruption since it builds from the on-disk states of the filesystem and relies on the well-protected p-log; thus, any corrupted data in memory is simply discarded and new values are recreated by replaying the p-log.

We inject memory corruption into the four major memory regions of the filesystem process address space (as depicted by Figure 6.1): stack, heap, filesystem metadata (within DMA-able memory), and p-log. After completing the first half of the workload, the fault injection module derives the runtime memory layout from `/proc/self/maps`, injects memory corruption in a configured region, and then Nebula continues handling the rest of workload. We monitor the cases where the faults actually manifest into errors, including filesystem errors (and thus recovery) or application errors (e.g., the application warns or aborts). Table 6.4 presents

the results.

Each experiment corrupts a 4KB chunk of memory in a specific region, except for the stack (64B). The total size of the stack region is much smaller compared with others, so a large corruption size leads to the same amount of work for recovery (as long as a particularly problematic portion is corrupted), while a 4KB memory corruption in other regions is adequate for generating diverse cases. The correctness of filesystem metadata and data is also checked. Our experiments thus exhaustively cover the corruption of all these memory regions, with a total of 18,100 memory corruption injection experiments, including the 3,373 cases where an error occurs, which are included in the table.

Nebula successfully performs a restart, continues handling application workloads, and ensures the correctness of filesystem metadata in all cases, demonstrating the robustness of the restart mechanism and p-log.

In a small amount of cases, four cases in stack region and five cases in heap region, the filesystem data are not recovered properly, which is due to limitations in our current implementation. Once the kernel monitors the exit of the main process, it signals the main process, which runs the procedure to rescue the data pages in the signal handler, thus corrupting a particular region in stack and heap halts the procedure. We expect further improvement that moves the rescue procedure (read-only) to the kernel space to address this issue. An eBPF program is a promising lightweight solution to safely run this procedure in the kernel space.

6.6.5 Common-Case Overheads

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

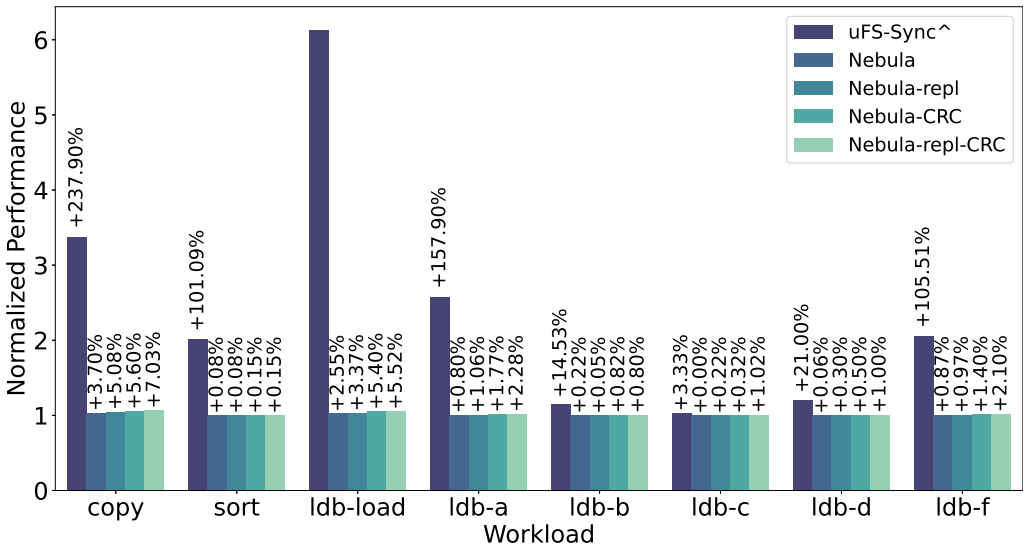


Figure 6.10: **Performance Overhead of uFS-Sync^ and Nebula Variants**. Nebula-repl writes to 2 CRC'ed p-logs; Nebula-CRC adds CRCs to filesystem in-memory structures (updated on all writes; checked on all reads); Nebula-repl-CRC combines replication and CRCs.

6.6.5.1 Performance

Figure 6.10 shows the performance overheads of uFS-Sync^ and uFS (with variants to protect structures with CRC and replicate the p-log) normalized to uFS^ for a range of data-intensive applications: Copy, Sort, and LevelDB for Load and five YCSB workloads. We use sufficiently long workloads to trigger garbage collection in uFS.

The slowdown of uFS-Sync^ is significant, peaking at 6x slower than uFS^. Copy performs many writes and meta-data updates; therefore, uFS-Sync^ has high overhead (3.3x). Sort represents applications that are data and computation intensive; therefore, the slowdown of uFS-Sync^ is not as severe (2x). The performance of LevelDB depends on the write-activity in the workload: Load is the most write-intensive and has the most slowdown; YCSB-B, C, D are all read-dominated and have low overhead; YCSB-A, F have more balanced read/write ratios and intermediate

Workload	Baseline Mem (MB)	Crc dentry blocks	Crc data bitmap	Crc I bitmap	Nebula 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%

Table 6.5: **Memory Overhead Summary**. 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.

overheads.

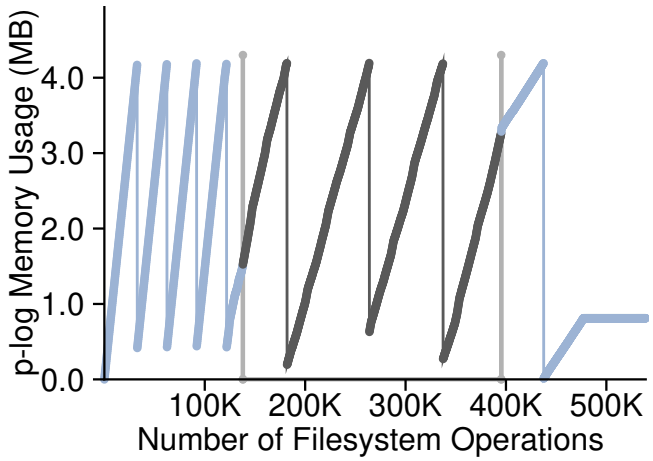
We have also implemented Membrane-style replay which requires extra full sync to handle fsync and directories operations like creat. For LevelDB load and Copy workloads, Membrane-style replay has high overhead compared with the baseline uFS, 1.33x and 3.43x, respectively.

For all workloads, the overhead of Nebula is low; in the worst cases of very write-intensive workloads (Copy and LevelDB-Load) the overhead of Nebula with no extra memory protection is less than 4%, while adding both p-log replication and CRC protection to essential data structures raises it to 7%; the overhead of all other workloads is below 1%, or 2% with full memory protection.

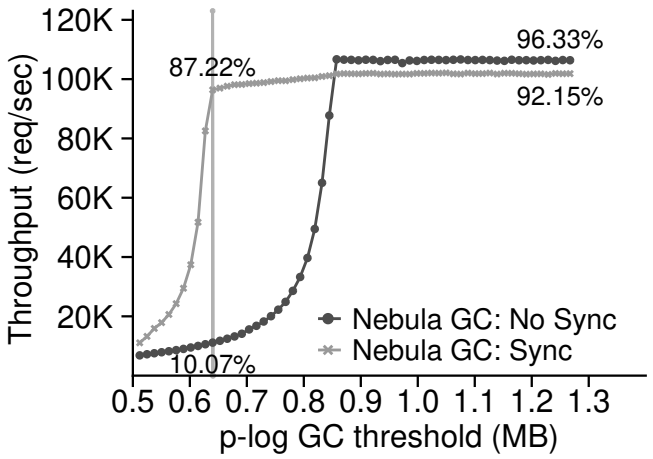
6.6.5.2 Memory Overhead

We next evaluate the extra memory overhead incurred by Nebula to provide transparent recovery. Nebula adds memory to uFS^ in two ways: for the p-log (and its replica) and for CRC checksums to detect corruptions.

The memory cost of the p-log is proportional to the number of requests



(a) p-log Memory Usage (LevelDB). Workloads for the Three Phases: SeqWrite, YCSB-A, and YCSB-C.



(b) p-log Memory Threshold Trade-off

Figure 6.11: **p-log Memory Usage (LevelDB).** *(a) shows that memory usage varies across workloads and that garbage collection effectively reclaims log entries. (b) illustrates the trade-off between the memory threshold for garbage collection and performance.*

that have not been garbage collected. Figure 6.11(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), Nebula performs garbage collection, reclaiming operations that will not be replayed on a p-crash; the p-log does not shrink to 0 because some file descriptors remain open as LevelDB runs. The write-intensive Load workload fills the p-log faster than read-write YCSB-A; similarly, read-only YCSB-C adds fewer operations to the p-log.

Figure 6.11(b) illustrates the trade-off between performance and the maximum size allocated to the p-log 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, LevelDB throughput is unacceptably low if the p-log threshold is too small. GC-Sync enables a smaller p-log compared to GC-NoSync (e.g., 0.65MB instead of 0.85MB), but slightly reduces LevelDB performance (92% of the baseline instead of 96%) due to the 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 6.11(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 1B CRC for each inode is embedded within existing inode structure and therefore does not require extra memory. Thus, CRC adds minimal memory overhead (at most 0.02%).

6.6.6 Recovery Performance

In our final experiments, we show recovery time for uFS^, uFS-Sync^, and Nebula in Figure 6.12; since recovery time depends on filesystem state, 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); the y-axis is the time for recovery after that p-crash point. While all ap-

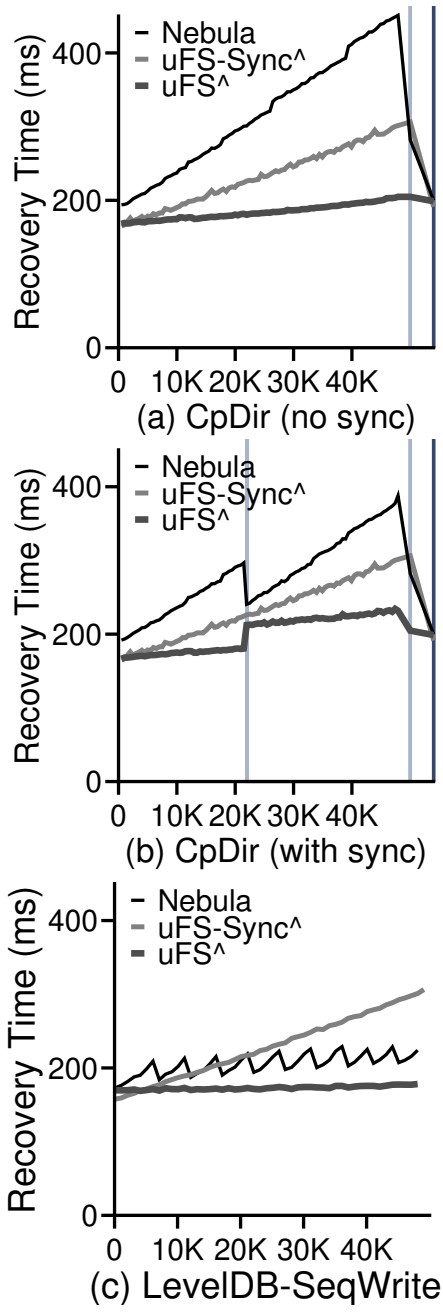


Figure 6.12: **Recovery time with uFS^, uFS-Sync^, and uFS.** The light blue vertical lines indicate the time where a background sync occurs; the deep blue (last) vertical lines indicate a checkpoint. X-axis is the timing (# of ops) of a p-crash.

proaches recover in less than 500ms thanks in part to kernel-coordinated speculative restart, the more state that exists within uFS at the p-crash, the longer recovery takes.

For uFS^, recovery time is relatively constant. Note that even though uFS^ recovers, 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 destroy a larger uFS address space.

The recovery time for uFS-Sync^ and Nebula includes the recovery time of uFS^. For uFS-Sync^, recovery time increases as the applications run longer because more operations have been persisted to the s-log. The time for recovery becomes minimal at the end after uFS performs a s-log checkpoint (marked by the last vertical line).

Finally, the recovery time of Nebula 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. This reduction is illustrated in CpDir when a background sync is performed at the first vertical line; and in LevelDB-Load periodically when a foreground sync occurs. Thus, depending on the size of the p-log and s-log, recovery may be faster in Nebula or uFS-Sync^.

