Correctness and Performance with Declarative System Calls

by

Anthony Rebello

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: September 20th, 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
Shivaram Venkataraman, Assistant Professor, Computer Sciences
Matthew D. Sinclair, Assistant Professor, Computer Sciences and
Affiliate, Electrical & Computer Engineering

To my parents and sister,
for their unwavering support and the confidence they instilled in me.

Acknowledgments

As a believer in Christ, I am grateful to Him for blessing me with the opportunities, experiences, and people (mentioned below) that led to completing this dissertation.

This endeavor would not have been possible without the support of my advisors, Andrea and Remzi Arpaci-Dusseau. Although I initially entered UW–Madison to pursue an MS, they graciously allowed me the flexibility to extend my studies with a fifth semester, giving me a valuable "test run" before committing to the PhD track—a decision that ultimately gave me the confidence to continue.

Andrea and Remzi have consistently encouraged my enthusiasm for system and tool building, even when it wasn't strictly necessary for a project, understanding that it made the journey more enjoyable for me. In our weekly meetings over the years, they have helped me identify and develop my strengths while offering actionable feedback that has made me a better researcher. Andrea's meticulous feedback on every draft has greatly enhanced my ability to structure and write research articles, while Remzi's thoughtful critique has sharpened my presentation skills. Together, they've always encouraged me to dig deeper and follow rabbit holes, a characteristic of the process that I've thoroughly enjoyed.

I am especially grateful for the welcoming environment they fostered, where I felt comfortable asking any question—whether unusual, seemingly

trivial, or tangential—and for their patience and kindness in enduring them all. Thank you, Andrea and Remzi, for your patience, kindness, and encouragement. I am truly grateful to have you both as my advisors.

I am also grateful to my committee: Michael Swift, Shivaram Venkataraman, and Matthew Sinclair, for their contributions throughout this process. Mike's detailed feedback on my dissertation significantly helped to refine and improve my work. Shivaram provided me with the opportunity to teach the CS537 Operating Systems class, an experience that helped me realize my passion for teaching. His interesting questions and insightful feedback also contributed to the development of my research. Matt's questions and remarks during my defense helped me understand the broader scope of my work.

I am particularly thankful to the friends I made during the MS program starting in 2017: Om Bathija, Gautham Sunjay, Nayela Zeba, Shruthi Nambiar, and Karthik Chandrashekar. You have all been pillars of support that I could lean on during difficult times. I have enjoyed our fun get-togethers and look forward to more of them in the future. A special thanks to Om Bathija—the world's best roommate—for nudging me to have that first conversation with Remzi!

When many of my friends graduated with their master's degrees, I found myself needing to build new connections as I continued with my PhD. I am grateful to the friends I made during this time in the department: Arjun Singhvi, Anjali, Kartik Sreenivasan, Vinay Banakar, and Suchita Pati. Arjun, thank you for both technical and fun conversations over chai and biryani, for reading my drafts, listening to my practice talks, and always looking out for me. As my intern host, you ensured I was set up for success in the industry by highlighting my achievements and guiding me throughout the process. Your support has meant the world to me. Also, I'm still taking good care of your car. Anjali, Kartik, and Suchita, for the countless conversations, trying out all the social activities Madison has to

offer, and for always showing up to celebrate each other's milestones in the PhD program. From celebrating our wins together to being there for one another during the difficult times, your friendship has made this journey unforgettable. Vinay, you joined the program at a time when I already had a set of friends and didn't think I'd make many more (I can be quite reserved like that). But you, being such a social and kind person, made it happen naturally. I'm so happy it worked out because I've truly enjoyed our friendship. You've always kept me in the know, been the bridge to a whole new set of friends in Madison, and, most importantly, you've been a wonderful friend throughout this journey.

To the incredible members of my research group at UW Madison—Abigail Matthews, Aishwarya Ganesan, Chenhao Ye, Guanzhou Hu, Jing Liu, John Shawger, Kai Mast, Kaiwei Tu, Kan Wu, Keren Chen, Ramnatthan Alagappan, Ruohui Wang, Sambhav Satija, Shawn Zhong, Suyan Qu, Tyler Harter, Vinay Banakar, Vojtech Aschenbrenner, Yifan Dai, and Yuvraj Patel—thank you for your invaluable feedback on my research during our group meetings and for the fun, engaging hallway discussions. I'm grateful to have been part of such a supportive and collaborative environment.

I would like to express my deepest appreciation to the seniors and alumni of my research group: Aishwarya Ganesan, Jing Liu, Kan Wu, Ramnatthan Alagappan, Tyler Harter, and Yuvraj Patel. Alongside Andrea and Remzi, Ram and Aishwarya have been incredibly generous with their time, acting as advisors-in-support even after they graduated. From the time I was considering pursuing a PhD to my next steps after completing it, they've always offered thoughtful advice and kept me in mind for opportunities. Jing has been an amazing office mate, co-author, and friend. I'm grateful for our time spent critiquing each other's ideas, practicing presentations, and fun conversations.

To my colleagues in the broader systems group—Anjali, Arjun Bala-

subramanian, Ashwin Poduval, Bijan Tabatabai, Deepak Sirone, Hayden Coffey, Konstantinos Kanellis, Mark Mansi, Shyam Murthy, and Sujay Yadalam—along with many others mentioned earlier—thank you for the engaging weekly systems meetings and the spontaneous conversations by the water cooler or while passing by each other's offices. It was always interesting to hear your different perspectives, as we understood each other's work but had unique areas of focus, leading to some really fun and thought-provoking discussions.

To the staff at the Computer Sciences department, thank you for your support throughout my time at the university. A special thanks to Angela Thorp, the graduate program manager, who ensured that even when I was focused on meeting tight deadlines and could easily overlook procedures, I had the right resources and guidance to meet all my milestones without complications.

Timothy Boals, Kate Watkin, Barbara Sella, Maria Joseph, Sushilika Salvi, Tanvi Tavarna, and Ruchira Ghosh, thank you for helping me maintain a balance between work and relaxation. During times when deadlines kept me from flying home for the holidays, I'm especially grateful for the holiday lunches and dinners I got to share with you. You all made my stay in Madison fun-filled and truly memorable, and I'm so happy to have made such wonderful friends in the community outside of the university setting.

To the incredible people I've had the pleasure of interning with in industry—Naveen Sharma, Hassan Wassel, Dan Gibson, and Ian Rae from Google; Annie Foong and Jeremy Tang from Intel—I'm grateful to have had some of you as friends, mentors, or both. You made my time spent in industry not only enjoyable but also rewarding, helping me improve my skills while ensuring I truly enjoyed the experience. The experience was so energizing that it recharged me when resuming my PhD work after the summer.

Abhishek Shivanna, Arisa Paul, Prashanth Ellina, and Rohan Ramanath, even though you've been miles away, you've always stayed in touch, checked in on me, and been a positive influence in my life. In times of difficulty, you've never hesitated to lend a helping hand, and I am so grateful for your unwavering support. Prashanth, you were one of my early mentors, and you helped me realize my love for systems, showing me what I could do with my skills beyond just solving programming puzzles. Abhishek, you've been a friend and senior who has looked out for me since high school, always sharing opportunities and fostering a fun, healthy competition that has pushed both of us to improve. I'm incredibly fortunate to have you all in my life, and your friendship means the world to me.

Last, but certainly not least, I thank my family for their unwavering support. My parents worked tirelessly to ensure I had the freedom to pursue the field I love, recognizing my strengths early on and always offering the encouragement I needed to hone the skills I was passionate about. Gitanjali, thank you for being an incredible elder sister and my role model. You sparked my interest in programming and taught me about for loops—I now use them frequently! I also want to thank my brother-in-law, Karan Ramakrishna, for your support and for being a positive force during my PhD journey, reminding me to take time to enjoy the experience and not let the demands of the PhD overshadow everything else in life. I am forever grateful for all of you.

Contents

List of Tables

List of Figures

Abstract

Modern applications, capable of running on a range of devices from servers to smartphones, require persistent data storage to maintain state across restarts, power loss, or crashes. Applications rely on system calls provided by the operating system, particularly the file system, to achieve this. However, with devices running various operating systems and file systems, developers aim for portability, writing code that can run across multiple platforms without modification.

While compilers do a great deal for user-space code, translating it efficiently for different processors, they offer little when it comes to system calls, typically performing a basic one-to-one translation. Subtle differences in internal system behavior, especially during failures, can affect the system's state and lead to incorrect outcomes, potentially causing data loss or other critical issues. These inconsistencies also introduce variations in performance across platforms. As a result, applications may experience critical failures or data loss when assumptions about system behavior are incorrect, and may also run inefficiently due to performance variations.

In this dissertation, we aim to bring the benefits that compilers provide to user-space applications—such as ensuring that optimizations preserve semantic correctness while selecting the most efficient instructions for different processors—to interactions between applications and the file system. We begin by challenging the assumption that system calls behave

consistently across platforms. Through a detailed study of `fsync` failures, we demonstrate that incorrect assumptions can result in data loss and show the necessity for file-system-specific error handling.

Next, we examine application interactions with file systems, discovering common patterns, which we refer to as *intentions*. Finally, leveraging these insights, we develop HSL (The High-Level System Language), a declarative language that allows developers to express these intentions. HSL handles failures at a per-file-system level to prevent data loss and ensure correctness. Additionally, it selects the most appropriate system calls based on runtime information, such as the target file system, to achieve optimal performance across diverse platforms.

1 Introduction

The applications we interact with on a daily basis do not run directly on hardware. Devices such as servers, desktops, laptops, tablets, and phones host multiple applications that require access to resources like the processor, memory, storage, and network. To manage this resource sharing, an underlying system is needed. This is where the operating system comes in, serving as a mediator between the applications and the hardware [4]. Applications interact with the operating system through system calls [59, 60], which allow them to request the resources they need.

Many applications require their data to be persisted, allowing recovery after a crash or power loss. This process involves accessing storage media like hard drives or SD cards, where the operating system plays a crucial role. A specific subcomponent of the operating system, known as the file system, is responsible for managing the storage of data. The file system exposes the abstraction of files and directories, enabling applications to organize and persist their data. Applications interact with the file system through system calls, requesting the reading and writing of files on the storage media.

Operating systems often support multiple file systems, each with its own characteristics and features. For example, a single operating system might support file systems like ext4, XFS, or Btrfs [72, 98, 113]; the Linux kernel documents over 20 different file systems [58]. These file systems may differ in durability, how they handle failures, and performance, but all

provide the same basic abstraction of files and directories for applications to interact with.

A web browser provides a common example of an application that interacts with the file system [47]. While primarily used for browsing, browsers also save state—such as cookies, bookmarks, and browsing history—so users can pick up where they left off. Browsers write state to disk during operation and read it upon startup. Additionally, when users download files, browsers write them to disk. Some browsers are designed for specific operating systems (e.g., Safari for macOS), while others (e.g., Firefox, Chrome) are cross-platform and designed for portability across different operating systems.

Beyond web browsers, many other applications (e.g., databases, spreadsheets, word processors) are designed to run on multiple platforms. Some of these applications, including browsers, may rely on embedded databases to manage their stored data. For developers aiming to reach as wide an audience as possible, writing portable software is essential, ensuring their applications and dependencies run seamlessly on different systems.

In modern software development, compilers play a crucial role in ensuring portability. Developers write code in high-level languages, and the compiler translates this code into machine instructions for specific systems. Compilers guarantee semantic correctness and provide numerous optimizations, ensuring that even poorly written code can perform efficiently. However, compilers are limited when it comes to handling system calls, which are necessary for interacting with the underlying operating system.

Different operating systems expose different system call interfaces for tasks such as file reading. The Portable Operating System Interface (POSIX) [64] standardizes these interfaces, enabling developers to write portable code. While POSIX guarantees certain functionalities, it hides the internal workings of the operating system and their numerous file systems, preventing fine-tuned optimizations. As a result, a portable implemen-

tation might not perform as well as one tailored to a specific operating system and file system. Moreover, without insight into the underlying details, developers might be unaware of potential failure modes, further complicating debugging and reliability.

For some applications, the performance trade-offs introduced by portability are negligible. However, applications that frequently interact with the file system, especially those concerned with durability (e.g., financial systems), cannot afford data loss. While losing a downloaded file is an inconvenience, losing a bank transaction is a serious issue. Furthermore, such applications often require high performance to process large volumes of transactions quickly. Unfortunately, compiled code doesn't account for the specifics of the file system it will run on, meaning developers face a dilemma: prioritize correctness and performance by tailoring the application to a specific system or opt for portability. By choosing portability, developers may sacrifice efficiency and, more critically, risk compromising durability and reliability, as it's difficult to anticipate how different systems handle critical operations like data persistence.

This dissertation explores how to bring the advantages compilers offer for user-space code to the realm of system calls and file-system interactions. We pose the question: *Can a declarative language for file-system interactions improve the correctness and performance of portable applications?* We answer in the affirmative through *HSL, the High-Level System Language*. HSL captures the intent behind an application's file-system interactions in a declarative way. Unlike traditional compilers, HSL performs its transformations at runtime, when it has knowledge of the exact file system it will operate on. While traditional compilers ensure that code transformations maintain semantic correctness, HSL goes further by optimizing system calls for reliability, ensuring data loss prevention, and then for efficiency, by understanding the underlying file-system behavior and the developer's intent.

This dissertation is structured around three key parts. The first part focuses on understanding the systems on which applications run. To generate efficient system calls, we must first explore the behavior of the underlying file systems. While processors have well-documented specifications that allow compilers to optimize code, file systems lack this transparency. Standards, such as POSIX, do not account for all scenarios, particularly around durability and failure handling. In this part, we study these systems, with a focus on durability issues relating to `fsync` failures.

The second part of this dissertation examines how applications interact with the file system. By analyzing these interactions, we identify recurring patterns, much like how compilers detect opportunities for optimization in user-space code. We categorize these patterns into what we call "intentions"—high-level tasks that applications perform, such as ensuring durability (e.g., using redo logs) or carrying out operations that span multiple system calls to achieve a larger goal (e.g., reading multiple parts of a file). Understanding these intentions allows us to develop strategies for optimizing file system interactions, ensuring both correctness and efficiency in the execution of these tasks.

The final part presents the design, implementation, and evaluation of HSL, the High-Level System Language. This section builds on the insights from both the first and second parts, using the knowledge gained about file system behavior to ensure correct handling of durability-related intentions, while also incorporating the patterns of system interaction identified earlier. HSL is constructed to express these high-level intentions using a declarative language model. Middleware optimizations are applied to these intentions, and the runtime system selects the most efficient sequence of system calls based on the specific file system in use. This part also details the benchmarking and testing processes that validate HSL's reliability and efficiency across different systems, ensuring both correctness and performance in real-world scenarios.

We now introduce each of these three parts in turn.

1.1 Understanding `fsync()` Behavior

Applications that care about data must care about how data is written to stable storage. Issuing a series of `write` system calls is insufficient. A `write` call only transfers data from application memory into the operating system; the OS usually writes this data to disk lazily, improving performance via batching, scheduling, and other techniques [4, 74, 105, 107].

To update persistent data correctly in the presence of failures, the order and timing of flushes to stable storage must be controlled by the application. Such control is usually made available to applications in the form of calls to `fsync` [36, 82], which forces unwritten ("dirty") data to disk before returning control to the application. Most update protocols, such as write-ahead logging or copy-on-write, rely on forcing data to disk in particular orders for correctness [15, 17, 41, 50, 81, 126].

Unfortunately, recent work has shown that the behavior of `fsync` during failure events is ill-defined [124] and error prone. Some systems, for example, mark the relevant pages clean upon `fsync` failure, even though the dirty pages have not yet been written properly to disk. Simple application responses, such as retrying the failed `fsync`, will not work as expected, leading to potential data corruption or loss.

In this chapter, we ask and answer two questions related to this critical problem. The first question relates to the file system itself: why does `fsync` sometimes fail, and what is the effect on file-system state after failure?

To answer this first question, we run carefully-crafted micro-workloads on important and popular Linux file systems (ext4 [73], XFS [114], and Btrfs [99]) and inject targeted block failures in the I/O stream using `dm-loki`—our custom built device-mapper target for deterministic fault injection. We then use `blockviz`—a block trace visualization tool that

enriches block access patterns with file-system specific information, to examine the results. We provide the traces generated by `blockviz` to serve as reference for current file system error-handling behavior.

Our findings show commonalities across file systems as well as differences. For example, all three file systems mark pages clean after `fsync` fails, rendering techniques such as application-level retry ineffective. However, the content in said clean pages varies depending on the file system; ext4 and XFS contain the latest copy in memory while Btrfs reverts to the previous consistent state. Failure reporting is varied across file systems; for example, ext4 data mode does not report an `fsync` failure immediately in some cases, instead (oddly) failing the subsequent call. Failed updates to some structures (e.g., journal blocks) during `fsync` reliably lead to file-system unavailability. And finally, other potentially useful behaviors are missing; for example, none of the file systems alert the user to run a file-system checker after the failure.

The second question we ask is: how do important data-intensive applications react to `fsync` failures? To answer this question, we build CuttleFS, a FUSE file system that can emulate different file system `fsync` failures. CuttleFS maintains its own page cache in user-space memory, separate from the kernel page cache, allowing application developers to perform durability tests against characteristics of different file systems, without interference from the underlying file system and kernel.

With this test infrastructure, we examine the behavior of five widely-used data-management applications: Redis [94], LMDB [85], LevelDB [68], SQLite [117] (in both RollBack [109] and WAL modes [110]), and PostgreSQL [85] (in default and DirectIO modes). Our findings, once again, contain both specifics per system, as well as general results true across some or all. Some applications (Redis) are surprisingly careless with `fsync`, not even checking its return code before returning success to the application-level update; the result is a database with old, corrupt, or

missing keys. Other applications (LMDB) exhibit false-failure reporting, returning an error to users even though on-disk state is correct. Many applications (Redis, LMDB, LevelDB, SQLite) exhibit data corruptions; for example, SQLite fails to write data to its rollback journal and corrupts in-memory state by reading from said journal when a transaction needs to be rolled back. While corruptions can cause some applications to reject newly inserted records (Redis, LevelDB, SQLite), both new and old data can be lost on updates (PostgreSQL). Finally, applications (LevelDB, SQLite, PostgreSQL) sometimes seemingly work correctly as long as the relevant data remains in the file-system cache; when said data is purged from the cache (due to cache pressure or OS restart), however, the application then returns stale data (as retrieved from disk).

We also draw high-level conclusions that take both file-system and application behavior into account. We find that applications expect file systems on an OS platform (e.g., Linux) to behave similarly, and yet file systems exhibit nuanced and important differences. We also find that applications employ numerous different techniques for handling `fsync` failures, and yet none are (as of today) sufficient; even after the PostgreSQL `fsync` problem was reported [124], no application yet handles its failure perfectly. We also determine that application recovery techniques often rely upon the file-system page cache, which does not reflect the persistent state of the system and can lead to data loss or corruption; applications should ensure recovery protocols only use existing persistent (on-disk) state to recover. Finally, in comparing ext4 and XFS (journaling file systems) with Btrfs (copy-on-write file system), we find that the copy-on-write strategy seems to be more robust against corruptions, reverting to older states when needed.

1.2 Intentions in the Wild

File systems play a critical role in managing the storage and retrieval of data in modern applications. However, applications often rely on a complex set of system calls to achieve higher-level operations, which makes it challenging to directly express their intentions to the underlying system. This chapter explores how applications interact with file systems, analyzing patterns and extracting key insights from observed behaviors.

While studying the challenges of ensuring data durability in the face of `fsync` failures (the previous section), it was made clear that existing system calls are not always sufficient to meet the full intentions of applications. Many tasks require the orchestration of multiple system calls, sometimes with intricate ordering to enforce atomicity or durability [82]. In this study, we expand on these observations by asking: what are the common dialogues between applications and the file system?

To conduct this study, we developed Ikhnaie, a specialized tool designed to assist in tracing and visualizing application behaviors in relation to file-system interactions. While Ikhnaie focuses on capturing system call sequences alongside their user-space counterparts (ordinary function calls), it does not automatically identify higher-level patterns or dialogues. Instead, Ikhnaie provides valuable insights that make the process of source-code inspection and documentation review easier, allowing us to manually categorize and analyze patterns that represent the application's intentions.

Our analysis spans a wide range of applications, including key-value stores, databases, version control systems, build systems, and command-line utilities. By selecting multiple applications from each domain, we aimed to uncover domain-specific interaction patterns. Additionally, we focused on applications that have been widely studied in the systems research community, providing a comprehensive view of how diverse workloads interact with the file system.

We observe that applications often engage in a "dialogue" with the

underlying file system, where multiple system calls are required to accomplish a higher-level task. When these dialogues are observed across different applications, they can be generalized into what we term "intentions". These intentions represent common patterns of operations, such as single-file reads (according to some access pattern), multi-chunk writes, and multi-file operations. Additionally, we identify the filtering operation, where applications read data but only process relevant parts, which becomes especially useful when working with remote systems; performing filtering closer to remote storage reduces data transfer costs.

The chapter also sheds light on the intentions behind more complex operations like Manifests and Redo Logs. Manifests provide an atomic view of a set of files, ensuring consistency across files, while Redo Logs are crucial for maintaining durability and atomicity during updates. However, implementing some of the intentions we've found pose challenges, as there are multiple ways to achieve them, which can lead to confusion in selecting the best approach. Addressing these challenges requires careful consideration of performance trade-offs and system-specific optimizations; a cognitive burden for the developer.

This study demonstrates that many common file system operations require complex system call sequences that are not directly supported by current file system interfaces. By categorizing and understanding these intentions, we hope to inform the design of future file systems and system call interfaces. These insights can also serve as a guide for developers to write more efficient and correct file system interactions, ultimately leading to more robust and high-performing applications.

1.3 HSL

Our findings from both the study of intentions and the impact of `fsync` failures provide not only insights for improving future file systems or

exposing new interfaces but also the foundation for an innovative intermediate system: *HSL, the High-Level System Language*. HSL is designed to bridge the gap between applications and file systems by addressing the durability challenges revealed by `fsync` failures for the relevant intentions, while also optimizing all operations for efficiency. Drawing from these two areas, HSL offers a more reliable and efficient way for applications to interact with file systems.

In the final part of this dissertation, we describe the design and implementation of HSL, followed by an evaluation of its durability claims, its individual features, and its application in real-world scenarios. Similar to how compilers for high-level programming languages enable portable and efficient user-space code, HSL leverages a declarative language approach to generate portable, robust file-system interactions. Developers can substitute some or all file-system-related system calls with HSL scripts, which resemble the execution of SQL queries. These scripts use *verbs*, informed by our intention study, and *collections*, a uniform method for providing arguments to verbs.

Internally, HSL scripts (the front-end) undergo transformations (the middle-end) and are ultimately executed through the most efficient sequence of system calls (the back-end). In the context of durability, we introduce dedicated verbs for physical redo logging and manifest files. Fault-injection testing demonstrates that our implementations handle file system errors robustly, avoiding the failures that applications without HSL face, and addressing all issues highlighted in the `fsync` study; in tests using HSL, *we did not observe any invalid states*, such as data corruption, data loss, or false failures.

From a performance perspective, we benchmark various implementations of intentions across different workloads to determine the optimal system calls for each scenario. For example, we find that `copy_file_range()` [19] is the preferred method for copying large contiguous file

sections, while copying scattered smaller portions is more nuanced, depending on whether the data resides in the page cache. When data is cached, `sendfile()` [106] proves significantly faster, particularly on XFS and Btrfs.

We also introduce the concept of *transports*, separating what functionality is required from the file system from how that functionality is requested. While conventional system calls remain the default, newer asynchronous interfaces such as `io_uring` [7] offer alternative ways to interact with the system. Through benchmarking, we identify performance crossover points and incorporate heuristics, allowing HSL to choose the best transport at runtime based on the workload, ensuring efficient execution.

Finally, we evaluate HSL's application in real-world scenarios through five case studies, highlighting both benefits and limitations:

1 Copying a file using coreutils `cp` to an ext4 file system is 1.36x faster with HSL, while performance matches existing solutions in all other cases.

2 Trimming audio WAV files shows significant benefits from HSL's COPY intention, with performance greatly improved after modifying the application design to ensure copies align with a 4KB block boundary.

3 Generating Merkle proofs, commonly used in blockchains, is up to 2x faster with HSL due to its seamless transition to the `io_uring` transport.

4 HSL matches the current performance of LMDB, ensuring compatibility with existing systems and future adaptability. While there are no faster alternatives for writing to the page cache, HSL allows for future optimizations without requiring changes to LMDB.

5 SELECT queries in SQLite that involve index lookups achieve up to a 5.5x speedup with HSL. Although integrating HSL required modifications to SQLite to support batched read operations, these changes resulted in significant query performance improvements.

1.4 Contributions

Here, we list the main contributions of this dissertation.

- **dm-loki and CuttleFS.** We provide two tools for fault injection: `dm-loki`, a loadable kernel module device-mapper target for deterministic fault injection at the block layer (targeting specific blocks or sectors), and `CuttleFS`, a FUSE file system for deterministic fault injection at the file layer (targeting specific file offsets). Together, these tools enable controlled testing of block- and file-level failures.

- **Revealing File System Behavior Under `fsync` Failures with `blockviz` Traces.** Using `dm-loki` to inject block and sector-level faults, we analyze the behavior of three file systems (ext4, XFS, and Btrfs) under `fsync` failures. Our contribution includes detailed visualizations of their responses as block traces, both under normal conditions and with injected faults, offering insight into their recovery and failure-handling mechanisms.

- **Analyzing Application Strategies and Failures in Response to `fsync` Errors.** Using `CuttleFS`, we inject faults into files used by applications and emulate file system behaviors to study how applications respond to `fsync` errors on ext4, XFS, and Btrfs. We report findings of data loss, corruption, and false failures (where applications incorrectly report failures despite successful file-system states).

- **Ikhnaie: Tracing File System and Application Dialogues, and Classifying Intentions.** We design and implement Ikhnaie, a tracing tool with moderate overhead that captures both user-space function calls and system calls to study interactions between applications and file systems. Our contribution includes a detailed classification of common application intentions, along with example code listings, providing insight into how applications structure their system call sequences to achieve higher-level operations.

- **HSL.** We design and implement HSL, a declarative language that translates developer-specified intentions into optimized sequences of system calls, using runtime information to choose the best execution path. We provide reference implementations for two durability-related intentions: redo logs and manifests, demonstrating through fault-injection testing that HSL's use of file-system-specific error handling prevents data loss. Additionally, we present the results of benchmarking various implementations of intentions, showing where each excels and where alternative approaches perform better; HSL automatically selects the best approach at runtime. Finally, we evaluate HSL through real-world case studies, highlighting both its benefits and limitations.

1.5 Overview

We briefly describe the structure of the contents following this chapter.

Background and Motivation. Chapter 2 provides relevant background on system calls and current approaches to achieving portability. We then explore how misunderstanding `fsync()` has lead to data loss, the limitations of file system portability, and the importance of utilizing runtime information.

Understanding `fsync()` Failures. Chapter 3 presents our study on how file systems react to device failures during `fsync`, and how applications respond to these failures.

Intentions in the Wild. Chapter 4 explains our methodology for identifying interactions between applications and file systems, and presents our findings through a classification of common intentions.

HSL: A declarative language for File System Intentions. Chapter 5 details the overall design of HSL, including implementation specifics for its verbs and optimizations.

Evaluation. Chapter 6 extends the previous chapter by evaluating HSL's correctness and performance. We provide fault-injection results that test HSL's durability (for physical redo logs and manifests), along with benchmarking results across various workloads to demonstrate the advantages of HSL's features. Real-world case studies are also presented.

Related Work. In Chapter 7, we discuss prior research relevant to this dissertation, including crash testing, fault injection studies, system call studies, and efforts to enhance system call performance.

Conclusions and Future Work. Chapter 8 summarizes the dissertation on a per-chapter basis, outlines key lessons learned, and suggests possible directions for future work.

2
Background and Motivation

Computing environments (desktops, laptops, tablets, phones) today are designed to run multiple applications simultaneously, each performing their functions without worry of interference from other applications, despite sharing the same resources: memory, the cpu, network interfaces for the internet, or local storage. This is achieved through the help of an underlying system that has full control of these resources—the operating system. In this chapter, we provide background on how an application interacts with the operating system specifically in the context of storage. We provide details on what happens when an application reads or writes a file, and how developers ensure the application works on multiple systems (portability). We show that persisting data is hard to get right, portability makes it difficult to do so correctly and efficiently, and motivate the need to use runtime information.

2.1 Application and File System Interaction

Instead of checking each and every instruction an application wishes to execute on the processor—a performance issue—the operating system runs in a "system-space" or "kernel-space" which can execute instructions such as using the network card to communicate with the internet or issue i/o to a disk. The applications we run today such as web browsers, music

players, file explores, and even databases that power other applications on top of them, all run in a lower privilege mode—"user-space" where executing the above instructions is prohibited directly by the hardware. Applications can therefore execute all other instructions on the processor directly (*limited direct execution*).

When an application needs to perform an operation that requires access to hardware resources, such as reading from a disk, writing to a file, or accessing the network, it cannot do so directly. Instead, applications *request* the operating system to perform the task on their behalf through ***system calls***. On system call invocation, the application moves into "kernel-space" where the operating system executes the right function depending on the arguments passed, and then transitions back to "user-space" when returning the result to the application. While there are many system calls (over 300 in Linux), we focus on those interacting with storage devices.

Applications that interact with storage media do not do so directly. First, they rely on the operating system to perform these actions through system calls as mentioned earlier. Second, applications use the operating system's file abstraction instead of interacting with the raw storage as binary data. The operating system provides a view of files and directories, allowing the application to perform operations such as listing all files in a directory and retrieving or storing content from or to a particular file, instead of manually retrieving bytes at certain locations on the disk and interpreting them.

In addition to providing a file abstraction, the operating system contains and supports various file systems, each having its own way of organizing and interpreting the on-disk contents. Different file systems are designed and optimized with different goals, such as performance, security, or compatibility. Examples include ext4, XFS, Btrfs, exFAT, and FAT32. We now cover how an application interacts with file systems to read or write files, specifically on Linux.

Reading and Writing Files. Applications that wish to read or write file contents use system calls to do so. However, given the variety of file systems, Linux provides a uniform interface through VFS—the virtual file system, allowing applications to interact with files without worrying about the underlying file system type. Additionally, POSIX (Portable Operating System Interface) provides a set of standardized APIs to ensure consistency across different operating systems and file systems. By adhering to POSIX standards, developers can interact with files and directories in a uniform manner, regardless of the underlying file system. File systems however can have non-portable functionality which we cover later when discussing portability(§2.2). We now discuss how an application uses standard POSIX interfaces to read and write files, and the path taken from system call to the disk.

An application that wishes to read or write to a file must first obtain a *descriptor* for the file using the `open()` system call[1]. If the application has the required permissions to read or write the file, or even to create the file in a particular directory, a file descriptor is returned; otherwise, the system call fails and the return value is set accordingly. With the file descriptor, the application can then read or write to the file.

The anatomy of a read system call. When an application wants to read data from a file, it uses the `read()` system call or its variants. While every read system call transitions into the kernel, not all of them require i/o to the disk. The operating system maintains a cache of frequently or most-recently accessed file contents. Every file is divided into equal sized segments called "pages" which are typically 4KB in size. These pages are cached in the operating system's page cache.

On `read()`, the VFS first checks if the requested file region is in the page cache. If present, no i/o with the disk is required, and the data is copied to the application's user-space buffer. If some or all of the requested region is

[1]The application can list all files in a directory with the `readdir()` system call.

not in the cache, VFS passes the request down to the underlying file system. Similar to files, disks divide their capacity into equal sized segments called "sectors" and "blocks", typically 512B and 4KB respectively; we cover their guarantees when discussing writes. The file system consults its internal state to identify the blocks for the requested region (a block-to-offset mapping), and issues an i/o read request for the same. The data is then stored in the page cache and copied to the application's user-space buffer.

As an optimization, VFS may detect a certain access pattern such as reading a file sequentially and may request more than what the application actually needs (readahead logic). Therefore, more data is stored in the page cache for future use by the application, but only the region requested is copied to the application's buffer. Applications also use similar techniques in user space, reading more than necessary to minimize system calls. The C standard i/o library exposes functions (`fopen`,`fread`) and maintains an internal buffer to read more than actually requested. However, we focus mainly on the interactions between user-space and kernel-space, not between applications and their user-space libraries.

The anatomy of a write system call. When an application wants to write data to a file, it uses the `write()` system call or its variants. We first cover the common case of writes to the page cache (buffered writes), and discuss the other methods later. Unlike reads, as write operations can modify metadata that is file-system specific, VFS always routes the write operation to the underlying file system. When writes are less than a page, the page must first be read if it is not already in the page cache. In cases where an entire page is modified, no i/o is necessary. The file system updates the contents for the file, but only in the page cache. If no offset-to-block mapping exists, depending on the file system (or configuration options) the block is allocated immediately or later just before issuing an

i/o write request[2]. Applications that read the file will have access to the updated contents served through the page cache. However, there is now a mismatch between the data in memory (page cache) and on the device. A system crash or power loss can result in the data not actually reaching the disk.

Ensuring writes reach the disk. Linux maintains a write-back page cache where buffered writes are stored in the cache; future reads return data from the cached pages. The previously written pages are later written to disk. Linux identifies these pages by marking them "dirty" when a write modifies them. Periodically, a flusher thread writes the dirty pages to disk and marks them clean.

As disks can fail (§2.2.1), it is possible that only some of the blocks are written to disk. Modern disks however provide some guarantees, specifically sector atomicity. Disk manufacturers ensure that writes to a sector either contain the old value or the new value, and not something in between. However, blocks or pages that are larger than a sector do not have such guarantees. Additionally, the file system maintains its own data structures that span multiple blocks. To attain the same atomicity guarantees but over multiple blocks, file systems implement crash consistency techniques such as journaling [87].

When a write request reaches the disk, it may still not be considered *stable* to survive power loss. Disks may have a volatile write cache to absorb write requests faster. To force the write request to be written to non-volatile storage, a `FLUSH` i/o request must be sent. Alternatively, the write request can set the Force Unit Access (`FUA`) flag which instructs the device to by-pass the volatile cache for that request. File systems use these commands as part of crash consistency strategies. One such strategy is to

[2] As some files are only temporarily written and soon deleted, file systems developed an optimization called *delayed allocation*—delaying the allocation of a block until it needs to be written out to disk.

issue a FLUSH so that all data blocks in the volatile write cache are written out. Then another FLUSH is issued after metadata is written to the journal for similar reasons. Finally, a sector indicating end of journal entry is written with the FUA flag set. Any crash during the operation results in discarding the entire journal entry (ignored during recovery). However, a crash after the operation completes successfully can be recovered by reading the journal entry. To prevent the journal from growing too large, the entries written out periodically during a checkpoint operation and removed from the journal (described further in relevant chapters).

Applications that have strict durability requirements and wish to persist data immediately cannot wait for the periodic activities of the operating system. Instead, they use the `fsync()` system call which writes the dirty pages only for the file they provide as argument while ensuring the same crash consistency guarantees.

As mentioned previously, not all writes behave as described. A file opened with `O_SYNC` or on a file system mounted with synchronous writes (`-o sync`) will ensure each write is written to non-volatile storage, ensuring crash-consistency as mentioned previously. An application can avoid `fsync()` in such cases, but makes the trade off of every write inducing i/o requests. A file opened with `O_DIRECT` bypasses the page cache (for both reads and writes), but without `O_SYNC` the written data may reside in the device's volatile write cache. Even if there are no dirty pages (as in the `O_DIRECT` case), an `fsync()` issues a `FLUSH` i/o request on devices with a volatile write cache.

Techniques that minimize system calls. System calls are not the only way to read and write files. Applications may do so either with the help of the kernel, or by taking control of the device entirely. In the former, they use the `mmap()` system call to map the file (or portions of it) into user-space memory which they can modify directly. In the latter, applications use kernel-bypass techniques where the kernel is initially involved to

grant access to the device and the application uses user-space drivers to communicate directly with the hardware; typically with a library file system that provides equivalent file abstractions. We do not expand further on these as they are not in the scope of this dissertation.

2.2 Portability in Cross-Platform Applications

Most software developed today is available for download as ready-to-use packages or installers for various platforms. Additionally, open-source software enables end users to download and build the application from source code. As there are multiple operating systems and many more file systems for each, applications rely on standards like POSIX to make cross-platform code easier to write and maintain. However, POSIX does not eliminate all challenges in portability. In this section, we provide an overview of how portability is currently achieved.

Cross-platform software can be categorized depending on whether it runs directly (native binary) or indirectly. For the latter, it is an intermediate form that runs atop another system (e.g., the Java Runtime Engine) which handles portability and runs directly. In both categories, we focus on the portions that run directly.

Modern compilers handle most of the heavy lifting by converting source code in high-level languages to executable binaries. While the code goes through multiple optimization passes resulting in efficient binaries, the gains relate to transformation of user-space code or selecting the most efficient assembly instructions for the same. Compilers can generate efficient user-space code but translate system calls as is. The onus is on the developer to choose the correct supported system call or use a library that does so.

As mentioned previously, portability for system interaction through system calls is commonly achieved with the help of interface standard-

ization. However, operating systems (and file systems) are sometimes only partially compliant or compliant but also provide faster interfaces. In other words, they aim to provide compatibility which does not always guarantee efficiency.

Due to versioning and backwards compatibility, an newer version of Linux may have better system calls; both are POSIX compatible but using the better system calls can be faster. Applications that intend to support both versions typically use conditional compilation. In C and C++, conditional compilation is achieved by checking for certain definitions through `#ifdef` and its variants; the definitions are made available before compilation with a configure script. For example, the configure script tests for the availability of the `pread()` system call. If present, it uses the same instead of the `lseek()` and `read()` combination. Similar checks are done for all system call variants that it wishes to use.

Conditional compilation is also used when compiling for different operating systems. For example, `fsync()` on macOS behaves differently than on Linux. To ensure the data is moved to non-volatile storage, `ioctl(F_FULLSYNC)()` must be called on macOS. Through conditional compilation, the developer can choose the correct strategy; they do have to be aware of them though.

Unfortunately, while conditional compilation helps choose system calls for different operating systems, it does little for file systems. The code compiled for a given operating system can be run on any file system the operating system supports. Some file systems may support all modern system calls while others only support the basic ones. Applications may take a try-by-failure approach to discover them as the unsupported ones return an `ENOTSUPP` error.

The more difficult task is when they are supported, but we don't understand them completely. As previously described, correct usage of `fsync()` on multiple operating systems requires knowledge of what macOS does

on `fsync()`. Achieving correctness on multiple file systems may be even harder.

2.2.1 Misunderstanding `fsync()` Causes Data Loss.

Applications that manage data must ensure that they can handle and recover from any fault that occurs in the storage stack. Recently, a PostgreSQL user encountered data corruption after a storage error and PostgreSQL played a part in that corruption [38]. Because of the importance and complexity of this error, we describe the situation in detail.

PostgreSQL is an RDBMS that stores tables in separate files and uses a write-ahead log (*wal*) to ensure data integrity [86]. On a transaction commit, the entry is written to the log and the user is notified of the success. To ensure that the log does not grow too large (as it increases startup time to replay all entries in the log), PostgreSQL periodically runs a checkpoint operation to flush all changes from the log to the different files on disk. After an `fsync()` is called on each of the files, and PostgreSQL is notified that everything was persisted successfully, the log is truncated.

Of course, operations on persistent storage do not always complete successfully. Storage devices can exhibit many different types of partial and transient failures, such as latent sector errors [10, 57, 103], corruptions [9], and misdirected writes [65]. These device faults are propagated through the file system to applications in a variety of ways [54, 89], often causing system calls such as `read()`, `write()`, and `fsync()` to fail with a simple return code.

When PostgreSQL was notified that `fsync()` failed, it retried the failed `fsync()`. Unfortunately, the semantics for what should happen when a failed `fsync()` is retried are not well defined. While POSIX aims to standardize behavior, it only states that outstanding IO operations are not guaranteed to have been completed in the event of failures during `fsync()` [84]. As we shall see, on many Linux file systems, data pages

that fail to be written, are simply marked clean in the page cache when `fsync()` is called and fails. As a result, when PostgreSQL retried the `fsync()` a second time, there were no dirty pages for the file system to write, resulting in the second `fsync()` succeeding without actually writing data to disk. PostgreSQL assumed that the second `fsync()` persisted data and continued to truncate the write-ahead log, thereby losing data. PostgreSQL had been using `fsync()` incorrectly for 20 years [124].

After identifying this intricate problem, developers changed PostgreSQL to respond to the `fsync()` error by crashing and restarting without retrying the `fsync()`. Thus, on restart, PostgreSQL rebuilds state by reading from the *wal* and retrying the entire checkpoint process. The hope and intention is that this crash and restart approach will not lose data. Many other applications like WiredTiger/MongoDB [76] and MySQL [78] followed suit in fixing their `fsync()` retry logic.

This experience leads us to ask a number of questions. As application developers are not certain about the underlying file-system state on `fsync()` failure, we study what happens when `fsync()` fails. How do file systems behave after they report that an `fsync()` has failed? Do different Linux file systems behave in the same way? What can application developers assume about the state of their data after an `fsync()` fails? Thus, we perform an in-depth study into the `fsync()` operation for multiple file systems (§3.1).

We then study how data-intensive applications react to `fsync()` failures (§3.2). Does the PostgreSQL solution indeed work under all circumstances and on all file systems? How do other data-intensive applications react to `fsync()` failures? For example, do they retry a failed `fsync()`, avoid relying on the page cache, crash and restart, or employ a different failure-handling technique? Overall, how well do applications handle `fsync()` failures across diverse file systems?

Studying file-system `fsync()` behavior highlights the non-standardized

post-failure characteristics. File systems are not bound to any one specific strategy as POSIX does not place restrictions on what state must be after the failure. Like with macOS, developers must now understand how `fsync()` behaves not just on different operating systems but on different file systems as well. However, as described previously, conditional compilation is insufficient in such scenarios as different file systems may be used after the application is compiled.

2.2.2 Limitations in Conditional Compilation

As described previously, portability (on operating systems and file systems) makes correctness hard to get right. And even when applications achieve correctness, portability—through conditional compilation—makes it difficult to extract all the performance a file system has to offer. While conditional compilation helps choose *an* available interface, it need not be *the best* interface. Some cases like `pread()` are always better than their individual counterparts, but the same cannot be said for all interfaces.

Within an operating system such as Linux, the same compiled application may run on different file systems and each may have different performance profiles for the available interfaces. In some cases like `copy_file_range()`, the interface may exist but not be supported by all file systems. Applications like `cp` employ the try-by-failure approach, first attempting to use the faster `copy_file_range()` before falling back to an available albeit slower interface. The choice of system calls is fixed after compilation; an implicit assumption of a performance hierarchy among the interfaces.

Unfortunately, the above approaches limit performance in applications that frequently interact with file systems. Unless developers constrain usage to specific file systems, the assumption of a clear performance hierarchy breaks. For example, using `io_uring` for writes to the page cache can be faster on XFS but hurts performance on ext4 [100, 101].

The assumption breaks even on the same file system as the type of workload can change the performance hierarchy of interfaces. We profiled a workload that copies random 4KB pages between two files on XFS using two implementations for the copy: (1) `read+write` (2) `copy_file_range`. The former does far better when the files are cached and the latter when they aren't. Furthermore, there may exist file systems with alternative faster interfaces that are overlooked.

These observations lead us to survey how popular applications commonly interact with the file system, both for durability and general functionality (§4).

2.3 A Case for Runtime Information

Armed with the knowledge of what applications want from the file system, and how those file systems behave, we ask the question: Can applications achieve portable correctness and efficiency? Answering it requires knowing the file system we run on at run time.

The previously mentioned limitations can be overcome by deferring the choice of system call from compile-time to runtime, where we know the exact underlying environment. An application can query runtime information such as the file system and its mount options to select the right system call, or even use non-standard interfaces for custom file systems. However, such an approach requires developers to have in-depth knowledge of all file systems they wish to support. We explore an alternative approach: a layer of indirection that makes the right choice for the developer.

While indirection through a library addresses the above mentioned limitations, optimizations are confined to within a single library call[3] overlooking benefits across library calls. Instead, a domain-specific language

[3]One could employ static analysis techniques and transform code through custom compiler passes to merge library calls.

approach allows the intermediate layer to analyze a larger context and provides benefits similar to modern compilers—portability and a repository of ever-growing optimizations. While domain-specific languages can be external (with an independent interpreter/compiler) or internal (using functions and syntax of the host language), both forms constitute a language with the ability to transform multiple equivalent library calls, similar to multi-query optimizations [104]. Furthermore, new compound operations optimized for systems in the future would require modifications to use a new library function; language approaches need only update the compiler.

Thus, we aim to answer this question of achieving portable correctness and efficiency by building a system that uses runtime information with a language that decides how to best perform common tasks (§5), and evaluate its effectiveness towards the same (§6).

3 Understanding Fsync Failures

In recent years, misunderstandings surrounding the `fsync()` system call have led to data loss (§2.2.1), highlighting the need for a deeper exploration of its failure modes and their implications.

In this chapter, we address two critical questions to improve our understanding of `fsync`: First, why does `fsync` fail, and what impact does such a failure have on file-system state (§3.1)? Second, how do popular data-intensive applications react when `fsync` fails (§3.2)?

By exploring these questions, we aim to determine whether applications can take proactive measures to handle `fsync` failures effectively.

This chapter is based on the paper, *Can Applications Recover from `fsync` Failures?*, published in ACM Transactions on Storage [93].

3.1 File System Study

Our first study explores how file systems behave after reporting that an `fsync` call has failed. We begin with our methodology for the study, followed by our findings for three Linux file systems (ext4, XFS, and Btrfs).

3.1.1 Methodology

To understand how file systems should behave after reporting an `fsync` failure, we begin with the available documentation. The `fsync` man

pages [36] report that `fsync` may fail for many reasons: the underlying storage medium has insufficient space (*ENOSPC* or *EDQUOT*), the file descriptor is not valid (*EBADF*), or the file descriptor is bound to a file that does not support synchronization (*EINVAL*). Since these errors can be discovered by validating input and metadata before initiating write operations, we do not investigate them further.

We focus on errors that are encountered only after the file system starts synchronizing dirty pages to disk; in this case, `fsync` signals an *EIO* error. *EIO* errors are difficult to handle because the file system may have already begun an operation (or changed state) that it may or may not be able to revert.

To trigger *EIO* errors, we consider single, transient, write faults in line with the fail-partial failure model [88, 89]. When the file system sends a write request to the storage device, we inject a fault for a single sector or block within the request. Specifically, we build a kernel module device-mapper target called `dm-loki` that intercepts block-device requests from the file system and fails a particular write request to a particular sector or block while letting all other requests succeed; this allows us to observe the impact on an unmodified file system.

3.1.1.1 Workloads

To exercise the `fsync` path, we create three simple workloads that are representative of common write patterns seen in data-intensive applications.

Single Block Update ($\mathbf{w_{su}}$): open an existing file containing three pages (12KB) and modify the middle page. This workload resembles many applications that modify the contents of existing files: LMDB always modifies the first two metadata pages of its database file; PostgreSQL stores tables as files on disk and modifies them in-place. Specifically, w_{su} issues system calls in the following sequence: `open`, `lseek(4K)`, `write(4K)`, *`fsync`*, *`fsync`*, `sleep(40)`, `close`. The first `fsync` forces the dirty page

to disk. While one `fsync` is sufficient in the absence of failures, we are interested in the impact of `fsync` retries after a failure; therefore, w_{su} includes a second `fsync`. Finally, since ext4, XFS, and Btrfs write out metadata and checkpoint the journal periodically, w_{su} includes a sleep for 40 seconds.

Multi Block Append ($\mathbf{w_{ma}}$): open a file in append mode and write a page followed by an `fsync`; writing and fsyncing is repeated after sleeping. This workload resembles many applications that periodically write to a log file: Redis writes every operation that modifies its in-memory data structures to an append only file; LevelDB, PostgreSQL, and SQLite write to a write-ahead-log and `fsync` the file after the write. w_{ma} repeats these operations after a delay to allow checkpointing to occur; this is realistic as clients do not always write continuously and checkpointing may occur in those gaps. Specifically, w_{ma} issues system calls in the following sequence: `open` (in append mode), `write(4K)`, *`fsync`*, `sleep(40)`, `write(4K)`, *`fsync`*, `sleep(40)`, `close`.

Multi File Create ($\mathbf{w_{dir}}$): create a new file within a directory and then `fsync` both the file and the directory. This workload resembles file creation in many applications that care about durability. The ALICE framework [82] analyzes multiple applications and lists vulnerabilities that arise from not issuing an `fsync` on the parent directory after creating and calling `fsync` on a file. w_{dir} repeats these operations after a delay to allow checkpointing to occur; a realistic scenario as applications often create files periodically. Specifically, w_{dir} issues system calls in the following sequence: `open(dir)`, `creat(file1)`[1], `fsync(file1)`, `fsync(dir)`, `sleep(40)`, `creat(file2)`, `fsync(file2)`, `fsync(dir)`, `sleep(40)`, `close(file1,file2,dir)`.

[1]We use the notation `creat` for conciseness but we actually use the `open` system call with flags `O_WRONLY | O_CREAT`.

3.1.1.2 dm-loki

To study file system behavior on `fsync` failure, we require a tool that injects failures deterministically. For example, always failing the i^{th} write to a particular sector or block. Additionally, as file systems may overwrite a block multiple times, capturing disk state before and after an experiment is insufficient. We require the content of each read or write request. We built `dm-loki` [25], a loadable kernel module device-mapper target to satisfy both requirements.

Contrast with current fault injection device-mapper targets like `dm-error` and `dm-flakey` [24], `dm-loki` can change its fault injection configuration dynamically via messages through the `dmsetup message` [26] interface. A user may start `dm-loki` without any failure points and then send a message to the target to start failing certain sectors or blocks. Fault injection for a particular block is expressed as character sequences where the index in the sequence is incremented every time the block is written to. `dm-loki` decides to fail a particular access if the character at the current index indicates failure. For example, the sequence string `wwxxw` describes a pattern where the first two writes succeed, the third and fourth fail, and all writes after succeed. For a specific block or sector failure sequence, the lowercase letters `w` and `x` at an index `i` decide whether the i^{th} request is sent to the underlying device or failed. For accesses greater than the string length, we refer the last character to decide.

A user may also enable or disable request logging using the `dmsetup message` interface. `dm-loki` logs all read and write requests with associated data and flags to a file. Additionally, a user may inject "tags" via messages which are also logged. Injecting tags with a system call name and arguments right before its invocation allows us to identify the origin of each request. For example, all requests immediately preceded by a thirty second sleep tag implied that the requests were generated periodically for checkpointing.

3.1.1.3 blockviz

File systems need not treat all block write failures equally. Data block write failures may not be treated the same way as metadata block write failures. Additionally, file systems may treat different types of metadata differently. While `dm-loki` provides the functionality to inject a failure and log all BIO requests, it has no file-system level context about the specific sector or block. We build `blockviz`, an interactive jupyter notebook [61] widget in python that takes BIO requests logged by `dm-loki` as input and enriches them with file-system specific information. We describe `blockviz`'s main features that help us in characterizing file-system behavior.

`dm-loki`'s ability to inject "tags" into the logs make it easier to visualize requests with `blockviz`. In our workloads, before issuing a system call, we inject a "tag" to `dm-loki`, specifying the system call and its arguments. `blockviz` visualizes the traces with tags, making it easier to identify the origin of every BIO request. Furthermore, as an interactive widget, clicking a particular request provides more file system specific information. Using a combination of existing tools such as `debugfs` and `xfs_db`, and custom code to parse metadata blocks (such as XFS headers and Btrfs tree nodes), `blockviz` provides more information about every block in the trace.

As a particular block may be read from or written to multiple times, `blockviz` allows a user to compare different blocks in the trace. Since `dm-loki` logs all data read or written, `blockviz` creates checksums of the data for fast searches; a useful feature when trying to match content written in a journal block to content written to a metadata block during checkpointing. `blockviz` also allows metadata specific comparisons such as highlighting differences between two inode table entries in ext4 or identifying bitmap differences.

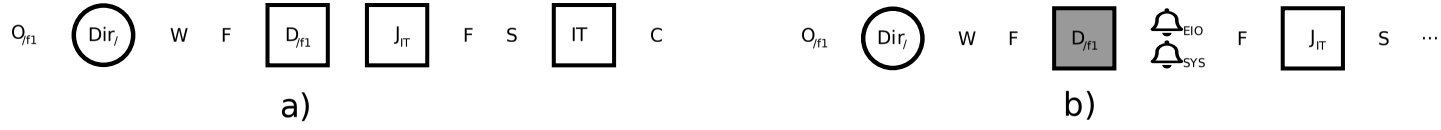

Figure 3.1: **Sample blockviz traces and how to interpret them**:
The figure shows two sample traces from blockviz for w_{su} on ext4 ordered mode. The first (a) represents a normal run without any failures while the second (b) contains a grey shaded block indicating a write failure. System calls are represented by their first letter (in boldfaced font): **O**pen, **W**rite, **F**sync, **S**leep, **C**lose. In some cases, a system call contains the file or directory path. As w_{su} opens file `/f1`, the first symbol is $\mathbf{O}_{/f1}$. BIO read requests are depicted using circles ○ and write requests with squares □. The letters within a BIO request are file-system dependent and are explained when first used.

In this blockviz ext4 trace:

$\text{Dir}_{/}$ is the directory data block for the root(`/`) directory. $D_{/f1}$ is the data block for file `/f1`.
IT is a block that contains the inode table entry for `/f1`.
J_{IT} is a journal block containing the inode table entry.

In workloads that have multiple data block writes such as w_{ma}, we use `D'` for the data corresponding to the second `write` system call. Trace (b) contains a grey shaded block ■ indicating that the write to data block $D_{/f1}$ failed.

It is followed by two bells that symbolize notifications:

EIO: The immediate left system call (the `fsync`) is failed and errno is set to EIO.
SYS: The error is written to syslog.

As seen at the end of trace (b), if the rest of the trace is similar to the trace without failures, we use a set of dots indicating an ellipsis. For better readability, a dotted line separates this caption from the main text.

Figure 3.1 shows two sample traces from blockviz for ext4 ordered mode running w_{su}. We provide interpretations for these traces as they are used frequently in our findings (§3.1.2).
Figure 3.1a can be read as follows:

1 `open(/f1)` triggers a read request for the root directory data block.

2 There are no BIO requests during `write`.

3 On `fsync`, the data block for `/f1` is written to disk and the inode table is journaled.

4 There are no BIO requests during the second `fsync`.

5 During `sleep`, the journaled inode table is written to its actual location.

6 There are no BIO requests during `close`.

Figure 3.1b can be read as follows:

1-2 Same as Figure 3.1a.

3 On `fsync`, the data block write for `/f1` fails and nothing is journaled. The user is notified of the failure through a syslog entry and `fsync` returns -1 with errno set to EIO.

4 The second `fsync` writes the inode table to the journal.

5-6 Same as Figure 3.1a.

BIO request traces from `blockviz` contain too much low-level information. For instance, journaling in ext4 ordered mode involves writing a journal descriptor block that describes the following blocks, the actual block data to be journaled, a BIO flush request, and finally, a journal commit block with the Force Unit Access (FUA) flag set. For simplicity and conciseness, our traces in this paper do not include the flush requests and BIO flags. For ext4 specifically, we also omit the journal descriptor and commit blocks from the traces.

3.1.1.4 Experiment Overview

We run the workloads on three different file systems: ext4, XFS, and Btrfs, with default mkfs and mount options. We evaluate both ext4 with metadata ordered journaling (data=ordered) and full data journaling (data=journal). We use an Ubuntu OS with Linux kernel version 5.2.11.

We run mkfs on loop devices. Since our workloads are small, the loop devices are backed by files of size 1GB (images). The images are created using the `dd if=/dev/zero` command to ensure a clean initial zero state. The 1GB size ensures that the block size of the file systems is 4KB by default[2]. Since workloads w_{su} and w_{ma} require an existing file to operate on, we mount the file system, create an existing file of required size, and unmount. The images are now considered ready for the workloads.

For each file system and workload, we conduct experiments as follows:

We create a loop device (say loop0) from the prepared image using the `losetup` [70] command. Then, using the `dmsetup` command, we setup a device-mapper device `/dev/dm/lokidev` that forwards all requests to the `dm-loki` target. We then run the workload with no fault points configured.

For each file system and workload, we first trace the block write access pattern. We then repeat the workload multiple times, each time configuring the fault injector to fail the i^{th} write access to a given sector or block. We only fail a single block or sector within the block in each iteration. We use `blockviz` to analyze the traces and SystemTap [115] to examine the state of relevant buffer heads and pages associated with data or metadata in the file system.

[2]A smaller file size can change the block size to 1KB on ext4.

3.1.1.5 Behavior Inference

We answer the following questions for each file system:
Basics of `fsync` Failures:

- Q1 Which block (data, metadata, journal) failures lead to `fsync` failures?
- Q2 Is metadata persisted if a data block fails?
- Q3 Does the file system retry failed block writes?
- Q4 Are failed data blocks marked clean or dirty in memory?
- Q5 Does in-memory page content match what is on disk?

Failure Reporting:

- Q6 Which future `fsync` will report a write failure?
- Q7 Is a write failure logged in the syslog?

After Effects of `fsync` Failure:

- Q8 Which block failures lead to file-system unavailability?
- Q9 How does unavailability manifest? Does the file system shutdown, crash, or remount in read-only mode?
- Q10 Does the file suffer from holes or block overwrite failures? If so, in which parts of a file can they occur?[3]

Recovery:

- Q11 If there is any inconsistency introduced due to `fsync` failure, can fsck detect and fix it?

[3]In file-system terminology, a hole is a region in a file for which there is no block allocated. If a block is allocated but not overwritten with the new data, we consider the file to have a *non-overwritten block* and suffer from *block overwrite failure*.

3.1.2 Findings

We now describe our findings for the three file systems we have characterized: ext4, XFS, and Btrfs. Our answers to our posed questions are summarized in Table 3.1.

3.1.2.1 Ext4

The ext4 file system is a commonly-used journaling file system on Linux. The two most common options when mounting this file system are *data=ordered* and *data=journal* which enable ext4 ordered mode and ext4 data mode, respectively. Ext4 ordered mode writes metadata to the journal whereas ext4 data mode writes both data and metadata to the journal.

Ext4 ordered mode: We give an overview of ext4 ordered mode by describing how it behaves for our three representative workloads when no failures occur.

Single Block Update ($\mathbf{w}_{\mathtt{su}}$). When no fault is injected and `fsync` is successful, ext4 ordered mode behaves as follows. During the `write` (Step 1), ext4 updates the page in the page cache with the new contents and marks the page dirty. On `fsync`, the page is written to a data block; after the data-block write completes successfully, the metadata (i.e., the inode with a new modification time) is written to the journal, and `fsync` returns 0 indicating success (Step 2). After the `fsync`, the dirty page is marked clean and contains the newly written data. On the second `fsync`, as there are no dirty pages, no block writes occur, and as there are no errors, `fsync` returns 0 (Step 3). During `sleep`, the metadata in the journal is checkpointed to its final in-place block location (Step 4). No writes or changes in page state occur during the `close` (Step 5). The trace for this experiment can be seen in Figure 3.2a.

If `fsync` fails (i.e., returns -1 with errno set to *EIO*), a variety of write problems could have occurred. For example, the data-block write could

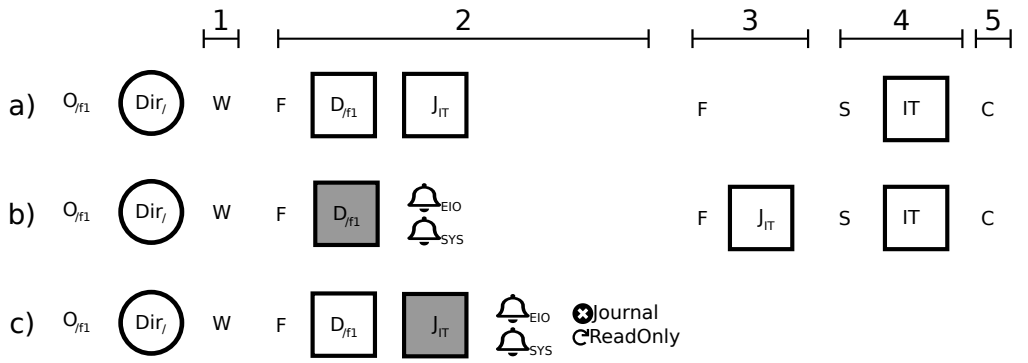

Figure 3.2: **Blockviz traces for w_{su} on ext4 ordered mode**:
The figure shows three traces corresponding to different fault injection configurations of dm-loki.

a) when no faults are injected: `open(/f1)` triggers a read request for the directory data block `/`. On `fsync`, the data block for `/f1` is written to disk and the Inode Table (IT) is written to the Journal (J_{IT}). During `sleep`, the Inode Table is checkpointed.

b) dm-loki configured to fail the data block write: On data block write failure, the error is logged to syslog (🔔_{SYS}) and fsync fails with errno=EIO (🔔_{EIO}).

c) dm-loki configured to fail the journal block write: On journal block write failure, in addition to the syslog and EIO notifications, ext4 aborts the journal (⊗Journal) and remounts in read-only mode (↻ReadOnly).

The figure is also annotated with steps (the first horizontal row with lines and numbers) that are referred to in the main text.

. . . . . .

have failed (trace in Figure 3.2b); if this happens, ext4 does not write the metadata to the journal. However, the updated page is still marked clean and contains the newly written data from Step 1, causing a discrepancy with the contents on disk. Furthermore, even though the inode table was not written to the journal at the time of the data fault, the inode table containing the updated modification time is written to the journal on the

		`fsync` Failure Basics					Error Reporting		After Effects			Recovery
		Which block failure causes `fsync` failure?	Is metadata persisted on data block failure?	Which block failures are retried?	Is the page dirty or clean after failure?	Does the in-memory content match disk?	Which `fsync` reports the failure?	Is the failure logged to syslog?	Which block failure causes unavailability?	What type of unavailability?	Holes or block over-write failures? If yes where do they occur?	Can fsck help detect holes or block over-write failures?
		Q1	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**	**Q8**	**Q9**	**Q10**	**Q11**
ext4	**ordered**	data, jrnl	yes [A]		clean [B]	no [B]	immediate	yes	jrnl	remount-ro	NOB, anywhere [A]	no
	data	data, jrnl	yes [A]		clean [B]	no [B]	next [C]	yes	jrnl	remount-ro	NOB, anywhere [A]	no
XFS		data, jrnl	yes [A]	meta	clean [B]	no [B]	immediate	yes	jrnl, meta	shutdown	NOB, within [A]	no
Btrfs		data, jrnl	no		clean	yes	immediate	yes	jrnl, meta	remount-ro	HOLE, within [D]	yes

[A] Non-overwritten blocks (Q10) occur because metadata is persisted despite data-block failure (Q2).

[B] Marking a dirty page clean (Q4) even though the content does not match the disk (Q5) is problematic.

[C] Delayed reporting (Q6) of `fsync` failures may confuse application error-handling logic.

[D] Continuing to write to a file after an `fsync` failure is similar to writing to an offset greater than file size, causing a hole in the skipped portion (Q10).

Table 3.1: **Behavior of Various File Systems when `fsync()` Fails.** The table summarizes the behavior of the three file systems: ext4, XFS, and Btrfs according to the questions posed in Section 3.1.1.5. The questions are divided into four categories mentioned at the top. For questions that require identifying a block type, we use the following abbreviations: Data Block (data), Journal Block (jrnl), Metadata Block (meta). In Q9, Remount-**ro** denotes remounting in read-only mode. In Q10, "anywhere" and "within" describe the locations of the holes or non-overwritten blocks (NOB); "within" does not include the end of the file. Entries with a superscript denote a problem.

second `fsync` in Step 3. Steps 4 and 5 are the same as above, and thus the inode table is checkpointed.

Thus, applications that read this data block while the page remains in the page cache (i.e., the page has not been evicted and the OS has not been rebooted) will see the new contents of the data; however, when the page is no longer in memory and must be read from disk, applications will see the old contents.

Alternatively, if `fsync` failed, it could be because a write to one of the journal blocks failed (trace in Figure 3.2c). In this case, ext4 aborts the journal transaction and remounts the file system in read-only mode, causing all future `writes` to fail.

Multi Block Append ($\mathbf{w_{ma}}$). This next workload exercises additional cases in the `fsync` error path. If there are no errors and all `fsyncs` are successful (trace in Figure 3.3a), the multi-block append workload on ext4 behaves as follows. First, during `write`, ext4 creates a new page with the new contents and marks it dirty (Step 1). On `fsync`, the page is written to a newly allocated on-disk data block; after the data-block write completes successfully, the relevant metadata (i.e., both the inode table and the block bitmap) are written to the journal, and `fsync` returns success (Step 2). As in w_{su}, the page is marked clean and contains the newly written data. During `sleep`, the metadata is checkpointed to disk (Step 3); specifically, the inode contains the new modification time and a link to the newly allocated block, and the block bitmap now indicates that the newly allocated block is in use. The pattern is repeated for the second `write` (Step 4), `fsync` (Step 5), and `sleep` (Step 6). As in w_{su}, there are no write requests or changes in page state during `close` (Step 7).

An `fsync` failure could again indicate numerous problems. First, a write to a data block could have failed in Step 2 (trace in Figure 3.3b). If this is the case, the `fsync` fails and the page is marked clean; as in w_{su}, the page cache contains the newly written data, differing from the on-disk block

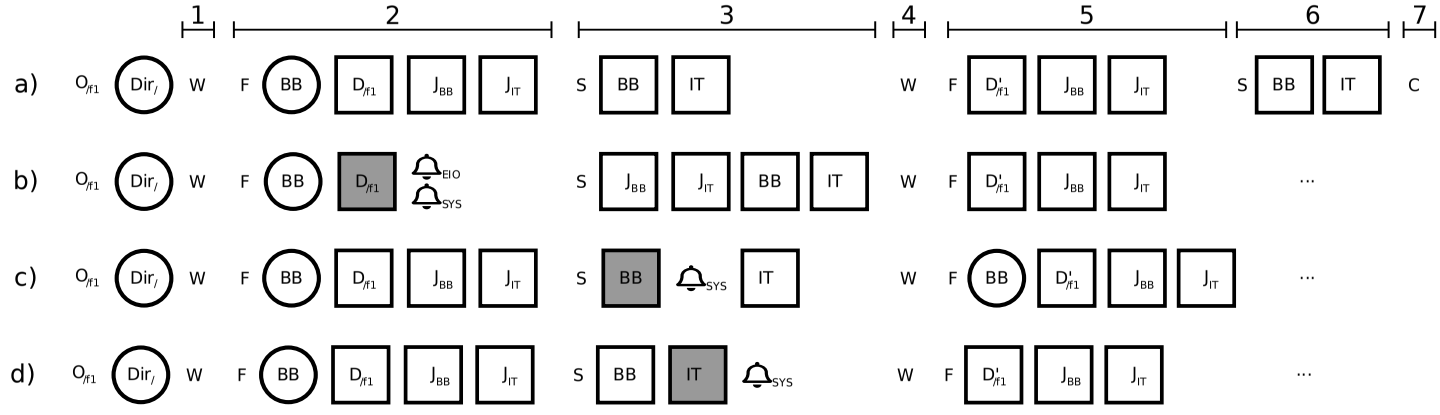

Figure 3.3: **Blockviz traces for w_{ma} on ext4 ordered mode**:
The figure shows four traces corresponding to different fault injection configurations of dm-loki.

a) when no faults are injected: Since w_{ma} appends, ext4 must allocate new blocks to the inode. Allocation is done on `fsync` as ext4 ordered mode uses delayed allocation by default. To allocate a block, the data block bitmap (BB) is read from disk if not already cached. While similar to w_{su}, an fsync in w_{ma} involves writing both inode table (IT) and block bitmap (BB) to the journal and later to their actual locations during checkpointing.

b) dm-loki configured to fail the data block write: While the data block bitmap (BB) and inode table (IT) are not journaled after data block failure, they are still journaled and written to disk during checkpointing.

c) dm-loki configured to fail the data block bitmap (BB) write: Failures during checkpointing are only logged to syslog while checkpointing continues. A stale block bitmap is re-read from disk after bitmap block failure.

d) dm-loki configured to fail the inode table (IT) write: Unlike c), inode tables are never read again from disk despite failure.

that contains the original block contents. The inode table and block bitmap are first journaled and then written to disk in Step 3; thus, even though the data itself has not been written, the inode is modified to reference this block and the corresponding bit is set in the block bitmap. When the workload writes another 4KB of data in Step 4, this write continues oblivious of the previous fault and Steps 5, 6, and 7 proceed as usual.

Thus, with a data-block failure, the on-disk file contains a non-overwritten block where it was supposed to contain the data from Step 1. A similar possibility is that the write to a data block in Step 5 fails; in this case, the file has a non-overwritten block at the end instead of somewhere in the middle. Again, an application that reads any of these failed data blocks while they remain in the page cache will see the newly appended contents; however, when any of those pages are no longer in memory and must be read from disk, applications will read the original block contents.

An `fsync` failure could also indicate that a write to a journal-block failed. In this case, as in w_{su}, the `fsync` returns an error and the following `write` fails since ext4 has been remounted in read-only mode.

Because this workload contains an `fsync` after the metadata has been checkpointed in Step 3, it also illustrates the impact of faults when checkpointing the inode table and block bitmap. We find that ext4 reacts differently to block bitmap and inode table write failures (traces in Figure 3.3 c and d). In both cases, the failure is only logged to syslog, checkpointing proceeds to write other metadata, and the following fsync does not return an error. However, when ext4 fails to write the block bitmap, it marks the associated buffer head `!uptodate`, indicating that a future read must first retrieve the on-disk contents. On `fsync` in Step 5 (or `write` in Step 4 if there is no delayed allocation), ext4 must query the block bitmap to allocate a new block, reloading the stale on-disk block bitmap. With no more write failures, the fsync in Step 5 succeeds and checkpointing proceeds to write the new block bitmap - a version where only the bit for

the second block in the file is set. The filesystem is now in an inconsistent state, with an inode pointing to a block whose bit is not set in the bitmap. While `fsck` can fix this inconsistency, it has to run in force mode (`fsck -f`) as ext4 incorrectly marks the filesystem clean on unmount.