6.7 Summary and Conclusions

In this chapter, we present Nebula, the variant of uFS that realizes the fault tolerance potential of semi-microkernel architecture. Returning to the question asked in the beginning of this chapter, can applications using the filesystem seamlessly continue after a server process crash with Nebula?

The answer is a resounding yes. Nebula equips uFS with a range of mechanisms designed for p-crash recovery, leveraging the fact that the host OS, the on-disk state, and the failed process memory can all contribute to the recovery, a unique opportunity arising in the p-crash model.

To address the aforementioned challenges, Nebula introduces three mechanisms: kernel-coordinated speculative restart, lightweight protection of in-memory states, and the p-log data structure for exit activation. These are combined to achieve a fast, robust, seamless p-crash recovery machinery that does not sacrifice the performance benefits of uFS.

We demonstrate Nebula through extensive fault injection experiments, including fail-stop errors and memory corruptions. Our benchmark suite, which crashes the filesystem server to generate a large number of realistic state gaps under real-world application workloads, is the first to analyze and evaluate the state gap problem and the recoverability of all prefix operation sequences. We also show that Nebula recovers quickly (in less than 500 ms) and incurs negligible overhead in the common path (less than 7%) even under challenging workloads.

Finally, relying on full system crash recovery to handle both process crashes and power failures is worth reconsideration because it misses the opportunities for better guarantees, availability, and performance. We believe that Nebula represents a step forward in the design of fault-tolerant microkernels and user-space system services. The process crash recovery machinery we have designed is beneficial, and the principles and mechanisms we introduce can be applied more broadly.

7
When Slow is Good

We have alluded to the velocity and customization benefits of the semi-microkernel approach earlier (§3.2.2), which we have leveraged to build uFS and Nebula. The approach allows for fast development, release, upgrade, and customization of various versions of the same system.

But why do we need yet another version of uFS? One perhaps obvious answer is to fix a bug and release a new version. When a bug is fixed, one might wish (though impossible) to go back in time and start the filesystem server using the fixed code, avoiding any adverse consequences caused by the bug.

The question then arises: can we have a version of uFS that is more correct (i.e., free of bugs) by design, allowing Nebula to run a bug-free code path during the recovery (i.e., replay)? We refer to this approach of using a more robust version of the same system to recover from runtime errors as *Robust Alternative Execution* (*RAE*).

The main benefit of RAE is that it allows the system to recover from deterministic software bugs, as replaying the p-log with the original code path would trigger the same bug and fail the recovery. Recovering from deterministic errors is fundamentally difficult for generic recovery when using the original implementation to replay the workload [127].

For the version of uFS utilized for RAE, *slow is good*, because by omitting any designs and optimizations for performance (thus being slow),

the system can be extremely simple, making it hard to go wrong. We refer to this version of the system as a *shadow filesystem*.

We explore the idea of RAE by building a prototype of the shadow filesystem for uFS and integrating it into Nebula. In this last chapter of exploration, we go back to the beginning, where the filesystem is single-threaded and minimally functional. The questions now are: How simple can the shadow filesystem be while still being able to run the operations in the p-log? And how much recovery time does it take? (§7.2 and §7.3)

The chapter is organized as follows. We first present a bug study of Linux kernel filesystem (ext4), showing the prevalence of both deterministic and non-determinsitic bugs (§7.1.1). We then introduce the robust alternative execution approach and the shadow filesystems (§7.1.2). We present the design of uFS-Shadow, a shadow filesystem for uFS, that is incorporated into Nebula(§7.2). Finally, we demonstrate that uFS-Shadow allows Nebula to recover from deterministic errors. We also evaluate the recovery time and implementation efforts of uFS-Shadow(§7.3).

7.1 Motivation and Approach

We present a mini-study of filesystem bugs in Linux ext4 that motivates the RAE approach and then describe the properties and strategies of shadow filesystems for RAE.

7.1.1 Deterministic Bugs in a Kernel Filesystem

One common assumption is that deterministic bugs are rare in mature softwares after extensive testing and debugging [54, 185]. However, a number of deterministic bugs in Linux kernel filesystems are reported and fixed every year. When a bug is reported with a reproducer including a sequential workload, it is deterministic.

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

Table 7.1: **Study of filesystem bugs (Linux ext4)**. Bugs that do not have reproducers, or are related to the interaction with IO (e.g., multiple inflight requests), or are related to threading, are classified as non-deterministic. Bugs are classified as Unknown in their consequence when the commit message does not contain clear clues of external symptoms. The columns present the numbers of bugs according to each consequence. WARN indicates the bug hits a WARN_$*$() path, the suggested substitute of BUG() in the Linux kernel. We collect the bugs by filtering the ext4's subtree's git log with the mentioning of "bugzilla" or "reported by", so the year of a bug corresponds to the year of its fix. (256 bugs in total since 2013).

We study 256 bugs in the Linux ext4 filesystem and categorize them. As shown in Table 7.1, deterministic bugs are prevalent (165/256), and a significant portion cause crashes or warnings that are detected as runtime errors (89/165).

Figure 7.1 presents the number of deterministic bugs by the year of fixes. More bugs are fixed in recent years for two reasons. First, advances in testing reveal more vulnerabilities with the proper workloads, especially in input sanity checks [100]. Second, new kernel features such as blk-mq, page folios, and iomap [47, 48, 60] introduce new bugs.

One notable type of deterministic bug occurs when a user mounts a crafted disk image and issues operations to trigger crash (e.g., a null-pointer dereference or use-after-free) in the kernel [39, 123, 209]; such images can bypass FSCK [94], leading to crashes from malicious attackers. One example of operation sequence that triggers a kernel crash in ext4 is shown in Figure 7.2.

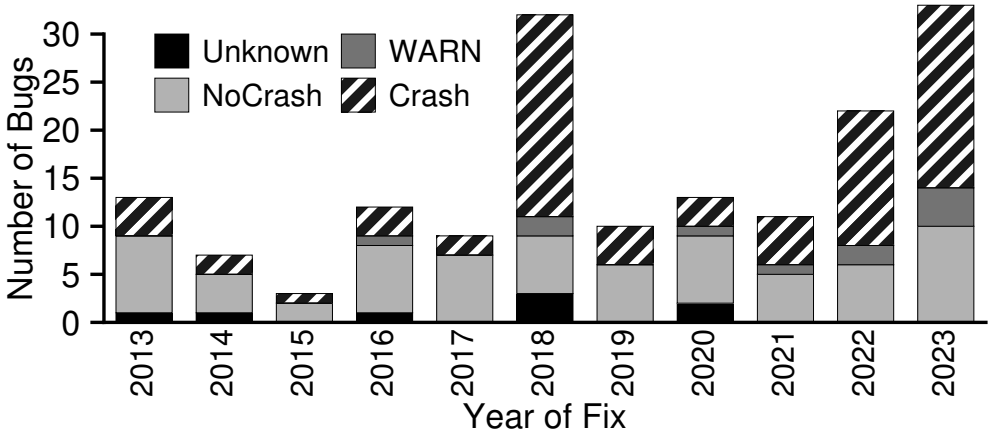


Figure 7.1: **Number of deterministic bugs (fixed) by year**. Examples of NoCrash consequences include data corruption, performance issue, permission issue, freeze, deadlock, etc.

```
#/bin/bash
mount -o loop tmp32.img mnt  # a corrupted image
mv mnt/foo/bar mnt/foo/Yzo... OFz
mv mnt/foo/Yzo... OFz  mnt/foo/AId... 7oF
```

Figure 7.2: **An Example of Error-Induced Sequence**. According to CVE 2022-1184 [2], such a sequence triggers a user-after-free in the Linux kernel (ext4).

N-version programming [13–15] (NVP) is a classic approach that can handle deterministic bugs. NVP advocates the independent development of several versions of software with the same specification, running them simultaneously to generate output by combining the decision of each version (via voting). Despite its conceptual advantage of detecting and masking one version's fault, the assumption of statistically independent failures does not usually hold [102]. Further, maintaining and executing multiple versions (often, at least three) incurs excessive overhead.

We propose a different approach – *Robust Alternative Execution* – to handle deterministic and non-deterministic runtime errors for a given

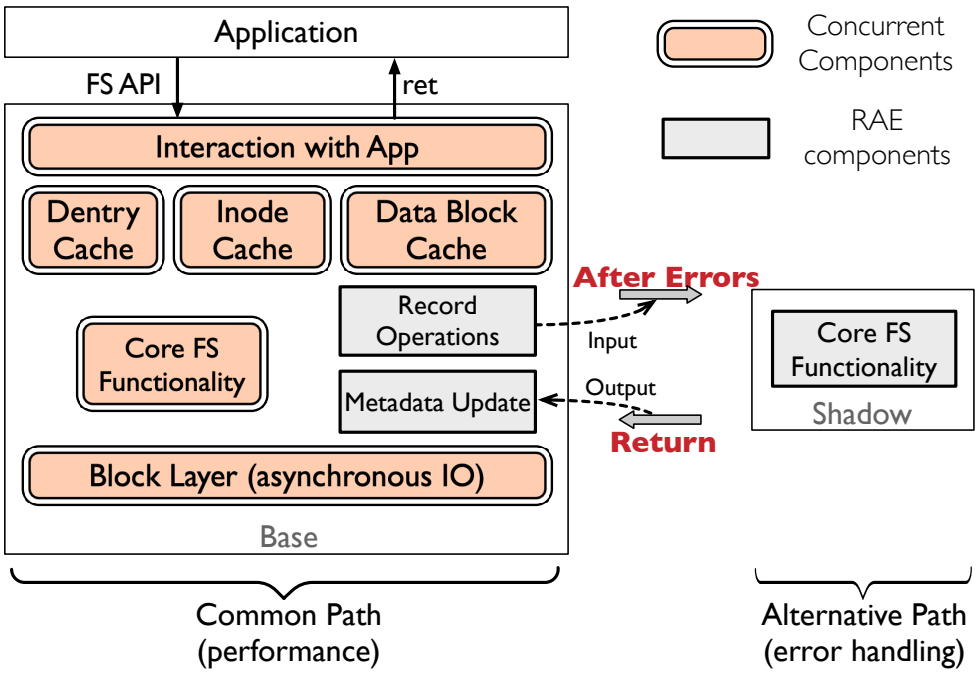


Figure 7.3: **Filesystem architecture with shadow filesystems**. The left side shows the constructs of modern concurrent and performance-oriented filesystems; the right side shows the shadow. The arrows indicate the control transfer between the base and the shadow.

base filesystem via a shadow filesystem. As shown in Figure 7.3, after detecting an error, a shadow is launched as the temporary substitute of the base to execute the problematic filesystem operation sequence and return correct metadata updates to a restarted base.

7.1.2 A Practical Approach: Robust Alternative Execution

We advocate robust alternative execution (RAE), a practical approach to improving the reliability of existing high-performance filesystems [112] via shadow filesystems. The shadow filesystem is a simple implementation of the base filesystem that focuses on correctly handling all workloads without concerning itself with performance.

The main intuition of this approach is to separate the filesystem execution into a common path and a alternative path and run one implementation of the filesystem for each path. The two implementations share the same on-disk format and can both execute the same API (e.g., POSIX). While the base is optimized for performance in the common path, the shadow strives for robustness.

As shown in Figure 7.3, RAE requires a component to record the error-induced sequence. After an error is detected, the in-memory metadata and file descriptors in the base are reset by a restart. A separate shadow filesystem process is launched to execute the recorded operations (reading any data required from the disk), and the shadow filesystem produces new (and correct) metadata structures that are directly used by a rebooted base. As such, the rebooted base starts from the recovered metadata and file descriptors without needing to re-execute the error-triggering operation sequence.

Compared to NVP, RAE has a significant conceptual difference between the two implementations, as shown in Figure 7.3. The shadow only needs the core functionality of the filesystem and does not need to include other components in the base designed for performance. Such a large difference and simplification are possible because the shadow is only executed in the alternative path (i.e., rare cases).