We do not observe such inconsistencies with inode table write failures as ext4 ignores the `!uptodate` flag on inode table buffer heads. Despite being `!uptodate`, ext4 continues to read and write to the latest in-memory inode table. Future successful writes to the on-disk inode table are guaranteed to have all the changes.

We note that for none of these `fsync` and metadata checkpoint failures does ext4 ordered mode recommend running the file system checker; furthermore, running the checker does not identify or repair any of the preceding problems. Finally, future calls to `fsync` never retry previous data writes that may have failed; neither are failed metadata writes during checkpointing. These results for ext4 ordered mode are all summarized in Table 3.1.

The ext4 file system also offers functionality to abort the journal if an error occurs in a file data buffer (mount option `data_err=abort`) and remount the file system in read-only mode on an error (mount option `errors=remount-ro`). However, we observe that the results are identical with and without the mount options. [4]

Multi File Create ($\mathbf{w_{dir}}$). While w_{su} and w_{ma} address data-block, inode-table, and data-block-bitmap failures, w_{dir} exercises failures related to directory data blocks and inode bitmap blocks (trace in Figure 3.4a). If there are no errors and all `fsyncs` are successful, the multi-file create workload on ext4 behaves as follows. First, during `open(dir)`, the directory data block is read from disk if not already cached (Step 1). Next, on `creat(file1)`, to allocate a new inode, ext4 reads the corresponding inode bitmap block from disk if not already cached (Step 2). Ext4 proceeds to

[4] We verified our observations by reproducing them using standard Linux tools and have filed a bug report for the same [80].

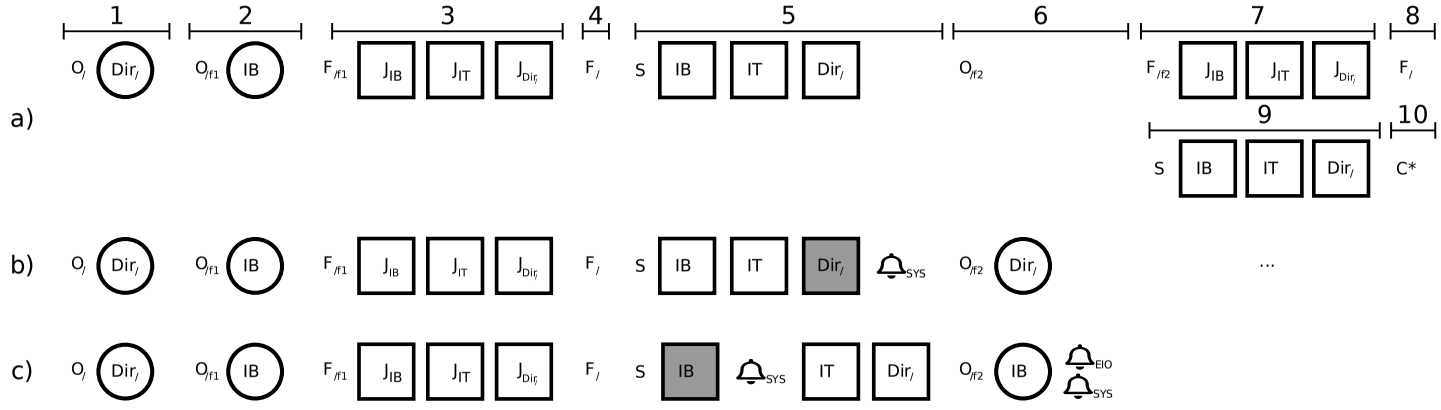

Figure 3.4: **Blockviz traces for w_{dir} on ext4 ordered mode**:
The figure shows three traces corresponding to different fault injection configurations of dm-loki.

a) when no faults are injected: Since w_{dir} creates new inodes, when the first file is created ($O_{/f1}$) the inode bitmap (IB) is read from disk if not already in cache. We use C_* to denote closing all open file descriptors (`/`, `/f1`, and `/f2`).

b) dm-loki configured to fail the directory data block write: A stale version of the directory data block is read from disk (as seen after $O_{/f2}$) if ext4 encounters a write failure during checkpointing for the same block.

c) dm-loki configured to fail the inode bitmap block (IB) write: A stale version of the inode bitmap is read from disk, but ext4 fails the inode creation ($O_{/f2}$), logging the error to syslog and setting errno to EIO.

modify the following data structures and marks them dirty: inode bitmap for file1's inode, inode table entry for file1 and dir, dir's directory data block that contains file1's name-to-inode mapping. On `fsync(file1)`, the dirtied metadata is written to the journal and `fsync(file1)` returns success (Step 3). Ext4 treats directory data blocks as metadata, so unlike w_{su} and w_{ma}, this workload does not write data blocks to disk during an `fsync`. On `fsync(dir)`, we observe no read or write requests as `dir` is already synced during `fsync(file1)` (Step 4). During `sleep`, the metadata is checkpointed to disk (Step 5); specifically, the inode bitmap has a previously cleared bit set, the inode table entry for the directory has a new modification time and updated size, the inode table entry for the file is initialized, and the directory data block has a new name-to-inode mapping. The pattern is repeated for `creat(file2)` (Step 6), `fsync(file2)` (Step 7), `fsync(dir)`, and `sleep`. As in the previous two workloads, we observe no bio requests during `close`.

As `fsync` in Step 3 and Step 7 only involve journal-block writes, similar to w_{su} and w_{ma}, a block write failure during `fsync` in w_{dir} will always return an error and trigger a remount in read-only mode.

Because this workload contains a `sleep` in Step 5, it also illustrates the impact of faults when checkpointing the inode table, inode bitmap, and directory data block. Inode table failures behave exactly as described for w_{ma}. Similar to block bitmap write failures, inode bitmap and directory data block write failures both mark the associated buffer heads `!uptodate` and trigger a read of the stale on-disk version during `creat(file2)` in Step 6. However, directory data block failures are problematic while inode bitmap failures are benign.

We find that files may disappear from directories even while the file system is running. After a directory data block write failure during checkpointing (trace in Figure 3.4b), because of the `!uptodate` flag, ext4 reads and modifies a stale version during `create(file2)`; the in-memory name-

to-inode mapping for file1 is lost and the inode for file1 is an orphaned inode[5]. Future calls to `readdir(dir)` either directly or through the `ls` command will not contain `file1`. Although ext4 does not prompt us to run a checker, running `fsck -f` can detect orphaned inodes and place them in the `lost+found` directory. However, applications that encode information in the filename still suffer data loss.

Since ext4 must refer to both the inode bitmap and inode table when allocating a new inode, it detects the inconsistency and fails the system call. Additionally, to prevent further errors on `creat`, ext4 locks the entire group described by the inode bitmap and recommends running `fsck` to set the bit and unlock the group. This trace can be found in Figure 3.4c.

Ext4 Data Mode: Ext4 data mode differs from ordered mode in that data blocks are first written to the journal and then later checkpointed to their final in-place block locations.

As shown in Table 3.1, the behavior of `fsync` in ext4 data mode is similar to that in ext4 ordered mode for most cases: for example, on a write error, pages may be marked clean even if they were not written out to disk, the file system is remounted in read-only mode on journal failures, meta-data failures are not reported by `fsync`, and files can end up with non-overwritten blocks in the middle or end.

However, the behavior of ext4 data mode differs in one important scenario. Because data blocks are first written to the journal and later to their actual block locations during checkpointing, the first `fsync` after a write may succeed even if a data block will not be successfully written to its permanent in-place location. As a result, a data-block fault causes the second `fsync` to fail instead of the first; in other words, the error reporting by `fsync` is delayed due to a *failed intention* [48]. This trace can be seen in Figure 3.5.

[5]Orphaned inodes are inodes that can never be accessed as no directory points to them.

Figure 3.5: **Blockviz trace for $\mathbf{w_{ma}}$ on ext4 data mode, data block failures**:
Unlike ext4 ordered mode, data blocks are written to the journal during `fsync` ($J_{D_{/f1}}$) and the data block bitmap is read on `write` instead of `fsync` as delayed allocation is disabled. On data block write failure during checkpointing, the error is logged to syslog. The second fsync writes the data and metadata from the second `write` to the journal but fails the `fsync` call with errno set to EIO.

3.1.2.2 XFS

XFS is a journaling file system that uses B-trees. Instead of performing physical journaling like ext4, XFS journals logical entries for changes in metadata.

Figure 3.6a shows a trace of XFS without any failures for w_{su}. As shown in Table 3.1, from the perspective of error reporting and `fsync` behavior, XFS is similar to that of ext4 ordered mode. Specifically, failing to write data blocks (trace in Figure 3.6b) leads to `fsync` failure and the faulty data pages are marked clean even though they contain new data that has not been propagated to disk; as a result, applications that read this faulty data will see the new data only until the page has been evicted from the page cache. Similarly, failing to write a journal block will cause `fsync` failure (trace in Figure 3.6c), while failing to write a metadata block will not. XFS remains available for reads and writes after data-block faults.

XFS handles `fsync` failures in a few ways that are different than ext4 ordered mode. First, on a journal-block fault, XFS shuts down the file system entirely (Figure 3.6c) instead of merely remounting in read-only mode; thus, all subsequent read and write operations fail. Second, XFS retries metadata writes when it encounters a fault during checkpointing; the retry limit is determined by a value in `/sys/fs/xfs/*/error/metadata/*/max_retries`; its value is infinite by default. If the retry limit is exceeded, XFS again shuts down the file system. We provide traces for w_{ma} in Figure 3.7 to highlight the retries.

The multi-block append workload illustrates how XFS handles metadata when writes to related data blocks fail. If the write to the first data block fails, XFS writes no metadata to the journal and fails the `fsync` immediately. When later data blocks are successfully appended to this file, the metadata is updated which creates a non-overwritten block in the file corresponding to the first write. However, if no new data blocks are successfully appended, the on-disk metadata is not updated to reflect

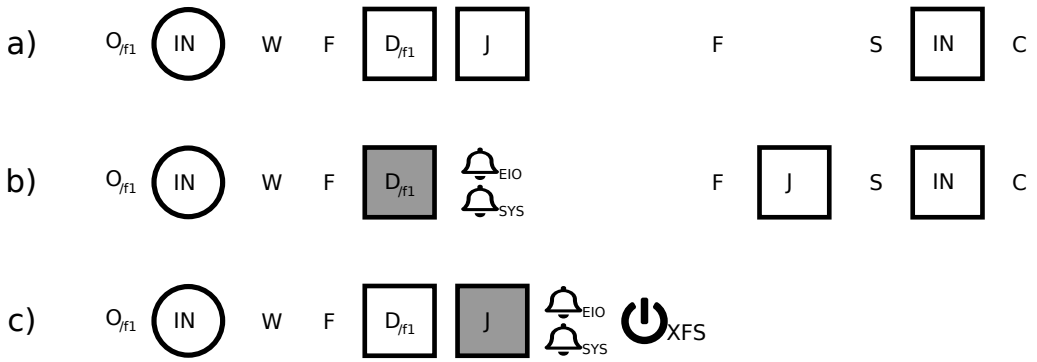

Figure 3.6: **Blockviz traces for $\mathbf{w}_{su}$ on XFS**:
The figure shows three traces corresponding to different fault injection configurations of dm-loki.

a) when no faults are injected: On `open`, XFS reads inode information from disk if not already cached (IN); it includes directory entries. On `fsync`, like ext4, XFS writes the data block and then journals metadata related to the changes. During a checkpoint, the inode information (IN) with updated mtime is written to disk.

b) dm-loki configured to fail the data block write: XFS immediately fails the `fsync` after a data block failure. However, the updated mtime is journaled in the second `fsync` and checkpointed during the `sleep`.

c) dm-loki configured to fail the journal block write: On journal block failure, XFS fails the fsync and shuts down the file system ($\mathbf{\Phi}_{\text{XFS}}$).

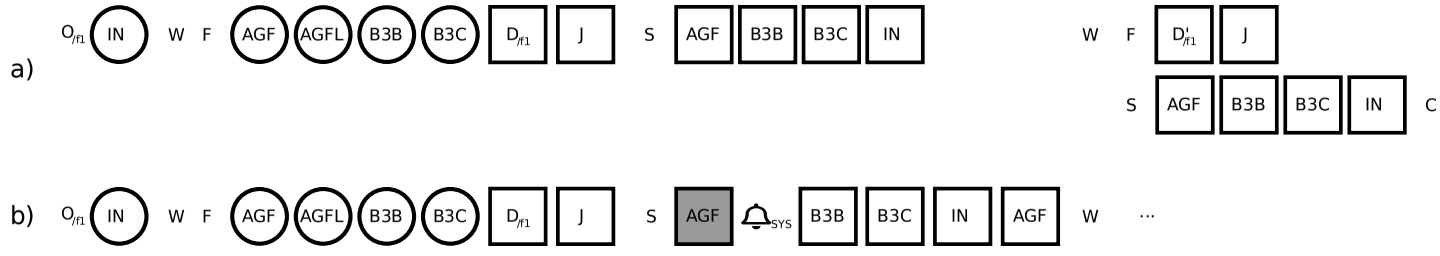

Figure 3.7: **Blockviz traces for w_{ma} on XFS**:
The figure shows two traces corresponding to different fault injection configurations of dm-loki.

a) when no faults are injected: Like ext4 ordered mode, XFS uses delayed allocation and must allocate new blocks to the inode since w_{ma} is appending to the file. Unlike ext4's free space block bitmap, XFS tracks free space for every allocation group (AG - groups of inodes) using two B+trees which need to be read from disk if not cached. First, it reads the allocation group free space block (AGF) which contains information about the B+trees. Next, it reads the allocation group free list block (AGFL) which contains pointers to free space for growing the B+trees. It then reads the two B+trees, the first is sorted by block number (B3B) to quickly find space closer to a given block. The second tracks space by size (B3C) to quickly find free space of a given size. On `fsync`, the data block is written and the free space changes are journaled along with the inode changes. During checkpointing, these changes are written to their actual disk locations. Since the B+trees did not need to grow, there are no modifications to AGFL. With no failures, the pattern repeats for the next `write`, `fsync`, and `sleep`.

b) dm-loki configured to fail a metadata block during checkpointing: During checkpointing, XFS retries the failed write to the allocation group free space block (AGF). The retry limit is configurable and set to infinity by default. However, if the limit is reached, XFS shuts down the file system and recommends running `fsck`. While this trace shows faults for AGF, we observe similar behavior for failures on B3B, B3C, and IN.

any of these last writes (i.e., the size of the file is not increased). [6] Thus, while in ext4 a failed write always causes a non-overwritten block, in XFS, non-overwritten blocks cannot exist at the end of a file. However, for either file system, if the failed blocks remain in the page cache, applications can read those blocks regardless of whether they are in the middle or the end of a file.

During checkpointing, since XFS either shuts down or retries writes on metadata failures, we do not observe the same inconsistencies as described for ext4 when running the multi-file create workload w_{dir}.

3.1.2.3 Btrfs

Btrfs is a copy-on-write file system that avoids writing to the same block twice except for the superblock which contains root-node information. Figure 3.8a provides a trace of w_{su} without any failures along with a description of Btrfs's data structures. At a high level, some of the actions in Btrfs are similar to those in a journaling file system: instead of writing to a journal, Btrfs writes to a log tree to record changes when an `fsync` is performed; instead of checkpointing to fixed in-place locations, Btrfs writes to new locations and updates the roots in its superblock. However, since Btrfs is based on copy-on-write, it has a number of interesting differences in how it handles `fsync` failures compared to ext4 and XFS, as shown in Table 3.1.

Like ext4 ordered mode and XFS, Btrfs fails `fsync` when it encounters data-block faults (trace in Figure 3.8b). However, unlike ext4 and XFS, Btrfs effectively reverts the contents of the data block (and any related metadata) back to its old state (and marks the page clean). Thus, if an application reads the data after this failure, it will never see the failed

[6] To be precise, the mtime and ctime of the file are updated, but not the size of the file. Additional experiments removed for space confirm this behavior.

operation as a temporary state. As in the other file systems, Btrfs remains available after this data-block fault.

Similar to faults to the journal in the other file systems, faults to Btrfs's log tree can result in a failed `fsync` and a remount in read-only mode. However, as seen in Figure 3.8c, Btrfs can recover from log-tree failures by attempting a full-tree commit immediately after the failure (as opposed to periodically during checkpointing). If the full-tree commit succeeds, Btrfs ignores the log-tree failure and returns success for `fsync`. However, if there were another failure during the full-tree commit, Btrfs would fail the `fsync` and remount in read-only mode. As Btrfs also performs a full-tree commit periodically during checkpointing, unlike ext4 and XFS, faults during checkpointing (trace in Figure 3.8d) result in a remount in read-only mode.

The multi-block append workload illustrates interesting behavior in Btrfs block allocation. If the first append fails, the state of the file system, including the B-tree that tracks all free blocks, is reverted. However, the next append will continue to write at the (incorrectly) updated offset stored in the file descriptor, creating a hole in the file. Since the state of the B-tree was reverted, the deterministic block allocator will choose to allocate the same block again for the next append operation. Thus, if the fault to that particular block was transient, the next `write` and `fsync` will succeed and there will simply be a one block hole in the file. If the fault to that particular block occurs multiple times, future writes will continue to fail; as a result, Btrfs may cause more holes within a file than ext4 and XFS. However, unlike ext4 and XFS, the file does not have block overwrite failures.

During checkpointing, since Btrfs remounts in read-only mode on metadata write failures, we do not observe the same inconsistencies as described for ext4 when running the multi-file create workload $w_{\texttt{dir}}$.

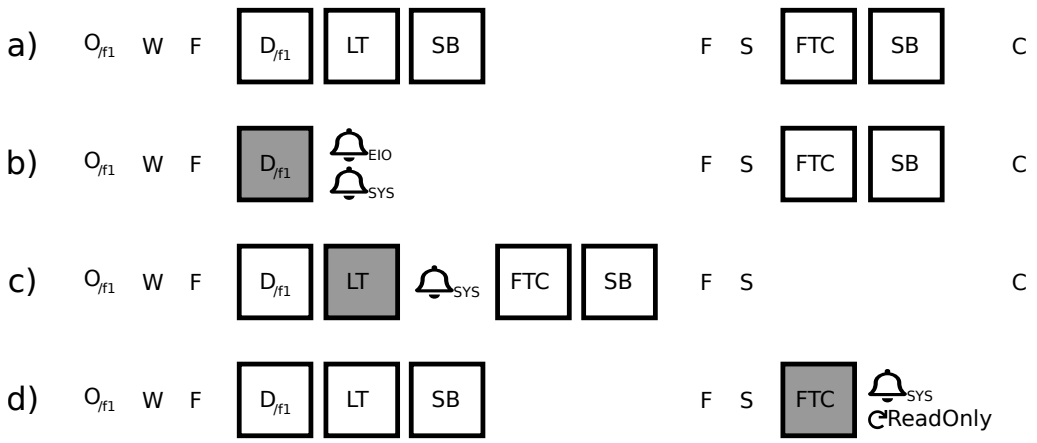

Figure 3.8: **Blockviz traces for w_{su} on btrfs**:
The figure shows four traces corresponding to different fault injection configurations of dm-loki.

a) when no faults are injected: In a copy-on-write file system like btrfs, any modification to data or metadata involves creating a copy of the modified nodes in the tree. To avoid too much redundant I/O by forcing repeated copy-on-write for modified parts of the trees, Btrfs journals fsync-triggered copy-on-writes to a log tree (LT). The super block (SB) is also updated as it contains a reference to the updated log tree root. During `sleep`, btrfs checkpoints state by performing a full tree commit (FTC) which involves writing all modified trees and deleting items from the log tree, followed by a write to the superblock which contains references to all the tree roots.

b) dm-loki configured to fail the data block write: On data block failure, btrfs fails the `fsync` and reverts state. Unlike ext4 and XFS, we observe no write requests during the second `fsync`. However, the modification and reversal trigger an unnecessary full tree commit during `sleep`.

c) dm-loki configured to fail a block write in the log tree: When btrfs encounters a log tree write failure, it logs the error to syslog and starts a full tree commit. Since we only fail one particular block, the full tree commit succeeds and fsync does not fail. As there are no changes after the last full tree commit, there are no write requests during `sleep`.

d) dm-loki configured to fail a block write during a full tree commit: If btrfs encounters a write failure during a full tree commit, it logs the error to syslog and remounts in read-only mode (ReadOnly). We observe this behavior for any full tree commit, both periodically (in w_{su}, w_{ma}, w_{dir} during `sleep`) and triggered on log tree failures as seen in subfigure c. When triggered due to log tree failures, the fsync fails with errno set to EIO.

3.1.2.4 File System Summary

We now present a set of observations for the file systems based on the questions from Section §3.1.1.5.

File System Behavior to `fsync` Failures. On all the three file systems, only data and journal-block failures lead to `fsync` failures (Q1). Metadata-block failures do not result in `fsync` failures as metadata blocks are written to the journal during an `fsync`. However, during a checkpoint, any metadata failure on XFS and Btrfs lead to unavailability (Q8) while ext4 logs the error and continues.[7]

On both modes of ext4 and XFS, metadata is persisted even after the file system encounters a data-block failure (Q2); timestamps are always updated in both file systems. Additionally, ext4 appends a new block to the file and updates the file size while XFS does so only when followed by a future successful `fsync`. As a result, we find non-overwritten blocks in both the middle and end of files for ext4, but in only the middle for XFS (Q10). Btrfs does not persist metadata after a data-block failure. However, because the process file-descriptor offset is incremented, future `writes` and `fsyncs` cause a hole in the middle of the file (Q10).

Among the three, XFS is the only file system that retries metadata-block writes. However, none of them retry data or journal-block writes (Q3).

All the file systems mark the page clean even after `fsync` fails (Q4). In both modes of ext4 and XFS, the page contains the latest write while Btrfs reverts the in-memory state to be consistent with what is on disk (Q5).

We note that even though all the file systems mark the page clean, this is not due to any behavior inherited from the VFS layer. Each file system registers its own handlers to write pages to disk (`ext4_writepages`, `xfs_vm_writepages`, and `btrfs_writepages`). However, each of these handlers call `clear_page_dirty_for_io` before submitting the bio request and

[7] Ext4's error handling behavior for metadata has unintended side-effects but we omit the results as the rest of the paper focuses on data-block failures.

do not set the dirty bit in case of failure in order to avoid memory leaks[8], replicating the problem independently.

Failure Reporting. While all file systems report data-block failures by failing `fsync`, ext4 ordered mode, XFS, and Btrfs fail the immediate `fsync`. As ext4 data mode puts data in the journal, the first `fsync` succeeds and the next `fsync` fails. (Q6). All block write failures, irrespective of block type are logged in the syslog (Q7).

After Effects. Journal block failures always lead to file-system unavailability. On XFS and Btrfs, metadata-block failures do so as well (Q8). While ext4 and Btrfs remount in read-only mode, XFS shuts down the file system (Q9). Holes and non-overwritten blocks (Q10) have been covered previously as part of Q2.

Recovery. None of the file systems alert the user to run a file-system checker. However, as Btrfs records intentionally created holes as zero-byte extents, holes created through fsync failures (as seen in w_{ma}) can be detected by btrfsck due to missing zero-byte extent information (Q11).

While file systems may differ in how they handle failures, it is important that they all have bit-to-state consistency. If the content in memory does not match the disk, then the dirty bit must be set. Additionally, if a file system acknowledges that data is committed with a successful return code, it must never change that decision with a later operation (e.g., checkpointing).

3.2 Application Study

We now focus on how applications are affected by `fsync` failures. In this section, we first describe our fault model with CuttleFS, followed by a description of the workloads, execution environment, and the errors we

[8]Ext4 focuses on the common case of users removing USB sticks while still in use. Dirty pages that can never be written to the removed USB stick have to be marked clean to unmount the file system and reclaim memory [23].

look for. Then, we present our findings for five widely used applications: Redis (v5.0.7), LMDB (v0.9.24), LevelDB (v1.22), SQLite (v3.30.1), and PostgreSQL (v12.0).

3.2.1 CuttleFS

We limit our study to how applications are affected by data-block failures as journal-block failures lead to unavailability and metadata-block failures do not result in `fsync` failures (§3.1.2). Our fault model is simple: when an application writes data, we inject a single fault to a data block or a sector within it.

We build CuttleFS[9] [20] - a FUSE [53] file system to emulate the different file-system reactions to failures defined by our fault model. Instead of using the kernel's page cache, CuttleFS maintains its own page cache in user-space memory. Write operations modify user-space pages and mark them dirty while read operations serve data from these pages. When an application issues an `fsync` system call, CuttleFS synchronizes data with the underlying file system.

CuttleFS has two modes of operation: trace mode and fault mode. In trace mode, CuttleFS tracks writes and identifies which blocks are eventually written to disk. This is different from just tracing a `write` system call as an application may write to a specific portion of a file multiple times before it is actually flushed to disk.

In fail mode, CuttleFS can be configured to fail the i^{th} write to a sector or block associated with a particular file. On `fsync` failure, as CuttleFS uses in-memory buffers, it can be directed to mark a page clean or dirty, keep the latest content, or revert the file to the previous state. Error reporting behavior can be configured to report failures immediately or on the next

[9]Cuttlefish are sometimes referred to as the "chameleons of the sea" because of their ability to rapidly alter their skin color within a second. CuttleFS can change characteristics much faster.

`fsync` call. In short, CuttleFS can react to `fsync` failures in any of the ways mentioned in Table 3.1 (Q4,5,6). Additionally, CuttleFS accepts commands to evict all or specific clean pages.

We configure CuttleFS to emulate the failure reactions of the file systems studied in Section 3.1.2. For example, in order to emulate ext4 ordered mode and XFS (as they both have similar failure reactions), we configure CuttleFS to mark the page clean, keep the latest content, and report the error immediately. Henceforth, when presenting our findings and referring to characteristics emulated by CuttleFS, we use $\text{CuttleFS}_{\text{ext4o,xfs}}$ for the above configuration. When the page is marked clean, has the latest content, but the error is reported on the next `fsync`, we use $\text{CuttleFS}_{\text{ext4d}}$. When the page is marked clean, the content matches what is on disk, and the error is reported immediately, we refer to it as $\text{CuttleFS}_{\text{btrfs}}$.

3.2.2 Workloads and Execution Environment

We run CuttleFS in trace mode and identify which blocks are written to by an application. For each application, we choose a simple workload that inserts a single key-value pair, a commonly used operation in many applications. We perform experiments both with an existing key (update) as well as a new key (insert). The keys can be of size 2B or 1KB.[10] The values can be of size 2B or 12KB. We run experiments for all four combinations. The large keys allow for the possibility of failing a single sector within the key and large values for pages within a value. Since SQLite and PostgreSQL are relational database management systems, we create a single table with two columns: keys and values.

Using the trace, we generate multiple failure sequences for each of the identified blocks and sectors within them. We then repeat the experiment multiple times with CuttleFS in fault mode, each time with a different

[10] As LMDB limits key sizes to 511B, we use key sizes of 2B and 511B for LMDB experiments.

failure sequence and file-system reaction. In order to observe the effects after a fault, we dump all key-value pairs before and after the workload.

We look for the following types of errors when performing the experiments:

- **OldValue (OV):** The system returns the new value for a while but then reverts to an old value, or the system conveys a successful response but returns the old value later on.
- **FalseFailure (FF):** The system informs the user that the operation failed but returns the new value in the future.
- **KeyCorruptions (KC)** and **ValueCorruptions (VC):** Corrupted keys or values are obliviously returned.
- **KeyNotFound (KNF):** The system informs the user that it has successfully inserted a key but it cannot be found later on, or the system fails to update a key to a new value but the old key-value pair disappears as well.

We also identify the factors within the execution environment that cause all these errors to be manifested. If an application maintains its own in-memory data structures, some errors may occur only when an application restarts and rebuilds in-memory state from the file system. Alternatively, the manifestation of these errors may depend on state changes external to the application, such as a single page eviction or a full page cache flush. We encode these different scenarios as:

- **App=KeepGoing:** The application continues without restarting.
- **App=Restart:** The application restarts either after a crash or a graceful shutdown. This forces the application to rebuild in-memory state from disk.

- **BufferCache=Keep:** No evictions take place.
- **BufferCache=Evict:** One or more clean pages are evicted.

Note that BufferCache=Evict can manifest by clearing the entire page cache, restarting the file system, or just evicting clean pages due to memory pressure. A full system restart would be the combination of App=Restart and BufferCache=Evict, which causes a loss of both clean and dirty pages in memory while also forcing the application to restart and rebuild state from disk.

Configuring CuttleFS to fail a certain block and react according to one of the file-system reactions while the application runs only addresses App=KeepGoing and BufferCache=Keep. The remaining three scenarios are addressed as follows. To simulate App=Restart and BufferCache=Keep, we restart the application and dump all key-value pairs, ensuring that no page in CuttleFS is evicted. To address the remaining two scenarios, we instruct CuttleFS to evict clean pages for both App=KeepGoing and App=Restart.

3.2.3 Findings

We configured all five applications to run in the form that offers most durability and describe them in their respective sections. Table 3.2 summarizes the per-application results across different failure characteristics.

Note that these results are only for the simple workload that inserts a single key-value pair. A complex workload may exhibit more errors or mask the ones we observe.

Redis: Redis is an in-memory data-structure store, used as a database, cache, and message broker. By default, it periodically snapshots in-memory state to disk. However, for better durability guarantees, it provides options for writing every operation that modifies the store to an append-only file

	A=KeepGoing	A=Restart
BC=Keep	⧹	⟋
BC=Evict	|	−

ext4o,xfs = { clean, differs, immediate } ext4d = { clean, differs, next `fsync` } btrfs = { clean, matches, immediate }

Applications		ext4o,xfs OV	ext4o,xfs FF	ext4o,xfs KC	ext4o,xfs VC	ext4o,xfs KNF	ext4d OV	ext4d FF	ext4d KC	ext4d VC	ext4d KNF	btrfs OV	btrfs FF	btrfs KC	btrfs VC	btrfs KNF
Redis		−		−	−	−	−		−	−	−	≠				≠
LMDB			−				+			+						
LevelDB			⟋				−		+	+	+					
SQLite	Rollback		+		+			+		−	−		✳			
	WAL		⟋							−	−					
PostgreSQL	Default		≠				−				−					
	Direct I/O						−				−					

Table 3.2: **Findings for Applications on `fsync()` Failure.** The table lists the different types of errors that manifest for applications when `fsync` fails due to a data-block write fault. The errors (OV, FF, KC, VC, KNF) are described in §3.2.2. We group columns depending on how a file system reacts to an `fsync` failure according to our findings in §3.1.2 for Q4, Q5, and Q6. For example, both ext4 ordered and XFS (ext4o,xfs) mark a page **clean**, the page **differs** in in-memory and on-disk content, and the `fsync` failure is reported **immediately**. For each application, we describe when the error manifests, in terms of combinations of the four different execution environment factors (§3.2.2) whose symbols are provided at the top left corner. For example, OldValue manifests in Redis in the first group (ext4-ordered, XFS) only on (A)App=Restart,(BC)BufferCache=Evict. However, in the last group (Btrfs), the error manifests both on App=Restart,BufferCache=Evict as well as App=Restart,BufferCache=Keep, depicted as a combination of the two symbols.

(*aof*) [95] and how often to `fsync` the *aof*. In the event of a crash or restart, Redis rebuilds in-memory state by reading the contents of the *aof*.

We configure Redis to `fsync` the file for every operation, providing strong durability. Thus, whenever Redis receives a request like an insert operation that modifies state, it writes the request to the *aof* and calls `fsync`. However, Redis trusts the file system to successfully persist the data and does not check the `fsync` return code. Regardless of whether `fsync` fails or not, Redis returns a successful response to the client.

As Redis returns a successful response to the client irrespective of `fsync` failure, FalseFailures do not occur. Since Redis reads from disk only when rebuilding in-memory state, errors may occur only during App=Restart.

On CuttleFS$_{ext4o,xfs}$ and CuttleFS$_{ext4d}$, Redis exhibits OldValue, KeyCorruption, ValueCorruption, and KeyNotFound errors. However, as seen in Table 3.2, these errors occur only on BufferCache=Evict and App=Restart. On BufferCache=Keep, the page contains the latest write which allows Redis to rebuild the latest state. However, when the page is evicted, future reads will force a read from disk, causing Redis to read whatever is on that block. OldValue and KeyNotFound errors manifest when a fault corrupts the *aof* format. When Redis restarts, it either ignores these entries when scanning the *aof*, or recommends running the *aof checker* which truncates the file to the last non-corrupted entry. A KeyCorruption and ValueCorruption manifest when the fault is within the key or value portion of the entry.

On CuttleFS$_{btrfs}$, Redis exhibits OldValue and KeyNotFound errors. These errors occur on App=Restart, regardless of buffer-cache state. When Redis restarts, the entries are missing from the *aof* as the file was reverted, and thus, the insert or update operation is not applied.

LMDB: Lightning Memory-Mapped Database (LMDB) is an embedded key-value store which uses B+Tree data structures whose nodes reside

in a single file. The first two pages of the file are metadata pages, each of which contain a transaction ID and the location of the root node. Readers always use the metadata page with the latest transaction ID while writers make changes and update the older metadata page.

LMDB uses a copy-on-write bottom-up strategy [69] for committing write transactions. All new nodes from leaf to root are written to unused or new pages in the file, followed by an `fsync`. An `fsync` failure terminates the operation without updating the metadata page and notifies the user. If `fsync` succeeds, LMDB proceeds to update the old metadata page with the new root location and transaction ID, followed by another `fsync`.[11] If `fsync` fails, LMDB writes an old transaction ID to the metadata page in memory, preventing future readers from reading it.

On CuttleFS$_{ext4o,xfs}$, LMDB exhibits FalseFailures. When LMDB writes the metadata page, it only cares about the transaction ID and new root location, both of which are contained in a single sector. Thus, even though the sector is persisted to disk, failures in the seven other sectors of the metadata page can cause an `fsync` failure.[12] As mentioned earlier, LMDB writes an old transaction ID (say ID1) to the metadata page in memory and reports a failure to the user. However, on BufferCache=Evict and App=Restart (such as a machine crash and restart), ID1 is lost as it was only written to memory and not persisted. Thus, readers read from the latest transaction ID which is the previously failed transaction.

LMDB does not exhibit FalseFailures in CuttleFS$_{ext4d}$ as the immediate successful `fsync` results in a success to the client. Instead, ValueCorruptions and OldValue errors occur on BufferCache=Evict, regardless of whether the application restarts or not. ValueCorruptions occur when a block containing a part of the value experiences a fault. As

[11] To be precise, LMDB does not do a write followed by an `fsync` for metadata page updates. Instead, it uses a file descriptor that is opened in O_SYNC mode. On a write, only the metadata page is flushed to disk. On failure, it uses a normal file descriptor.

[12] CuttleFS can fail the i^{th} write to a sector or block(§3.2.1). We observed FalseFailures in LMDB when CuttleFS was configured to fail writes to sectors in the metadata pages.

LMDB `mmaps()` the file and reads directly from the page cache, BufferCache=Evict such as a page eviction leads to reading the value of the faulted block from disk. OldVersion errors occur when the metadata page experiences a fault. The file system responds with a successful `fsync` initially (as data is successfully stored in the ext4 journal). For a short time, the metadata page has the latest transaction ID. However, when the page is evicted, the metadata page reverts to the old transaction ID on disk, resulting in readers reading the old value. KeyCorruptions do not occur as the maximum allowed key size is 511B.