The shadow filesystem has three properties: it is simple, it can afford to be slow, and it is more robust, achieved mainly through two strategies.
Simple yet functional implementation: Modern filesystems (including uFS) are complex, primarily due to optimizations for concurrency, caching, and asynchronous interaction with storage devices. The software architecture (Figure 7.3) of modern filesystems (including uFS and kernel filesystems) consists of an interface layer for (un)marshaling and user interaction (e.g., IPCs and VFS), a cache for inodes, data blocks, and directory entries, and numerous concurrency-related optimizations. A block layer (e.g., blk-

mq) interacts with storage devices asynchronously, employing various mechanisms and policies for performance. In between, the core functionality component handles the semantics of the filesystem metadata and data, interacting with those concurrent and complex components to acheive better performance. A filesystem implementation's complexity and likelihood of bugs are exacerbated by the interactions with these performance components; worse, those performance components keep evolving, causing more bugs, as shown in our bug study.

To improve robustness, a shadow eliminates all performance optimizations, focusing instead on correctness. It thus aims to be the simplest possible yet functional implementation of the base filesystem. To reduce complexity and the likelihood of bugs, a shadow is not interactive with users, does not have concurrency, does not have sophisticated caching structures and policies (e.g., LRU 2Q [117]), performs IO synchronously, and does not write to devices, as shown on the right side of Figure 7.3.

Therefore, the first vehicle for shadow's robustness is simplification. Such simplification reduces the size of the codebase, naturally reducing the likelihood of bugs. The cost of simplification is low performance, and the shadow intentionally avoids any performance techniques.

Extensive runtime checks: The shadow can further improve its robustness by adding extensive runtime checks because the performance cost of the checks is affordable in the alternative path.

For example, an input check can resolve the CVE we discussed above that triggers a use-after-free. It is not practical to assume developers will add source code to check for the validity of every function's input (e.g., assertions), because it will make the code less clean and incur overhead. Furthermore, assertions and checks are usually removed for performance while executing the base in production.

The shadow implementation, instead, encourages as many runtime checks as possible. For example, the validity of each function's input can

be checked, and invariants after completing one operation can also be checked, which is helpful to detect silent corruption due to hardware. Other helpful checks include memory sanitizers.

RAE can be applied to existing high-performance kernel filesystems as our argument of why the shadow could be different, could be simple, and thus could be robust also holds for a kernel filesystem, which shares the software architecture as shown in Figure 7.3.

RAE naturally fits into Nebula as one step further to handle deterministic bugs in addition to all the faults that Nebula already recovers from. Nebula, with its restart and exit activation data structure, p-log, provides the convenient infrastructure to utilize a shadow implementation of uFS. The p-log already records the workload that triggers an error, and we build uFS-Shadow to explore the benefits and challenges of RAE. The rest of this chapter will be focusing on uFS-Shadow.

7.2 uFS-Shadow

We build on Nebula to realize RAE. The practical difference between RAE and Nebula (as described in §6.3.6) is that the transformed p-log is replayed by uFS-Shadow, whereas Nebula utilizes the base uFS's source code to replay the p-log. With uFS-Shadow, the result of the replay (i.e., the state gap) is returned to the restarted uFS. The base uFS and uFS-Shadow never run simultaneously, and they shares the same on-disk data structures. We use uFS and the base, and uFS-Shadow and the shadow interchangeably.

7.2.1 Design

We mainly discuss the design of uFS-Shadow in the context of replay and its interaction with the base.

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

In constrained mode, uFS-Shadow executes completed operations, possibly performs disk reads, and produces output along with a set of modified data structures. Constrained mode also cross-checks with the output of the original execution. Discrepancies in output are reported; whether or not to continue can be configured. For inode number and file descriptor allocation, uFS-Shadow validates if the value produced by uFS is usable, rather than performing its own allocation (which could lead to a different value). The shadow omits operations that returned an error by uFS.

In autonomous mode, uFS-Shadow executes the in-progress operations whose return values have not been seen by the client. In this mode, uFS-Shadow must make policy decisions such as allocating new inode numbers (as opposed to simply validating the decisions made by uFS).
Hand-off back to uFS: The base filesystem must provide well-tested interfaces to absorb the output of uFS-Shadow: a set of file descriptors and on-disk metadata structures. To implement the interfaces, uFSreads these structures and reuses its existing logic to place them into its cache, marked as dirty. After the hand-off, the restarted base resumes execution and admits new operations, at which point all state within the base filesystem is correct and up to date.

The output of uFS-Shadow should not be persisted to the disk used by uFS, because it requires uFS-Shadow to implement a crash consistency protocol to ensure safety in case of a power failure. However, the crash consistency protocol is a major source of complexity in filesystems (§4.3).
Limitation: The main limitation of uFS-Shadow is in cases where an error

causes a wrong value to be returned to the application before the error is detected. For example, if it returns a 1000-byte read when the file only has 500 bytes, even though uFS-Shadow can produce the supposedly correct results, it cannot correct the returned value. Therefore, uFS-Shadow still assumes that the fault is detected before incorrect results are returned to the applications.

7.2.2 Contrasting Base and Shadow

uFS and uFS-Shadow are different in several ways.

Performance optimizations: uFS-Shadow omits all the performance components in uFS. Specifically, uFS-Shadow does not use a dentry cache (§4.2), and instead always performs path lookup from the root inode and scans the directory entries. uFS-Shadow does not utilize the complex structures like inode cache and buffers caches; instead, it uses a simple data structure (e.g., a hashmap without replacement) to manage filesystem structures read from disk during recovery.

uFS-Shadow is strictly single-threaded; it does not deal with locking and concurrency (§4.2). uFS-Shadow busy wait for the completion of the device IO (via SPDK's polling), avoid asynchronous IO and the associated complexity (§4.1).

API support: The shadow filesystem supports the same set of filesystem operations as the base. The exception is those that persist data to disk, such as `fsync`. We omit the sync family API for simplicity, to avoid interacting with the crash consistency protocol. The process-crash model enables such simplification because the restarted base can absort the results of the shadow, which is better to be through memory (e.g., via tmpfs) such that the metadata updates in uFS-Shadow do not need to be flushed to the persistent device. If the base fails in the middle of fsync, our current design relies on the uFS-Shadow for the prefix operations and the base to perform fsync again after the hand-off.

Core functionality: For a given operation sequence, the output at the API level and the effects to semantic states (e.g., file descriptors and metadata) must be equivalent between uFS and the uFS-Shadow. While more low-level policy decisions might differ, the two must agree on essential invariants. For example, allocating ten data blocks for a 4K write can be valid behavior for a particular base filesystem, but the specific blocks allocated might differ, leading to different data bitmaps.

7.3 Evaluation

uFS-Shadow is implemented in around 2.3K lines of code in C++. The current implementation adds the runtime check for the precondition and postcondition for each operation's execution. We run all the experiments in this chapter using the same setup as in §6.6. Nebula with uFS-Shadow can recover from both transient fault and deterministic software bugs.

Our evaluation in this section answer the following questions:

- Can uFS-Shadow offer better robustness than Nebula? Especially, can it recover from deterministic errors?
- Is uFS-Shadow simple to implement? How does the implementation effort compare with uFS?
- How does the recovery performance of Nebula with uFS-Shadow compare to that of Nebula with uFS?

7.3.1 Benchmarks and Methodology

We first use the benchmarks to evaluate Nebula and run all the fault injection experiments described in Table 6.3. We also design a error injector to emulate deterministic software bugs. The injector can trigger a null-pointer-deference in two ways: (1) by a specific workload sequence pat-

Workload	# of ops	uFS-Shadow	
		Fail after each op	Fail during an op
		S_OK	S_OK
Sort	5327	5327	5327
Sort (w/s)	5327	5327	5327
CpDir	82	82	82
CpDir (w/s)	82	82	82
Unzip	77	77	77
Unzip (w/s)	77	77	77
SQLite	154	154	154
SQLite (w/s)	154	154	154
LevelDB	1997	1997	1997
LevelDB (w/s)	1997	1997	1997

Table 7.2: **Transparent Recovery of Applications.** A single p-crash is inserted after (or during) each system call of the five benchmark applications. With uFS-Shadow, all applications are correct.

tern (e.g., ten writes followed by a rename, and rename with a certain pathname), or (2) by a variable's specific value upon access. The failure is deterministically triggered by the configured conditions.

We create 25 deterministic error benchmark, covering various patterns that stress different code paths of uFS. We also refer to the ext4 filesystem bugs we studied (Table 7.1) for workload patterns.

We compare uFS-Shadow with Nebula for recovery performance. We also compare uFS-Shadow with uFS for implementation effort, discussing the lines of code needed for each component.

7.3.2 The Goodness: Robustness and Simplicity

We first evaluate the robustness and simplicity of uFS-Shadow.

7.3.2.1 Robustness

We first run the 300,000+ fault injection experiments with transient p-crash points (Table 7.2). In all the cases, uFS-Shadow recover from the

Length of Sequence	# of Cases	Base	Shadow
1	5	0	5
1 – 10	5	0	5
10 – 50	5	0	5
50 – 100	5	0	5
>100	5	0	5

Table 7.3: **Recoverability under Error-Induced Sequence** *Last two columns show the number of cases that Base and Shadow recover from.*

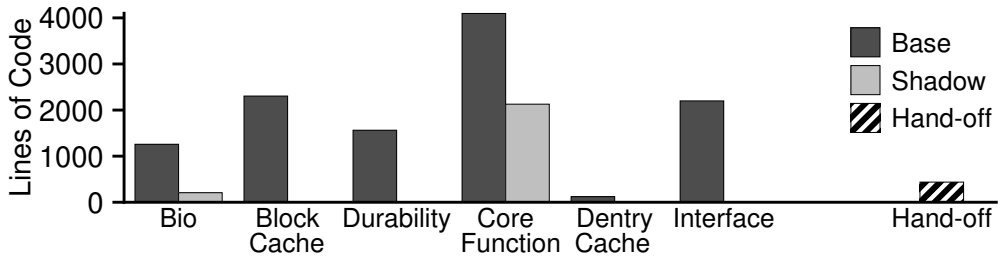


Figure 7.4: **Lines of code comparison between the base and the shadow**. *We show the components described in Figure 7.3.*

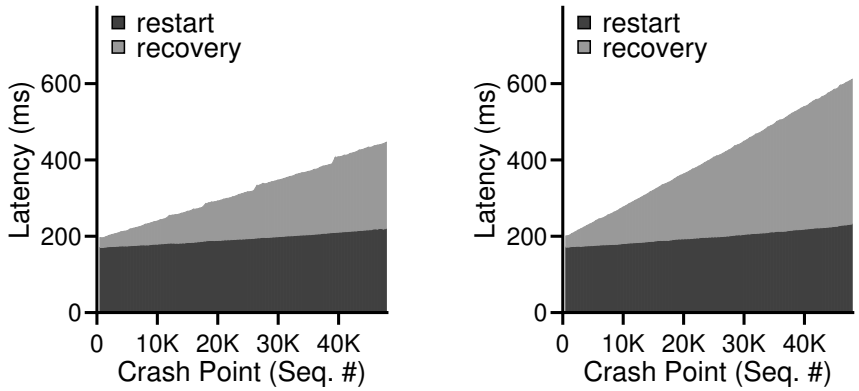
transient p-crashes successfully (S_OK), demonstrating that uFS-Shadow is properly functional as desired.

We demonstrate that uFS-Shadow recovers from deterministic errors, as shown in the Table 7.3. The fault injection benchmarks include workloads whose length are within the five ranges, five cases for each range.

In all the total 25 cases, uFS-Shadow successfully recovers from the deterministic errors, whereas Nebula cannot recover from any of them.

7.3.2.2 Simplicity

We compare the implementation efforts of uFS-Shadow and uFS, as shown in Figure 7.4. Overall, the 2.3K lines of code in uFS-Shadow are significantly fewer than the approximately 35K lines of code in uFS(i.e., 7%). The uFS codebase includes other facilities commonly needed for complex systems, such as debugging and logging. We also don't include the features of load management in the Figure 7.4. The comparison of imple-



(a) Use the base code. (b) Use the shadow code.

Figure 7.5: **Recovery performance comparing the base and the shadow**. *Recovery time (y-axis) is shown in milliseconds. X-axis shows the number of operations before the crash, i.e., number of operations executed during recovery.*

mentation efforts shows that uFS-Shadow is much simpler than uFS.

First, highly-optimized components like the interface (including the IPC message ring), bio (non-blocking I/O), and block caches add significant complexity to uFS. In contrast, uFS-Shadow has a much smaller codebase by omitting these components.

Second, the core functionality implementation is much simpler in uFS-Shadow, with only around half the code of uFS. The reasons are: uFS-Shadow is single-threaded, does not interact with those complex components, notably does not need to deal with non-blocking I/O which greatly simplifies the implementation, and is single-threaded.