As CuttleFS$_{\text{btrfs}}$ reports errors immediately, it does not face the problems seen in CuttleFS$_{\text{ext4d}}$. FalseFailures do not occur as the file is reverted to its previous consistent state. We observe this same pattern in many of the applications and omit them from the rest of the discussion unless relevant.

LevelDB: LevelDB is a widely used key-value store based on LSM trees. It stores data internally using MemTables and SSTables [33]. Additionally, LevelDB writes operations to a log file before updating the MemTable. When a MemTable reaches a certain size, it becomes immutable and is written to a new file as an SSTable. SSTables are always created and never modified in place. On a restart, if a log file exists, LevelDB creates an SSTable from its contents.

We configure LevelDB to `fsync` the log after every write, for stronger durability guarantees. If `fsync` fails, the MemTable is not updated and the user is notified about the failure. If `fsync` fails during SSTable creation, the operation is cancelled and the SSTable is left unused.

On CuttleFS$_{\text{ext4o,xfs}}$, as seen in Table 3.2, LevelDB exhibits FalseFailures only on App=Restart with BufferCache=Keep. When LevelDB is notified of `fsync` failure to the log file, the user is notified of the failure. However, on restart, since the log entry is in the page cache, LevelDB includes it

while creating an SSTable from the log file. Read operations from this point forward return the new value, reflecting FalseFailures. FalseFailures do not occur on BufferCache=Evict as LevelDB is able to detect invalid entries through CRC checksums [33]. Faults in the SSTable are detected immediately and do not cause any errors as the newly generated SSTable is not used by LevelDB in case of a failure.

On CuttleFS$_{\text{ext4d}}$, LevelDB exhibits KeyNotFound and OldVersion errors when faults occur in the log file. When inserting a key-value pair, `fsync` returns successfully, allowing future read operations to return the new value. However, on BufferCache=Evict and App=Restart, LevelDB rejects the corrupted log entry and returns the old value for future read operations. Depending on whether we insert a new or existing key, we observe KeyNotFound or OldVersion errors when the log entry is rejected. Additionally, LevelDB exhibits KeyCorruption, ValueCorruption, and KeyNotFound errors for faults that occur in the SSTables. Ext4 data mode may only place the data in the journal and return a successful `fsync`. Later, during checkpointing, the SSTable is corrupted due to the fault. These errors manifest only on BufferCache=Evict, either while the application is running or on restart, depending on when the SSTable is read from disk.

SQLite: SQLite is an embedded RDBMS that uses BTree data structures. A separate BTree is used for each table and index but all BTrees are stored in a single file on disk, called the "main database file" (*maindb*). During a transaction, SQLite stores additional information in a second file called the "rollback journal" (*rj*) or the "write-ahead log" (*wal*) depending on which mode it is operating in. In the event of a crash or restart, SQLite uses these files to ensure that committed or rolled-back transactions are reflected in the *maindb*. Once a transaction completes, these files are deleted. We perform experiments for both modes.

SQLite RollBack: In rollback journal mode, before SQLite modifies its user-space buffers, it writes the original contents to the *rj*. On commit, the *rj* is `fsyncd`. If it succeeds, SQLite writes a header to the *rj* and `fsyncs` again (2 `fsyncs` on the *rj*). If a fault occurs at this point, only the state in the user-space buffers need to be reverted. If not, SQLite proceeds to write to the *maindb* so that it reflects the state of the user-space buffers. *maindb* is then `fsyncd`. If the `fsync` fails, SQLite needs to rewrite the old contents to the *maindb* from the *rj* and revert the state in its user-space buffers. After reverting the contents, the *rj* is deleted.

On CuttleFS$_{\text{ext4o,xfs}}$, SQLite Rollback exhibits FalseFailures and ValueCorruptions on BufferCache=Evict, regardless of whether the application restarts or not. When faults occur in the *rj*, SQLite chooses to revert in-memory state using the *rj* itself as it contains just enough information for a rollback of the user-space buffers. This approach works well as long as the latest contents are in the page cache. However, on BufferCache=Evict, when SQLite reads the *rj* to rollback in-memory state, the *rj* does not contain the latest write. As a result, SQLite's user-space buffers can still have the new contents (FalseFailure) or a corrupted value, depending on where the fault occurs.

SQLite Rollback exhibits FalseFailures in CuttleFS$_{\text{ext4d}}$ for the same reasons mentioned above as the `fsync` failure is caught on the second `fsync` to the *rj*. Additionally, due to the late error reporting in CuttleFS$_{\text{ext4d}}$, SQLite Rollback exhibits ValueCorruption and KeyNotFound errors when faults occur in the *maindb*. SQLite sees a successful `fsync` after writing data to the *maindb* and proceeds to delete the *rj*. However, on App=Restart and BufferCache=Evict, the above mentioned errors manifest depending on where the fault occurs.

On CuttleFS$_{\text{btrfs}}$, SQLite Rollback exhibits FalseFailures for the same reasons mentioned above. However, they occur irrespective of whether buffer-cache state changes due to the fact that the contents in the *rj* are

reverted. As there is no data in the *rj* to recover from, SQLite leaves the user-space buffers untouched. ValueCorruptions cannot occur as no attempt is made to revert the in-memory content.

SQLite WAL: Unlike SQLite Rollback, changes are written to a write-ahead log (*wal*) on a transaction commit. SQLite calls `fsync` on the *wal* and proceeds to change in-memory state. If `fsync` fails, SQLite immediately returns a failure to the user. If SQLite has to restart, it rebuilds state from the *maindb* first and then changes state according to the entries in the *wal*. To ensure that the *wal* does not grow too large, SQLite periodically runs a Checkpoint Operation to modify *maindb* with the contents from the *wal*.

On CuttleFS$_{\text{ext4o,xfs}}$, as seen in Table 3.2, SQLite WAL exhibits FalseFailures only on App=Restart with BufferCache=Keep, for reasons similar to LevelDB. It reads valid log entries from the page cache even though they might be invalid due to faults on disk.

On CuttleFS$_{\text{ext4d}}$, SQLite WAL exhibits ValueCorruption and KeyNotFound Errors when there are faults in the *maindb* during a Checkpoint Operation for the same reasons mentioned in SQLite Rollback.

PostgreSQL: PostgreSQL is an object-relational database system that maintains one file per database table. On startup, it reads the on-disk tables and populates user-space buffers. Similar to SQLite WAL, PostgreSQL reads entries from the write-ahead log (*wal*) and modifies user-space buffers accordingly. Similar to SQLite WAL, PostgreSQL runs a checkpoint operation, ensuring that the *wal* does not grow too large. We evaluate two configurations of PostgreSQL: the default configuration and a DirectIO configuration.

PostgreSQL Default: In the default mode, PostgreSQL treats the *wal* like any other file, using the page cache for reads and writes. PostgreSQL notifies the user of a successful *commit* operation only after an `fsync` on the *wal* succeeds. During a checkpoint, PostgreSQL writes data from its user-

space buffers into the table and calls `fsync`. If the `fsync` fails, PostgreSQL, aware of the problems with `fsync` [39], chooses to crash. Doing so avoids truncating the *wal* and ensures that checkpointing can be retried later.

On CuttleFS$_{ext4o,xfs}$, PostgreSQL exhibits FalseFailures for reasons similar to LevelDB. While App=Restart is necessary to read the entry from the log, BufferCache=Evict is not. Further, the application restart cannot be avoided as PostgreSQL intentionally crashes on an `fsync` failure. On BufferCache=Keep, PostgreSQL reads a valid log entry in the page cache. On BufferCache=Evict, depending on which block experiences the fault, PostgreSQL either accepts or rejects the log entry. FalseFailures manifest when PostgreSQL accepts the log entry. However, if the file system were to also crash and restart, the page cache would match the on-disk state, causing PostgreSQL to reject the log entry. Unfortunately, ext4 currently does not behave as expected with mount options `data_err=abort` and `errors=remount-ro` (§3.1.2.1).

Due to the late error reporting in CuttleFS$_{ext4d}$, as seen in Table 3.2, PostgreSQL exhibits OldVersion and KeyNotFound Errors when faults occur in the database table files. As PostgreSQL maintains user-space buffers, these errors manifest only on BufferCache=Evict with App=Restart. During a checkpoint operation, PostgreSQL writes the user-space buffers to the table. As the fault is not yet reported, the operation succeeds and the *wal* is truncated. If the page corresponding to the fault is evicted and PostgreSQL restarts, it will rebuild its user-space buffers using an incorrect on-disk table file. The errors are exhibited depending on where the fault occurs. While KeyNotFound errors occur in other applications when a new key is inserted, PostgreSQL *loses existing keys on updates* as it modifies the table file in-place.

PostgreSQL DIO: In the DirectIO mode, PostgreSQL bypasses the page cache and writes to the *wal* using DirectIO. The sequence of operations during a transaction commit and a checkpoint are exactly the same as the

default mode.

FalseFailures do not occur as the page cache is bypassed. However, OldVersion and KeyNotFound errors still occur in CuttleFS$_{\text{ext4d}}$ for the same reasons mentioned above as writes to the database table files do not use DirectIO.

3.3 Discussion

We now present a set of observations and lessons for handling `fsync` failures across file systems and applications.

#1: Existing file systems do not handle fsync failures uniformly. In an effort to hide cross-platform differences, POSIX is intentionally vague on how failures are handled. Thus, different file systems behave differently after an `fsync` failure (as seen in Table 3.1), leading to non-deterministic outcomes for applications that treat all file systems equally. *We believe that the POSIX specification for `fsync` needs to be clarified and the expected failure behavior described in more detail.*

#2: Copy-on-Write file systems such as Btrfs handle fsync failures better than existing journaling file systems like ext4 and XFS. Btrfs uses new or unused blocks when writing data to disk; the entire file system moves from one state to another on success and no in-between states are permitted. Such a strategy defends against corruptions when only some blocks contain newly written data. *File systems that use copy-on-write may be more generally robust to `fsync` failures than journaling file systems.*

#3: Ext4 data mode provides a false sense of durability. Application developers sometimes choose to use a data journaling file system despite its lower performance because they believe data mode is more durable [30]. Ext4 data mode does ensure data and metadata are in a "consistent state",

but only from the perspective of the file system. As seen in Table 3.2, application-level inconsistencies are still possible. Furthermore, applications cannot determine whether an error received from `fsync` pertains to the most recent operation or an operation sometime in the past. *When failed intentions are a possibility, applications need a stronger contract with the file system, notifying them of relevant context such as data in the journal and which blocks were not successfully written*.

#4: Existing file-system fault-injection tests are devoid of workloads that continue to run post failure. While all file systems perform fault-injection tests, they are mainly to ensure that the file system is consistent after encountering a failure. Such tests involve shutting down the file system soon after a fault and checking if the file system recovers correctly when restarted. *We believe that file-system developers should also test workloads that continue to run post failure, and see if the effects are as intended.* Such effects should then be documented. File-system developers can also quickly test the effect on certain characteristics by running those workloads on CuttleFS before changing the actual file system.

#5: Application developers write OS-specific code, but are not aware of all OS-differences. The FreeBSD VFS layer chooses to re-dirty pages when there is a failure (except when the device is removed) [34] while Linux hands over the failure handling responsibility to the individual file systems below the VFS layer (§3.1.2.4). *We hope that the Linux file-system maintainers will adopt a similar approach in an effort to handle `fsync` failures uniformly across file systems.* Note that it is also important to think about when to classify whether a device has been removed. For example, while storage devices connected over a network aren't really as permanent as local hard disks, they are more permanent than removable USB sticks. Temporary disconnects over a network need not be perceived as device removal and re-attachment; pages associated with such a device can be

re-dirtied on write failure.

#6: Application developers do not target specific file systems. We observe that data-intensive applications configure their durability and error-handling strategies according to the OS they are running on, but treat all file systems on a specific operating system equally. Thus, as seen in Table 3.2, a single application can manifest different errors depending on the file system. *If the POSIX standard is not refined, applications may wish to handle `fsync` failures on different file systems differently.* Alternatively, applications may choose to code against *failure handling characteristics* as opposed to specific file systems, but this requires file systems to expose some interface to query characteristics such as "Post Failure Page State/Content" and "Immediate/Delayed Error Reporting".

#7: Applications employ a variety of strategies when fsync fails, but none are sufficient. As seen in Section 3.2.3, Redis chooses to trust the file system and does not even check `fsync` return codes, LMDB, LevelDB, and SQLite revert in-memory state and report the error to the application while PostgreSQL chooses to crash. We have seen that none of the applications retry `fsync` on failure; application developers appear to be aware that pages are marked clean on `fsync` failure and another `fsync` will not flush additional data to disk. Despite the fact that applications take great care to handle a range of errors from the storage stack (e.g., LevelDB writes CRC Checksums to detect invalid log entries and SQLite updates the header of the rollback journal only after the data is persisted to it), data durability cannot be guaranteed as long as `fsync` errors are not handled correctly. *While no one strategy is always effective, the approach currently taken by PostgreSQL to use direct IO may best handle `fsync` failures.* If file systems do choose to report failure handling characteristics in a standard format, applications may be able to employ better strategies. For example, applications can choose to keep track of dirtied pages and re-dirty them by

reading and writing back a single byte if they know that the page content is not reverted on failure (ext4, XFS). On Btrfs, one would have to keep track of the page as well as its content. For applications that access multiple files, it is important to note that the files can exist on different file systems.

#8: Applications run recovery logic that accesses incorrect data in the page cache. Applications that depend on the page cache for faster recovery are susceptible to FalseFailures. As seen in LevelDB, SQLite, and PostgreSQL, when the *wal* incurs an `fsync` failure, the applications fail the operation and notify the user; In these cases, while the on-disk state may be corrupt, the entry in the page cache is valid; thus, an application that recovers state from the *wal* might read partially valid entries from the page cache and incorrectly update on-disk state. *Applications should read the on-disk content of files when performing recovery.*

#9: Application recovery logic is not tested with low level block faults. Applications test recovery logic and possibilities of data loss by either mocking system call return codes or emulating crash-restart scenarios, limiting interaction with the underlying file system. As a result, failure handling logic by the file system is not exercised. *Applications should test recovery logic using low-level block injectors that force underlying file-system error handling.* Alternatively, they could use a fault injector like CuttleFS that mimics different file-system error-handling characteristics.

4
Intentions in the Wild

In the previous chapter (§3), we studied the effects of `fsync()` failures on real-world applications. In doing so, we observed that the `fsync()` system call is necessary but not sufficient for what the application intended to do. Applications use `fsync()` as a way to persist modifications made previously through other system calls. These applications devise update protocols with the strategic use of `fsync()` both as a persistence mechanism and as a barrier before future writes to guarantee—albeit not completely effectively—durability. Additionally, data is read from persisted files in a specific way to achieve similar goals.

The reason for specific protocols with careful ordering and barriers is due to the lack of specific file-system interfaces to accomplish the same task. Due to the fine-grained nature of system calls, applications realize complex operations through a dialogue with the underlying system. While observed in §3.2 for five applications, prior work has documented similar behavior in 7 additional systems [82]. In this chapter, we ask the question: How do applications commonly interact with the underlying file system? Specifically, what other *dialogues* do applications have with the file system, requiring multiple system calls.

We begin with a study of a variety of systems, involving the examination of source code and documents. We describe Ikhnaie, a tool to aid in tracing and visualizing dialogues in applications. We then summarize our findings after categorizing commonly observed dialogues which we term

as intentions; operations an application *intends* to perform but is spread out over multiple system calls.

4.1 Hunting for Application Dialogues

We choose applications from a diverse set of domains: key-value stores, relational databases, embedded and client-server architectures, version control systems, build systems, and classic command-line utilities. In order to discover domain-specific patterns, we study mode than one application per domain. We prioritize candidate applications that have been studied or evaluated against in the past by the systems research community.

As some applications are too large to manually inspect all their source code, we build a tracing and visualization tool—Ikhnaie—to help narrow our focus. We first describe the motivation for ikhnaie, due to limitations with existing tools, and then describe its design and implementation details.

Ikhnaie does not automatically discover dialogues, but provides the tools that help us explore applications. As ikhnaie only helps to make our inspection of applications easier, we do not perform any evaluation of the tool. Instead, we explain how we use ikhnaie to find application dialogues (§4.2) and then generalize common dialogues into *intentions* (§4.3).

4.1.1 Why Ikhnaie?

To identify dialogues with the underlying file system, we absolutely require system call traces for every application we study. While it is possible to identify patterns solely through system call traces, what the application does in user-space can provide more helpful context. For example, a repeated `read()` may be due to a function that wishes to read data from multiple locations, or due to a caller making repeated calls to the callee that issues the system call. The extra context from user-space functions helps

set boundaries over system call sequences to help us gauge what might be part of a dialogue and what is the beginning of an entirely different dialogue.

We considered both `strace` and gcc's `-finstrument-functions` to get both system calls and user-space information; neither were sufficient. The system call tracing tool—`strace`—is capable of printing the user stack trace[1]; i.e., the user space functions that led to the system call. However, examining only stack traces makes it impossible to differentiate between multiple system calls from the same user function from multiple invocations of the user function. The GCC flag `-finstrument-functions` allows developers to provide a custom function definition which will be called on every other function entry and exit. While a function definition could be provided to log entry/exit events, profile functions, and dump the call stack, it cannot reliably identify the location of the call site (the location within the parent/caller function). Call-site information such as locus, basic-block, and scope can provide additional details to differentiate between two identical function call stacks. We build Ikhnaie, to handle the capturing of system calls and sufficient user-space function trace events to distinguish between multiple identical call stacks.

4.1.2 Ikhnaie: Design & Implementation

We now describe `Ikhnaie`[2] - a collection of tools built to acquire rich trace information from applications. The main goal of Ikhnaie is to make it easier to study the file-system interactions of complex applications with large code bases. Specifically, we built and designed Ikhnaie with the following goals:

[1]The command `strace -k` prints the execution stack trace of the traced process after every system call.

[2]Ichnaea (Ikhnaie) the goddess of tracing and tracking. Her name was derived from the Greek verb ichneuô meaning "to trace" or "to track".

Correctness. As a tracing tool, Ikhnaie must only provide visibility and not alter the behavior of the application under trace. Applications that are traced with Ikhnaie must also pass any test suites coupled with the application. Additionally, the tracer must avoid emitting records generated due to tracer logic. For example, a tracer using the `write()` system call to persist trace records must ensure that the `write` is not part of the final records to be analyzed.

Acceptable performance. Although Ikhnaie is a tracing tool and need not have a low overhead as long as application behavior is not altered, we do require an acceptable level of performance; not for evaluation but to complete our study on real workloads at a reasonable pace. A previous implementation of Ikhnaie automated gdb catchpoints and breakpoints. Apart from the loss of call-site information, there was a performance overhead due to `ptrace` involvement. A YCSB-A workload on LMDB for 1 million records suffered a throughput reduction from 60kops/sec to 150ops/sec. The absolute time taken to acquire traces was 1.8 hours while the actual benchmark took 16 seconds. Such problems cannot be alleviated by scaling down the workload as certain paths are executed only after certain data limits are reached. For example, applications like LevelDB and SQLite trigger compaction and write-ahead-log checkpointing respectively at certain thresholds[3].

Ikhnaie consists mainly of four components: ikgimple.so during compile time, libiklog.so and iksys.bpf during run time, and ikanalyze.py for post processing and analysis. We describe each of them in the remainder of this section.

[3]Such thresholds can be changed and workloads can be scaled down on a case-by-case basis but limit our ability to study system call behavior at different levels of scale and void our claim of real-world representative workloads.

4.1.3 Compile-time changes with ikgimple.so

`ikgimple.so` is a GCC compiler plugin [43] that injects event-emitting logic when the application is compiled. Specifically, `ikgimple.so` registers with GCC as a compiler pass right after the control-flow graph is created, operates on each function, and iterates over the basic blocks in GCC's internal representation—GIMPLE. Unlike `-finstrument-functions`, `ikgimple.so` injects instructions (explained below) at both the call site and function entry/exit site. For every instruction injected, `ikgimple.so` creates a unique identifier (an integer) and associates locus information (file, line number, column number, etc), basic-block membership, and scope membership within the function. It stores this information in a database that can be looked up during post processing by ikanalyze.py.

The injected function calls are of four types: `FunctionEnter`, `FunctionExit`, `CallsiteEnter`, and `CallsiteExit`, which are defined elsewhere in a shared library—`libiklog.so`. All four function definitions must accept a single integer parameter—the unique identifier mentioned earlier.

While built for GCC, `ikgimple.so` can be ported to LLVM [67]. Additionally, applications that use languages that have frontends for the above two compilers (such as C, C++, Objective C, Go) [42] should be traceable. However, we currently restrict scope to applications that use C and C++.

The application's build configuration is modified to include ikgimple.so as a plugin. Once compiled with gcc, the output binary will contain code that invokes functions defined in libiklog.so.

4.1.4 Run-time tracing with libiklog.so and iksys.bpf

As described above, `ikgimple.so` injects function calls at function and call-site boundaries which are invoked by the application during runtime. The definitions for these functions reside in `libiklog.so`.

The `libiklog.so` library is responsible for efficient event storage. A

call to one of the injected functions results in an event that needs to be stored. To avoid the overhead of a system call per event, `libiklog.so` memory-maps a newly created tracefile and writes events to the buffer. On possible buffer overflow, the file is unmapped, extended, and remapped. Using thread-local storage(TLS [44, 120]), `libiklog.so` maintains a file and buffer for every thread. While system call usage can also be minimized using buffered IO with `fopen,fwrite`, such an approach leads to extra occasional `write()` calls internally that are harder to ignore from trace results.

As `ikgimple.so` can only inject code in files that are compiled, it misses internal library calls; e.g., the occasional `write()` from `fwrite()` when the buffer is full. However, unlike `-finstrumentfunctions`, we emit events at the call site indicating that `fwrite()` is called. Rather than the exact callstack within a library, we focus on the system calls that emanate from the library function. To do so, we also run `iksys.bpf`—an eBPF application that emits system call events.

Both `libiklog.so` and `iksys.bpf` emit timestamps with events, ensuring correct reassembly from the two sources. Specifically, we require a monotonically increasing counter shared by both user and kernel space. On x86 architectures, the `RDTSC` instruction is an ideal candidate. As accurate time measurements are unnecessary, the generally associated instructions to avoid reordering and pipeline flushing such as `CPUID`, fences, or `RDTSCP` are unnecessary as well. Unfortunately, the current Linux kernel's BPF virtual machine is its own architecture separate from x86. We rely on the BPF helper function `bpf_ktime_get_ns()` to get nanosecond timestamps in the kernel and `clock_gettime` with `CLOCK_MONOTONIC` in user space; both appear to use the same internal clock source and work as expected, albeit slower than `RDTSC`.

The minimal system calls generated by `libiklog.so` on file creation, mapping, and extension are preceded by instructions to `iksys.bpf` to ig-

nore them. Rather than complicating in-kernel BPF logic, `iksys.bpf` emits "ignore markers" which is understood by the post-processing library—ikanalyze.py.

The recording of events does add overhead in both additional function calls and periodic writes of the buffer to disk. The same YCSB-A LMDB workload took 30 seconds to trace, nearly double of the original time. However, waiting 30 seconds was acceptable for us, and far better than the wait with gdb.

4.1.5 Post-processing and analysis with ikanalyze.py

After an application finishes running a workload, all the trace files and compiler generated output need to be processed. Specifically, during compilation ikgimple.so generates a database of locus information, i.e., a mapping from function call site id to the function name, file name, and position within the file. And during run time, libiklog.so and iksys.bpf generate the call-site events and system call trace files for each thread. ikanalyze.py examines the files containing function and call-site events, system call events, and markers indicating events to ignore. It enriches these traces with the locus information generated during compilation by `ikgimple.so`.

Additionally, ikanalyze.py prunes unnecessary events generated by `libiklog.so` that never lead to system calls. Pruning is necessary to reduce the data we analyze. We could not perform this step at compile time as certain application designs utilize callback functions in libraries; code execution through callback functions or function pointers cannot be easily detected during compilation. As performance overhead is minimal and storage is plenty, `ikgimple.so` injects function and call-site boundary events on every function available to it, leaving pruning for later.

We analyze the pruned traces using scripts and functions that are part of ikanalyze.py. Written in Python, with Numpy and Pandas, ikanalyze.py

contains functions to gather statistics, search for patterns we input, and generate visualizations that can help narrow our examination of source code.

4.2 Finding Dialogues with Ikhnaie

As described previously, we choose applications from a diverse set of domains (§4.1). In this chapter, we explain our process of finding dialogues in those applications and how Ikhnaie fits in.

Choosing workloads for a given application.
While simple command-line utilities have a specific task (`cp` to copy a file), larger complex applications tend to behave differently depending on the given workload. The different behavior can result in a different set or number of system calls, affecting the survey. We choose standard workloads that are frequently used by the research community. For example, the TPC-H and TPC-DS benchmarks for relational databases, and YCSB for key-value stores. For applications that do not have standard benchmarks, we study behavior on a range of workloads representative of common use cases. For example, when copying a file with `cp`, using arguments where the destination file exists or where both files are on the same or different file system.

Using Ikhnaie.
We first modify the application build process to include ikgimple.so during compilation. Then, we run the application with a workload, which loads libiklog.so at runtime. Simultaneously, we start iksys.bpf to capture system call traces.

Once the workload completes, we are left with the user-space function traces and system call traces. We use ikanalyze.py to prune the user level traces. Now, we launch a jupyter notebook environment to programmati-

cally (but not automatically) explore both traces using helper functions in ikanalyze.py.

First, we focus on the system call traces. We run the workload at different scales and compare the system call traces for each. Varying the scale allows us to identify system calls that are constant, proportional, or occasional. It is important to note that some system calls may exhibit characteristics belonging to more than one of these categories. However, this initial categorization provides a general overview of system behavior, which serves as a foundation for more in-depth analysis.

Having identified some system calls to analyze further, we use ikanalyze.py to find all functions that lead to all or a subset of those system calls. While `main()` is one of those functions, we sort functions by their depth in the call stack to find the nearest common functions that contain those subset of system calls; we also visualize the same as flamegraphs.

We then take a look at each of the above common functions found, one at a time. Sometimes, the function name makes it easy to identify what it does and we look at the source code directly to confirm its behavior. For others, we dig deeper by using ikanalyze.py to generate a graphviz visualization of function and system calls stemming from a specific call site invoking one of those common functions.

The graphviz visualization provides us with a tree whose leaves are system calls. The intermediate nodes are function call sites. The edges are annotated with a range of how many times a particular invocation happens which could indicate a loop. Each node also embeds the location in source code to make it easier for us to navigate and read the related source code. The following are two examples that walk through our workflow.

Example #1: LMDB.

Figure 4.1 is a graphviz rendering of a callsite within the LMDB application with YCSB workloads. We had noticed that `pwrite64()`, `lseek()`, and `writev()` all scale with changing the scale of the workload. We

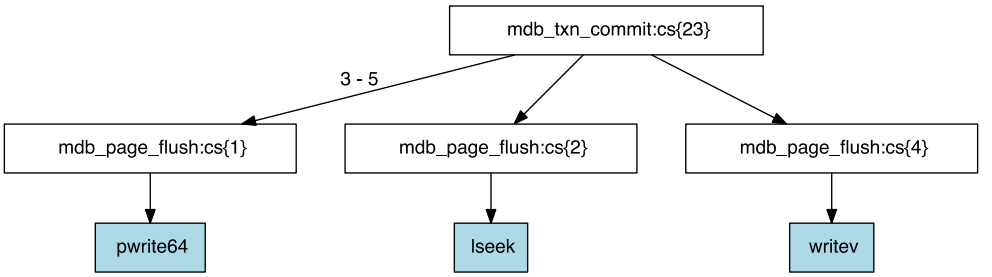

Figure 4.1: **Visualizing portions of LMDB with Ikhnaie:** We start with an observation of repeated `pwrite64` syscalls and ask Ikhnaie to give us a visualization of that region. Ikhnaie first focuses on that region to identify the system calls and functions, and then scans the entire log to provide information on that region for the entire workload. In the above call-graph visualization, the leaves (colored) correspond to system calls. Every other node is a call site, i.e., a function name and a position within the function. In this figure, all three system calls are invoked by the same function (`mdb_page_flush`) at different call sites: cs{1}, {2}, and {4}. The figure also shows that cs{1} is invoked multiple times (3–5 times), followed by one `lseek` and one `writev`. All three call sites are invoked as part of a single execution of the `mdb_txn_commit` function. The image gives us context to decide whether we need to dig further. While not depicted, each of the nodes in the call graph are hyperlinked to the source location, allowing us to immediately take a look at the source code. In this specific case, LMDB maintains a list of dirty pages (pages that are modified during a transaction) that need to be flushed to disk. As LMDB keeps track of dirty pages in sorted order, sometimes the dirty pages are contiguous, leading to an `lseek()` and a single `writev()`. In other cases, each non-contiguous page can be written out with a `pwrite()`. Upon source-code inspection, we also notice pre-processor macros that depend on compile time flags. For example, building with `MDB_USE_PWRITEV` changes the pattern to always have one leaf `pwritev()` repeated multiple times.

dug deeper using ikanalyze.py and identified a single call site within the `mdb_txn_commit` function—the function used to commit a transaction. The visualization helped us confirm that this is a pattern that scales with the number of transaction commits and is worth inspecting further.

Looking at the source code, we found compile-time preprocessor directives that could be used to change the system call LMDB uses. The `lseek()` and `writev()` could be changed to a `pwritev()` if `MDB_USE_PWRITEV` is provided as a compile-time definition. More importantly, we were able to identify that `mdb_page_flush` writes one or more dirty pages to disk inside a loop. Each page is located at different memory locations and all pages to be written need not be contiguous. We term this operation a *scattered write* (described later), and look for the same in other applications.

Example #2: SQLite.
Figure 4.2 is the graphviz rendering of a callsite in SQLite. We followed a process similar to LMDB but running a TPC-H workload. As seen in the figure, SQLite has many more abstraction layers than LMDB: seekAndRead, unixRead, sqlite3OsRead, readDbPage, and getPageNormal. Like LMDB, a `USE_PREAD` definition changes seekAndRead to use `pread()` instead. Internally, when SQLite needs to read a specific page in the database (when navigating its B-Tree nodes), it calls the getPageNormal function. The left sub-tree in the figure was not originally visible. We modified ikanalyze.py to also use the unpruned data to show us more context—SQLite first checks if the page is in its own application cache.

The visualization was helpful in identifying key functions to study further. With documentation and source code, we concluded that SQLite operates on a row-by-row bases. The sqlite3BtreeNext function moves to the next row, which sometimes requires moving to another leaf node in the B-Tree at which point a page needs to be read. SQLite uses its pager to read the page (sqlite3PagerGet), which internally checks its cache or issues a system call to fetch the data.

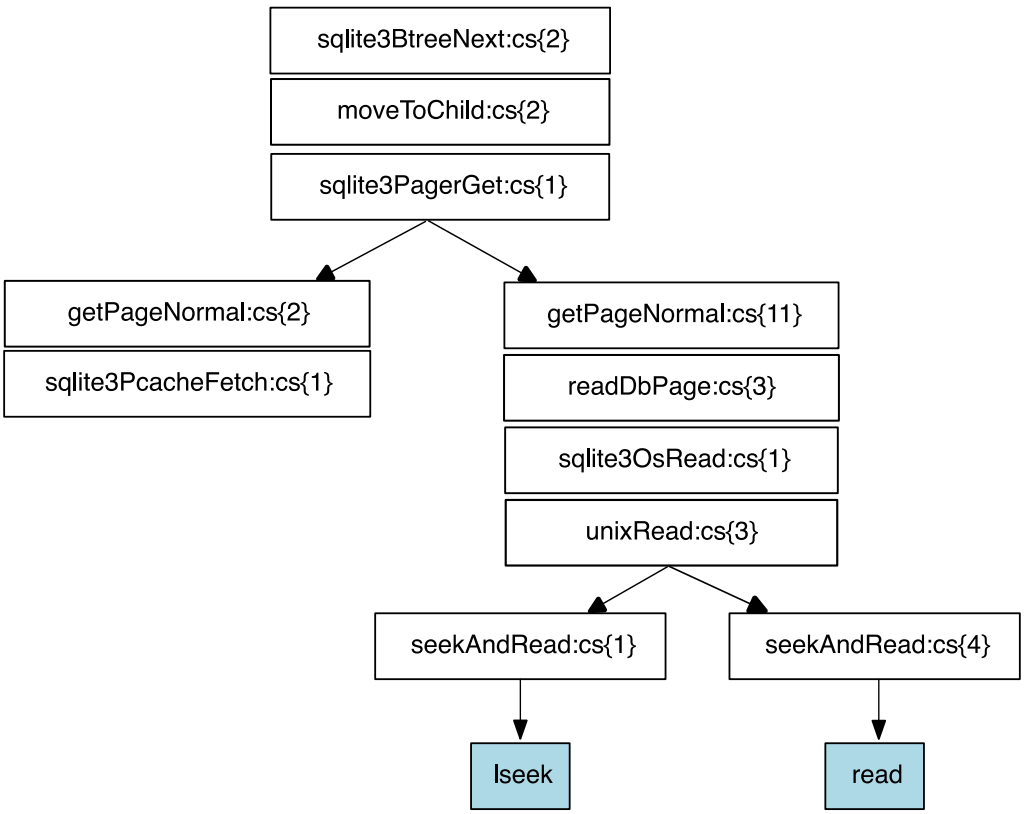

Figure 4.2: **Visualizing portions of SQLite with Ikhnaie:** We observed repeated patterns of `lseek()` followed by `read()` when running SQLite. At first, we were unsure whether multiple lseek-read pairs appear together or one at a time. With Ikhnaie, the visualization tells us that every pair appears one at a time and is the result of a `sqlite3BtreeNext` function call. As observed in the figure, the system calls are hidden under a layer of one-to-one portability abstractions: `sqlite3OsRead`, `unixRead`, followed by `seekAndRead`. With the help of Ikhnaie, SQLite design documents, and source code, we learn the following: SQLite uses B-Tree data structures whose nodes are called "pages", and rows in a database table reside in B-Tree leaf pages. SQLite uses `sqlite3BtreeNext` to move to the next row, which may be on the same page or on a different page. If the new page is in the page cache (left subtree in the figure), no file-system interaction is necessary. If the page has to be read from disk, the code takes the right subtree path resulting in a single `lseek()` and `read()` to the target page. As with LMDB in Figure 4.1, a compile-time flag (`USE_PREAD`) changes this behavior to use a single `pread()` instead.

If we were to only look at the `read()` system calls, we would conclude they were *scattered reads* (similar to LMDB scattered writes). However, further inspection revealed that it operates one-by-one and those pages cannot be read in bulk. An intermediate page must be read to determine the offset of the leaf page—a pointer chasing workload.

4.3 Findings

We repeat the process described previously (§4.2) on multiple applications. We referred to source code and documentation for all of them; but not all required Ikhnaie. After using Ikhnaie for a few, we knew what to look for in other applications, and searched for their existence. We created categories for our observations and now present them here.

We selected applications from multiple domains: key-value stores (Redis, LMDB, LevelDB, RocksDB), relational databases (SQLite, PostgreSQL, MySQL), build systems (make, ninja), version control systems (git), and common command-line applications (cp, cat, ls, tac, tar, vim). While we also observed metadata interactions (`readdir()`, `stat()`) for some applications (ls, make, ninja, git), we focus on and categorize interactions related to retrieving or storing data.

The data being retrieved or stored may be on a single file or spread across multiple files. When storing, applications may overwrite (update) existing data, or write new data. These applications may require the stored data to only be made visible as a whole (no partial reads), and ensure it survives power failures. We describe these categories in detail below, providing examples for each. Table 4.1 and 4.2 are a summary of single and multi-file intentions.

Intention				Applications
Single File Reads	Single Contiguous			ninja, git
	Multi-Chunk	Content Independent	Forward Sequential	grep, cat, log recovery
			Reverse Sequential	tac, sql ORDER BY DESC
			Strided	fixed-length record file processing
			Scattered	SQLite
		Content Dependent		SQLite, PostgreSQL
	Filtered			grep, csplit, logstash, sql SELECT queries in SQLite, PostgreSQL
Single File Writes	Add	Prepends, Inserts		-
		Appends	Sequential	Redis, SQLite, LevelDB
	Overwrite	Content Independent	Scattered	SQLite, LMDB
	Atomicity and Durability	Application Specific		LMDB, Redis
		Generic	Copy-modify-replace	vim, LevelDB
			Physical Redo Logging	SQLite

Table 4.1: **Summary of Single-File Intentions:** Their categories and occurrence in applications.

Intention				Applications
Multiple File Operations	Homogenous	Reads		grep, LevelDB, PostgreSQL
		Writes		PostgreSQL, exporting data-sets in Python
	Heterogenous	Moves		SQLite, PostgreSQL checkpointing (always implemented as copy-and-remove)
		Copies	Few large contiguous	cp, tar, ar, audio trimming
			Many small scattered	SQLite checkpointing in physical redo logs
	Atomicity and Durability	Physical Redo Logging		PostgreSQL
		Manifest Files		LSM key-value stores (LevelDB, RocksDB)

Table 4.2: **Summary of Multi-File Intentions:** Their categories and occurrence in applications.

```
// Implementation 1
    lseek(fd,offset,SEEK_SET);
    read(fd,buffer,size);
// Implementation 2
    pread(fd,buffer,size,offset);
```

Listing 4.1: Two implementations of a single contiguous read into a single buffer

```
struct iovec iov[2] = {
    {.iov_base=buffer1,.iov_len=size1},
    {.iov_base=buffer2,.iov_len=size2},
}
// Implementation 1
    lseek(fd,offset,SEEK_SET);
    readv(fd,iov,2);
// Implementation 2
    preadv(fd,iov,2,offset);
```

Listing 4.2: Two implementations of a single contiguous read into multiple buffers

4.3.1 Single File Reads

4.3.1.1 A Single Contiguous Read

When reading data from a single file, some applications only require a single contiguous portion or the entire file. For example, the ninja build tool reads the entire dependency file into a single string, and git reads the entire HEAD file whose contents refer to the name of the currently active branch (or commit).

Listing 4.1 demonstrates a single contiguous read into a single buffer (e.g., a string in ninja). It contains two implementations of the same intention; they differ due to interface availability. When `pread()` is available, the second implementation is used.

Listing 4.2 also demonstrates a single contiguous read. However, unlike Listing 4.1, while the bytes read from the file is contiguous, the memory

buffers are not. For example, an image processing application that processes different strips of an image in parallel may want them read into different buffers. The intention of what is being read from the file remains the same, but there is an application design difference in where it should be stored. Listing 4.2 also contains two implementations depending on the existence of `preadv()`.

Throughout this section, we will see intentions that are realized using different interfaces. Within each of the previous listings, the differences are *implementation differences*. However, while both listings perform the single contiguous read of a file, they have *application design differences*.

4.3.1.2 Multi-Chunk Reads

Applications that read data from multiple different locations in a file typically read more than one byte from each location; they read a contiguous portion from each location—a chunk. Applications either know the exact locations of each chunk beforehand (content-independent traversals), or must read one or more chunks to identify subsequent ones (content-dependent traversals)—application design differences.

Content-independent traversals have more application design differences based on the *access pattern* of the traversal. The most common forms are Forward Sequential, Reverse Sequential, Strided, and Scattered.

Forward Sequential. Listing 4.3 provides two implementations of Forward Sequential reads, where the future reads are always the next chunk sequentially. Common examples include `grep`, `cat`, and recovery logic (§4.3.3.1) that reads a log (Redis, LevelDB).

Reverse Sequential. Listing 4.4 provides one implementation of Reverse Sequential reads, where the future read is the previous chunk in the sequence. It can also be implemented with `lseek()` and `read()` instead of `pread()` but omitted from the listing. Common examples include `tac`,

```
// Implementation 1
    lseek(fd,0,SEEK_SET);
    do {
        ret = read(fd,buffer,bufsize);
        if (ret < 0) // error
        if (ret == 0) break; // EOF
        // process buffer
    } while (1);

// Implementation 2
    offset = 0;
    do {
        ret = pread(fd,buffer,bufsize,offset);
        ...
        // process buffer
        offset += ret;
    } while (1);
```

Listing 4.3: Two implementations of a Multi-Chunk Forward Sequential Read

```
    offset = lseek(fd,-1*bufsize,SEEK_END);
    // Or use stat() and set offset accordingly
    do {
        ret = pread(fd,buffer,bufsize,offset);
        ...
        // process buffer
        offset -= bufsize;
    } while (offset >= 0);
```

Listing 4.4: Multi-Chunk Reverse Sequential Reads

and some uses of `ORDER BY DESC` in relational databases (SQLite) where many B-Tree leaf pages happen to be contiguous.

While both the above sequential accesses can be expressed as a single contiguous read, there are application design differences; the application wants to minimize memory usage and operate in chunks.

Strided. Listing 4.5 provides one implementation of Strided reads, where the future read is some fixed distance away from the current chunk.

```
1      do {
2          ret = pread(fd,buffer,bufsize,offset);
3          ...
4          // process buffer
5          offset += stridelength;
6      } while (ret > 0);
```

Listing 4.5: Multi-Chunk Strided Reads

```
1 for (ptr=page_list_head; ptr!=NULL; ptr=ptr->next) {
2   offset = ptr->page_no * page_size;
3   pread(fd, ptr->buffer, page_size, offset);
4 }
```

Listing 4.6: Multi-Chunk Scattered Reads

While not observed in the applications listed above, it is a category of traversal that is used when working with multi-dimensional on-disk arrays. The atmospheric and oceanographic communities use workloads that access only particular fields in netCDF or HDF5 formats which requires skipping over other fixed-length fields; NASDAQ also exports fixed-width text files of stock exchange data.

While sequential accesses can fall into single contiguous or multi-chunk reads depending on application design, strided accesses (even though sequential with gaps) are always multi-chunk reads requiring multiple system calls. This is due to a limitation in current interfaces. The scatter-gather interfaces (`readv()`, `preadv()`, `preadv2()`) require that all IOV arrays be filled; the skipped-over regions between strides must be read into some memory region. Applications avoid reading between strides through multiple system calls.

Scattered. Listing 4.6 provides one implementation of Scattered reads, where chunks are scattered across the file with no observable pattern. In the listing, an application treats the file as an on-disk list of "pages" or "chunks". It constructs a linked list of the pages it wishes to read and then

```
node = get_root_node(btree);
do {
    pread(btree->on_disk_fd,node->buf,
        /*buffer size*/ btree->nodesize,
        /*offset*/ node->pgno * btree->nodesize);
    if (is_leaf(node)) break;
    // intermediate node, find next level node
    node = get_child(node, key);
} while (node != NULL);
```

Listing 4.7: Content-Dependent Reads: B-Tree Traversal

loops over the list issuing a read for each page.

SQLite and LMDB maintain B-Tree data structures where each node is a page or a chunk in the file. As LMDB uses `mmap()` for reads, we focus on SQLite as an example, where tables and indices are represented as B-Trees. Any workload that performs a breadth first traversal over a tree (e.g., gathering statistics about data distribution in an index) examines nodes level by level. Since intermediate nodes contain pointers to all their immediate child nodes, reading nodes of the next level are a content-independent scattered read.

Content-dependent traversals require inspecting the content of the previous chunk to identify the next chunk to be read. Listing 4.7 provides one implementation of content-dependent traversals—a depth-first search to identify a key in a leaf of an on-disk B-Tree. Applications like SQLite store tables as B-Trees and accessing a row involves traversing the intermediate nodes from the root of the tree to the leaf node containing the row. However, the exact path to each node can only be determined by inspecting the contents of the previous node—a pointer chasing workload.

Applications have come up with techniques to improve overall performance in some multi-chunk reads. For content-independent traversals, applications ask the file system to prefetch the chunks using `fadvise()` or `readahead()` system calls.

```
while !EOF
    read(fd, chunk, chunksize);
    for line in get_lines(chunk, chunksize):
        if keyword not in line:
            continue
        // process line
```

Listing 4.8: Filtering reads with a specific keyword

For content-dependent traversals, an application must change its design to perform some content-independent traversals. For example, applications designed for fast data retrieval often construct an index over their data. Therefore, they can first iterate over the index in a content-dependent manner, identifying all the locations of the chunks they wish to retrieve. Then, they can perform a scattered read over the locations identified. Concretely, SQLite can perform a content-dependent traversal to identify specific leaf nodes and later use a content-independent scattered read for those leaf nodes. Unfortunately, there is currently no benefit in doing so because scattered reads are not a primitive that applications can use; they stick to full content-dependent traversals.

4.3.1.3 Filtered Reads

Applications do not always require all the data they read. Some utility applications (grep, csplit) scan through data sequentially to find records matching certain criteria. Another class of applications that perform filtering are log file analysis tools (e.g., logstash) that search for specific keywords in logs for further analysis. However, as the file system is unaware of this criteria (e.g., a keyword, or the record delimiter format), all the data must be transferred to the user-space application which then discards bytes that don't fit the criteria. Reading unfiltered data can be costlier, especially if the underlying file system is not local.

Consider the pseudocode example in Listing 4.8 that behaves similar

```
1  int fd = open("/path/to/file", O_WRONLY|O_APPEND);
2  // Implementation 1 (a single contiguous buffer)
3  write(fd, buffer, bufsize);
4
5  // Implementation 2 (multiple scattered buffers)
6  struct iovec iov[2] = {
7      {.iov_base=buffer1,.iov_len=size1},
8      {.iov_base=buffer2,.iov_len=size2},
9  }
10 writev(fd, iov, 2)
```

Listing 4.9: Two implementations of appending data to a file

to grep. An entire chunk is first copied into a userspace buffer and lines that do not have a keyword are discarded. While inexpensive on a local file system, the application can be run on FUSE file systems backed by remote storage. Cloud storage providers like AWS and Google provide FUSE endpoints but also charge for network egress costs. While users may not notice runtime differences (modern network speeds), the discarded lines increase costs. When the underlying system has compute capabilities (AWS Lambda, Google Cloud Functions), the application can perform the filtering closer to storage and only transfer relevant lines[4].

Row-order databases like SQLite and PostgreSQL place all columns of a row together; SQL SELECT queries over a few columns result in reading all columns. However, as these applications maintain a user-space cache, future queries over the same rows are served by the cache and therefore not an immediate problem.

4.3.2 Single File Writes

4.3.2.1 Adding New Data

While new data can be added either to the beginning (prepends), the mid-

[4]The cost to executing lambdas may be far lesser than network costs if the keyword occurrences are rare, e.g. searching for errors in logs.

dle (inserts), or end of the file (appends)[5], applications most commonly append data as it is less restrictive (Listing 4.9).

Prepending and inserting data is costly as existing data must be pushed downwards to make space for the incoming data. Popular file systems like ext4 and XFS do offer mechanisms to do so "punching a hole" in the file; but such techniques only work if the incoming data occupies a full block (commonly 4KB) and is being inserted at a block boundary. We did not see evidence of their usage but mention them for completeness (more details in §4.4.2).

Listing 4.9 provides two common forms of appending to files differing in application design. The first uses a single contiguous buffer in memory while the second uses multiple different buffers. Appends are commonly seen in applications that write to a log either for auditing or recovery purposes (e.g., Redis, SQLite, LevelDB).

4.3.3 Overwriting Existing Data

Regardless of whether chunks are content-dependent or not when read, the application performing writes has access to the content and file offsets before issuing them, i.e., writes are content-independent. Single chunk updates or multi-chunk updates following sequential patterns can be achieved with a single system call: `write()`, `writev()`, or `pwritev()`. Multi-chunk strided and scattered updates require multiple system calls.

While strided updates are not as common as their read counterparts, scattered updates are frequently observed. Applications like LMDB using a copy-on-write B-Tree structure must update all nodes in the tree all the way to the root for a single leaf node update. These nodes (especially the intermediate nodes) are scattered across the file, requiring multiple system calls to complete the update.

[5]For simplicity, we do not cover sparse files. However, some applications do make use of sparse files, adding data to existing gaps (holes) within a file.

```
1  // sorted order to reduce write syscalls
2  Page page = sorted_dirty_pages->head;
3
4  int iovcnt = 0;
5  struct iovec iov[MAX_IOV];
6  iov[iovcnt].iov_base = page->buffer;
7  iov[iovcnt].iov_len = page_size;
8  ++iovcnt;
9
10 Page startpage = previous_page = page;
11 page = page->next;
12 for(; page!=NULL; page=page->next) {
13     bool not_contiguous =
14         (page->pgno != previous_page->pgno + 1);
15     bool limit_reached = (iovcnt == MAX_IOV);
16     bool do_write = not_contiguous || limit_reached;
17
18     if (do_write) {
19         offset = startpage->pgno * page_size;
20         pwritev(fd, iov, iovcnt, offset);
21         iovcnt = 0;
22         startpage = page;
23     }
24
25     iov[iovcnt].iov_base = page->buffer;
26     iov[iovcnt].iov_len = page_size;
27     ++iovcnt;
28     previous_page = page;
29 }
30 // ... write remaining iov
```

Listing 4.10: Overwriting Existing Data: Scattered Writes of LMDB dirty pages with an optimization for contiguous pages

Listing 4.10 describes the scattered write performed by LMDB. LMDB keeps track of the pages it needs to write in a linked list sorted by page number. The sorted order allows LMDB to perform an optimization: when a contiguous sequence of pages is found, a single `pwritev()` system call can be used, reducing the total system calls used to write all pages.

4.3.3.1 Atomicity and Durability

Applications face two additional challenges when writing to files:

1. Ensuring new and existing readers see either the old or new data during normal operation (atomicity)
2. Ensuring new readers see the last successfully committed data after recovering from a crash (durability)

Atomicity. While contiguous writes using a single `writev()` are atomic, the underlying file (or operating) system imposes limits on the maximum number of scattered memory buffers (`IOV_MAX`, often 1024). Larger writes require multiple system calls which are not guaranteed to be atomic.

Although earlier versions of Linux (pre 5.14) supported mandatory locks, they have since been removed. All file locks are now advisory and cannot force atomic views across write system calls. In the absence of underlying OS support, applications that wish to provide atomicity "globally" do so by updating a copy of the file and renaming it (e.g., vim and LevelDB). Existing readers (accessing via open file descriptors) would read an old consistent copy while new readers resolve the path to the newly modified file.

Alternatively, developers may choose to follow a specific access protocol under the assumption that only multiple instances of their application may be used—no guarantees are made with respect to accesses by other applications. We term such expectations as "app-local write atomicity".

App-local write atomicity can be achieved in multiple ways. Some are heavily integrated into the application's design (like LMDB) while others are generic (SQLite, vim, gedit). Additionally, when limited to a single process, user-space reader/writer locks are sufficient.

Durability. Providing atomicity guarantees when data must be written to disk is much more difficult. Devices can fail at any moment either entirely or partially, and transient errors disappear after a while.

In some cases, the failure causes an entire system crash. In cases where it does not, applications running on the system may or may not be notified of write failures. In all such scenarios, applications that intend to provide durability guarantees must have mechanisms to ensure proper recovery from states that break previously-mentioned atomic states. To do so, applications build on top of the `fsync()` system call, writing data in a particular order, injecting `fsync()` strategically so that recovery to valid states is possible.

To satisfy both atomicity and durability requirements, some applications (e.g., LMDB, Redis, LevelDB) use techniques that are integrated with their internal data structures, unique to their design. Others (e.g., SQLite, PostgreSQL) use a more generic design (called physical redo logging) but implement them independently. In both cases however, `fsync()` is a key component and misunderstanding post-failure behavior has led to data loss (§3). We first describe application specific approaches followed by the generic intention.

Application-specific approaches. LMDB achieves atomicity by maintaining two B-Tree root pointers, one for all readers and one for a single writer. The readers always use the last modified root. During a transaction, the writer—using a copy-on-write approach—constructs a new tree consisting of new nodes for the modified nodes, and links to unmodified nodes in the readers' tree. The transaction is committed when the writer's root pointer is updated to point to the new tree. Therefore, existing readers can continue reading from the old reader root while new ones use the new root. This approach also simplifies durability—all modified nodes are first written and then synced to disk with `fsync()` before the root pointer is written and synced.

Unlike LMDB which uses a single file, Redis and LevelDB record all operations that modify state to a separate file—a log. An `fsync()` is called on the log to ensure the operations are persisted. When the application restarts, the log is examined to recreate state before the crash. As the entries record logical operations, applications replay each entry in the same order they were written to the log. In doing so, they modify their internal data structures to re-attain their state before the crash. This approach—termed logical redo logging—is specific to each application as only the application knows how to interpret each entry in the log and modify its data structures appropriately.

4.3.3.2 Physical Redo Logging

We now describe a more generic approach to app-local atomicity and durability—the physical redo log intention used by SQLite and PostgreSQL. Similar to logical redo logging, changes are written to the log and `fsync()` is called. However, instead of logical changes, applications record the actual bytes that would go into the original file and the location of where they should go—physical changes.

The original file—termed a backing file—is divided into fixed-size chunks[6]. An operation that results in the application modifying chunks leads to those chunks being written to the log; the backing file is left untouched. Over time, as some chunks reside in the log, the application maintains a data structure to determine where to read the latest persisted chunk from. During recovery, the application examines the log to identify which chunks are present so that they are used for reads instead of the ones in the backing file.

To restrict a log from growing too large, the application periodically updates the backing file with all the changes from the log—conventionally

[6]Applications like SQLite and PostgreSQL use the term "pages" instead of chunks. These pages are often configured to have the same size as each page in the page cache.

called checkpointing. To prevent invalid non-atomic states for readers while checkpointing takes place, applications use locks in user space, on entire files, or on regions of files. For example, SQLite acquires an exclusive file lock (either through `flock()`,`fcntl()`, or `lockf()`) during checkpointing while readers acquire a shared lock.

Like previously described intentions, every application re-implements its own version. Some differ in the layout of chunks and their location in the log, and what bookkeeping data structures are used. However, the intentions are the same—chunks are written and synced to the log, reads are served either from the log or backing file, and a backing file is updated with content from the log. More importantly, they each fail to implement it correctly, leading to data loss (§3.2.3).

4.3.4 Multiple Files

Operations on multiple files can be categorized as homogenous or heterogenous, depending on whether they perform the same operation on all files or not.

4.3.4.1 Homogenous Operations

In addition to the operations on a single file, some applications read data across multiple files. Many applications know the files they wish to read from in advance, e.g., grep using the command-line parameters; others are content-dependent. For example, LevelDB maintains a manifest file containing the paths of all other files it reads, and PostgreSQL stores each table and index in separate files. Either way, an application that wishes to read data from multiple files is currently forced to make multiple system calls with a file-descriptor for each file separately.

The same limitation holds true when writing data across files, often when generating new data according to a specific format. For example,

exporting multiple dataframes in python to their own csv files. Additionally, the atomicity and durability challenges with single files also apply to multiple files.

4.3.4.2 Heterogenous Operations

Applications that perform both reads and writes on multiple files commonly do so to copy data. While data that is read can be processed (e.g., encryption, compression) before writing, there are many that only require copying. Sometimes, the file that data is being copied from is deleted (`unlink()`) after the copy, indicating a move. However, move operations are always implemented as a copy-and-delete. We expand on them later (§4.4.2).

We observe the copy operation in two distinct workloads: few large contiguous copies or many small scattered copies. Large contiguous copies are most commonly seen by copying files in their entirety (e.g., coreutils cp), or growing existing files (e.g., coreutils tar, binutils ar). Partial yet large and contiguous copies are also a frequent occurrence especially in the media post-processing industry, to trim or crop audio.

Small scattered copies are frequently seen in applications that use physical redo logging. To prevent the redo log from growing too large, the contents from the log must be written to the main file periodically. Conventionally called a checkpoint operation, applications read the redo log and write the physical chunks to their target locations in the main file, and then reset the redo log (a move).

4.3.4.3 Atomicity and Durability

The atomicity and durability challenges faced by applications writing to a single file also extends to multiple files. We identify two distinct workloads requiring atomicity and durability guarantees involving multiple files: updating multiple existing files, and creating a set of new files.

Relational database servers like PostgreSQL store each database table and index as a separate file. User queries that modify multiple tables as part of a transaction must atomically change rows in all the involved tables—atomically change multiple files. These applications implement a version of *physical redo logging* described earlier (§4.3.3.2). Instead of a single backing file, the log contains chunks for multiple files. Every chunk's associated location contains both the name of the backing file and its position within that file.

Unlike PostgreSQL, LSM key-value stores (LevelDB and RocksDB) maintain a set of immutable SST files for each level. Modifications to the store result in new SST files, and a compaction operation generates a new set of SST files. The individual SST files are never modified, but some may be reused as part of the newly compacted set. Existing readers must continue to have access to the old SST files, while new readers obtain the new set. Although they utilize app-specific logical redo logging for the updates themselves, the approach to changing the set of SST files after compaction is generic—applications maintain MANIFEST files that describe an entire set.

4.3.4.4 MANIFEST Operations

To the file system, a MANIFEST is a regular file whose contents do not hold special meaning. However, to the application, a manifest points to many existing files; a dependency not under the protection domain of crash-consistent file systems. Therefore, applications construct careful update protocols to change manifests.

After creating the new SSTs, LevelDB also creates a new manifest file to reflect the new set. To ensure that the entire set is updated atomically, LevelDB maintains a pointer (a file containing the path) to the latest manifest called CURRENT. LevelDB creates a new pointer (CURRENT.tmp) that points to the new manifest, and then renames it to CURRENT. The

atomic rename allows new readers to obtain the new manifest when reading CURRENT. After the rename, unused SST files and old manifests are unlinked to reclaim storage space.

Despite proper use of `fsync()` to ensure SSTs are persisted, as observed in Chapter 3, the directory entries themselves can disappear on ext4 resulting in data loss. We evaluate the effects of disappearing directory entries on LevelDB's manifest files in Chapter 5.

Like physical redo logging, every application implements its own version of manifests. As the files a manifest links to are application specific, each application has a custom format to encode the information associated with each file. However, a more generic manifest (designed and implemented in Chapter 5) can use a generic key-value format where the key is the application-specific information and the value is the file reference.

4.4 Challenges in Implementing Intentions

Based on the findings mentioned above, we now describe what a developer takes into account to ensure correctness or performance when implementing these intentions.

4.4.1 Correctness

Ensuring atomicity and durability when modifying one or more files requires understanding of the guarantees of the underlying system. As atomicity guarantees at the primitive level do not translate to a sequence of primitives (a single write vs multiple writes), developers use techniques like clone-modify-rename or force mutual exclusion through file locks to achieve the same.

Developers carefully insert `fsync()` operations both as persistent points and ordering barriers to ensure correct recovery on failure. However, proper `fsync()` usage requires understanding `fsync()` behavior on the

target file system. Currently, developers who are aware of certain `fsync()` issues (such as syncing the parent directory as well), take a lowest common denominator approach. Using the same example, an `fsync()` to the parent directory is issued even if the file system guaranteed directory entry persistence when the inode was `fsync()`ed.

Unfortunately, even having a mechanism to detect and code for the target file system does not eliminate the need to be aware of each file system's idiosyncrasies. As newer features are developed, existing file systems may acquire better (or worse) guarantees, or newer file systems may offer different ones as tradeoffs for performance; a cognitive burden for the developer.

While many applications share the same intention (using redo logs or manifests), they all implement the intention independently. As shown in Chapter 3, even modern widely-used applications written by experienced developers have data loss.

Developers of new applications currently do not have concrete reference implementations that implement these intentions correctly.

4.4.2 Performance: Designing Around Expensive Operations

As mentioned previously, certain intentions (prepends, inserts, and moves) are conspicuously absent. We do not observe them in applications we study and conclude that they are rarely, if ever, used directly. The underlying reason for this absence is due to well known facts about the costs associated with these operations.

Attempting to perform a prepend or insert would necessitate shifting all subsequent data within a file to accommodate new data. The hole-punching technique mentioned previously lacks widespread awareness and even then, has strict block alignment and size constraints.

Move operations are most commonly performed as a data copy followed by a delete—the most portable form of a move. When data moves across file systems or even across devices, a copy must be performed. As the files an application works with may not all reside on the same file system, the copy-and-delete technique offers a simple (and efficient enough) strategy.

If the move is indeed on the same file system, there should also be a faster copy on the same file system. For example, remote file systems (e.g., NFS) allow copying at the server without multiple round trips to the client. Therefore, costly data operations are minimized by the copy leaving just a metadata operation to remove the source file. While some file systems like ext4 have a method to swap extents (commonly used by defragmentation tools), they too have strict block alignment and size constraints.

These limitations are well understood by developers and system architects. As a result, application design inherently avoids such costly operations. This preemptive adaptation results in the absence of evidence for these operations in practice. The knowledge of their costliness is so ingrained that applications are designed to work around these limitations from the very beginning.

4.4.3 Performance: Implementing Intentions

Many of the intentions described previously have no unique implementation. Consequently, a developer is faced with multiple questions when choosing how to implement them, such as:

1. What primitives does the underlying system support?

   Different operating systems support different primitives, and file systems within an operating system may have special purpose primitives for the task. A developer must be aware of their existence in order to use them. These are often done after the fact in open source

code when someone realizes an interface exists and opens an issue to ask to support it.

2. What sequences of primitives are faster, when multiple exist to perform the same task?

   Given two implementations that perform the same task, one may not always be better than the other. Different implementations may work better on different file systems and may depend on workload characteristics such as page cache occupancy or data alignment. Identifying the best implementation requires profiling.

3. How to execute the sequence of primitives?

   Most primitives supported by file systems are conventionally exposed as system calls. Some file-system specific primitives are multiplexed over a single ioctl system call. Modern systems support alternate ways to execute primitives through asynchronous or batching interfaces. We term such methods of execution as transports—communicating what primitive needs to execute to the underlying system.

   Transports like io_uring use system calls to enter the kernel but the primitives themselves are prepared beforehand in user space and executed within the kernel without repeated user-kernel boundary crossings. However, the overheads of using this transport may outweigh the benefits depending on the workload. Currently, developers adopt a use-one-transport approach inside an application instead of choosing the best transport per task. Identifying the best transport for a task requires profiling.

4. Can context be provided to help the underlying system accelerate the operation?

> Applications that understand their workload characteristics may have additional data that can help the underlying system do a better job. Developers often use `fadvise()` to hint at sequential access or mark cold regions of a file whose page cache contents can be evicted.

Future improvements to file systems can introduce more primitives or change the dynamics of performance between existing ones. It is challenging for developers to keep track of these contributions and modify their applications to take advantage of them.

5

HSL: A declarative language for File System Intentions

In the previous chapter (§4), we observed that applications make multiple system calls to the underlying file system for one specific high-level task—an intention. Often times, there are multiple interfaces that can be used to implement the intention, and developers are required to select the most efficient interfaces. In some cases, a developer prioritizes durability above performance, and must know the inner workings of the interfaces to ensure correct crash-recovery behavior.

While selecting the most efficient interface is already challenging for a single underlying system, the complexity increases significantly when considering portability across different file systems and operating systems. Ensuring that the intention is correctly implemented with the desired performance and durability characteristics on various platforms requires in-depth knowledge of each system's nuances and behavior.

From Chapter 3, we concluded that durability is hard to get right. Despite adhering to POSIX standards, different file systems behave differently when `fsync()` fails. Unless the developer knows (1) the file system the application is being run on, and (2) understands its failure characteristics, there may be cases of data loss. In this chapter, we describe HSL, a language with runtime features which does precisely that—detect the underlying file system at runtime and handle durability correctly.

However, HSL is not *only* for getting durability correct. Despite sharing the same system call interface, file systems vary in how well they perform for different workloads; some even have special purpose non-standard interfaces. As HSL can detect the underlying file system, it can also improve performance by choosing the most efficient system calls for that system.

In this chapter, we show how applications can use HSL for those intentions to minimize data loss and improve performance. We start with the design of HSL (§5.1) followed by implementation details (§5.2).

5.1 Design

The goals of HSL are modeled around the benefits developers obtain by using high-level languages and their compilers. The compiler generates semantically equivalent correct code for every target architecture it supports and contains a growing repertoire of optimizations, making it easy to generate efficient code. HSL attempts to bring those benefits to applications that frequently interact with the underlying storage system. Specifically, we design HSL with the following goals in mind:

Reduce Cognitive Burden. HSL should make it easy for developers to express intent without having to know non-standard details about every file system they wish to support.

Portable Efficiency. As a portability layer, HSL must use the correct operations on systems. However, it must also choose the *most efficient* operation, which may differ across systems or versions for the same system.

Incremental Integration. Developers should not have to migrate an entire application to reap the benefits of HSL. HSL should co-exist with existing functionality in applications that are partially modified.

Standalone Sufficiency. While capable of being used alongside regular system calls in existing applications (incremental integration), developers should also be able to completely switch over to HSL without the need to use system calls at all.

Extensibility. Developers should be able to easily extend HSL to support their own specialized file systems with non-standard interfaces.

The remainder of this section describes the design of HSL.

5.1.1 Design Overview

HSL is composed of three main parts: a front, middle, and back end; similar to modern compiler design [67]. First, applications describe system interaction in a declarative language (HSL Script) to the HSL Frontend which performs syntax checks and converts the user-specified script into an intermediate representation (HSL Bytecode)[1]. Next, the HSL Middle-end performs a series of transformations over HSL Bytecode, adding instructions for correctness and combining multiple instructions for performance. Lastly, the HSL Backend selects the most efficient mechanism to execute each instruction. We describe each of the three parts below.

5.1.2 The HSL Frontend

The HSL Frontend consists of a few library functions that allow applications to interact with HSL, and the declarative language (HSL Script) that captures system interaction at a high level. The library functions are designed to resemble the APIs of popular SQL relational databases [77, 116]. In C for example, a SQLite [117] developer defines a SQL query and

[1]HSL Script is meant to be programmer friendly while HSL Bytecode is middle-end friendly. Performing optimization passes on HSL script would require complex regular expressions.

stores it in a string (`const char *`) variable. The developer then calls `sqlite3_prepare` which compiles the string query into SQLite's internal representation and returns a reference to it. Note that this compilation is done at runtime and not when the application is compiled. If the SQL query requires access to application data stored in variables, the developer calls `sqlite3_bind`. Finally, the developer calls `sqlite3_step` with the prepared reference as argument, to execute the query and read results. The HSL frontend provides similar functions: `HSL_compile`, `HSL_bind`, and `HSL_execute`.

As HSL is intended to be an interface to the underlying system, we choose to make HSL Script an external domain-specific language (DSL), independent from any one particular programming language. Applications that wish to use an embedded or internal DSL can do so through bindings that internally construct the script similar to object-relational mappers for SQL. An application provides a string in HSL Script format to `HSL_compile`, uses `HSL_bind` to provide arguments that are used in the script, and calls `HSL_execute` to run the script.

While the current version of HSL requires the use of `HSL_bind` at runtime, future versions can utilize compiler plugins to auto generate the boilerplate code and check HSL syntax when the application is being compiled. Listing 5.1 describes both versions with a simple example of reading 4096 bytes into a buffer. While HSL can be used as demonstrated, we intend to use it in situations that require multiple system calls; simple one-off system calls can still be used natively alongside HSL. The remainder of this subsection focuses on the HSL Script language and future listings will use the concise version without boilerplate code.

The HSL Script language is designed to capture system interaction in an easy to read high-level format. It consists of verbs, collections, statements, hints, and directives. Verbs declare the action required and may be accompanied with arguments (e.g., read, write, copy). When arguments are

```
1  void read_4K(int fd, char *buf) {
2    const char *str = "READ fd, buf, 4096;";
3    HSL_script_t script = HSL_compile(str);
4    HSL_bind(script, "fd", &fd);
5    HSL_bind(script, "buf", buf);
6    HSL_execute(script);
7  }
8
9  // When using compiler plugins to autogenerate
10 // boilerplate code equivalent to the above.
11 void read_4K(int fd, char *buf) {
12   HSL_script_t script(
13     "READ fd, buf, 4096;"
14   );
15   HSL_execute(script);
16 }
```

Listing 5.1: Using HSL in an application

required, simple forms may use direct arguments as Listing 5.1. However, as seen in §4, many intentions use the same action but are required to use different system calls due to the nature of their parameters (e.g., `pread()`, `preadv()`). We introduce collections as a way of decoupling actions from arguments. A statement combines verbs with their arguments. Hints and directives allow applications to provide domain-specific knowledge that cannot be inferred but help in triggering certain optimizations.

Table 5.1 is a summary of HSL's frontend features. We expand on each component below, providing examples of what a HSL script looks like for intentions described in §4.

5.1.2.1 Verbs and Collections

HSL verbs declare the intended action and are named as such. Simple actions such as reading and writing may share the same names as system calls but may not necessarily call that exact system call. For example, `read`, `pread`, `readv`, `preadv`, and `preadv2` all use the same verb—`READ`.

HSL Feature		Description
Collections	{ } @ name	Use an unordered collection
	[ ] @ name	Use an ordered collection
Verbs	READ	Read from one or more files
	WRITE	Write to one or more files
	APPEND	Append to one or more files
	FSYNC	Call fsync on one or more files
	COPY	Copy entire files or regions within files
	REDOLOG_*	Redo Logging Operations: Open, Read, Append, Sync, Apply
	MANIFEST_*	Manifest File Operations: Open, Add, Remove, Commit, Keys, Read
Hints	.expect	Inform of future read access or access pattern
Directives	.filter	Filter a read buffer according to delimiter and keyword
	.filter_endpoint	Like .filter but perform compute closer to storage e.g. AWS Lambda closer to S3.

Table 5.1: **Summary of HSL Frontend Features:** The table provides a description for each frontend feature. The verbs that work on *one or more* arguments use collections to do so.

```
// A simple single contiguous read.
// No collections necessary.
READ fd, buf, 4096;

// Reading contiguous pages of size 4K into
// different buffers (instead of one or more readv).
// 'records' is the name of the collection.
READ fd, [buf, 4096]@records;

// Reading scattered pages instead of
// multiple preadv system calls.
READ fd, {buf, offset, 4096}@pages;
```

Listing 5.2: Using collections with verbs: Reading from files.

```
// Appending records to a file
APPEND fd, [buf,sz]@records;
// Performing scattered writes to a file
WRITE fd, {buf, offset, bufsz}@pages;
// Syncing a file
FSYNC fd;
```

Listing 5.3: Writing to files.

```
// Reading 4KB from many different files
READ {fd, buf, 4096}@fileset;
```

Listing 5.4: Homogenous operations on multiple files.

```
// Copying entire files
COPY src_fd, dst_fd;
// Copying a single range
COPY src_fd, dst_fd, src_offset, dst_offset, size;
// Copying multiple ranges
COPY src_fd, dst_fd,
    {src_offset, dst_offset, size}@ranges;
```

Listing 5.5: Copying data.