Third, the hand-off takes around 450 lines of code, which is simple enough for extensive testing.

The implementations of uFS-Shadow and uFS are fairly different, and importantly, the uFS-Shadow codebase is small enough to be easily understood, tested, and maintained.

7.3.3 The Slowness: Recovery Performance

We now evaluate the recovery performance of uFS-Shadow, understanding the cost of the shadow filesystem, as shown in Figure 7.5. We run the write intensive CpDir workload. As in the previous evaluation (§6.6), we inject a p-crash after 500 system calls (x-axis); the y-axis shows the recovery time after that p-crash point. We disable the background sync for these experiment to focus on the time take for each to run a long sequence. The gray region shows the time to run the operations in Figure 7.5.

Overall, the recovery time of uFS-Shadow compared with Nebula is much slower. However, the slowdown is not as significant as orders of magnitude – recovery time is around 1.6x. One reason is that Nebula only uses a single thread to replay the p-log. The recovery time of uFS-Shadow and Nebula are both proportional to the number of sequences needed to replay. We believe that this slowdown is acceptable in the alternative path.

7.4 Summary and Conclusions

In this chapter, we explore robust alternative execution (RAE), which enables Nebula to recover from deterministic bugs. We build uFS-Shadow, a shadow filesystem for the base uFS implementation, and incorporate it into Nebula to replay the p-log during p-crash recovery. After an error is detected, another version of the filesystem code is invoked to run the workload that fails the base uFS. The filesystem execution is thus separated into two paths: the base filesystem (uFS) handles most workloads in the common path, optimizing for performance, while the shadow filesystem (uFS-Shadow) runs in the alternative path, striving for robustness.

We demonstrate the robustness of uFS-Shadow through fault injection experiments that include both transient and deterministic errors, and uFS-Shadow successfully recovers from all of them. We show that uFS-

Shadow reduces the codebase to 7% of the base uFS. The recovery time is around 1.6x compared with using the base.

Our observation of filesystem software architecture is that the complexity of filesystems lies in the performance components and their interactions, as well as durability, which are more likely to introduce bugs. In contrast, an implementation that provides the core functionality (i.e., accessing on-disk states and executing APIs) is quite simple.

Existing high-performance filesystems continue to evolve and inevitably introduce more bugs. RAE is a practical approach to improve the reliability of these filesystems via a separate implementation that is more robust by being simple and slow. Slow is good when running in the alternative path and when striving for robustness and availability.

Finally, we believe that the velocity and customization benefits have more potential in the framework of running different versions of filesystems and combining them in the uncommon path, such as for failure diagnosis.

8 Related Work

In this chapter, we discuss other works related to this dissertation. We begin with recent efforts in OS architectures (§8.1) and discuss how our semi-microkernel approach compares to them. We then cover works around modern filesystems (§8.2), including new filesystem architectures for emerging devices, improvements in filesystem multi-core scalability, and other efforts in user-level filesystem development. Finally, we describe relevant works in fault tolerance (§8.3), including sources of errors, how other filesystems improve fault tolerance and reliability, and general system fault tolerance and reliability.

8.1 OS Architectures and Construction

We have discussed the earlier evolutions of microkernels (§2.1); we focus on the recent works exploring new OS architectures in this section.

Optimizing the IO path has drawn great attention in recent years in response to high-performance IO devices. Similar to our architecture, several pioneering works such as IsoStack, Snap, TAS, and Shenango [99, 134, 155, 178] explore the benefits of semi-microkernels in the network domain. Among them, Snap [134] is a user-level network stack that supports a large scale of Google's infrastructure. Compared with networking semi-microkernels, filesystems are more complex and have more cou-

pling with the host OS kernel, such as in memory management, user permissions, and process management (e.g., for permissions). This is because networking is a transport layer, while a filesystem is a storage layer that needs to retrieve data, whereas networking can forget the data (i.e., it is memoryless).

Another direction for accelerating the I/O path to overcome the Linux kernel's significant overhead is kernel-bypass libraries. These OSes, such as Arrakis [157], IX [20], ZygOS [163], and Demikernel [215], integrate hardware direct-access into the applications, offering high performance. They require other techniques such as virtualization (e.g., SR-IOV) to manage the sharing and protection of the devices across multiple applications.

In addition to the kernel-bypass library OSes in the I/O path, the Rust language inspires recent exploration of general OS architectures. Redleaf [151] explores leveraging Rust's memory safety as the isolation mechanism to build a microkernel. Theseus [24] is a single-address-space and single-privilege OS that redesigns OS runtime models and abstractions according to Rust's ownership model, allowing resource usage safety to be enforced by the language. As noted by the authors [24], rebuilding an entire OS to support modern applications takes a significant amount of time. Our work relies on the process boundary to provide isolation between the OS services and the host monolithic kernel.

General library OSes, notably unikernels [105, 133, 160], tailor and integrate the entire OS directly with applications to form single-address-space applications that are efficient to run in the cloud (e.g., on a hypervisor). The most relevant insight from these works is the compatibility issue: porting the applications and extracting the performance benefits requires significant effort and expert knowledge [105, 109] when a completely new OS is introduced. Our approach is more focused, designed specifically for one critical subsystem – the filesystem – and pro-

vides POSIX compatibility.

Another recent success in microkernels is the HongMeng OS [32], driven by industry, which took seven years and many engineers to build and has been deployed in millions of devices. The IPC frequency issues are mentioned to be much more problematic (i.e., 70x higher) when the OS becomes more general-purpose and supports applications in smartphones. The compatibility issues are also highlighted. Semi-microkernel filesystems, by design, can avoid these issues by focusing on a single subsystem and relying on the host kernel for other system services.

8.2 Modern Filesystems

uFS draws on a broad range of recent work in filesystems. We first discuss systems that explore new filesystem architectures; then we present systems that address scalability; finally, we examine related work on user-level filesystem development.

8.2.1 New Filesystem Architectures

Emergent devices (such as NVM and SSDs [90, 211]) have placed a spotlight on kernel overheads and have motivated researchers to revisit filesystem architecture. One approach is to enable applications to directly access the device via user-level libraries, sometimes bypassing a centralized and trusted entity. Because library-based solutions avoid the high cost of trapping into and out of the kernel [56, 95, 97, 107, 169, 202, 216], they generally provide high performance as compared to traditional kernel filesystems.

However, there are challenges with the library-based approach [97]. For example, to maintain filesystem integrity, the manipulation of metadata requires the involvement of a trusted entity, either to update the metadata or to validate the updates done by the library. Thus, maintain-

ing metadata integrity not only slows down metadata-intensive operations, but also complicates the write path, as metadata updates are intertwined with data operations in the traditional filesystem interface (e.g., an append to a file also changes its size).

One early example of this approach is found in Aerie [202], whose library can directly access filesystem data but is read-only for metadata; a separate trusted user-level process takes care of metadata updates and inter-process sharing via a distributed locking mechanism. Strata [107] decouples layout and access methods of different devices via data migration between media. The Strata library accelerates performance by appending to a per-process private NVM log. The library also maintains a DRAM cache for structures (such as inodes) to improve read performance and acquires leases from the trusted entity for shared-file access. ZoFS [56] offloads filesystem functionality into the user's address space, where the library can directly update any data or metadata. To enforce security and permission, ZoFS includes a cooperative protocol between trusted library instances based on Intel MPK [4], but assumes the library is trusted. Similarly, KucoFS [38] equips the library with a per-file range lock to accelerate intra-process concurrent writing to a file. The library directly translates naming into device location, such that the trusted kernel only needs validation instead of costly look-ups for metadata updates.

SplitFS [95] proposes another approach where the library handles data operations and a kernel NVM filesystem (ext4-DAX) processes metadata operations. The SplitFS library improves performance by replacing data copying with linking pages and avoiding page faults on the write path. However, it has the same performance problem for metadata operations as a kernel filesystem. NOVA [207], a DAX kernel filesystem, provides atomic filesystem operations for NVM via an atomic mmap; it optimizes device access performance through per-core structures but still suffers from kernel overhead above the VFS layer.

Finally, a different approach is to push filesystem functionality further down into the devices themselves. For example, DevFS [97] pushes the filesystem entirely into the device, thus providing direct access and serving as a centralized, trusted entity, but at a cost: the device must provide the full filesystem API – a large change from today's devices – and also be able to serve filesystem needs with limited resources (device CPU and memory). Follow-on work on CrossFS [169] distributes filesystem functionality across hardware, software, and firmware, but equires significant changes to device firmware.

A filesystem semi-microkernel differs from these approaches in that it retains the same key property of kernel-based filesystems: trust is centralized (in server software) instead of being distributed (across library, trusted process, OS, and hardware) and no special hardware is required. As such, it is relatively straightforward to implement, and can deliver scalable high performance across the entire filesystem API.

8.2.2 Filesystems and Multicore Scalability

Researchers have been studying the limitations of OS scalability [25, 43, 53, 69]; most of them find that the poor scalability of applications is primarily attributed to the OS. The kernel scalability bottleneck usually stems from some highly contended lock, leading to significant effort to introduce fine-grained locks and resolve the subsequent concurrency bugs [130]. The most recent Linux kernel filesystem scalability study [144] explores how the design of each kernel filesystem and the VFS layer affects application scalability, which leads to a conclusion of "speculating scalability is precarious." It is thus natural for a semi-microkernel like uFS to consider a scalable-by-design approach.

Clements *et al.* [43] take a principled approach by using the *scalable commutativity rule* to reason about system scalability. ScaleFS [23] follows these scalability guidelines, implementing concurrency-optimized data

structures and a per-core private operation log for durability. Scalability is also a critical design point in recent library-heavy filesystems [38, 169, 206], commonly introducing fine-grained concurrency control into the libraries. Unlike ScaleFS, uFS generally uses per-core partitioned structures (no locking needed) and a global journal (with a small critical section).

Several kernel filesystems [50, 84, 96] exploit data partitioning for better scalability. SpanFS [96] and Hare [84] partition both files and directories into cores in a static manner. uFS, instead, dynamically changes the mapping of files (excluding directories) into cores. Recent work in WAFL [50] incrementally re-architects a kernel filesystem for scalability by sharding stripes of files to cores and multi-granularity partitioning of the directory tree based on the request type. Like uFS, message passing is used for users to submit requests and communication between filesystem threads. WAFL incorporates a scheduling policy that chooses a filesystem thread with more requests into the kernel CPU scheduler, which shares the same purpose as uServer's load balancing and core allocation. The data mapping in WAFL remains static and exploits NetApp's enterprise data to accelerate the common workload scalability. uFS's dynamic data mapping mechanism relies on runtime performance monitoring, and similar optimization based on offline workload characteristics could also apply.

8.2.3 User-level Filesystems

Much efforts have been made to facilitate user-space developement of filesystems. FUSE [66] has been the *de facto* framework for user-level filesystem development. However, FUSE-based filesystems focus on functionality (e.g., ssh-based remote file access, encryption, etc.). Performance is a well-known weakness for FUSE filesystems, arising from its design. Efforts have also been made to improve FUSE performance, such as XFUSE [80],

and RFUSE [41].

Recently, Bento [142] provides user-space development and debugging without performance cost, by downloading the memory-safe filesystem directly into the kernel. Despite matching kernel filesystem performance (i.e., ext4), Bento still suffers from the performance overhead in VFS and other kernel subsystems, whereas uFS outperforms ext4 along numerous axes. Furthermore, Bento restricts the choice of language (to Rust) whereas uFS could be developed in any language framework.

8.3 Fault Tolerance

Our work to improve the fault tolerance of a filesystem semi-microkernel in Nebula and uFS-Shadow benefits from the literature on fault tolerance and reliability in systems. We first discuss the root causes of faults that our fault model is based on. Then, we discuss filesystem fault tolerance and reliability. Finally, we examine general system fault tolerance and reliability.

8.3.1 Hardware and Software Faults

Hardware corruption is real. Sridharan *et al.* [183] performed a detailed study of DRAM and SRAM faults. The study 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 [214].

More recently, Alibaba [204], Google [79] and Meta [55] reported that CPU core faults can lead to silent data corruption. Spanner, in particular, has encountered the consequences of silent corruption due to bad cores [17].