The exact set of system calls eventually used is determined by both the verb and the arguments. When arguments are enclosed in curly (`{}`) or square (`[]`) brackets, they represent an unordered or ordered collection respectively. After the closing brace is an `@` followed by a name for the collection, so that applications can bind vectors or arrays as arguments. Ordered collections have to execute each entry in the order provided, while unordered collections can be executed in any order. When appending to a file, the buffers may have to be written in a specific order, requiring the use of ordered collections. However, when performing a multi-chunk scattered read (§4.3.1.2), the application may not care about the order so long as all buffers are filled before it proceeds. In such cases, the unordered collection would be more appropriate as the absence of an ordering constraint could increase opportunities for optimization by the HSL middle-end and backend. Listing 5.2 demonstrates the use of verbs and collections.

The separation of action from argument aligns with HSL's goal to reduce cognitive burden for developers. By providing all arguments for a single verb, the developer need not reorganize code for the underlying system's limitations. For example, scatter-gather interfaces like `readv()` can only accept a maximum of `IOV_MAX` entries at a time, requiring special handling for bigger entries. The same listing (Listing 5.2) also demonstrates how single contiguous reads and multi-chunk scattered reads can be expressed in HSL. Listing 5.3 demonstrates the use of verbs and collections towards single file writes, both adding data and overwriting data. The scattered write is equivalent to LMDB's implementation of writing dirty pages from Listing 4.10), where optimizations for contiguous pages are handled internally by HSL. Additionally, developers can use `FSYNC` to sync file data. Unlike the `fsync()` system call, the HSL middle and backend will be responsible for proper implementation of file syncing and providing consistent behavior semantics to the user regardless of the

```
1  //  Open log for basefile and chunksize
2  //  store handle in variable rfd
3  REDOLOG_OPEN rfd, basefile, chunksize;
4  //  Read latest stable chunks through rfd
5  //  Provide chunk_id and buffer through collections
6  REDOLOG_READ rfd, {id, buffer}@chunks;
7  //  Append modified chunks to the log
8  REDOLOG_APPEND rfd, {id, buffer}@chunks;
9  //  Sync the log
10 REDOLOG_SYNC rfd;
11 //  Apply all updates from log to basefile
12 REDOLOG_APPLY rfd;
```

Listing 5.6: Using redo logs.

```
1  //  Open a manifest file at the desired path
2  MANIFEST_OPEN mfd, path;
3  //  Add file reference with an associated key
4  MANIFEST_ADD mfd, key, path/to/file;
5  //  Remove a key (and its file reference)
6  MANIFEST_REMOVE mfd, key;
7  //  Persist changes as a new immutable set
8  MANIFEST_COMMIT mfd;
9  //  Returns a list of all keys in the set
10 MANIFEST_KEYS mfd, num_keys, keys;
11 //  Obtain a file descriptor to read the
12 //  file associated with key
13 MANIFEST_READ mfd, key, fd;
```

Listing 5.7: Using manifest files.

underlying file system. Homogenous operations on multiple files are also captured through collections (Listing 5.4). Heterogenous intentions such as copying (Listing 5.5) has its own verb. However, the exact copy workload (few large ranges vs many small ranges) is detected and handled internally by HSL.

HSL is designed to be extensible, allowing developers to add their own dedicated custom verbs for intentions not covered by us. While intentions can be *inferred* through middle-end and backend passes, some intentions

```
READ fd, buffer, size;
.expect SEQ
.expect RSEQ
.expect STRIDE length
```

Listing 5.8: Expecting sequential or strided reads.

such as those relating to atomicity and durability require finer control. We provide dedicated verbs for redo logs (Listing 5.6) and manifests (Listing 5.7). These verbs also reduce a developer's cognitive burden, offering a simple failure model that hides the failure characteristics of independent file systems. Section 5.2 provides details on their internals.

5.1.2.2 Statements, Hints, and Directives

Statements are verbs followed by arguments (which could be collections), terminating with a semicolon. While the HSL middle and backend can pick the most efficient system calls for statements based on action and argument, awareness of how scripts are executed by the application can lead to better optimizations. Hints and constraints allow applications to convey such information to HSL. Applications can also introduce new middle-end passes and backend implementations for custom hints and directives, allowing for domain-specific transformations.

Both hints and constraints start with a period followed by an identifier and is associated with the previous statement. The key difference between hints and constraints is optionality. Transformation passes may use hints to perform better optimizations, which when absent may result in lower performance. However, directives are restrictions that the transformation passes must not violate to ensure desired behavior.

Expect Hint for Better Read-Ahead. While developers prioritize using data structures that yield sequential access patterns, they cannot entirely eliminate non-sequential accesses. Traditionally, applications that wish

```
READ fd, buffer, size, offset;
.expect next_offset
```

Listing 5.9: Expecting scattered reads.

```
READ fd, chunk, chunksz;
.filter \n, keyword, callback_fn, SEQ|RSEQ
.filter_endpoint s3fs http://url/to/aws/lambda
```

Listing 5.10: Filtered reads.

to inform the kernel of these access pattern use `fadvise()`. The `expect` hint allows developers to do so similarly through HSL. However, unlike `fadvise()`, HSL chooses to communicate access patterns only if doing so is beneficial. Listings 5.8 and 5.9 demonstrate the use of the expect hint in different multi-chunk reads (§4.3.1.2).

Filter Directive. As described in §4.3.1.3, some applications read data into their user-space buffers and perform filtering logic, discarding records that don't meet some criteria. HSL provides the `filter` directive to accomplish the same, where users can bind criteria and a callback function to be called for every record matching the criteria; the callback return value controls future iterations. Additionally, users specify the direction of traversal–sequential or reverse sequential. The `filter` directive exposes a uniform interface, which is beneficial when applications are deployed in many different environments. While running on a native Linux file system may see no benefits, remote file systems may offer methods to move compute closer to storage. The `filter` directive also accepts `filter_endpoints` which can execute filtering logic on the remote file system. Developers can therefore use the same interface and need only add endpoints when they are available. Listing 5.10 demonstrates the use of the filter directive with an AWS Lambda function when the underlying store is AWS S3.

5.1.2.3 Preparing for the next phase

In addition to reducing cognitive burden and being extensible, applications control the execution of HSL Scripts through `HSL_execute`. This allows applications to run existing code with system calls alongside HSL, satisfying our goal of incremental integration. Additionally, HSL could provide standalone sufficiency by implementing all existing system calls following the same action argument separation philosophy.

The frontend converts a HSL script which is a sequence of characters into a binary representation to avoid transformations through string matching. Additionally, it performs syntax checks and organizes the binary data such that each statement and following hints and directives are associated. The binary representation is then passed to the middle-end which can perform transformations which are then executed by the backend. This indirection (contrast with directly executing system calls) through HSL Scripts allows HSL to achieve portable efficiency.

5.1.3 The HSL Middle-end

As the backend operates on each verb independently, the middle-end is where bytecode is transformed with larger context (multiple verbs). The HSL Middle-end exists to perform correctness and performance transformations over the frontend generated bytecode. While the initial bytecode is a single basic block, transformations that rely on runtime information inject control-flow instructions to execute other basic blocks. We first describe how runtime information can be accessed, followed by two example peephole optimizations.

5.1.3.1 Accessing Runtime Information

The transformations performed by the middle-end currently rely on runtime information that is in one of two categories: file-system traits or

file-descriptor traits. The design can be extended in the future to utilize device traits.

File-system traits. The middle-end can transform bytecode differently depending on the underlying file system. For example, POSIX makes no mention of whether directory entries must be persisted when `fsync()` is called on a newly created file. While the safest solution is for applications to always include an `fsync` to the parent directory, it is unnecessary (added cost[2]) on file systems like ext4, XFS, and btrfs. One possible method of conditionally executing the parent directory `fsync` is to check the file-system name or type against a known set at runtime. However, such a method requires updates to the middle-end pass whenever a new file-system with similar characteristics is identified. Additionally, file systems can change their behavior depending on the options provided to them when mounted. To address these issues, middle-end transformations check if the underlying file system possesses specific characteristics (traits). For the same example, the middle-end transformation does not need to know what the file system is but whether it persists the directory entry –a trait we call "safe-file-flush". We require that file systems only change their traits on remount, or to forcefully close all open file descriptors before doing so, minimizing frequent querying of traits. However, traits must be queried on every `open` system call as even two files in the same directory can reside on different file systems through softlinks.

File-descriptor traits. While most transformations rely on file-system traits, there are some that also require information specific to the descriptor. Using the previously mentioned example, knowing the safe-file-flush file-system trait is insufficient. Files opened with `O_CREAT` are not always created; they may already exist. The transformation should insert an `fsync` to the parent directory only if the file is newly created and the file system does not support the safe-file-flush trait. Other traits include flags

[2]For devices with volatile caches, some file systems always issue a device cache flush on `fsync()` even when there are no dirty pages [37].

used to open the file descriptor. For example, knowing that a file was opened with `O_SYNC` can allow transformations to ignore the `fsync` that follows a `write`.

We identify traits through an `ioctl()` to a custom kernel module, which are then cached to minimize lookups.

5.1.3.2 Peephole Optimizations for Standard Verbs

In a one-to-one mapping of verb to backend execution, there is an added responsibility on the developer to choose the right verbs. For example, the developer must use the `COPY` verb instead of a read followed by a write. However, one of the goals in HSL is to provide optimizations similar to what modern compilers do for poorly written user-space code. We introduce a copy-verb-combiner pass that looks for read and write verbs that use the same buffers and replace them with the copy verb. The backend can then decide the best way to perform the copy. More such passes can be added to the middle-end, building a repository of optimizations incrementally like those found in modern compilers.

5.1.3.3 Peephole Optimizations for User-Defined Verbs

Applications with specific patterns of system calls may build special underlying file systems (or modify existing ones) to support them. These niche operations are only applicable when on a specific system and do not require dedicated verbs until a majority of systems decide to support it. Developers who wish to use such operations can introduce a backend implementation for the verb as well as a middle-end pass to replace relevant bytecode. Later, we expand on one such optimization—concatenation—in §5.2.3.2.

5.1.4 The HSL Backend

The HSL backend is responsible for executing bytecode generated by the middle-end. It consists of two parts: a fixed interpreter and backend modules. The interpreter iterates over the bytecode, calling the right backend module to execute a verb, and handles control flow operations generated by the middle-end based on runtime parameters. The backend modules choose the best way to execute a verb when assumptions match the runtime parameters. While we currently use an interpreter to execute bytecode, the design supports implementations that transpile and compile or JIT compile bytecode. Execution through a backend module involves choosing the right operation supported by the underlying system (Instruction Selection), and the right way of communicating with the underlying system (Transport Selection). We describe both below.

5.1.4.1 Instruction Selection

A given bytecode operation maps one-to-one to code to be executed. An optimization must be limited to just that operation and any optimizations that were to be done across operations should have been done in the middle-end. Using previous examples, a `READ` verb could use `read()` or its variants, and similarly for WRITE. When ordered collections are involved, the default implementation would try to reduce the number of system calls if offsets are contiguous and use the Scatter-Gather variants; unordered collections sort offsets before applying the same optimization.

Instruction selection for a `COPY` is more involved, requiring inspection of arguments at runtime to decide the instructions based on the workload. Additionally, the implementation uses runtime information such as the file system being used and page cache occupancy. We provide details on how `COPY` is implemented in §5.2.3.1. Similarly, custom verbs for niche optimizations are also implemented in backend modules.

5.1.4.2 Transport Selection

A transport is the mechanism through which we execute operations on the underlying system. While the prevalent transport is POSIX system calls for portability, HSL is designed to work with different transports. As new transports are developed, upgrading HSL to one that uses the transport (or developing your own) allows all HSL code to use the new transport.

An example transport that exists only on recent Linux kernels (since v5.1) is `io_uring` [7]. HSL applications can choose to always use io_uring or let the backend decide when it is beneficial, choosing the best among multiple transports with runtime information.

Another example transport is through shared memory. Kernel-bypass file systems as user-space processes use shared-memory ring buffers and provide a client library for applications to communicate in the designed protocol. HSL-enabled applications can easily use kernel-bypass file systems by defining the protocol for communicating verbs in the backend.

We implement the standard system call transport and the `io_uring` transport (details in §5.2.3.5).

5.2 Implementation

We implement HSL primarily in C++, using ANTLR to parse the HSL Frontend. The middle-end and backend are implemented purely in C++, while the kernel module for trait detection is implemented in C.

We now describe the implementation details of the intentions themselves. We first describe two high-level intentions catering to correctly persisting data: redo logging and manifests. We then focus on details relating to performance.

5.2.1 Redo Logs in HSL

As described in §4.3.3.1, some applications require that updates to files be performed atomically and guarantee that the update survives a power failure. If implemented correctly, a redo log satisfies these requirements. While there is no guarantee of arbitrary applications interacting with the data, threads or processes owned by the application will always follow the same protocol; redo logs satisfy app-local atomicity and durability.

We start with a description of the semantic guarantees redo logs must provide, followed by the API HSL exposes. We then discuss the internal implementation details and strategies for handling different file systems.

Semantic Guarantees

When an operation modifies application state, the application first writes the modification to the log. The modification must be *persisted* to the log before an acknowledgement to the user so that it can be read again in case of a crash or restart. On restart, the state before a shutdown or crash is reconstructed by replaying items from the log. As logical redo logs are application specific, we provide custom HSL verbs for a generic physical redo log.

Applications that use physical redo logs typically write data to files in *chunks*. For example, relational databases like SQLite and PostgreSQL store B-Trees as files on disk. These databases divide a file into fixed size chunks (also called pages) which represent B-Tree nodes; this file is called the *base* or *backing* file. An operation that modifies state here translates to a modification of one or more of these chunks; the modified chunks are written to the log. Later, for read operations, the newly written content in the log must be used instead of the chunks in the backing file.

All chunks that are written to the log file as part of an update operation must be written atomically. Doing so avoids invalid states that occur by reading old state for some of the chunks from the backing file and new state for the rest from the log file. As applications may wish to perform

more than one update as part of a transaction, all chunks modified for all the updates within that transaction must be written atomically.

If the redo log reports a successful persistence of the chunks, then future reads of those chunks must serve the new content present in the log. A corollary is that if the redo log indicates it couldn't persist the chunks, then future reads must serve the old content for *all* chunks as part of that transaction. Additionally, until the redo log reports a successful persistence, all previous reads must serve the old content.

Over time, as the redo log grows, future updates may write chunks that are already present and persisted. The above guarantee of serving old content now corresponds to the previous persisted chunk in the log, not the backing file. Generically, we use the term "stable chunk" for chunks that have been persisted. Initially, the stable chunks are in the backing file. After successful log persistence, the stable chunk is in the redo log. The redo log must always serve the latest stable chunks.

HSL API for Physical Redo Logging

Listing 5.6 demonstrates the use of physical redo logs through HSL Verbs. With `REDOLOG_OPEN`, a log is opened/created referencing a backing file (e.g., the main B-Tree file). When chunks from the backing file are to be modified, they are appended to the log with `REDOLOG_APPEND`. When appending one or more chunks, the data must be accompanied by chunk identifiers. Since the backing file is divided into fixed sized chunks, the identifiers are the indices of these chunks in the backing file. If chunk 0 is the first chunk starting at offset 0, chunk n starts at `n*chunksize`. As the redo log guarantees atomicity, all chunks can be passed via unordered collections ({}) instead of ordered collections ([]).

While a single update involving multiple chunks can be appended with collections, as mentioned earlier, the application may wish to issue multiple such appends (multiple updates in a transaction) before guaranteeing persistence. We provide a separate command `REDOLOG_SYNC` to persist the

log after all appends have been made with REDOLOG_APPEND.

Over time, modified chunks will have their latest versions in the log and unmodified ones in the backing file; applications will have to read the latest versions. REDOLOG_READ provides a single interface that returns the latest stable data for one or more chunks. In line with the semantic guarantees, chunks that are being written to the log with REDOLOG_APPEND are not yet considered stable. These chunks will not be served to future reads through REDOLOG_READ unless they have been first made stable with a successful REDOLOG_SYNC.

After a HSL script finishes executing, like system calls, it returns an error code (0 for success). A non-zero code indicates an error. When the application receives a return code 0 for a script containing REDOLOG_SYNC, it can assume that the log is persisted in line with semantic guarantees. We use the REDOLOG_SYNC in its own script to guarantee that the error (if any) is specific to syncing.

While not part of the semantic guarantees, we provide REDOLOG_APPLY for practical reasons—to prevent the log from growing unbounded. As the application receives more and more updates, the log will tend to have multiple versions of stable chunks; the log may eventually grow much larger than the backing file itself. The REDOLOG_APPLY is meant to mitigate this issue by overwriting the chunks in the backing file with the latest stable content in the log. Doing so makes the backing file the latest stable reference allowing the log to be truncated—a checkpointing operation. Regardless of whether checkpointing succeeds or fails, the semantic guarantees mentioned previously must still hold.

Internals

In the following paragraphs, we describe one backend implementation of physical redo logging. However, like all HSL verbs, more backends can be implemented in the future. We begin with a description of the on-disk layout of our redo log implementation followed by details for each verb.

The on-disk layout of the redo log starts with a main header, which contains a unique identifier to this redo log implementation, allowing for future work to dynamically change the redo log implementation based on runtime statistics. It also includes information about the chunk size and the backing file to which it belongs.

Following the main header, the log consists of one or more chunk groups and each group begins with a chunk group descriptor. To ensure atomicity, the size of the chunk group descriptor is limited to the atomic sector size of the underlying storage device, which is typically either 512 bytes or 4096 bytes. The descriptor is structured as an array of 64-bit integers, with each integer encoding both a chunk ID and an epoch. The chunk ID and epoch for the i^{th} in the group can be obtained from descriptor[i]. After the chunk group descriptor, the actual content of all the chunks listed in the descriptor is stored sequentially.

On `REDOLOG_READ`, HSL searches an in-memory index for each chunk to be read. If present in the index, we read the chunk from the log. If absent, we read the chunk from the backing file. On application restart, HSL rebuilds the in-memory index by reading the chunk group descriptors.

On `REDOLOG_APPEND`, the next free entry from the descriptor is selected and marked in use by setting the chunk id and epoch. The chunk id identifies the chunk in the backing file while the epoch helps identify an atomic set of chunks; epochs are only incremented on `REDOLOG_SYNC`. While the chunk data is written to the log file, the descriptor is only modified within the application's user-space memory. An internal index that maintains the location of latest chunks in the log is not updated yet, so that `REDOLOG_READ` operations still read content from previous epochs.

On `REDOLOG_SYNC`, we first persist the log, which involves `fsync()`. A failure here results in a failure reported to the application. However, we still maintain semantic guarantees. Any failure to persist chunk data is easy to recover from as the descriptors are only modified in the application's

memory. However, as applications can continue operating (instead of using a crash-on-failure strategy), we also reset the in-memory descriptors.

Once the chunk data is successfully persisted, the descriptors must be written and persisted so that any future recovery logic can identify the chunks and their epochs. If all chunks are in a single chunk group, atomicity is guaranteed as the descriptor occupies a single sector.

If the chunks span multiple chunk groups, multiple descriptors need to be persisted *atomically* to prevent breaking semantic guarantees. To do so, we write the descriptors in reverse, i.e., starting with the last chunk group descriptor and persist them one by one.

Any persistence failure when writing descriptors also results in reporting failure to the application. Writing descriptors in reverse ensure a gap in the redo log as the most immediate descriptor after the last sync is not persisted yet. Recovery logic that builds state for future `REDOLOG_READ`s stops at the gap, discarding content that was reported to the application as a failure.

In the absence of any failures, the epoch is updated (for future `REDOLOG_APPEND`s), the internal index for chunks is updated, and error code 0 is returned. Any future `REDOLOG_READ`s for chunks that were part of this `REDOLOG_SYNC` are now served with this latest content as pointed to by the internal index.

On `REDOLOG_APPLY`, the latest chunk data from the log is copied to the backing file. The backing file is first persisted and any failure terminates the operation with a non-zero return code. Even if the backing file contains partial overwrites, semantic guarantees still hold as the log is still available with the latest data. In the absence of failure, the log is truncated, the internal index erased (so all reads are served from the backing file), and the epoch reset to 0.

Handling Different File-System Characteristics

When persisting any files as part of `REDOLOG_SYNC` or `REDOLOG_APPLY`, HSL takes into account the file system the log resides on. Using runtime information (on `REDOLOG_OPEN`), HSL records the file system for the log and the backing file. These files are usually on the same file system but we also handle the case where they are not.

On ext4 data mode, when an `fsync()` has to be issued, we issue a second `fsync()` to avoid failed intentions and detect failures correctly; a workaround for ext4 data mode delayed error reporting.

If `fsync()` fails when persisting chunk data, nothing more needs to be done as our implementation has not yet written the descriptors. However, if `fsync()` fails when persisting chunk descriptors, we must perform corrections. For chunks that span multiple descriptors, all descriptors and groups except the first to be modified are truncated.

For the first descriptor, we revert the page cache contents on ext4 and XFS as the contents do not match what is on disk. We maintain a previous "stable" version of the descriptor that is rewritten to revert the contents. An alternate implementation can open the file with `O_DIRECT` and read the sector again.

5.2.2 Manifests in HSL

As described in §4.3.4.4, applications like LevelDB work on a *set* of immutable files. Any changes to the set (adding new files or removing/replacing existing ones) must happen atomically, allowing existing readers to continue reading an old consistent set of files while new readers start using the new set. Manifest files—regular files whose data contains the names of files in the set—are the standard method of doing so. Here, we describe the semantic guarantees of manifest files, followed by the API HSL offers. We then discuss internals and file-system specific strategies.

Semantic Guarantees

We start with an initial set of files and a manifest that contains references to those files. While some applications may treat the manifest as a *set* of files, there may be cases where applications wish to associate some more information with each file. They could encode that information in the file name, but doing so makes it difficult to generically identify whether a new file must remove an existing file or be added as a new entry. Instead, we associate a key with each file and treat the manifest as a key-value map where keys are strings and values are filenames (also strings).

The set of files referenced by the manifest are assumed to be immutable. At any given time, the values (references to files) in the map must exist. When an application wishes to update the set of files it updates the manifest either by removing, inserting, or replacing a key-value pair.

Any existing readers of the manifest must not be affected by the update. For example, if a reader is currently iterating over all files in the manifest and reading them, it must not read any newly inserted pairs made by a recent writer. The set of files a reader accesses is determined and fixed at the time the reader opens the manifest. Therefore, when removing key-value pairs or replacing files, the files should not be deleted by the application as a reader of an old manifest version may be using it.

While a writer can update the manifest with one or more modifications, these updates must not be visible to other readers. However, the writer itself should be able to see those changes. Once all changes are made, the writer should be allowed to *persist* those changes so that they are visible to future readers. If the writer receives a successful return code for the persistence, future readers must read from the new version. A corollary is that failures during persistence must ensure future readers read the old existing version.

HSL API for Manifests

Listing 5.7 demonstrates the use of Manifests through HSL. With `MANIFEST_OPEN`, a manifest is opened or created at the given path. In the case of LevelDB, this would be the path to CURRENT; the entry point to its manifest (expanded on later).

As manifest files contain key-value pairs, `MANIFEST_ADD` and `MANIFEST_REMOVE` add or remove these pairs. As all modifications are only visible to the writer, replacing a value for a particular key is a remove followed by an add. The `MANIFEST_KEYS` verb allows the application to obtain a set of all keys in the manifest.

Using `MANIFEST_READ`, applications can obtain a read-only file descriptor for the file referenced by a given key. We initially provided a `MANIFEST_GET` verb to obtain the path to the file but removed it as opening of files and deletions of old files are handled by HSL.

With `MANIFEST_COMMIT`, all modifications to the set are written and persisted such that new readers with `MANIFEST_OPEN` view an updated set.

Internals

In the following paragraphs, we describe one backend implementation of manifests. We begin with a description of the on-disk layout followed by any implementation-specific details for each verb.

On-disk layout. The path provided to `MANIFEST_OPEN` is the root file. It does not contain they key-value pairs with file references. Rather, like LevelDB's CURRENT, it contains information about where readers can find the key-value pairs.

Like HSL Redo Logs, the root file begins with an implementation identifier (a 64-bit magic number) to identify this manifest backend implementation. Followed by the identifier is a list of 64-bit integers representing a timestamp since some standardized epoch (we use the unix 1970 epoch). We ensure that the root file is no larger than a single sector for the atomic sector write property.

The latest manifest file is obtained by selecting the latest timestamp and the filename is a combination of magic number and timestamp. Every manifest file is an on-disk hashmap representation where both keys and values are strings. The keys are user-provided keys, and the values are the file references.

On `MANIFEST_OPEN`, the root file is read to identify the latest manifest file. Then, the manifest file is read to construct an in-memory hashmap. We then iterate over all file references and keep open file descriptors to all of them, allowing us to read any files in the set even if later deleted by other applications. The entire process is wrapped in a critical section using an advisory shared file lock on the root file. Additionally, we keep a copy of the root file (a sector worth of memory) in application memory and remove any timestamps from it that do not exist as manifest files.

On `MANIFEST_READ`, we refer to the in-memory hashmap and duplicate the file descriptor previously opened.

On `MANFEST_ADD` **and** `MANIFEST_REMOVE`, only the in-memory hashmap is updated.

On `MANIFEST_COMMIT`, a new manifest file is created with the current timestamp. The in-memory hashmap is serialized to the manifest file and persisted. Any failure to persist returns a non-zero error code to the application. We remove the newly generated manifest file to avoid wasting space; keeping it does not break semantic guarantees.

Unlike LevelDB, we do not rely on atomically renaming the root file with a different root file. Instead, after the manifest file is successfully persisted, we proceed to update the root file *in place*. We acquire an advisory exclusive file lock on the root file and then add the newly generated timestamp to its contents. We then persist the root file while still holding the lock. On successful persistence, we remove obsolete files: file references that were removed and the old manifest file. Finally, we release the lock. Future readers on `MANIFEST_OPEN` will read the updated root file and

obtain the new set. We cover persistence failures later.

Note, we do not remove timestamps or rewrite the root file again. Instead, as mentioned earlier, `MANIFEST_OPEN` will remove the timestamps in its copy of the root file which will be persisted on the next `MANIFEST_COMMIT`. While the root file is the size of a sector and can accommodate 63 timestamps (assuming 512B sector and first 8 bytes for the magic number), regular usage will limit timestamps to two.

After all changes are complete, the application persists the changes with `MANIFEST_COMMIT`. During this operation, with the current timestamp a new manifest file is created and the hashmap is serialized to disk.

Handling Different File-System Characteristics

Like `REDO_LOG`s, persistence of individual files is handled with correct use of `fsync()`—double on ext4 data mode.

We handle the case where directory entries on ext4 can silently disappear (§3.1.2.1). To ensure correct behavior on ext4, we first force a checkpoint operation in `MANIFEST_COMMIT`, before acquiring the exclusive lock and updating the root file. When running on Linux 5.14 or later, we use `ioctl(EXT4_IOC_CHECKPOINT)` to trigger the checkpoint from user space. Unfortunately, on older versions, apart from writing a custom kernel module to trigger checkpoints, we must wait 30 seconds on a default mounted ext4 file system for the metadata (directory data blocks) to be written to disk. Once the checkpoint has completed, we re-read the directory entries. Only after we confirm that the manifest file and all entries it references are not missing, we proceed to update the root file.

Since updating the root file does not change directory entries (unlike LevelDB atomically renaming CURRENT), we need only ensure correct fsync handling. Like `REDO_LOG`s, when on ext4 and XFS, we revert the root file's contents in the page cache if `fsync()` fails, while holding the exclusive lock. As the root file is always less than a sector, the on-disk state is guaranteed to be the old state and does not need special handling.

5.2.3 HSL Features for Performance

5.2.3.1 Instruction Selection with `COPY`

As described in §5.1.2, HSL provides a COPY verb for direct use and also has a peephole optimization pass to transform existing code that it detects to be a copy. As there are multiple ways to copy data from one file to another, we first describe them and then explain how HSL chooses the right implementation.

The simplest (and oldest most portable) method to copy data is a `read()+write()`. Data is first read from the source file into a buffer and the buffer is then written to the destination. To avoid unnecessary copies, `sendfile()` accepts both the source and destination file descriptors and writes a given number of bytes from source to destination[3]. To take advantage of copy-acceleration techniques such as copy-on-write extent sharing or server-side copying, `copy_file_range()` was introduced in Linux 4.5 but majorly revised in Linux 5.3.

Deciding which of the above three approaches to choose can depend on whether the files are on the same file system. As `copy_file_range()` is unsupported on different file systems[4], we default to `sendfile()` as it avoids multiple copies and user-kernel transitions. When on the same file system, the type of workload matters. As mentioned in §4.3.4.2, we observe two distinct workloads: few large range copies and many small range copies. Through benchmarking (shown in §6.2.2.1 when evaluating COPY), we observe `copy_file_range()` outperforms the rest for large range copies. However, for many small range copies, it depends on the file system and page-cache occupancy. As ext4 does not perform such acceleration, we default to `copy_file_range()`. XFS and Btrfs perform copy accelera-

[3]Early versions of Linux only allowed the destination descriptor to refer to a socket. As of Linux 2.6.33, file descriptors can be used as well.

[4]As of Linux 5.19, file system copies can be achieved on different file systems provided that they are of the same type. However, acceleration is limited as they may not reside on the same underlying device.

tion through copy-on-write extent sharing, previously available through `ioctl_ficlonerange()`, internally, always using `remap_file_range()` for the copy. As our benchmarks show that `remap_file_range()` performs worse for small range copies if the data is in the page cache, we add heuristics to choose between `copy_file_range()` and `sendfile()`, using `cachestat()`[5] to guage cache occupancy.

5.2.3.2 `CONCAT`: A User-Defined Verb

As HSL is designed to allow user-defined verbs which replace existing verb sequences through peephole optimizations (§5.1.3.3), we provide an example of its usage for concatenating files. The concatenation of files—commonly performed by the coreutils `cat` utility—reads data from multiple files and appends them to a destination file in a given order. In terms of a copy workload, the concatenation belongs to the *few large range copies* category.

The existing coreutils `cat` utility tries to use `copy_file_range()`, a try-by-failure approach that falls back to `read()+write()`. However, `copy_file_range()` has a restriction that the destination file descriptor cannot be opened with the `O_APPEND` flag. While HSL applications can benefit from using the `COPY` verb as described previously, some file systems may support faster methods of concatenation. The operation itself is not as widespread to necessitate a new verb; too many verbs can increase cognitive burden. Instead, such niche optimizations can be performed internally by transforming the `COPY` verbs into a `CONCAT` verb in the middle-end for file systems that have such interfaces. Currently, we inject code in the middle-end to test whether the file-system matches some known file systems that can execute the `CONCAT` verb. An alternate method for the future would be to define it as a trait—concatenation acceleration. We

[5]Available since Linux 6.5

now describe how `CONCAT` is implemented in the backend, focussing on two FUSE file systems: s3fs and gcsfs.

Cloud object storage such as S3 and GCS provide FUSE file-systems: s3fs and gcsfs respectively. While FUSE support for `copy_file_range()` has been available since 2018, s3fs and gcsfs do not support it yet. Both S3 and GCS could implement an efficient `copy_file_range()` which creates a multipart upload that references the source objects. However, until s3fs and gcsfs officially support the implementation, one could modify the file system and use `copy_file_range()`, or directly implement the multipart upload in the HSL backend. Furthermore, multipart uploads require that the parts be greater than 5MB; small files will have to be downloaded, concatenated, and re-uploaded. Although concatenation for S3 cannot be further optimized, GCS has a dedicated concatenation interface: the `compose()` API.

GCS supports `compose()` that takes between 1 and 32 objects and creates a new composite object. Additionally, it does not have the 5MB constraint seen with multipart uploads. Unfortunately, despite GCS supporting `compose()`, the FUSE file-system gcsfs does not. When using gcsfs, the HSL backend directly issues the `compose()` request to GCS bypassing gcsfs. Future implementations can modify gcsfs to accept an `ioctl()` for compose and have the HSL backend invoke the ioctl instead. Internally, if the number of files exceed 32, HSL uses intermediate files and finally calls `compose()` on the intermediate files—a reduce operation.

5.2.3.3 The `.expect` Hint

Conventionally, when an application wants to tell the kernel how it expects to use a file handle, it uses the `fadvise()` system call so that the kernel can employ an appropriate readahead and caching techniques. Among the `fadvise()` flags, `FADV_SEQUENTIAL` is used for forward-sequential accesses and `FADV_WILLNEED` (also used by `readahead()`) is used for arbitrary ac-

cess. We do not focus on the other flags as they either disable readahead or evict pages. Our current implementation focuses on ensuring future reads are already in the page cache, using the `EXPECT` hint in HSL.

As described in §5.1.2.2, hints unlike directives in HSL can be ignored. We ignore hints when the underlying system already performs the readahead by default. Sometimes, arbitrary accesses happen to be sequential or strided within the thresholds of the default readahead logic. In such situations, an `fadvise()` is unnecessary.

Additionally, hints are only helpful if the next expected read request appears after sufficient time for the system to complete the expected readahead. If the time between reads is significantly lower than the time required to read from disk, the `fadvise()` is wasteful, adding an extra system call overhead. Since measuring time since the last `fadvise()` or querying the cache with `cachestat()` before every read adds overhead, we use non-blocking reads. The `preadv2()` with `RWF_NOWAIT` sets errno to `EAGAIN` if it has to read data from backing storage, a signal that the `fadvise()` was not effective. The backend then switches to an implementation that ignores the hint, which can periodically switch back to see if things change.

5.2.3.4 The `.filter` Directive

As described in §4.3.1.3, applications do not always require all the data they read. As shown in Listing 4.8, they make multiple system calls, filling up a buffer, only to discard records that don't meet some criteria. The HSL `FILTER` directive is an alternative way to accomplish the same in HSL.

When the HSL backend detects a `FILTER` directive associated with a `READ` verb, it executes the callback function. Additionally, we also use the same logic associated with the `EXPECT` hint, calling `fadvise()` with the necessary flags depending on the access pattern provided. A forward-sequential pattern uses `FADV_SEQUENTIAL` and the reverse uses `FADV_WILL NEED` calculating the offset based on buffer size.

As some filtering workloads are performed over objects stored in the cloud exposed via FUSE (s3fs, gcsfs), reading extra data increases network egress costs and—depending on network bandwidth—runtime. If a filter endpoint is provided, the backend first checks if the file is cached locally to use the local approach mentioned above. If the data must be fetched over the network, the provided endpoint is used to run the filtering logic at the source before returning the filtered items, minimizing network traffic.

We currently implement support for AWS Lambda functions as endpoints. For our filtering workload, we deploy a function written in Python that minimally accepts the object name, offset, direction of scan, target buffer size, delimiter, and keyword. While the function is limited in the number of bytes it returns after filtering (the target buffer size), it can read much more before filtering. For example, if a given word only occurs on the first and last line of a file, a single lambda invocation would suffice provided both lines fit within the buffer.

However, we identify two limitations with AWS Lambda as endpoints. First, the functions are expected to run within a certain time limit (up to 15 minutes [66]). Second, the function output must be less than a given limit (6MB or 20MB depending on whether requests are synchronous or not). Therefore, the size of buffers used by AWS Lambda endpoints share these limitations. While we demonstrate the effectiveness of filters with AWS, other cloud providers or even self-hosted object storage (e.g., ceph) may not have these limitations. When provided with a large buffer, the backend will issue multiple requests that satisfy the limits.