Software bugs also are prevalent, leading to crashes or memory corruption. Lu *et al.* [130] studied bug-fixing patches of several kernel filesystems, revealing that around 20% of bugs lead to machine crashes. Data corruption accounts for the largest percentage (~40%) of bug consequences, although it is unclear how many of the bugs cause in-memory (vs. on-disk) corruption. Huang *et al.* [81] studies patches of the Linux memory management subsystem. Two findings are especially relevant: first, regardless of a component's maturity, bugs are still quite common. Second, the consequences include a significant number of crashes.

8.3.2 Filesystem Fault Tolerance

Filesystem fault tolerance can be improved by enhancing detection to fail fast, avoiding further propagation. Recon [65] and WAFL [106] both check in-memory filesystem structures to detect semantic violations when committing updates to disk. Both are complementary to Nebula as they aim for error detection.

Recovering from runtime errors is another critical aspect of fault tolerance. Membrane [185] designs a restarting framework for kernel filesystems by using sophisticated techniques to unwind the threads, unmount the filesystems, and replay logged operations. However, Membrane requires additional flushing in the common path. Membrane also does not fully take advantage of a clean address space and is vulnerable to memory corruption. IceFS [131] adds failure isolation between clients; it can also prevent an error in a kernel filesystem from shutting down the entire machine. However, clients may notice data loss after recovery.

Nebula has similarities to the Rio File Cache [37], which preserves the kernel file cache and enables an automatic warm reboot when the OS kernel crashes. Rio tolerates runtime errors in the entire OS kernel but is vulnerable to corrupted semantic states in the cache that occur before the crash. Nova-Fortis [208] is a kernel NVM filesystem that tolerates

the corruption of NVM devices, highlighting and addressing the problem of memory corruption when filesystem data is in persistent media (i.e., NVM). Rio's file cache is essentially a persistent memory used by the restarted OS, and therefore, the memory corruption problem cannot be overlooked.

8.3.3 Filesystem Reliability

Improving filesystem reliability involves other important aspects in addition to fault tolerance. The main focuses are reducing the likelihood of bugs and ensuring the correctness of the filesystem. RAE requires a more reliable filesystem to handle the alternative path and can thus benefit from efforts such as testing and formal verification to achieve a more robust filesystem.

Many testing techniques and frameworks have been proposed for filesystems, such as fuzzing [100, 210] and model checking [124, 212]. Input generation is critical for testing filesystems, especially for semantic bugs. CrashMonkey [148] leverages effective heuristics about crash consistency, and DogFood [31] proposes a layered model. Our p-crash benchmark, which attempts to cover diverse workloads and each prefix sequence in one workload in evaluating Nebula and uFS-Shadow, has a similar goal of generating high-quality input and higher coverage. Testing the correctness of uFS-Shadow and ensuring that Nebula functions the same as uFS will also benefit from these techniques.

Formal verification is another approach to ensure filesystem reliability. FSCQ [35] is the first verified filesystem with crash consistency semantics. Other challenging properties like concurrency and transactions have also been verified [29, 30, 33, 218]. uFS-Shadow has interesting aspects for verification, as it eliminates many of the seemingly difficult properties for automatic verification in uFS, and a verified version of uFS-Shadow would be an interesting future work.

8.3.4 General System Fault Tolerance and Reliability

Nebula is also similar to previous efforts such as microreboot [27] and Rx [164]. However, microrebooting [27] 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). Redesigning the filesystem to be crash-only requires the filesystem's semantic states to be duplicated atomically after each operation, which is challenging to achieve correctly and may incur significant overhead due to the large amount of data. Rx [164] makes several retrying attempts from a checkpoint by altering various environmental factors (scheduling order, memory allocation, etc.) while replaying. Performing 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 [188] and Shadow Driver [189] are pioneering works that attempt to recover the kernel from a component (i.e., drivers) crash. Recovery Domains (RDs)[110] provide request-oriented recovery of the OS kernel; for instance, the effect of a system call or an interrupt can be recovered if an error occurs. RDs track and roll back the state changes of a particular request using undo logging and assume that a fault's influence is limited to a single request without affecting the rest of the kernel. RDs also leverage the recovery code path in the kernel source code, which is risky to execute if the memory is corrupted. Such techniques are effective if the error is detected at the request level before it is completed. Nebula makes more realistic assumptions about detection. Otherworld[54] *microreboots* the monolithic OS kernel and reuses the application's address space (i.e., a snapshot after the crash) in the new kernel. However, less work is done to recover the complex filesystem states inside the kernel, which is one main challenge Nebula addresses.

Some previous work on microkernels adds fault-tolerant machinery [22, 114]. Minix3 [192] builds a Reincarnation Server to restart failed pro-

cesses, but this can cause data loss when the filesystem fails [52]. CuriOS [52] stores OS service states in clients' address spaces with virtual memory-based protection to survive a service crash and support restart; however, the memory permissions and process isolation incur significant overhead. Specifically, both OSIRIS [22] and TxIPC [114] incorporate instruction-level undo logs to roll back problematic in-flight requests but cannot deal with bad states (e.g., memory corruption or buggy results) that occurred earlier than the current in-flight request.

Several works aim to improve the reliability of in-memory data structures and memory (de)allocation, and could perhaps be utilized within Nebula. The DieHard [21] memory allocator leverages replication and provides probabilistic memory protection; AMA [186] ensures that allocations in the filesystem will never fail by statically analyzing the amount of memory needed. Other approaches for protecting critical memory regions from corruption could be helpful [145, 156]. RESIN [126], which detects and mitigates memory leaks, could also be of use.

RAE draws inspiration from the concept of N-version programming [13–15] and recovery blocks [165, 176]. N-version programming, however, raises the complexity of the system and is not widely adopted due to the cost of maintaining multiple versions of the same software. Research has argued that creating diverse versions of the software is challenging, and the assumption of independence of failures has failed statistically. RAE is also similar to recovery blocks [165], where an *idealized fault tolerance component* is advocated. EnvyFS [18] realizes N-version programming in the context of filesystems, where various existing kernel filesystem implementations execute the same set of operations and defend against bugs and disk corruption via majority voting.

Using an alternative implementation for fault tolerance has been explored in other systems. Slicer [8], Google's auto-sharding system, incorporated a Backup Distributor that has reduced complexity (i.e., static

sharding) and is designed to be less error-prone. DIVA [12] proposes separating processor design into DIVA core and DIVA checkers; the checker is designed to reduce correctness concerns and complexity in the DIVA core's functionality, enabling detection and recovery from CPU errors at runtime. uFS-Shadow aims to tolerate broader types of errors, including deterministic bugs in complex stateful filesystems.

Rx-like [164] methods can also handle some deterministic errors by changing the environment of the software so that the error is no longer deterministic. However, this technique is more suitable for applications rather than system services like a filesystem, which acts more like an environment. The idea can be used to avoid deterministic hardware errors like bad cores and bad memory regions, which is an interesting direction to explore.

9 Conclusions and Future Work

In this chapter, we summarize each part of this dissertation (§9.1). We then present the lessons learned during the journey (§9.2), discuss the future work (§9.3), and finally conclude (§9.4).

9.1 Summary

This dissertation is comprised of four parts. In the first part, we explored the architectural advantage of high performance across broader workloads, addressing low latency and multi-core scalability. In the second part, we examined the architectural challenge of resource elasticity arising from the decoupled application and filesystem threads, focusing on balancing performance with CPU efficiency and adapting core numbers to changing workloads. In the third part, we investigated the architectural opportunity of fault tolerance by formulating the process crash model, introducing exit activation and other mechanisms to enable fast, robust, and seamless recovery. Finally, we proposed Robust Alternative Execution (RAE), which allows a slow but correct shadow filesystem to execute workloads that cause errors in the base high-performance filesystem, thereby improving system reliability. We now summarize each of these parts.

9.1.1 Functionality and High Performance

We designed and implemented uFS, a fully functional, high-performance, and crash-consistent user-space filesystem. We began by addressing the necessary issues for correctness, API support, and low latency with a single-threaded uFS server. uFS relies on the Storage Performance Development Kit (SPDK) to directly access ultra-fast NVMe SSDs in user space.

Three mechanisms were designed for single-thread performance, including a non-blocking filesystem thread (i.e., worker), which adopts an event-driven programming model, performing tasks such as polling and processing application requests, issuing hardware I/Os and processing completions, and handling background tasks, all in a non-blocking manner; a shared-memory-based inter-process communication (IPC), optimized for filesystem calls and modern CPU caches; and keeping kernel interaction off the critical path without compromising security.

uFS further achieved multi-core scalability through designs that allow multiple cores to operate independently as much as possible, following two principles: avoiding block-induced synchronization and separating designs for in-memory and on-disk data structures. uFS adopts a "shared-nothing" design for in-memory data structures, partitioning inodes across multiple cores so that each core can operate on its own set of inodes. Other in-memory data structures are also carefully designed to avoid blocking (e.g., locking). uFS uses a "shared-everything" design for writing to disk (i.e., crash consistency), where all workers share a single logical journal, and only a small non-blocking critical section is needed for allocating journal space, allowing multiple inflight journaling transactions. For simplicity, directory operations are handled only by a primary worker, while other operations can be handled by any worker.

To evaluate uFS, we created 32 single-operation microbenchmarks to assess various facets of uFS's performance and compared it with Linux

ext4, a well-optimized kernel filesystem. Through the microbenchmarks and macrobenchmarks (i.e., a web server and a mail server), we demonstrated that uFS performs well across a wide range of workloads, achieving low latency and excellent multi-core scalability.

9.1.2 Resource Elasticity

With optimal performance in mind, we designed and implemented the load management feature in uFS. The load management has two main goals: to control the number of cores used by uFS without compromising performance, and to dynamically adapt the number of cores to changing workloads.

Our load management mechanism is centered around a separate load management thread, which periodically collects runtime statistics from each uFS server worker and makes centralized decisions. We found that the combination of per-core effective CPU cycles (i.e., cycles spent on useful work) and congestion (i.e., request queuing delay) is an effective indicator of the load status, conveying both end-application performance and internal CPU efficiency. These two simple metrics are exported by each worker with negligible overhead, again, in a non-blocking manner.

We designed an algorithm to use the collected statistics to detect workload changes, decide the number of cores, and balance the load across cores. The algorithm predicts load status by addressing two orthogonal subproblems: load balancing – how to balance the load with a fixed number of cores, and core allocation – how many cores are needed assuming a balanced load. The predicted load status is then used for comparing the three configurations: adding one core, removing one core, and keeping the current number of cores, all with balanced load. Each worker enforces these decisions by shuffling inodes across cores.

We demonstrated the effectiveness of the load management feature using load balancing microbenchmarks and core allocation microbench-

marks, covering various changing factors of workload and combinations of workloads. Finally, we evaluated uFS under LevelDB with eight different workloads, showing that uFS outperforms Linux ext4 by a significant margin in scaled settings (by 1.3x to 4.6x) while controlling the number of cores used.

9.1.3 Process Crash Recovery via Exit Activation

In the third part, we discussed the process crash (i.e., p-crash) model of filesystem semi-microkernels and its implications, comparing it with the full system crash (i.e., s-crash) model of monolithic kernel filesystems. We explained why relevant filesystem applications cannot easily continue on a newly restarted filesystem server that relies only on s-crash recovery; this results from the state gap – the difference between the states perceived by the applications and those persisted to disk due to buffered updates.

We proposed an approach for process crash recovery, exit activation. Exit activation is code runs when a process crashes, before the failed process's memory is reclaimed by the OS. Exit activation allows the failed process's memory states (in addition to the on-disk states) and the OS to contribute to the recovery.

We built Nebula, a system that leverages exit activation to enable fast, robust, and seamless recovery from process crashes, appearing to applications as merely a small latency spike. Central to the recovery is an exit activation data structure – p-log (process log) – an in-memory log that is well-protected to ensure safe access by exit activation. The p-log is empowered by a novel algorithm AIM (Act/Ignore/Modify), which accurately captures sources of the state gap (i.e., system calls) without imposing extra behaviors on the original filesystem, such as heavyweight flushing. We found that naively replaying the filesystem calls cannot recover the proper state gap because the effects of filesystem calls are not

made durable in the same order as they are executed (and as perceived by the applications). However, ignoring some operations and modifying particular operations to another system call allows the state gap to be recovered through the original filesystem code; such transformations are also based on AIM.

Nebula incorporates a set of other mechanisms to support exit activation, especially for robustness and efficiency, including lightweight metadata protection to enhance detection, fast and safe per-core logging for common-path performance, and kernel-coordinated speculative restart that reduces recovery time by seconds. After a p-crash, control is transferred to the exit activation code before the failed process's memory is reclaimed, which rescues the p-log to the host OS, and then the fresh filesystem server replays the p-log to recover the state gap.

We systematically analyzed the consequences of the state gap problem for continued applications and evaluated Nebula's recoverability from state gaps via a benchmark suite that includes operation sequences from 24 synthetic workloads and 10 real-world application workloads (ranging in length from 77 to over 5,000 calls). We exhaustively injected p-crashes after each system call, covering a wide range of state gaps. We found that applications' reactions to state gap behaviors are diverse and unpredictable, possibly leading to severe consequences like silent data loss. We demonstrated that Nebula recovers from all the 30,000+ fault injections. Furthermore, we showed that Nebula recovers from corruptions of the failed filesystem's address space and incurs negligible overhead in the common path (less than 2% in most cases). Finally, Nebula recovery is fast ($\leqslant$500ms).

9.1.4 Robust Alternative Execution via Shadow Filesystems

In the last part, we aimed to improve the reliability of filesystems by recovering from virtually all runtime errors. The idea is to have a shadow filesystem that is slow but correct, which can execute workloads that cause errors in the base high-performance filesystem; we call this approach Robust Alternative Execution (RAE).

We studied the Linux ext4 filesystem bugs over the last 10 years and found that both transient errors and deterministic errors are prevalent. Furthermore, the number of bugs is increasing over time due to new features, optimizations, and refactoring.

We observed that the complexity of filesystems lies in the performance techniques and optimizations, such as caching, asynchronous I/O, concurrency, and crash consistency, which are more likely to introduce bugs. In contrast, the core functionality of filesystems, which only needs to understand the on-disk data structures and execute the filesystem APIs, is quite simple. Therefore, shadow filesystems can be more robust by design, omiting the features for performance.

We built uFS-Shadow, a shadow filesystem for the base uFS, which is incorporated into Nebula to replay the p-log during p-crash recovery. We showed that uFS-Shadow recovers from emulated deterministic bugs and greatly simplifies the codebase to 7% of the base uFS. The recovery time of uFS-Shadow is relatively slow, around 1.6x that of the base.

9.2 Lessons Learned

We now present the general lessons we learned while working on this dissertation.

9.2.1 Mechanisms First, and then Policies

During the design and implementation of uFS, Nebula, and uFS-Shadow, we generally followed the paradigm of building a strong and solid foundation and then adding policies on top of it.[1,2,3] In retrospect, this approach has proven beneficial, and we believe that such a practice is generally fruitful in system design and construction.

We first built the mechanisms for the no-blocking worker thread in uFS, which constructs the software in a event-driven programming model, encapsulating the event of polling, processing, and issuing I/Os, and handling IPCs. Then, policies of scheduling can be added, such as pre-fetching and batching. We also built the mechanisms of dynamically exchange inodes across cores and statistics collection before the right policies and algorithms for core allocation and load balancing.

In Nebula, the restarting mechanism is the foundation for recovery, and the exit activation data structure p-log, is the mechanism for capturing the state gap, while using base uFS or uFS-Shadow code to replay the p-log is the policy.

Such a methodology, perhaps a variant of the famous "separation of mechanisms and policies"[111], has several benefits. First, it allows us to separately understand the performance of the mechanisms (relatively stable) and the influence of the policies. The performance impact of the mechanisms tends to be less dependent on workloads, whereas policies are just the opposite.

Second, it helps to avoid premature optimizations[83]. Policy opti-

[1] "In system, it is always mechanism first." – Remzi Arpaci-Dusseau (advises during a research meeting).

[2] "You have the mechanism built, and can explore interesting policies..." – Andrea Arpaci-Dusseau (comment made during a talk about kernel interrupts in February 2020).

[3] Other said firsts concerning how to do research includes: "Most interesting fisrt" and "Simplest first". It occurs like "You know, there is depth-first search, breadth-first search, and in doing research, it is the most interesting first."

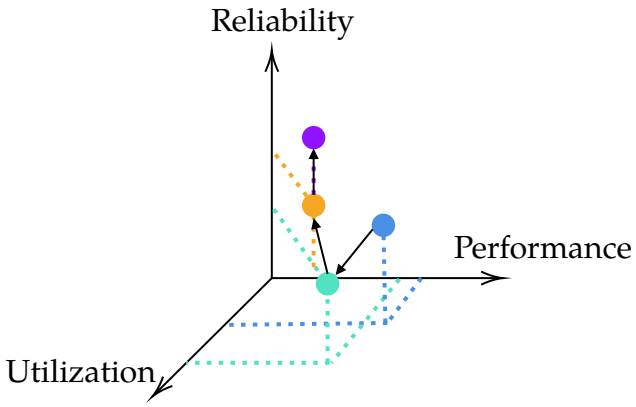


Figure 9.1: **Illusration of Three Dimensions, Trade-offs, and our Roadmap.** The arrows shows the path of §4 → §5 → §6 → §7 .

mizations play an important role in system performance [51], often imposing a number of tuning knobs that complicate the system and performance reasoning. When building a research system, it is neither practical nor necessary to enable all possible policies that might suit different workloads. Instead of chasing performance numbers by fine-tuning policies, addressing key research questions is more important; thus, policies should remain simple until approaching the core problems. Third, it helps to modularize the code, making it easier to maintain and extend, which is essential for research systems.

9.2.2 Understand the Extreme First, and then Trade-offs

Another lesson we learned is to understand the extremes first, which is perhaps more helpful for research than for production systems. For example, in the design of uFS, we first focused on understanding single-threaded performance before addressing multi-core scalability. We also examined the best possible performance before adding support for load management and CPU efficiency. This understanding of performance helped us make sense of the trade-offs involved in achieving fault toler-

ance and reliability. Finally, understanding the extreme level of simplicity in uFS-Shadow provided insights into the cost of reliability in terms of performance.

There are essentially three dimensions to consider for a system with specific functionalities: performance, utilization,[4] and reliability. Improving one dimension often comes at the cost of at least one other, and possibly both. For instance, improving utilization often comes at the expense of performance (e.g., load management for CPU efficiency). Enhancing performance typically adds complexity and reduces reliability. In contrast, improving reliability inevitably has a negative impact on performance. Reliability enhancements also require extra resources, which, in some sense, reduce utilization. For example, formally-verified systems often have lower performance, and performance optimizations is difficult to verify (e.g., asynchrony and concurrency). Figure 9.1 illustrates the works presented in this dissertation along these three dimensions.

Without understanding the extremes, it is difficult to make sense of the trade-offs we made in the design and implementation of these systems. Furthermore, focusing on and achieving one extreme can be an effective way to understand the essence of a problem, often leading to interesting research questions [154], which is beneficial for research.

9.2.3 Make Assumptions First, and then? – Some Assumptions cannot be Removed Incrementally

In doing research and building systems, we often need to make assumptions. Making assumptions is beneficial for making progress on a simplified problem. However, we believe that making assumptions is worth a second thought on the question of whether the assumption can be removed incrementally. Put another way, we need to ensure that the sim-

[4]Sharing and isolation can be regarded as forms of utilization.

plified problem is a subproblem of the final problem, but not a divergent or conflicting one.

For example, assumptions can often be incrementally removed when evaluating the performance of a given system. One can start by measuring single-threaded performance or a subset of APIs, then move on to multi-threaded performance, and finally address complex and dynamic workloads.

One interesting example we encountered is the assumption made for fault tolerance – the fault model. During our review of the literature, we found that the fault model is often not clearly defined, particularly with the notion of "fail-stop." But the question is, how fail-stop is it? It is virtually impossible to detect all faults immediately. The assumption of fail-stop is difficult to remove incrementally. Consider Nebula, if the fail-stop assumption holds, it implies that the memory states in the failed process are still intact, so the most straightforward way to recover the state gap is to directly read them from the failed process's memory. However, those memory states are fundamentally unsafe to access if the assumption does not hold (which could be the case), leading us to the trusted p-log approach. In this case, removing the assumption changes the entire problem definition.

I think that in the context of fault tolerance, assumptions are relatively more difficult to remove incrementally. The reason is that the root causes of faults are too diverse, including hardware faults, software bugs, and human errors, causing the essence of the problems to diverge. The issue with assuming fail-stop is that it does not assume what can happen, but rather assumes what cannot happen. Not to mention that the notion of fail-stop itself lacks a clear definition, as it depends on detection. We believe it is beneficial to *be explicit* – getting all of the assumptions out on the table [174]. We attempted to do so in Nebula (e.g.,Table 6.1 and Table 6.2).

9.2.4 Perspectives on Building Research Systems with a Clean Slate

Overall, building research systems with a clean slate is a rewarding experience, though more challenging than we initially thought.

First, building such a system requires a large amount of effort invested in the basic infrastructure, such as API compatibility, basic functionality, and so on. We need to address many problems that have already been solved in existing systems (e.g., challenges of event-driven programming model, page caches, etc.) before we can focus on the core research problems. In addition, much effort is also devoted to building benchmarks and the surrounding ecosystem. We developed a set of microbenchmarks and created tools to support fault injection experiments. The dilemma of velocity is that, on the one hand, building a user-space system is supposed to be easier and faster than building a kernel system, but on the other hand, we needed to start from scratch, selecting the right tools, customizing them for our needs, and building a wide range of utilities. For instance, we integrated tools like Linux perf, Intel Pin [88], and Intel PMU hardware counters [87]. We also built command-line tools for uFS and tools to check integrity.

Second, comparing apples to apples is challenging. As we have shown in this dissertation, the system has many components, and ours differs in many ways from existing systems. On the one hand, this is an opportunity to explore the design space and innovate. On the other hand, we also need to understand the performance of Linux ext4 and kernel storage stacks. The recovery machinery is also an entire system that needs to address several problems. However, some problems are dependent on the OS architectures (e.g., the restarting mechanism), some are dependent on the assumptions, and others are more general, like the state gap problem. Therefore, a more sensible approach is probably to compare according to these specific problems, rather than the entire system.

9.3 Future Work

In this section, we discuss directions that can extend the work presented in this dissertation.

9.3.1 Composable and Extensible Filesystems

Despite the benefits of customization, we did not fully explore the new approach to extending filesystems, compared with FUSE [66] and other new kernel filesystems. The issues with classic approaches like these are: i) the entire filesystem needs to be rebuilt, such as supporting all APIs together, and ii) performance components like the page cache and dentry cache must be shared by all filesystems.

It would be interesting to explore composable and extensible filesystems where the components (as shown in Figure 7.3) are built as processes that can be reused at runtime, each with its own focus, such as performance optimization for a particular workload or type of hardware. By dividing the filesystem process into several components, with multiple instances of each component, applications could choose the components they wish to use at runtime. For example, multiple buffer cache processes could be present, each with different data structures optimized for various hardware types, allowing applications to choose and switch the buffer cache they wish to use. In contrast, all filesystems in a monolithic kernel must share the same buffer cache. This is similar to the Trio architecture [217], but with an aim to make the components even more composable and extensible.

9.3.2 Formally Verified Shadow Filesystems

One interesting direction is to formally verify the shadow filesystem, uFS-Shadow. uFS-Shadow requires strong correctness guarantees, which can benefit from formal methods and recent automatic proof tools based on

Rust, such as Verus [108]. Additionally, the simplicity of the shadow filesystem simplifies the verification efforts, as difficult properties to verify – such as crash consistency, concurrency, and asynchrony – are omitted [121]. A formally verified shadow filesystem also suggests an interesting approach to bringing verified systems into practice (i.e., in the alternative path) and improving the reliability of mature systems.

9.3.3 Study and Regulate Kernel Warning

During our study of the kernel filesystem bugs in Chapter 7, we find that the Linux kernel subsystems have inconsistencies and various viewpoints on how to handle warnings [46, 49]. The tension is that, on the one hand, avoiding panic and "continue regardless" is essential for the OS kernel [116]. On the other hand, fail-fast is a good practice. There is no consensus on how to handle warnings, and what kind of warnings should be ignored, and what are their consequences. The discussion showed that different developers can have different opinions on when to use what. Studying and regulating kernel warnings will improve the reliability of Linux kernel, and we believe systematic study like fault injections, and automatic recovery procedures are beneficial.