While network egress costs are a factor, applications that optimize for latency might still perform faster if they have a fast network connection. The current implementation always uses the available endpoint if data needs to be fetched. However, a future backend can implement a dynamic selection between local and endpoint versions depending on user requirements.

5.2.3.5 Using the `io_uring` Transport on Collections

As described in §4.3.1.2, multi-chunk reads, writes, and operations on multiple files involve multiple system calls. The scatter-gather interfaces (`*readv()`,`*writev()`) are also limited to a maximum of `IOV_MAX` entries (1024). Collections in HSL allow us to capture all the arguments necessary to perform these operations irrespective of the underlying system's limitations. The HSL backend can then choose the appropriate transport to execute the verb associated with the collection. We currently implement two transports: the default system call transport and `io_uring`.

When deciding whether or not to use `io_uring`, the HSL backend currently considers the size of the collection, target file system, and whether write operations are involved. We do not use cache occupancy as the `cachestat()` system call works on contiguous ranges; scattered reads or writes would require multiple calls to determine cache occupancy increasing overheads especially for cached operations. As using `io_uring` requires one extra system call, the collection has to be larger than one entry. While there is no limit on the size of a collection, `io_uring` has a limit of 32K entries. HSL initializes the ring with 32K entries and operates on larger collections in chunks. Additionally, the ring uses registered file descriptors to avoid descriptor lookups for every operation.

For read operations, we always prefer the `io_uring` backend if the number of chunks is greater than or equal to 8. While uncached reads can still benefit from device parallelism even if the number of chunks is less than 8, it would hurt applications that perform few small cached reads. The speedup is more meaningful for larger number of chunks.

Regular writes that do not bypass the page cache through `DIRECT_IO` always write to the page cache, without any device io. However, the same logic we applied for reads does not apply entirely here. We benchmarked buffered writes through `io_uring` and found them to always be slower than individual system calls for each entry for ext4. Even though system

call transitions and descriptor lookups are minimized, writes to ext4 are processed through io workers and not executed on the ring *inline*. These writes are offloaded to a task queue—a large and unnecessary overhead. Future versions of `io_uring` may support flags to force inline execution. However, until then, we avoid using `io_uring` for writes to ext4.

Using the same benchmark, we observed that XFS and Btrfs can benefit from io_uring but only for small chunk sizes (4K or less) and a large number of chunks. Using io_uring on XFS is only beneficial if the number of chunks are greater than 1024; 16K for Btrfs. Additionally, the io_uring operations must be chained (ordered) and XFS requires a ring with the `SINGLE_ISSUER` and `DEFERRED_TASK` flags to avoid slowdowns. We maintain two separate rings: one with the flags for XFS and the other for Btrfs and read operations.

6 Evaluation

We now evaluate the benefits of using HSL. Specifically, attaining our design goal of portable efficiency—generating correct and efficient code for a given system.

We begin with a correctness evaluation (§6.1) to test if HSL takes into account the individual failure characteristics of each file system, thereby minimizing cases of data loss and corruption. We then proceed to evaluate the performance aspects of HSL (§6.2) to see if it uses the best available implementation for different file systems; especially when there is no single fast solution for all systems.

6.1 Correctness Evaluation

In this section, we focus on the atomicity and durability guarantees provided by the applications using the `REDO_LOG` and `MANIFEST` verbs. We begin with a description of a general methodology we follow when evaluating both classes of applications, followed by specifics and findings for each.

6.1.1 General Fault-Injection Methodology

As the majority of this section deals with correctness in the presence of failures, we describe the common fault-injection method used in our

experiments; specifics are mentioned in individual experiments. We run our experiments on Ubuntu 18.04 with Linux Kernel version 5.2.11[1].

For any application (using HSL or applications we compare against), we construct a pre-workload, workload, and post-workload script. The pre-workload script initializes state for the application workload; it may involve running the application to do so (e.g., creating a database table and populating some rows). The workload script performs the actual operations we wish to evaluate. For example, modifying a few entries. The post-workload script inspects the application state. In the same example, reading all rows and comparing against initialized state. The details of each script are mentioned in their respective experiment.

Our goal is to inject failures at the block or sector level of storage devices. However, using a fault-injection tool such as dm-loki (§3.1.1.2) is insufficient for deterministic fault injection. While applications may write the same content to the same files as per the pre-workload and workload scripts, the blocks allocated by the file system may be different. Similarly, the directory entries for the files an application creates may not reside on the same directory data blocks. Instead, we use CuttleFS (§3.2.1)—a FUSE file system designed to inject faults deterministically.

CuttleFS can be configured to fail the i^{th} write to a device block or sector associated with 1) a specific offset of a file or 2) a directory entry. CuttleFS can also be configured to emulate the behavior of specific file systems after a device block or sector write failure. The exact behavior and file systems are described in each experiment.

We start by mounting CuttleFS in *trace* mode, to log all modifications and accesses by the application. We run the three workload scripts, collect a trace of all device writes (and associated file offsets or directory entries) during the workload phase, and verify that the trace output is deterministic. We then repeat the experiment multiple times, each time failing the

[1]While kernel version should not make a difference when emulating behavior, the behavior we emulate was studied on systems belonging to that version.

i^{th} device write. To do so, each experiment first runs the pre-workload script and then changes CuttleFS to *fault* mode with a specific file-system behavior, configuring it to fail the i^{th} device write (given as a file offset or directory entry). We then run the workload script in this fault mode. Finally, we disable fault mode and run the post-workload script. As there may be changes in the execution environment such as pages being evicted from the page cache, we repeat the same set of experiments to account for such behavior. The exact changes in the execution environment are experiment specific and described in their respective sections.

If the application is notified of the failure, it may choose to ignore it, handle it by attempting to fix and continue, or crash. Irrespective of the decision, if the workload receives a successful return code from the application, the post-workload must see the new state. If the workload receives a failure return code, the post-workload must see the old state. The above two scenarios are the only valid outcomes; the rest are invalid outcomes that can be categorized as follows:

Old Value. When the workload receives a successful return code but post-workload sees the old state (data loss).

False Failure. The workload receives a failure return code but the post-workload sees new state. While seemingly innocuous, applications that do not perform idempotent operations and choose to retry failed operations can work incorrectly. For example, decrementing a particular value can result in a double decrement.

Corruption. Irrespective of the return code, if the post-workload does not see new state or old state, we term it as a corruption. This could be due to missing keys, missing values, newly found keys or values that were never part of the pre-workload and workload scripts, or the inability to access the database altogether.

6.1.2 Evaluating `REDO_LOG`

In this section, we evaluate physical redo logging. We ask the question: Does HSL's `REDO_LOG` lead to invalid outcomes? We build our own application that uses `REDO_LOG` and compare against two applications that use physical redo logging: SQLite configured in write-ahead-log (WAL) mode, and PostgreSQL's default configuration.

Methodology

We focus on injecting data-block failures when applications append to the log[2]. We configure CuttleFS to emulate ext4 (ordered and data mode), XFS, and Btrfs. Their reactions to data-block write failures are as follows:

Ext4 in ordered mode and XFS behave the same. They both mark the dirty pages clean but the content in the page cache differs from what is on disk; the page cache contains the latest write. Both file systems also respond to the failure immediately by returning a failed `fsync()` return code to the caller.

Ext4 in data mode works similarly in terms of marking the page clean and containing the latest write in the page cache. However, as the file system puts data into the journal first (and we do not inject journal block failures), the first `fsync()` succeeds but the *next* one fails.

Btrfs also marks the page clean but reverts the page cache contents so that it matches the disk. It also responds to the failure immediately by returning a failed `fsync()` return code to the caller.

After a fault, we emulate two execution environments. The first evicts clean pages from the page cache while the second retains them.

Applications and Workloads. For SQLite and PostgreSQL, we initialize the database and create a single two-column table (for keys and values)

[2]Other block failures include journal and metadata block failures. However, journal failures lead to file-system unavailability and metadata failures do not result in `fsync()` failures (§3.1.2).

and populate a few rows in the pre-workload script. We run two sets of experiments corresponding to two different workload scripts; the first inserts a new key and value, while the second updates an existing key's value. The post-workload script dumps all keys and values. Using the workload return code and post-workload output, we classify the outcome as valid or invalid using the previously mentioned categories.

To evaluate HSL's `REDO_LOG`, we create an application (Hslog) that uses HSL to update `N` chunks of a file within a single transaction. While not a database, Hslog exercises a fundamental building block in databases—atomically updating a file. Applications like SQLite and PostgreSQL use data structures such as on-disk B-Trees for each table. Updating or inserting rows involves modifying multiple B-Tree nodes and failure to persist all nodes can result in invalid outcomes. Like B-Tree nodes, our application updates chunks in a file. In SQLite and PostgreSQL, failure to persist all nodes *can* result in invalid outcomes, specifically the ones a post-workload cares about—ensuring keys and values are as intended. There could be cases where a B-Tree node failure did not result in an invalid outcome because of how the application interprets the data (e.g., redundancy). Hslog has a stronger constraint—every chunk (B-Tree node) must be persisted or every chunk must be reverted depending on success or failure.

For Hslog, the pre-workload script initializes a file with a few chunks. We run four different sets of experiments corresponding to different workloads that differ in number of transactions and failure handling. The first two workloads attempt to update `N` chunks in a single transaction and can either retry or crash on transaction failure. The next two workloads attempt two transactions, i.e., updating the same `N` chunks again in a different transaction and also differ in failure handling like above. The post-workload script examines the values of each chunk.

	A=KeepGoing	A=Restart
BC=Keep	\	/
BC=Evict	|	–

ext4o,xfs = { clean, differs, immediate } ext4d = { clean, differs, next `fsync` } btrfs = { clean, matches, immediate }

Applications	OV	FF	CC	OV	FF	CC	OV	FF	CC
SQLite (WAL)		/				–			
PostgreSQL (Default)		≠		–		–			
HSL `REDO_LOG` App (Hslog)									

Table 6.1: **Findings for Physical Redo-Logging Applications on `fsync()` Failure:** The table lists the different types of errors that manifest for applications when `fsync()` fails due to data-block write fault. All errors shown here are invalid outcomes, unwanted by the applications. The errors OV, FF, and CC stand for Old Value, False Failure, and Corruptions described in §6.1.2. We group columns according to file-system post-failure characteristics and mention the characteristics above each column. For example, on data-block write failure, both ext4 ordered and XFS (ext4o,xfs) mark the dirty page **clean**, the page **differs** in in-memory and on-disk content, and the `fsync` failure is reported **immediately**. For each application, we describe when the error manifests, in terms of execution environment factors. Applications may choose to keep continuing or restart, and the page cache (buffer cache) may or may not evict clean pages. For example, FalseFailures manifest in SQLite in the first group (ext4-ordered,xfs) only on (A)App=Restart,(BC)BufferCache=Keep. However, in PostgreSQL, the error manifests both on App=Restart,BufferCache=Keep and App=Restart,BufferCache=Evict, depicted as a combination of the two symbols.

		ext4o,xfs = { clean, differs, immediate }				ext4d = { clean, differs, next `fsync` }				btrfs = { clean, matches, immediate }			
		Valid Outcomes				**Valid Outcomes**				**Valid Outcomes**			
#tx	error strategy	Total	rc=0 New Value	rc=-1 Old Value	#Invalid	Total	rc=0 New Value	rc=-1 Old Value	#Invalid	Total	rc=0 New Value	rc=-1 Old Value	#Invalid
1	crash	40	4	36	0	40	4	36	0	40	4	36	0
	retry	40	32	8	0	40	32	8	0	40	32	8	0
2	crash	68	4	64	0	68	4	64	0	68	4	64	0
	retry	68	60	8	0	68	60	8	0	68	60	8	0

Table 6.2: **Findings for Hslog on `fsync()` Failure:** The table shows the number of valid and invalid outcomes that manifest in Hslog when `fsync()` fails due to data-block write fault. We group columns according to file-system post-failure characteristics and mention the characteristics above each column, similar to Table 6.1. Each row corresponds to a different workload which depends on the number of transactions and the error-handling strategy (crash or retry).

Findings

We run each application according to the workload scripts on CuttleFS and present our findings across different file-system post-failure behaviors in Table 6.1. For all applications, we observe no invalid outcomes on Btrfs as it reverts page cache contents and reports the error immediately. For the rest, we go through the failures on a per-application basis. The findings for SQLite and PostgreSQL are taken from Section 3.2.3 and Table 3.2, but described here again.

SQLite when configured in wal mode writes the modified b-tree pages to its write-ahead log. On ext4 ordered mode and XFS, it exhibits False Failures on App=Restart with BufferCache=Keep; i.e. when SQLite restarts but the page cache contents are not evicted. On block failure, SQLite is notified of the `fsync()` failure and returns a failure to the user (expected). But when SQLite restarts, it recovers from the log which has the new contents in the page cache, resulting in the new state; an invalid outcome.

On ext4 data mode, there is no false failure as the `fsync()` failure is not reported immediately. However, during a checkpoint operation, SQLite writes the contents from its write-ahead log to the main database. As data-block failure is not reported immediately, SQLite proceeds to truncate the log. When the pages are evicted from the cache and SQLite restarts, the latest content is lost which leads to corruption.

PostgreSQL in default configuration also uses a write-ahead log like SQLite. On ext4 ordered mode and XFS, it exhibits False Failures for the same reasons as SQLite. Additionally, it also exhibits False Failures when pages are evicted (BufferCache=Evict). As mentioned previously, not all data-block failures can lead to invalid outcomes. Depending on which block experiences the fault, PostgreSQL either accepts or rejects the log entry.

On ext4 data mode, PostgreSQL exhibits Old Value and Corruption errors. As PostgreSQL maintains user-space buffers, these errors man-

ifest only on BufferCache=Evict with App=Restart. Similar to SQLite, PostgreSQL's checkpoint operation assumes successful `fsync()` due to delayed error reporting and the log is truncated. When PostgreSQL restarts, it rebuilds its user-space buffers from the on-disk database which contain the old contents (as latest content has been evicted). Unlike SQLite, sometimes a value can occupy a whole page leading to the Old Value error.

Hslog does not have any invalid outcomes. When run on ext4 data mode, Hslog issues two `fsync()` system calls to force data blocks to be written to their eventual location, thereby also providing immediate error notification. In all file systems except Btrfs, as the page cache contains the latest contents even after failure, Hslog reverts the content of specific blocks—the chunk descriptors (§5.2.1), ensuring stable data is used to proceed further or on recovery. As the descriptor blocks are never greater than the underlying device's sector that provides atomicity, we do not encounter False Failures.

Table 6.2 provides more details on experiments with Hslog. We run a total of 648 experiments divided based on workload (number of transactions and error handling strategy) and file-system failure characteristics. In all cases, we only observe valid outcomes where a successful run results in latest state, and a failed run results in old (or previously stable) state. We describe them below.

On tx=1 and a error handling strategy to crash, Hslog immediately crashes upon receiving the failed returncode from HSL. The only valid outcomes with new value are those where no faults were injected. The multiple runs (4) were for different configurations deciding whether or not and when to evict the page cache. The remaining valid outcomes, where faults are injected, all result in a failure and retain old state. With a retry error handling strategy, the number of valid new value states increase as the retry succeeds (when the faulty block is no longer faulty). There are a few (8) valid old value states however, due to reaching a maximum retry limit (when the faulty block remains faulty). We observe the same

Implementation	#Writer States	#Reader States	#Total	#Valid	#Invalid
HSL Manifests v1	9	8	24310	9285	15025
HSL Manifests v2	8	7	6435	6435	0

Table 6.3: **Findings for Evaluating Atomicity on HSL Manifest:** The table shows the number of valid and invalid outcomes for two HSL `MANIFEST` implementation versions.

trends for tx=2, but with more experiments due to the increased data block writes for two transactions.

To conclude, Hslog uses runtime knowledge of file-system characteristics to avoid the invalid outcomes observed in SQLite and PostgreSQL.

6.1.3 Evaluating `MANIFEST`

In this section, we evaluate the atomicity and durability guarantees of `MANIFEST` implementations. We ask the question: can HSL's `MANIFEST` lead to invalid outcomes? We perform two sets of experiments. The first answers the above question in the absence of any failures, to identify and correct invalid outcomes with simultaneous readers and writers. The second answers the same question in the presence of directory data-block failures which cause directory entries to silently disappear on ext4 (§3.1.2.1).

6.1.3.1 Evaluating `MANIFEST` Atomicity

In the absence of failures, but with simultaneous readers and writers, our implementation of `MANIFEST` must provide the user with an atomic set of files (and their contents) as per the contents of the `MANIFEST`. In this section we evaluate whether the above characteristic holds true on an application that uses HSL's `MANIFEST` implementation. As part of this evaluation, we identified issues with HSL's implementation and **fixed** them, resulting in two versions; we evaluate each of them.

Methodology

While no faults are injected, we reuse some of the common methodology described previously; specifically, the use of pre-workload, workload, and the terminology of valid and invalid outcomes.

The pre-workload script creates an initial set of files and a manifest. The workload script runs a reader and writer application concurrently. The reader application opens the `MANIFEST`, and then reads each file it refers to. The writer updates the `MANIFEST` to include an additional entry, and also replaces an existing entry. The reader within the workload script replaces post-workload as it dumps the contents it reads which is then analyzed to classify the outcome.

To exhaustively explore all possible concurrency issues, we inject *custom breakpoints* at locations inside the reader and writer applications, and inside HSL's `MANIFEST` implementation. We limit the breakpoints to locations that interact with HSL or with the underlying file system as those are the places where changes become visible. We first run the experiment in *trace* mode where readers and writers print the location of the breakpoint and continue.

From the given breakpoint traces, we generate all valid interspersed reader-writer combinations. We then recompile readers and writers to pause themselves by raising a `SIGSTOP` at the breakpoint. We build an orchestrator script that issues `SIGCONT` to the right process based on the given combination.

We re-run the experiment multiple times, with the workload script using one of the reader-writer combinations each time. We then analyze the output of the reader to identify if it read a partial or consistent view of the manifest.

Findings

We created readers and writers for both versions of HSL's `MANIFEST` implementation and present our findings in Table 6.3. In the first version, the

	Inode Dealloc.	Total	Valid Outcome rc=0 New	Valid Outcome rc=-1 Old	Invalid Outcome rc=0 Old	Invalid Outcome rc=0 Corruption
LevelDB Manifests	Immediate	1728	96	0	96	1536
	Delayed	2816	640	0	640	1536
HSL Manifests	Immediate	448	56	392	0	0
	Delayed	448	56	392	0	0

Table 6.4: **Findings for Evaluating Durability on HSL Manifest:** The table shows the number of valid and invalid outcomes on two applications exercising the use of `MANIFEST` files: LevelDB and Hslmanifest. For each application we provide two results depending on whether an unlinked inode is deallocated immediately or later. We group columns based on valid and invalid outcomes as described in 6.1.1.

number of breakpoints in writers and readers were 9 and 8, resulting in 24310 possible combinations out of which only 9285 were valid outcomes. The remaining 15025 were invalid due to a single issue: the reader tries to read a file that does not exist any more.

We fixed the above problem by ensuring the reader always keeps open file descriptors to each of the files the manifest refers to. The reader does so inside a critical section when reading the manifest itself (§5.2.2).

In the second (fixed) version, the number of breakpoints reduce by one in both readers and writers as they now appear within the critical sections. In the resulting 6435 combinations, we observe no invalid outcomes.

6.1.3.2 Evaluating `MANIFEST` Durability

Our file-system study showed that directory entries can disappear on ext4 (§3.1.2.1). In this section, we evaluate whether HSL's `MANIFEST` implementation (the second version without any invalid atomicity outcomes from the previous section) is resilient to directory data block failures. As we did not study the repercussions of missing directory entries on appli-

cations earlier, we also provide an evaluation of LevelDB under the same failures.

Methodology

We focus on injecting block failures in directory data-blocks, i.e., blocks belonging to a directory that contains the name-to-inode mapping. To recapitulate, ext4 writes directory data blocks during periodic checkpoint operations. If the directory data-block write fails, ext4 marks the page as `!uptodate` which causes the stale block to be re-read from disk; the stale contents do not have the new entry. We configure CuttleFS to emulate this behavior. In *trace* mode, CuttleFS logs all directory entry updates i.e., adding new files, removing existing files, and replacing (renaming) files. In *fault* mode, during checkpointing, CuttleFS skips one or more of the updates.

As checkpointing happens periodically, we run multiple experiments triggering a checkpoint at different locations of the workloads. In addition to checkpointing, we emulate two execution environments. The first deallocates inodes immediately, representing behavior for files that do not have any open file descriptors or other hard links. The second delays inode deallocation as one of the above conditions may hold true. For example, an external process may snapshot the system (increasing the hard links of many files), or even just invoke `ls -l` at that instance which results in an open file descriptor.

We now describe the pre-workload, workload, and post-workload scripts for both applications: LevelDB and Hslmanifest.

LevelDB. The pre-workload script initializes a LevelDB database with a few (1000) key-value pairs. The workload script updates some (100) key-value pairs, followed by a compaction. The post-workload script re-opens the database and dumps all key-value pairs.

Hslmanifest. To evaluate HSL's `MANIFEST`, we create an application (Hslmanifest) that uses HSL's `MANIFEST` verbs to read and update a manifest. While

not a key-value store like LevelDB, Hslmanifest exercises the properties LevelDB requires from manifests—viewing a consistent set of immutable files.

The pre-workload script creates a set of files and initializes a `MANIFEST` with references to them. As described in Section 5.2.2, applications associate some key with the file reference. For example, LevelDB lists the set of Sorted String Table files (SSTs) that make up each level. In Hslmanifest, a key such as `L3IDX5` could refer to the fifth table in level 3. The pre-workload script initializes the manifest with three references: `k1:file1`, `k2:file2`, and `k3:file3`.

The workload script updates the `MANIFEST` by replacing a file for an existing reference (`k1:file1`→`k1:file100`), adding a new reference (`+k20:file20`), and removing an existing reference (`-k2:file2`).

The post-workload script reads the `MANIFEST` and the files it references. Using the above examples, a successful update by the workload script should result in a `MANIFEST` with references: `k1:file100`, `k3:file3`, and `k20:file20`. Additionally, we ensure the contents of each file are unique, allowing us to verify that the data corresponds to the correct file.

Similar to evaluation of `REDO_LOG`s, Hslmanifest has a strong constraint. While LevelDB identifies valid and invalid outcomes by analyzing key-value outputs, the post-workload in Hslmanifest examines the `MANIFEST` and its references directly.

Findings

We run both applications according to the workload scripts on CuttleFS and present our findings in Table 6.4. For both applications, we provide two sets of results corresponding to whether inodes were deallocated immediately or not. We walk through the results for both, starting with LevelDB.

LevelDB. For both immediate and delayed inode allocations, we observe some (96 and 640) valid outcomes, some (96 and 640) invalid outcomes as

old values, and many (1536) invalid outcomes as corruptions. As the fault is never detected and reported, LevelDB is never aware of any error and exits successfully. Therefore, there are no valid outcomes where LevelDB fails (rc=-1) and reverts to the old state.

The number of valid and invalid old value outcomes jump from 96 to 640 due to the change in environment, i.e., delaying inode deallocation that leads to more experiments. The old value errors stem from how LevelDB updates its `MANIFEST` file. It maintains a text file named CURRENT that contains the filename of the latest `MANIFEST` (similar to a softlink but implemented as a regular file). When the manifest has to be updated, LevelDB does not modify either of the files. It creates a new `MANIFEST.1` and a new `CURRENT.1` that contains the filename `MANIFEST.1`. Finally, it renames the `CURRENT.1` to `CURRENT`, so new readers atomically view a new manifest. Unfortunately, if the rename fails (as caught by our experiment), it leads to invalid outcomes as old values.

Regardless of the change in environment for inode deallocation, the number of corruptions are the same (1536). LevelDB failed to open the database in the post-workload script, logging the error: "CURRENT points to a non-existent file". The errors arise from two situations, explained using the same example above as follows:

1. The rename of `CURRENT.1` to `CURRENT` succeeded, but the directory entry `MANIFEST.1` disappeared.

2. The rename of `CURRENT.1` failed (i.e., `CURRENT.1` directory entry disappeared) but the previous `MANIFEST` was successfully deleted while cleaning up old files.

Hslmanifest. We do not observe any invalid outcomes in Hslmanifest. Out of 448 experiments, 392 fail and revert to old state, and a small amount (56) succeed—both valid outcomes.

HSL accomplishes this by forcing a checkpoint as part of the `MANIFEST` update protocol and re-reading directory entries to ensure they are present before the final update. Any missing directory entries are detected and reported as errors immediately. In such situations (rc=-1), Hslmanifest reverts back to the old state—a valid outcome. Additionally, the final update is not a rename like LevelDB does with `CURRENT` and `CURRENT.1`. Hslmanifest performs an in-place write to its equivalent of `CURRENT` within a critical section and if required, reverts page cache contents on `fsync` failure similar to `REDO_LOG`.

To conclude, while Hslmanifest has strict constraints and fails due to injected faults, it ensures that—unlike LevelDB—any reported success truly contains the new state.

6.2 Performance Evaluation

In this section we evaluate the performance benefits of HSL. As HSL is responsible for choosing the best underlying interface for performing a task, we first run experiments to showcase the benefits of each HSL feature (§6.2.2). Next, we perform case studies on real applications that interact with the file system (§6.2.3), describing the steps to modify them and the performance implications of using HSL.

6.2.1 Methodology and Experimental Setup

To showcase the benefits of each HSL feature, we have at least two implementations: one using the HSL feature and another that represents the usual method. When there are multiple solutions, we add them to the implementations we evaluate. As each HSL feature shines in different workloads, we construct different workloads to evaluate them independently; they are described when presenting findings for the feature. Our

goal when evaluating HSL features is to highlight sensitivity to the runtime environment, and show that HSL takes advantage of that sensitivity.

When performing case studies on applications, we evaluate the unmodified application against a version that uses HSL. We evaluate the coreutils `cp` utility, a WAV file audio trimmer, a Merkle proof generator, LMDB (a key-value store), and SQLite (a relational database). Similar to above, due to differences in domain, we construct and describe the workloads in their respective sections. Our goal in evaluating these applications is to assess the performance benefits to real-world applications and the effort it takes to use HSL in them.

Experimental Setup: We run our experiments on Cloudlab [28] c220g2 machines running Linux v6.5.7 with hyperthreading disabled and use the `performance` cpu governor. For workloads that run natively, we evaluate against the ext4, XFS, and Btrfs file systems with their default mount options [72, 98, 113]. For experiments that involve cloud providers, we use s3fs for Amazon AWS S3 [52, 102], and gcsfuse for Google Cloud Storage [45, 46].

6.2.2 Evaluating HSL Features

To showcase the benefits of each HSL feature, we run experiments that answer the following questions:

§6.2.2.1 Does HSL always choose the fastest copy interface?

§6.2.2.2 Can HSL improve performance of file concatenation for remote file systems?

§6.2.2.3 Are there any (and does HSL avoid) cases where prefetching (`fadvise()`) harms performance?

§6.2.2.4 Is using HSL's filter directive abstraction beneficial to applications?

§6.2.2.5 Are there any benefits to using modern transports like io_uring, and having HSL decide when to use it?

6.2.2.1 COPY Verb

As described in §5.1.4.1, HSL provides a COPY verb to copy data between regular files using the fastest among available interfaces. Here, we run experiments to see whether HSL always chooses the best interface. We ask and answer two questions: 1) If there is always a single best interface, does HSL use it? and 2) If the interface depends on runtime parameters, what are those parameters, and does HSL choose the right interface accordingly?

We use two distinct copy workloads as described previously (§4.3.4.2): a large single-range copy and small multi-range copies. In both cases, we focus on copies where source and destination files are on the same file system[3]. We run our experiments on three different file systems (ext4, XFS, and Btrfs) and vary the length of the contiguous portion. We repeat the same experiment in two settings: where files are cached or not. We now describe experiment details and our findings for each workload separately below.

Large Single-Range Copy

In addition to common experiment details above, we add another parameter: whether the destination file exists or not. We create applications that perform the single-range copy using existing interfaces: `read+write`, `sendfile`, and `copy_file_range`. As `read+write` requires reading into a buffer and writing that buffer, we vary the buffer sizes being used. We compare these three interfaces against an application that uses HSL to perform the copy.

Findings. We run experiments varying the `read+write` buffer from 4KB to 1MB, and the range length from 128KB to 100MB, and measure the latency

[3]The case study of coreutils `cp` evaluates copy performance on different file systems.

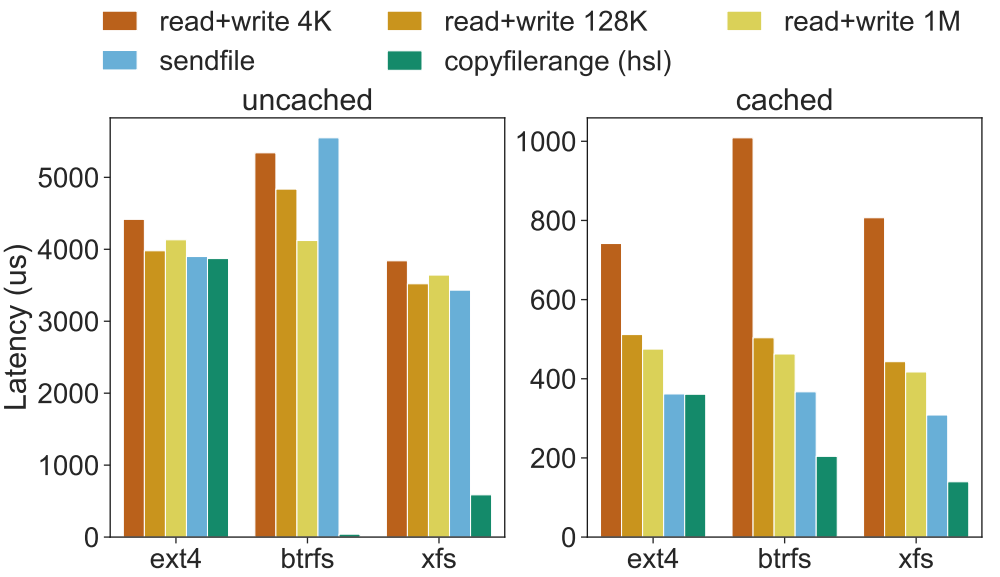

Figure 6.1: **Evaluating implementations to copy 1MB to an existing file on different file systems:** The figure shows the per page latency in microseconds (y-axis) for different local file systems (x-axis) performing a copy with different implementations. The two graphs correspond to whether the files were in the page cache (right: cached) or not (left: uncached).

in microseconds to perform the copy. Figures 6.1 and 6.4 document our findings.

As observed in Figure 6.1, when copying 1MB to a destination file that exists, `copy_file_range` and HSL (which always uses `copy_file_range` for large single-range copies) always have the lowest latency regardless of whether the files are cached or not.

Figure 6.2 presents our findings on XFS as a heatmap for all range lengths, both when destination exists or not, and when files are cached or not. Every cell in the heatmap represents the speedup of that implementation (column) over `read+write` with a 4KB buffer. The `read+write` with larger buffer sizes show slight speedups due to a increased buffers that reduce number of system calls. The `sendfile` shows slight speedup due

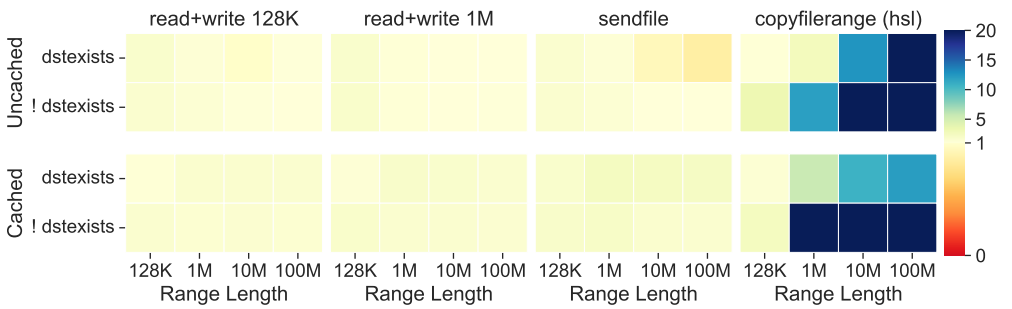

Figure 6.2: **Evaluating implementations of large single-range copies on XFS:** The figure shows the speedups of each copy implementation (four columns) relative to the `read+write` implementation using a 4KB buffer. The fourth column represents both `copy_file_range` and HSL. The two rows correspond to whether the files are uncached (top) or cached (bottom). Inside each heatmap, the y-axis represents whether the destination file exists or not (dstexists and !dstexists). The x-axis varies the length of the range being copied.

to avoiding copies to user-space buffers. However, the `copy_file_range` and HSL implementations always have the highest speedups.

We observe the same trend on ext4 and Btrfs. When copying a large single range, the `copy_file_range` system call performs a copy-on-write of the extents on XFS and Btrfs, minimizing data copy. On ext4, the performance matches `sendfile`.

For large single-range copies, there is indeed a single best interface: `copy_file_range`, and HSL uses it to perform the copy.

Small Multi-Range Copy

In addition to common experiment details, as the copy involves multiple ranges we vary the number of ranges involved in the copy. We limit experiments to source and destination offsets that exist in both files as most applications use this pattern to overwrite ranges, or grow the destination file to the maximum range before performing the copy. Similar to single-range copy, we compare HSL against `read+write`, `sendfile`, and `copy_file_range`. Unlike single-range copy, the `read+write` appli-

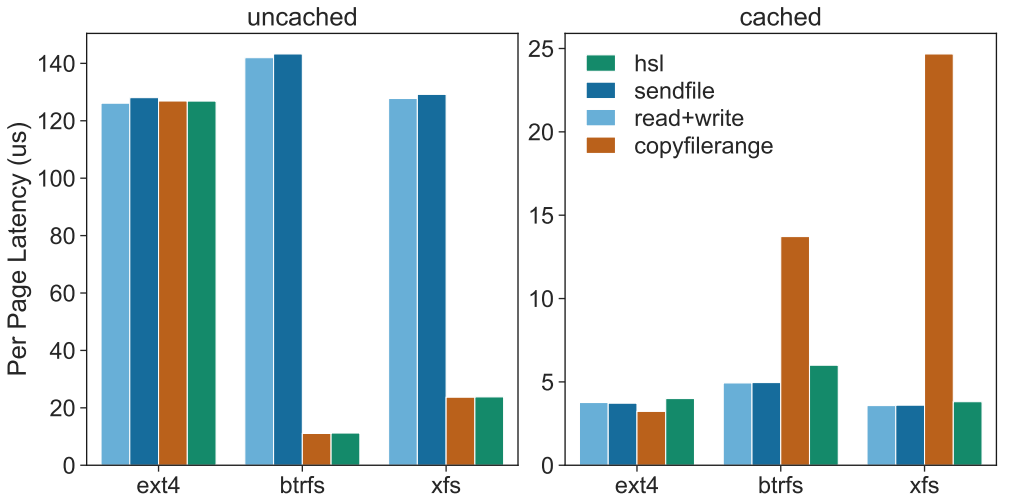

Figure 6.3: **Evaluating implementations to copy 1000 4KB pages on different file systems:** The figure shows the per page latency in microseconds (y-axis) for different local file systems (x-axis) performing a copy with different implementations. The two graphs correspond to whether the files were in the page cache (right: cached) or not (left: uncached).

cation always uses a buffer equal to the size of the range, requiring one `read+write` per range.