9.3.4 Generalized Exit Activation for Other Systems

We believe that a customized logic that can be safely executed before the failed process's memory is reclaimed by the OS is a powerful mechanism that can benefit more applications. For example, in the context of databases, exit activation can be used to achieve prefix recoverability [113]. Exit activation can also be used to avoid global fail-over in in-memory distributed systems. Furthermore, such a mechanism can be used for failure triage and diagnosis.

We envision that such an investigation could start with a study of the potential benefits of seamless process crash recovery in other contexts, such as in a single node of a distributed system. For example, in what percentage of cases can process crash recovery be used to avoid global fail-over in a safe manner, while in the rest it must be escalated to global fail-over?

9.4 Closing Words

In this dissertation, we have shown that the semi-microkernel approach for filesystems can achieve high performance on modern hardware and provide better fault tolerance for applications. Overall, such an approach is promising for rapid adoption of new hardware, customizations for applications, and better fault isolation.

Nevertheless, a filesystem is a complex piece of software, requiring a considerable amount of engineering effort to build, test, maintain, and support applications. Compatibility is also important. For instance, support for `mmap` and `fork` is essential for a wide range of applications, which goes beyond API compatibility and involves deep coupling with the host OS kernel (e.g., process management, memory management, etc.). Resource elasticity in uFS is one example where semi-microkernels need to take charge of CPU scheduling [134, 155].

Therefore, this approach is most suitable for cases where high performance and fault isolation on a single machine are important, and many applications can benefit from it, weighing the benefits against the engineering efforts. These could include powerful desktops, large-scale cloud VMs running small or medium applications, or a local filesystem serving as a backend for a distributed filesystem [9]. One interesting scenario is when reliability is critical, such as in vehicles [32], where a filesystem error not crashing the kernel is a life-saving property.

The demand for extending system services offered by the OS kernel is real, driven by the increasing volume of applications, the diversity of workloads, and modern hardware [86, 177]. The need for extending such services in a safe manner that does not easily crash the OS kernel is also significant, as evidenced by the prevalence of eBPF frameworks [161], which are referred to as another form of microkernel. Additionally, the demand for improving the reliability of the Linux kernel is real, as its complexity continues to increase [112]. We believe the techniques and mechanisms we have developed could be beneficial for these efforts to extend OS kernels.

Bibliography

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

[2] use-after-free flaw was found in fs/ext4/namei.c:dx_insert_block(). https://access.redhat.com/security/cve/cve-2022-1184.

[3] Folly: Facebook Open-source Library. https://github.com/facebook/folly.git, 2020.

[4] Intel® 64 and IA-32 Architectures Software Developer's Manual. https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html, 2020.

[5] Optimize ext4 file overwrites - perf improvement. https://lore.kernel.org/linux-ext4/cover.1600401668.git.riteshh@linux.ibm.com/, 2020.

[6] uFS Code and Benchmarks. https://research.cs.wisc.edu/adsl/Software/uFS/, 2021.

[7] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A

new kernel foundation for unix development. In *Proceedings of the USENIX Summer Technical Conference* (*USENIX Summer '86*), Atlanta, GA, June 1986.

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

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

[10] Nadav Amit, Amy Tai, and Michael Wei. Don't shoot down tlb shootdowns! In *Proceedings of the EuroSys Conference* (*EuroSys '20*), Heraklion, Greece, April 2020.

[11] Hagit Attiya, Eshcar Hillel, and Alessia Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. In *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 69–78, 2009.

[12] Todd M. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture* (*MICRO'99*), Haifa, Israel, nov 1999.

[13] Algirdas A. Avižienis. The Methodology of N-Version Programming. In Michael R. Lyu, editor, *Software Fault Tolerance*, chapter 2. John Wiley & Sons Ltd., 1995.

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

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

[16] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[17] 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.

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

[19] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, mar 2017.

[20] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dat-

aplane Operating System for High Throughput and Low Latency. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation* (*OSDI '14*), Broomfield, CO, October 2014.

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

[22] 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.

[23] 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.

[24] 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.

[25] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation* (*OSDI '10*), Vancouver, Canada, December 2010.

[26] 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.

[27] 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.

[28] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies* (*FAST '20*), Virtual conference, February 2020.

[29] Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai Zeldovich. Gojournal: a verified, concurrent, crash-safe journaling system. In *Proceedings of the 15th USENIX Conference on Operating Systems Design and Implementation* (*OSDI '21*), Virtual Conference, July 2021.

[30] Tej Chajed, Joseph Tassarotti, Mark Theng, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying the DaisyNFS concurrent and crash-safe file system with sequential reasoning. In *Proceedings of the 16th USENIX Conference on Operating Systems Design and Implementation* (*OSDI '22*), Carlsbad, CA, July 2022.

[31] 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.

[32] 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.

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

[34] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. Linux kernel vulnerabilities: state-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, 2011.

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

[36] J. Bradley Chen and Brian Bershad. The Impact of Operating System Structure on Memory System Performance. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles* (*SOSP '93*), Asheville, North Carolina, December 1993.

[37] 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.

[38] Youmin Chen, Youyou Lu, Bohong Zhu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiwu Shu. Scalable Persistent Memory File System with Kernel-Userspace Collaboration. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies* (*FAST '21*), Virtual conference, February 2021.

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

[40] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (*SOSP '13*), Nemacolin Woodlands Resort, Farmington, Pennsylvania, October 2013.

[41] Kyu-Jin Cho, Jaewon Choi, Hyungjoon Kwon, and Jin-Soo Kim. RFUSE: Modernizing userspace filesystem framework through scalable Kernel-Userspace communication. In *Proceedings of the 22nd USENIX Conference on File and Storage Technologies* (*FAST '24*), Santa Clara, CA, February 2024.

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

[43] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The Scalable Commutativity

Rule: Designing Scalable Software for Multicore Processors. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (*SOSP '13*), Nemacolin Woodlands Resort, Farmington, Pennsylvania, October 2013.

[44] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing* (*SOCC '10*), Indianapolis, IA, June 2010.

[45] Jonathan Corbet. Rust and C filesystem APIs. `https://lwn.net/Articles/958072/`.

[46] Jonathan Corbet. Warning about WARN_ON(). `https://lwn.net/Articles/969923/`.

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

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

[49] Jonathan Corbet. What to do in response to a kernel warning. `https://lwn.net/Articles/876209/`, 2021.

[50] Matthew Curtis-Maury, Vinay Devadas, Vania Fang, and Aditya Kulkarni. To waffinity and beyond: A scalable architecture for incremental parallelization of file system code. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (*OSDI '16*), Savannah, GA, November 2016.

[51] Cody Cutler, M. Frans Kaashoek, and Robert T. Morris. The benefits and costs of writing a POSIX kernel in a high-level language. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (*OSDI '18*), Carlsbad, CA, October 2018.

[52] 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.

[53] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (*SOSP '13*), Nemacolin Woodlands Resort, Farmington, Pennsylvania, October 2013.

[54] 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.

[55] 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.

[56] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. Performance and Protection in the ZoFS User-Space NVM File System. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles* (*SOSP '19*), Ontario, Canada, October 2019.

[57] Fred Douglis and John K. Ousterhout. Process Migration in the Sprite Operating System. In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 18–25, Berlin, West Germany, September 1987. IEEE.

[58] DPDK Development Team. Data Plane Development Kit. `https://www.dpdk.org/`, 2021.

[59] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. Using continuations to implement thread management and communication in operating systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (*SOSP '91*), Pacific Grove, CA, October 1991.

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

[61] Kevin Elphinstone and Gernot Heiser. From l3 to sel4 what have we learnt in 20 years of l4 microkernels? In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (*SOSP '13*), Nemacolin Woodlands Resort, Farmington, Pennsylvania, October 2013.

[62] NVM Express. NVM Express Specifications. `https://nvmexpress.org/specifications/`.

[63] 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.

[64] 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.

[65] 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.

[66] FUSE. Linux FUSE (Filesystem in Userspace) interface. `https://github.com/libfuse/libfuse`.

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

[68] Sanjay Ghemawhat, Jeff Dean, Chris Mumford, David Grogan, and Victor Costan. LevelDB. https://github.com/google/leveldb, 2011.

[69] Hossein Golestani, Amirhossein Mirhosseini, and Thomas F Wenisch. Software Data Planes: You Can't Always Spin to Win. In *Proceedings of the ACM Symposium on Cloud Computing* (*SOCC '19*), Santa Cruz, California, November 2019.

[70] Google. Fuchsia Zircon Kernel. `https://fuchsia.dev/fuchsia-src/concepts/kernel?hl=en`.

[71] Redha Gouicem, Damien Carver, Jean-Pierre Lozi, Julien Sopena, Baptiste Lepers, Willy Zwaenepoel, Nicolas Palix, Julia Lawall, and Gilles Muller. Fewer cores, more hertz: Leveraging High-Frequency cores in the OS scheduler for improved application performance. In *Proceedings of the USENIX Annual Technical Conference* (*USENIX '20*), Virtual Conference, June 2020.

[72] Cary G. Gray and David R. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles* (*SOSP '89*), Litchfield Park, Arizona, December 1989.

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

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

[75] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (*SOSP '11*), Cascais, Portugal, October 2011.

[76] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of µ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles* (*SOSP '97*), Saint-Malo, France, October 1997.

[77] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Minix 3: a highly reliable, self-repairing operating system. *SIGOPS Oper. Syst. Rev.*, 40(3):80–89, jul 2006.

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

[79] 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.

[80] Qianbo Huai, Windsor Hsu, Jiwei Lu, Hao Liang, Haobo Xu, and Wei Chen. XFUSE: An Infrastructure for Running Filesystem Services in User Space. In *Proceedings of the USENIX Annual Technical Conference* (*USENIX '21*), Virtual Conference, July 2021.

[81] 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.

[82] 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.

[83] Randall Hyde. The fallacy of premature optimization, 2006.

[84] Charles Gruenwald III, Filippo Sironi, M. Frans Kaashoek, and Nickolai Zeldovich. Hare: a file system for non-cache-coherent multicores. In *Proceedings of the EuroSys Conference* (*EuroSys '15*), Bordeaux, France, April 2015.

[85] Intel Inc. What is pcie 4 and why does it matter? `https://www.intel.com/content/www/us/en/gaming/resources/what-is-pcie-4-and-why-does-it-matter.html`.

[86] Samsung Inc. Samsung z-ssd sz985. `https://semiconductor.samsung.com/news-events/tech-blog/samsung-z-ssd-sz985/`.

[87] Intel. Performance Counter Monitor. `https://www.intel.com/content/www/us/en/developer/articles/tool/performance-counter-monitor.html`.

[88] Intel. Pin - A Dynamic Binary Instrumentation Tool. `https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html`.

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

[90] Intel Corporation. Intel Optane SSD 905P Series Specification. https://ark.intel.com/content/www/us/en/ark/products/series/129835/intel-optane-ssd-905p-series.html, 2018.

[91] Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '94, pages 151–160, 1994.

[92] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a highly scalable user-level TCP stack for multicore systems. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation* (*NSDI '14*), Seattle, WA, April 2014.

[93] Jonathan Corbet. Bugs and fixes in the kernel history. https://lwn.net/Articles/914632/.

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

[95] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles* (*SOSP '19*), Ontario, Canada, October 2019.

[96] Junbin Kang, Benlong Zhang, Tianyu Wo, Weiren Yu, Lian Du, Shuai Ma, and Jinpeng Huai. SpanFS: A Scalable File System on Fast Storage Devices. In *Proceedings of the USENIX Annual Technical Conference* (*USENIX '15*), Santa Clara, California, June 2015.

[97] Sudarsun Kannan, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Yuangang Wang, Jun Xu, and Gopinath Palani. Design-

ing a True Direct-Access File System with DevFS. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies* (*FAST '18*), Oakland, CA, February 2018.

[98] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. Failure Sketching: A Technique for Automated Root Cause Diagnosis of in-Production Failures. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles* (*SOSP '15*), Monterey, California, October 2015.

[99] 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.

[100] 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.

[101] 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.

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

[103] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Man-

gard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. *Communications of the ACM*, 63(7):93–101, 2020.

[104] Sungjoon Koh, Changrim Lee, Miryeong Kwon, and Myoungsoo Jung. Exploring system challenges of Ultra-Low latency solid state drives. In *10th USENIX Workshop on Hot Topics in Storage and File Systems* (*HotStorage '18*), Boston, MA, July 2018.

[105] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. Unikraft: fast, specialized unikernels the easy way. In *Proceedings of the EuroSys Conference* (*EuroSys '21*), Virtual Event, April 2021.

[106] 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.

[107] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A Cross Media File System. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles* (*SOSP '17*), Shangai, China, October 2017.

[108] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying rust programs using linear ghost types. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):286–315, 2023.

[109] Hugo Lefeuvre, Gaulthier Gain, Vlad-Andrei Bădoiu, Daniel Dinca, Vlad-Radu Schiller, Costin Raiciu, Felipe Huici, and Pierre

Olivier. Loupe: Driving the Development of OS Compatibility Layers. In *Proceedings of the 29th International Conference on Architectural Support for Programming Languages and Operating Systems* (*ASPLOS '24*), La Jolla, CA, USA, April 2024.

[110] 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.

[111] Roy Levin, Ellis Cohen, William Corwin, Fred Pollack, and William Wulf. Policy/mechanism separation in hydra. In *Proceedings of the 5th ACM Symposium on Operating Systems Principles* (*SOSP '75*), Austin, Texas, November 1975.

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

[113] Tianyu Li, Badrish Chandramouli, Jose M Faleiro, Samuel Madden, and Donald Kossmann. Asynchronous prefix recoverability for fast distributed stores. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1090–1102, 2021.

[114] 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.

[115] Jochen Liedtke. On micro-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (*SOSP '95*), pages 237–250, Copper Mountain Resort, CO, December 1995.

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

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

[118] Linux Kernel Organization, Inc. Linux IRQ handling. https://www.kernel.org/doc/html/v5.4/core-api/genericirq.html.

[119] Linux Kernel Organization, Inc. Linux Page Cache. https://www.kernel.org/doc/html/next/mm/page_cache.html.

[120] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg, and Raoul Strackx. Meltdown: reading kernel memory from user space. *Commun. ACM*, 63(6):46–56, may 2020.

[121] 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, page 15–22, 2024.

[122] 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.

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

[124] Yifei Liu, Manish Adkar, Gerard Holzmann, Geoff Kuenning, Pei Liu, Scott A. Smolka, Wei Su, and Erez Zadok. Metis: File System Model Checking via Versatile Input and State Exploration. In *Proceedings of the 22nd USENIX Conference on File and Storage Technologies* (*FAST '24*), Santa Clara, CA, February 2024.

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

[126] Chang Lou, Cong Chen, Peng Huang, Yingnong Dang, Si Qin, Xinsheng Yang, Xukun Li, Qingwei Lin, and Murali Chintalapati. RESIN: A Holistic Service for Dealing with Memory Leaks in Production Cloud Infrastructure. In *Proceedings of the 16th USENIX Conference on Operating Systems Design and Implementation* (*OSDI '22*), Carlsbad, CA, July 2022.

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

[128] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The linux scheduler: a decade of wasted cores. In *Proceedings of the EuroSys Conference* (*EuroSys '16*), London, United Kingdom, April 2016.

[129] LTP Team. The Linux LTP Project. `http://linux-test-project.github.io`, 2021.

[130] 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.

[131] 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.

[132] 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.

[133] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: library operating systems for the cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 461–472, New York, NY, USA, 2013. Association for Computing Machinery.

[134] 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.

[135] Chris Mason. The Btrfs Filesystem. `oss.oracle.com/projects/btrfs/dist/documentation/btrfs-ukuug.pdf`, September 2007.

[136] 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.

[137] Nicholas D Matsakis and Felix S Klock. The rust language. *ACM SIGAda Ada Letters*, 34(3):103–104, 2014.

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

[139] 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.

[140] Frank McSherry, Michael Isard, and Derek G. Murray. Scalability! But at what COST? In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems (HOTOS '15)*, Kartause Ittingen, Switzerland, May 2015.

[141] James D Meindl. Beyond Moore's Law: The Interconnect Era. *Computing in Science & Engineering*, 5(1):20–24, 2003.

[142] Samantha Miller, Kaiyuan Zhang, Mengqi Chen, Ryan Jennings, Ang Chen, Danyang Zhuo, and Thomas E. Anderson. High Velocity Kernel File Systems with Bento. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST '21)*, Virtual conference, February 2021.

[143] Samantha Miller, Kaiyuan Zhang, Danyang Zhuo, Shibin Xu, Arvind Krishnamurthy, and Thomas E. Anderson. Practical safe linux kernel extensibility. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HOTOS '19)*, Bertinoro, Italy, May 2019.

[144] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. Understanding Manycore Scalability of File Systems. In *Proceedings of the USENIX Annual Technical Conference* (*USENIX '16*), Denver, CO, June 2016.

[145] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *Proceedings of the USENIX Annual Technical Conference* (*USENIX '13*), San Jose, CA, June 2013.

[146] Jeffrey Mogul and K.K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-driven Kernel. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation* (*OSDI '96*), Seattle, WA, October 1996.

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

[148] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (*OSDI '18*), Carlsbad, CA, October 2018.

[149] MongoDB. MongoDB. `https://www.mongodb.org/`.

[150] MongoDB. MongoDB WiredTiger. `https://docs.mongodb.org/manual/core/wiredtiger/`.

[151] 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 *Proceed-*

ings of the 14th USENIX Conference on Operating Systems Design and Implementation (*OSDI '20*), Virtual Conference, November 2020.

[152] 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.

[153] 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.

[154] Diego Ongaro. *Consensus: Bridging theory and practice*. Stanford University, 2014.

[155] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation* (*NSDI '19*), Boston, MA, February 2019.

[156] Karthik Pattabiraman, Vinod Grover, and Benjamin G. Zorn. Samurai: Protecting Critical Data in Unsafe Languages. In *Proceedings of the EuroSys Conference* (*EuroSys '08*), Glasgow, Scotland UK, March 2008.

[157] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe.

Arrakis: The operating system is the control plane. *ACM Trans. Comput. Syst.*, 33(4), nov 2015.

[158] 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.

[159] The Next Platform. Ampere readies 256-core cpu beast, awaits the ai inference wave. `https://www.nextplatform.com/2024/04/16/ampere-readies-256-core-cpu-beast-awaits-the-ai-inference-wave/`.

[160] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the library os from the top down. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*, Newport Beach, CA, March 2011.

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

[162] 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.

[163] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks.

In *Proceedings of the 26th ACM Symposium on Operating Systems Principles* (*SOSP '17*), Shangai, China, October 2017.

[164] 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.

[165] Brian Randell and Jie Xu. The evolution of the recovery block concept. *Software fault tolerance*, 3:1–22, 1995.

[166] 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.

[167] 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.

[168] Anthony Rebello, Yuvraj Patel, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Can Applications Recover from fsync Failures? In *Proceedings of the USENIX Annual Technical Conference* (*USENIX '20*), Virtual Conference, June 2020.

[169] Yujie Ren, Changwoo Min, and Sudarsun Kannan. CrossFS: A Cross-layered Direct-Access File System. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation* (*OSDI '20*), Virtual Conference, November 2020.

[170] 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.

[171] Luigi Rizzo. netmap: A novel framework for fast packet I/O. In *Proceedings of the USENIX Annual Technical Conference* (*USENIX '12*), Boston, MA, June 2012.

[172] Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. Ffwd: Delegation is (much) faster than you think. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles* (*SOSP '17*), Shangai, China, October 2017.

[173] 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.

[174] Jerome Saltzer and M Frans Kaashoek. *Principles of Computer System Design: an Introduction*. Morgan Kaufmann, 2009.

[175] 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.

[176] Lui Sha et al. Using simplicity to control complexity. *IEEE Software*, 18(4):20–28, 2001.

[177] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in

the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *Proceedings of the USENIX Annual Technical Conference* (*USENIX '20*), Virtual Conference, June 2020.

[178] Leah Shalev, Julian Satran, Eran Borovik, and Muli Ben-Yehuda. IsoStack – Highly Efficient Network Processing on Dedicated Cores. In *Proceedings of the USENIX Annual Technical Conference* (*USENIX '10*), Boston, MA, June 2010.

[179] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems* (*ASPLOS X*), San Jose, CA, October 2002.

[180] Livio Soares and Michael Stumm. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation* (*OSDI '10*), Vancouver, Canada, December 2010.

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

[182] SQLite. SQLite transactional SQL database engine. `http://www.sqlite.org/`.

[183] 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.

[184] Michael Stonebraker. *Readings in Database Systems*. Morgan-Kaufmann, 1994.

[185] 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.

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

[187] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS File System. In *Proceedings of the USENIX Annual Technical Conference* (*USENIX '96*), San Diego, CA, January 1996.

[188] 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.

[189] 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.

[190] Amy Tai, Igor Smolyar, Michael Wei, and Dan Tsafrir. Optimizing storage performance with calibrated interrupts. In *Proceedings of the 15th USENIX Conference on Operating Systems Design and Implementation* (*OSDI '21*), Virtual Conference, July 2021.

[191] Andrew Tanenbaum, Raja Appuswamy, Herbert Bos, Lorenzo Cavallaro, Cristiano Giuffrida, Tomáš Hrubỳ, Jorrit Herder, and ERIK VAN DER. Minix 3: status report and current research. *login: The USENIX Magazine*, 2010.

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

[193] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A Flexible Framework for File System Benchmarking. *USENIX; login*, 41(1):6–12, 2016.

[194] Thomas N Theis and H-S Philip Wong. The End of Moore's Law: A New Beginning for Information Technology. *Computing in science & engineering*, 19(2):41–50, 2017.

[195] Linus Torvalds. Linux 2.6.29. `https://lkml.org/lkml/2009/3/25/632`.

[196] Avishay Traeger, Erez Zadok, Nikolai Joukov, and Charles P Wright. A nine year study of file system and storage benchmarking. *ACM Transactions on Storage (TOS)*, 4(2):1–56, 2008.

[197] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E. Porter. A Study of Modern Linux API Usage and Compatibility: What to Support When You're Supporting. In *Proceedings of the EuroSys Conference (EuroSys '16)*, London, United Kingdom, April 2016.

[198] Chia-Che Tsai, Yang Zhan, Jayashree Reddy, Yizheng Jiao, Tao Zhang, and Donald E. Porter. How to Get More Value From Your File System Directory Cache. In *Proceedings of the 25th ACM Sym-*

posium on Operating Systems Principles (*SOSP '15*), Monterey, California, October 2015.

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

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

[201] 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.

[202] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. Aerie: Flexible File-system Interfaces to Storage-class Memory. In *Proceedings of the EuroSys Conference* (*EuroSys '14*), Amsterdam, The Netherlands, April 2014.

[203] Robert Wahbe, Steven Lucco, Thomas Anderson, and Susan Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles* (*SOSP '93*), pages 203–216, Asheville, North Carolina, December 1993.

[204] 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.

[205] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding.

MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters. In *Proceedings of the 19th Symposium on Networked Systems Design and Implementation* (*NSDI '22*), Renton, WA, April 2022.

[206] Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson. Finding and Fixing Performance Pathologies in Persistent Memory Software Stacks. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems* (*ASPLOS '19*), Providence, RI, USA, April 2019.

[207] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies* (*FAST '16*), Santa Clara, CA, February 2016.

[208] 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.

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

[210] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing file systems via two-dimensional input space exploration. In *2019 IEEE Symposium on Security and Privacy* (*SP*), pages 818–834. IEEE, 2019.

[211] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An Empirical Guide to the Behavior and Use

of Scalable Persistent Memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies* (*FAST '20*), Virtual conference, February 2020.

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

[213] 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.

[214] 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.

[215] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, et al. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (*SOSP '21*), Virtual Event, Germany, October 2021.

[216] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies* (*FAST '19*), Boston, Massachusetts, February 2019.

[217] Diyu Zhou, Vojtech Aschenbrenner, Tao Lyu, Jian Zhang, Sudarsun Kannan, and Sanidhya Kashyap. Enabling high-performance and secure userspace nvm file systems with the trio architecture. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 150–165, Koblenz, Germany, October 2023.

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

[219] Khawar M. Zuberi, Padmanabhan Pillai, and Kang G. Shin. Emeralds: a small-memory real-time microkernel. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles* (*SOSP '99*), Kiawah Island Resort, South Carolina, December 1999.