Findings. We run experiments varying the range length from 4KB to 64KB, the number of ranges from 100 to 1000, and measure the latency in microseconds to perform the copy. Figures 6.3 and 6.4 document our findings.

In Figure 6.3, we compare the performance of all the implementations across the three file systems in copying a 1000 4KB ranges. While `read+write` and `sendfile` behave similarly in both the uncached and cached settings (all file systems), `copy_file_range` does not (XFS, Btrfs). In the uncached setting, `copy_file_range` is much faster for XFS and Btrfs as it uses copy-on-write on the extents. However, in the cached setting, it is much faster to copy the bytes from pages already in the cache than to modify the extent trees. This difference in performance does not occur on

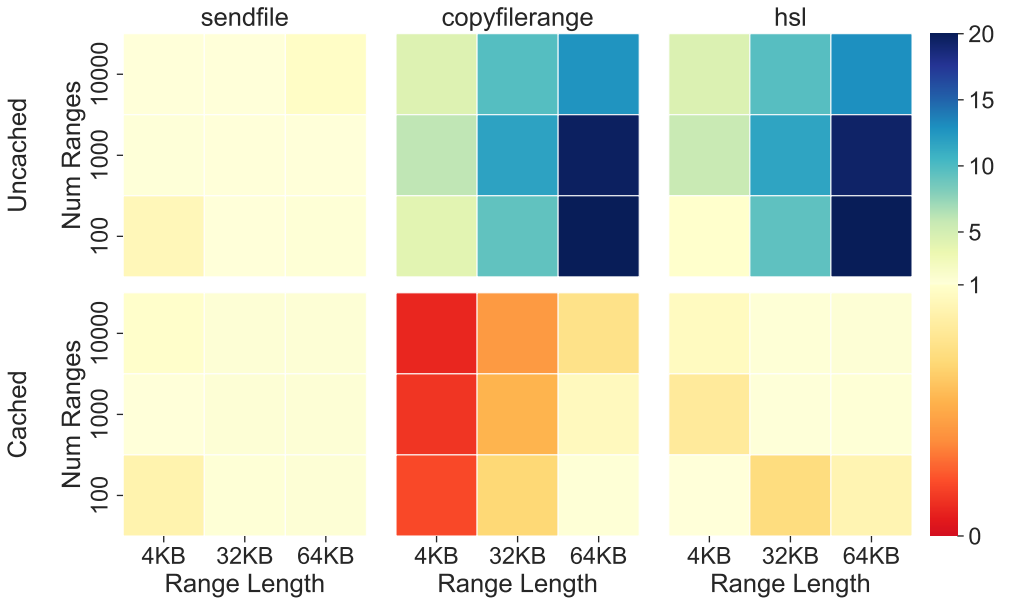

Figure 6.4: **Evaluating implementations of small multi-range copies on XFS:** The figure shows the speedups of each copy implementation (three columns) relative to the `read+write` implementation. The two rows correspond to whether the files are uncached (top) or cached (bottom). Inside each heatmap, the y-axis represents the number of ranges in the copy and the x-axis the length of each range being copied.

ext4 as `copy_file_range` is internally implemented similar to `sendfile`.

HSL matches `copy_file_range` performance in the uncached setting, and nearly matches `sendfile` in the cached setting with a minor overhead due to the extra `cachestat()` system call to gauge cache occupancy.

Figure 6.4 presents our findings on XFS as a heatmap for all range lengths and number of ranges, in both cached and uncached settings. Every cell represents the speedup of the implementation over `read+write`. As observed, `copy_file_range` has high speedups in the uncached setting but performs poorly in the cached setting. HSL obtains similar speedups in the uncached setting but avoids the slowdowns in the cached setting.

We observe the same trend on Btrfs but not on ext4. As ext4 uses

a `sendfile` equivalent implementation for `copy_file_range`, there is no difference in the cached and uncached settings.

For small multi-range copies, there is no single best interface on XFS and Btrfs. Depending on whether files are cached or not, HSL chooses the right interface on these file systems, obtaining speedups and avoiding slowdowns.

6.2.2.2 Niche Middle-end Optimizations: Concatenation

While not part of the standard set of verbs, HSL allows *custom verbs* that can have niche optimizations for specific file systems (§5.1.3.3). Concatenation is one such example, implemented on HSL with the `CONCAT` verb (§5.2.3.2) specifically for the remote file systems: s3fs and gcsfs. Here, we evaluate the performance benefits of using non-portable interfaces through HSL's `CONCAT` implementation over the standard `cat` command-line utility.

We construct a workload that concatenates three equal sized files—all uncached—into one new file. We use the `cat` utility—unmodified—as one method of performing the concatenation. We also build our own concatenation command line utility using HSL (hslcat).

We store the three equal sized files in an AWS S3 bucket and a Google Cloud Storage bucket. We then mount FUSE file systems for these object stores: s3fs and gcsfs respectively, and run both applications on each.

Findings. We run experiments varying the size of the input files and measure the time taken and network usage for the concatenation. Figure 6.5 documents our findings.

For small files (10MB) the difference between applications is negligible. However, as the file size increases, the runtime of the `cat` application grows; ~75 seconds to concatenate 3 500MB files on s3fs and ~58 seconds on gcsfs. This increase in runtime is due to the extra network traffic to perform the concatenation. As seen in the lower graph, 1.5GB (3x500MB) is transferred to concatenate the three existing files. While not shown in

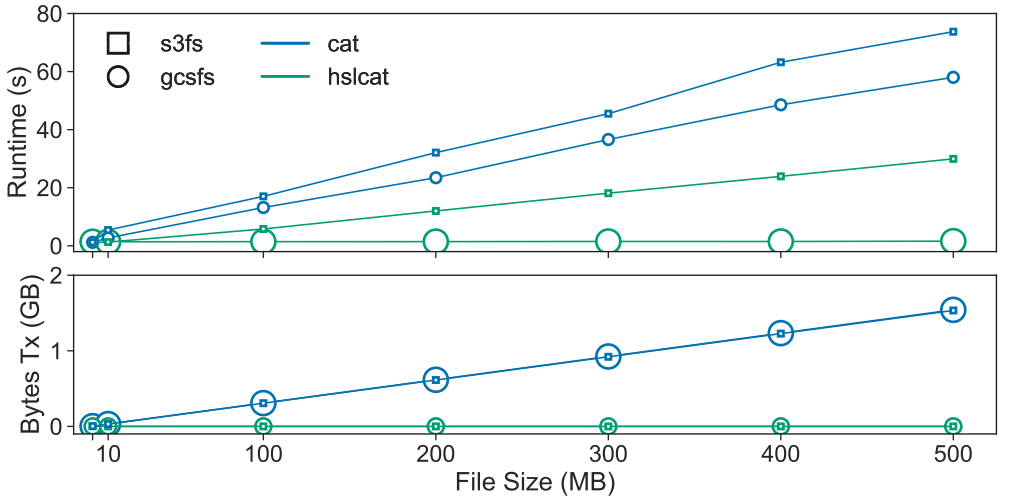

Figure 6.5: **Evaluating file concatenation on remote object storage:** The figure shows the time taken to concatenate three equal sized files into one on S3 and GCS, using cat and HSL. We vary file size on the x-axis, measure runtime in seconds (top y-axis), and measure bytes transferred (bottom y-axis).

the figure, as the files are not cached, 1.5GB (3x500MB) is also received over the network.

As seen in the figure, `hslcat` matches and soon exceeds the performance `cat`; ~29 seconds to concatenate 3 500MB files on s3fs and ~1.5 seconds on gcsfs. The lower graph explains the reason: on both file systems, less than 35KB is transferred or received. HSL identifies the file system and issues network requests to the remote object store directly (§5.2.3.2) as the current FUSE implementations do not support any faster non-standard interfaces. The runtime on gcsfs does not grow as Google Cloud Storage offers a single `compose` API for concatenation. Unfortunately, despite fewer network transfers, multiple network requests (which grow with file size) are made on AWS S3 leading to the increase in run time.

To conclude, implementing `CONCAT` in HSL to detect and use non-portable interfaces significantly reduces runtime and network costs.

6.2.2.3 Hints: Expecting Future Reads

As described in §5.1.2.2, HSL users can provide a `.expect` hint specifying the access pattern (sequential or reverse sequential) or arbitrary offsets. Here, we evaluate how beneficial the use of HSL's `.expect` is over applications that accomplish the same outside of HSL with the `fadvise()` system call. As `.expect` internally uses `fadvise()` selectively, we ask the question: Are there instances where using `fadvise()` hurts performance, and does HSL avoid them?

We construct a workload that repeatedly reads 4KB from a file and then performs computation for a specific duration of time. The file is not present in the page cache. The offset in the file to be read changes based on one of four access patterns: sequential, strided, reverse sequential, and scattered (§4.3.1.2). In sequential and reverse sequential, we increment (or decrement) the offset by 4KB. In strided, we increment the offset by a given stride length (greater than 4KB). In scattered, we select a random (4KB aligned) offset in the file.

We build two applications that can run the above workload. The first always uses `fadvise()`; sequential uses `FADV_SEQUENTIAL` before beginning the read-and-compute loop. The remaining use `FADV_WILLNEED` inside the loop right before the computation, giving the system some time to perform the prefetch before the read. The second application uses HSL with the `.expect` hint.

Findings. We run the workload with different access patterns on the two applications, varying the compute time between reads from 1us to 10ms, and measure the the throughput. We also vary the stride length in the strided access pattern from 8KB to 128KB. Figure 6.6 documents our findings. As computation time increases, the bytes read per second from the file decrease, leading to lower throughput.

We divide the figure into two halves: the top presents results for all the access patterns except small strides of 8KB and 16KB. We observe no

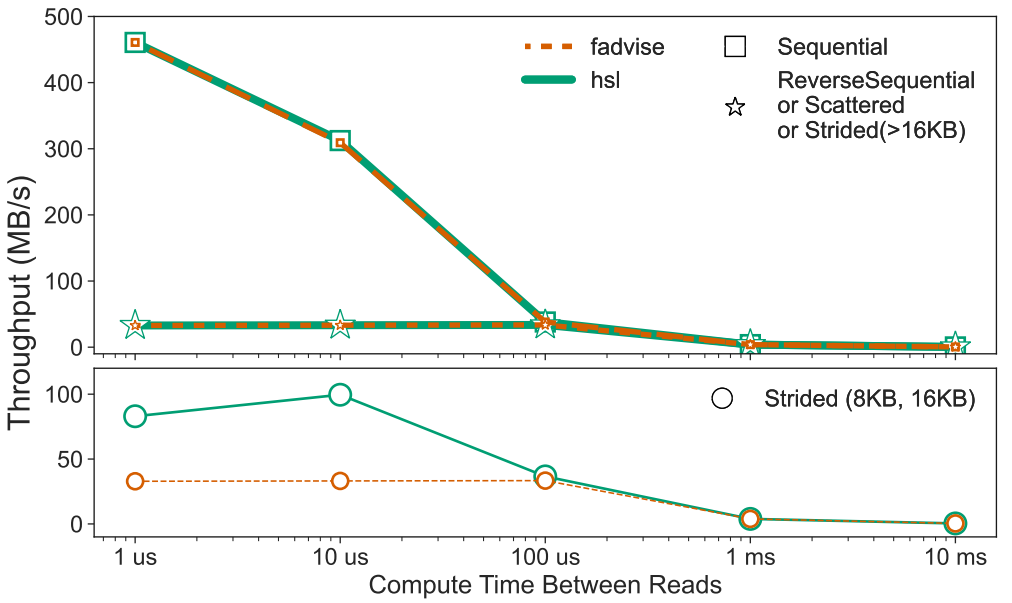

Figure 6.6: **Comparing HSL .expect hint with always issuing `fadvise()`:** The figure shows two graphs divided based on access pattern. In both graphs, the color identifies the application being used: always fadvise, or .expect with hsl. The markers represent access patterns. The top graph contains results for sequential, reverse sequential, scattered, and large strided (more than 16KB strides) access patterns. Sequential uses the square marker while the remaining three have the same results denoted by the star marker. The bottom graph contains results for small strided access patterns (8KB or 16KB) denoted by a circle marker. The y-axis (different for top and bottom) measures the throughput, and the x-axis (shared) varies the compute time between reads.

difference between the two applications. For lower compute time, the sequential access pattern has higher throughput as the device can service those requests faster compared to the other access patterns. At higher compute times, device request latency becomes less significant; the data is present in the page cache before the next read.

The bottom half presents results for small stride (8KB and 16KB) access. At lower compute times, the `fadvise()` implementation has half the throughput of the HSL implementation; ~40 and ~80 MB/s respectively. As HSL does not issue an `fadvise()` for small strides (because the underlying system readahead logic handles this case), it does not interfere or cancel the readahead, leading to higher throughput. As compute time increases, the differences disappear for the same reason stated above.

To conclude, using HSL's `.expect` hint is beneficial when there is small compute between reads. It selectively issues `fadvise()`, avoiding cases which can cause performance degradation.

6.2.2.4 Directives: Filtering

As described in §5.1.2.2, HSL users can perform filtering operations with the `.filter` directive, exposing a uniform interface to applications that perform filtering and choosing a more efficient implementation under the hood. Here, we evaluate whether there is any overhead to using this abstraction and whether there are any benefits to it. We evaluate both of HSL's `.filter` and `.filter_endpoint` directives to identify situations in which they are beneficial.

We construct a large text file of `\n` delimitted records containing keywords that occur at various frequencies throughout the file. We run a workload to find the first N occurrences of a given keyword, starting either at the beginning or end of the file. By changing the keyword in the workload, we change the frequency of matches in the file. An infrequent keyword would require scanning more of the file while a frequent key-

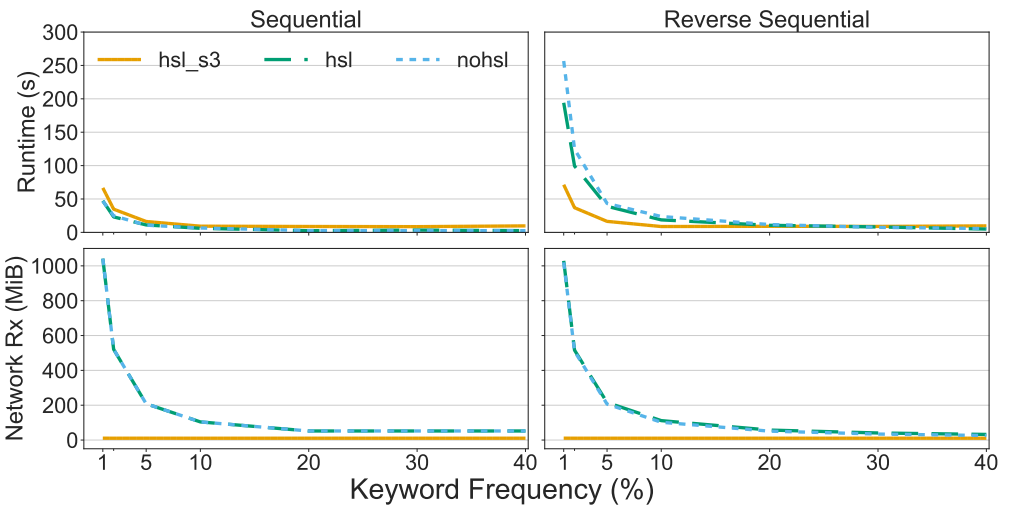

Figure 6.7: **Comparing .filter and .filter_endpoint against a traditional filter workload:** The figure shows four graphs divided into two columns based on access pattern (left:sequential and right:reverse sequential), and into two rows based on metric captured (top: runtime in seconds, bottom: bytes received over the network in MB). The colors of each line represent different applications: filtering without hsl (nohsl), filtering with hsl locally (hsl), filtering with remote compute through aws lambda (hsl_s3). All graphs use the same x-axis which denotes the frequency of the keyword used for the filter workload. The y-axis denotes the metric captured (top:runtime, and bottom: bytes received).

word will have matches immediately. We upload the large text file to AWS S3 and mount the bucket to a directory using s3fs.

We build three applications: nohsl, hsl, and hsl_s3. The first (nohsl) performs the filtering like a regular application that does not use HSL. The second (hsl) uses HSL's `.filter` directive to do the same. The third (hsl_s3) uses `.filter_endpoint` that points to an AWS Lambda function. All three applications share the same command line interface: they take the path to the file and a keyword as input.

Findings. We run the workload on the three applications for both sequential and reverse sequential access patterns. We vary the keyword used (a

frequency from 1 to 40%), and measure the total runtime to obtain the N matches as well as bytes received over the network. Figure 6.7 documents our findings.

In all three applications and both access patterns, runtime and network usage reduce with increased keyword frequency as the first N occurrences are found earlier in the file. Additionally, both hsl and nohsl download the same amount of data as they make the same accesses to s3fs. However, hsl does perform *slightly* faster than nohsl for smaller frequencies due to HSL using `fadvise` internally. Unfortunately, the speedup is not that significant case as s3fs downloads data in large chunks (50MB)

The hsl_s3 application performs worse than the other two in the sequential case because downloading the file sequentially[4]is faster than invoking multiple lambdas. However, hsl_s3 downloads fewer than 10MB of data as opposed to the other two, because it only downloads the lines that contain the keyword. Performance may be better on custom object-storage deployments like Ceph clusters with serverless capabilities as they do not share the same restrictions as AWS Lambda.

For lower keyword frequencies and reverse-sequential accesses, hsl_s3 outperforms the other two as the cost of invoking a lambda that can filter a larger portion closer to storage is faster than downloading and filtering. Similar to the sequential case, it downloads fewer than 10MB, minimizing network egress costs.

To conclude, the `.filter` directive matches performance of existing filtering approaches. The abstraction is beneficial as it allows applications to preserve the same interface but use a different underlying implementation. While `.filter_endpoint` does not always outperform the rest in performance, it always minimizes network traffic which could be a factor for certain users.

[4]Cloudlab has fast network bandwidth.

6.2.2.5 Collections: Scattered Reads and Writes

Collections in HSL (§5.2.3.5) allow applications to provide all arguments that share the same verb—vectored calls, which would otherwise have to be split across multiple system calls. By providing all arguments at once, HSL can choose between the io_uring and native system call transport. Here, we evaluate the advantages of letting HSL determine the transport as opposed to applications fixing to one at compile time. We ask and answer two questions: 1) if there is one superior transport that must always be preferred, does HSL use it? and 2) if it depends on operations and runtime parameters, does HSL change accordingly?

We construct two workloads: scattered reads and scattered writes, where multiple blocks of a file are read or written to. In both cases, we vary the number of blocks being read or written, and the size of each block. The blocks within each scattered read or write operation are selected at random. As reads can be served from the page cache, we run experiments both when the file is in the cache and when it is not.

For scattered writes to files opened in `O_SYNC()` mode, each write is handled synchronously by the device in both cases leading to the same performance regardless of transport. Instead, we focus on buffered writes that go directly to the page cache and only run experiments where the file is cached. As we are aware of poor write performance on io_uring depending on the flags that are passed, we use the flags that yield best performance: default flags for Btrfs, and `SINGLE_ISSUER | DEFERRED_TASK` for XFS. HSL does the same internally (§5.2.3.5).

We build three applications: syscall, io_uring, and hsl. The first two always use the transport they name; syscalls always uses traditional system calls (`pread()` and `pwrite()`) for scattered operations, and io_uring always uses the io_uring API. The last—hsl—uses HSL's `READ` and `WRITE` verbs with unordered collections for scattered reads and writes respectively. We run the workload on all three applications and on three file

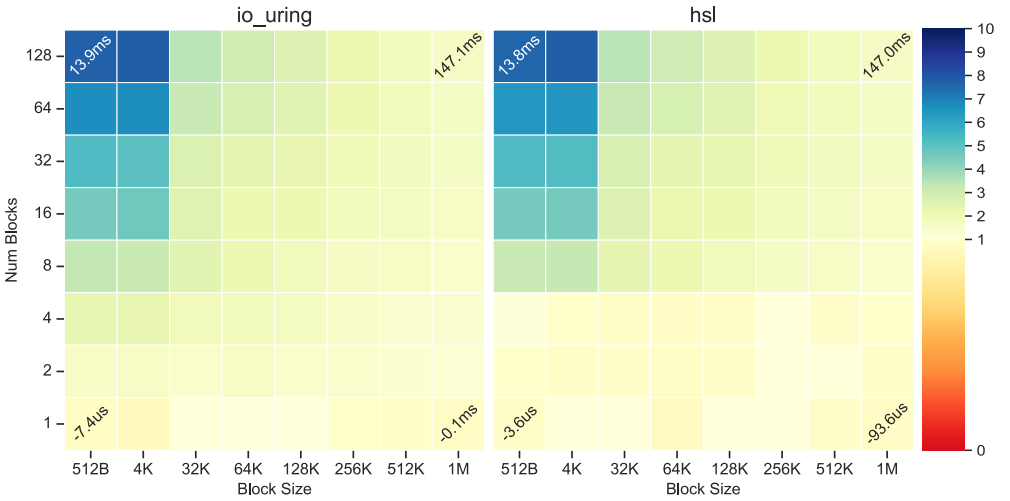

Figure 6.8: **Evaluating scattered uncached reads on ext4:** The figure shows the speedup obtained by two implementations over using traditional `pread()` system calls to perform scattered reads on a file that is not in the page cache. On the left is an implementation that always uses io_uring, while the right uses HSL which *selectively* uses io_uring. In each graph, the x-axis represents the size of every chunk (Block Size) in the scattered read while the y-axis represents the number of chunks (Num Blocks) in the scattered read. The color of each cell represents the speedup (greens and blues) or slowdown (oranges and reds) over the system call implementation. We annotate the boundaries with the difference to the system call implementation: the speedups (positive) or slowdowns (negative) are in microseconds for fewer chunks while in milliseconds for larger number of chunks.

systems: ext4, XFS, and Btrfs.

Scattered Read Findings. We run the scattered read workload on all three applications, varying the number of blocks, the size of each block, the file system, and whether the file is in the page cache. We show the results in terms of speedup relative to the syscall application. Figures 6.8 and 6.9 document our findings.

Figure 6.8 shows the speedup obtained over syscall on ext4, when the file is not cached. We observe similar trends on XFS and Btrfs. On the left,

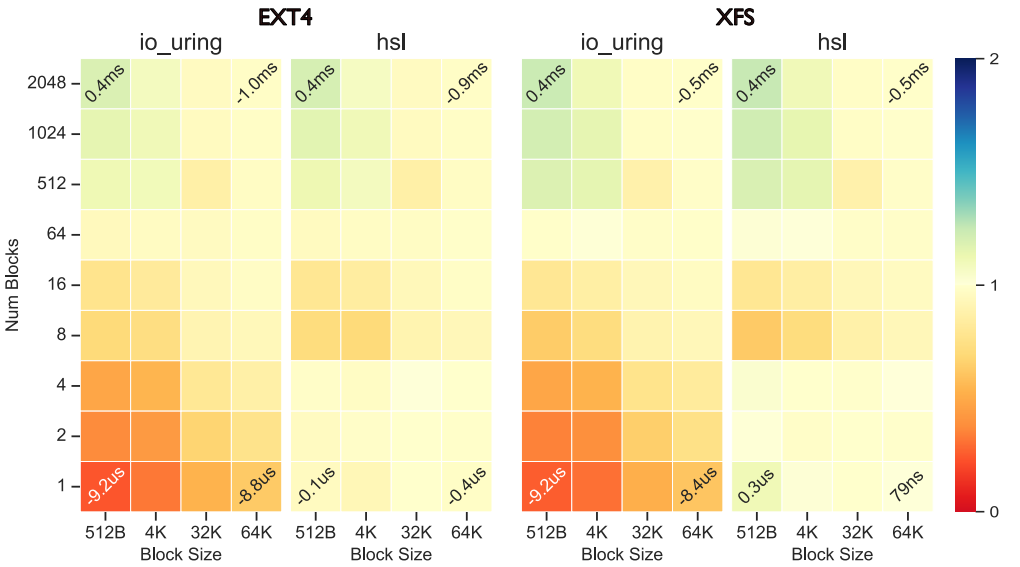

Figure 6.9: **Evaluating scattered cached reads on ext4 and XFS:** The figure shows four heatmaps that identify speedups and slowdowns of two scattered read implementations over traditional `pread()` system calls over a file that is fully in the page cache. Each half represents a different file system: ext4 and XFS respectively, and can be interpreted similar to Figure 6.8. As HSL is aware of poor cached read performance through io_uring, it uses the system call implementation for the smaller number of chunks.

always using io_uring is beneficial if the number of blocks is greater than one as the device's i/o parallelism is exploited. We observe a nearly 7x speedup for small block sizes (512B and 4KB) when the number of blocks exceed 128. Using io_uring for a single block is slower than using system calls due to the overhead of ring execution and no need for i/o parallelism.

Figure 6.9 shows the speedup obtained over syscall on ext4 and XFS, when the file is cached. We observe similar trends on Btrfs. Unlike the uncached case, using io_uring to read cached data is slow as there is no device i/o paralellism to exploit. Smaller benefits such as minimizing system calls (the user-kernel-user boundary crossings), and file-descriptor lookup add up only as the number of blocks increase. We increase the number

of blocks in the experiment from 128 to 2048 to observe the speedup. As block sizes increase, the speedup over system calls reduces as more time is spent in the copy.

In the uncached case, HSL achieves the same speedups as io_uring when the number of blocks are greater than or equal to 8; io_uring outperforms HSL for fewer number of blocks. However, the same setting allows HSL to avoid the slowdowns io_uring faces in the cached setting. HSL cannot determine cache occupancy to change runtime behavior as calling `cachestat()` adds overhead; instead, HSL uses a threshold of 8.

The threshold was chosen such that we minimize slowdowns for small cached operations, but maximize speedups if uncached. When system calls outperform HSL (cached reads of 8-64 blocks), the slowdowns are in microseconds. But in the uncached case, the speedups are in milliseconds. We expect the threshold to change on different systems, and HSL will discover and use those thresholds as part of installation.

Scattered Write Findings. Similar to scattered reads, we run the scattered write workload on all three applications, varying the number of blocks, size of each block, and the file system being run on. We show the results in terms of speedup relative to the syscall application. We omit results on ext4 as io_uring always performs worse than syscall; HSL always uses system calls on ext4 (§5.2.3.5). Figure 6.10 documents our findings for XFS and Btrfs.

Unlike scattered reads, io_uring performs poorly for fewer blocks on XFS and Btrfs. Additionally, the performance crossover point where io_uring outperforms syscalls is different on the two file systems. As seen in the figure, XFS performs better when the number of blocks exceed 1K, but Btrfs reaches that stage at 16K. For larger block sizes (greater than 4K), io_uring never outperforms system calls. As both block size and number of blocks increase, differences between transports diminish as most time is spent in copying bytes.

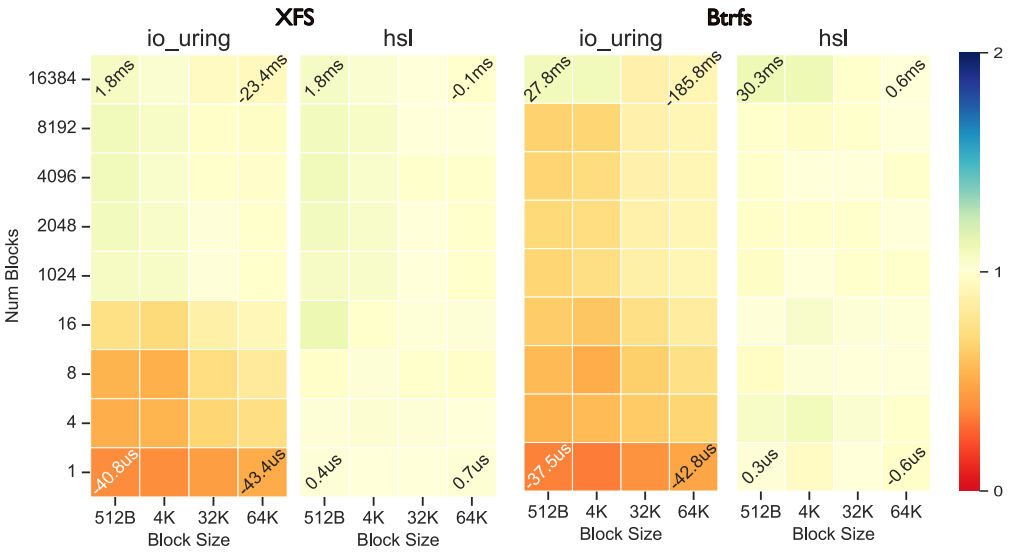

Figure 6.10: **Evaluating scattered writes on XFS and btrfs:** The figure shows four heatmaps similar to figure 6.9, but for scattered writes on XFS and btrfs. As HSL is aware of the performance equilibrium between io_uring and system call implementations for writes on XFS and btrfs, it sets different thresholds accordingly. HSL uses system calls for fewer than 1024 scattered reads in XFS but 16384 scattered reads in Btrfs. For larger chunk sizes (32K or more), HSL always uses system calls as the cost of copying data becomes the bottleneck.

HSL outperforms io_uring for fewer blocks and smaller block sizes as it uses system calls. As seen in the figure, HSL switches to io_uring at each file systems performance crossover points. Similar to scattered reads, we believe the crossover points will change on different systems (and versions), and will have to be discovered as part of installation through profiling.

To conclude, both io_uring and system call transports have their benefits and sticking to one specific transport can cause performance degradation on certain workloads. Through profiling, HSL identifies when to switch between the two, obtaining speedups (upto 7x for uncached scattered reads but less than 1.4x for writes) while minimizing slowdowns.

6.2.3 Application Case Study

In this section, we study five applications to evaluate HSL's ease of integration and the performance benefits it provides. We run experiments that answer the following questions:

§6.2.3.1 Can HSL be easily integrated into the coreutils `cp` utility, and does it make copies faster?

§6.2.3.2 Can HSL handle alignment issues when copying portions of files, as seen in audio trimming workloads?

§6.2.3.3 Can HSL reduce the merkle proof generation time on full nodes in blockchains?

§6.2.3.4 Can HSL be easily integrated into LMDB, and does it increase write throughput?

§6.2.3.5 Can HSL be easily integrated into SQLite, and does it speed up `SELECT` queries?

6.2.3.1 Coreutils cp

We begin our case studies with the coreutils `cp` command line utility; a simple but commonly used program. In addition to it being used on the command line terminal, applications also launch the utility as a process (e.g., inside a bash script, using `system()` in `c/c++`, or `subprocess()` in python). Here, we ask the question: how easy is it to modify `cp` to use HSL, and does using HSL provide any performance improvements? We begin by describing the modifications to `cp`, followed by experiment details to compare performance against the unmodified `cp`.

Modifying `cp`

We begin with an unmodified version of coreutils v9.4. Modifying `cp` to work with HSL was a simple modification of the `src/copy.c` file and modifying the `Makefile` to include the HSL library. We introduce changes

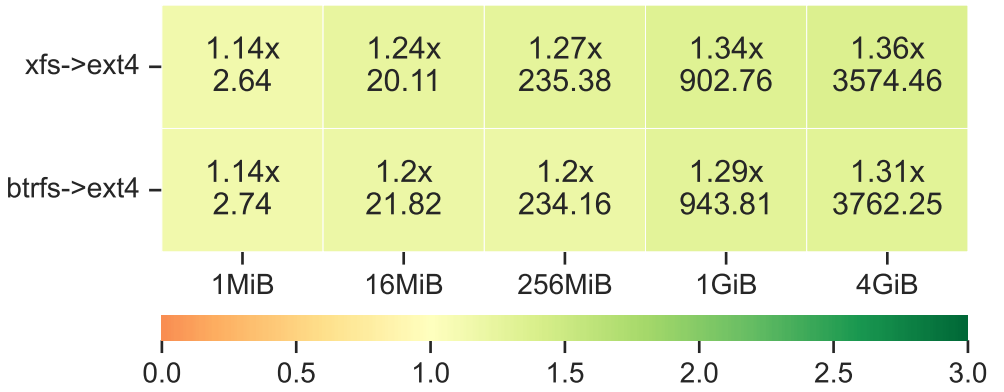

Figure 6.11: **Evaluating coreutils cp on different file systems:** The figure shows the speedup obtained by using a modifed version of coreutils cp that uses HSL over the unmodified version, when the source file is cached. On the x-axis, we vary the size of the file. On the y-axis, we vary source and destination file systems. Each cell is colored according to the speedup and is also annotated accordingly on the first line. The second line in each cell is the time taken by the unmodified version (in microseconds) to perform the copy.

to use the `COPY` verb in the `copy_reg` function: the function `cp` uses internally to copy regular files after it has processed command-line flags and checked that the existing files point to regular files.

Experiment Details and Workload

We build two versions of `cp`: an unmodified version and the one modified to use HSL. We then generate source files of different sizes and place them on three different file systems: ext4, XFS, and Btrfs. We run the `./cp <src> <dst>` command and measure the time for the process to finish.

Findings

We run the workload on both applications, with different `src` `dst` combinations to account for copies on same and different file systems. We repeat each combination twice to account for whether `src` is cached or not. Figure 6.11 reports our findings as a speedup using HSL relative to the unmodified `cp` application, when the source file is cached.

We only report results on different file systems where the destination is ext4 because all other combinations (and the uncached case) show no differences. As explained in §6.2.2.1, copying a file resembles a large single-range copy and therefore, `copy_file_range()` is the best strategy. For all cases where the source and destination are on the same file system, `copy_file_range()` is used both by the unmodified and HSL versions, resulting in similar results both in the cached and uncached cases.

When the source and destination are on different file systems, the unmodified version first attempts to use `copy_file_range()` before falling back to `read()` + `write()` (a try-by-failure strategy), while HSL uses `sendfile()`. For the uncached case, the time spent in disk i/o far outweighs any performance benefits provided by minimizing system calls and buffer copies with `sendfile()`, leading to similar results.

For the cached case, when the destination file resides on XFS or Btrfs, the common code to read or write pages used by `sendfile()`, `read()`, and `write()` tends to dominate, leading to similar results. But we do observe performance improvements (up to 1.36x) when the destination file resides on ext4.

Modifying `cp` to use HSL was incredibly simple but the performance improvements are observed in very few instances. However, as more file systems improve their read and write logic in the future, we may see similar performance improvements as observed with ext4. While the absolute improvement in speedups is only close to a millisecond (1.36x leads to a 907us difference from the unmodified version), applications like bash scripts tend to launch the `cp` application serially in loops; even small gains from HSL can accumulate significantly depending on the loop.

6.2.3.2 Audio Trimming

We now look at an application of *partial* copies—audio trimming, where a new file is created with new headers followed by a portion of the original

file. As we do not control the offset from where the copy begins, depending on user needs, the copy may or may not begin at a 4K (file-system block alignment) boundary. Here, we ask the question: Are there performance differences in copying aligned and unaligned data, and does HSL help increase performance? We begin with a brief background about audio trimming, followed by a description of the application and workloads, and then discuss our findings.

Background on WAV audio trimming

Data processing in the media industry often involves audio editing in post production such as adding effects, background music, or cropping raw audio recordings. Despite its large size footprint, the WAV format is widely used by audio engineers due to its high quality [1, 125]. We focus on WAV audio trimming which is used to select a portion of audio from a long recording.

An audio trim operation is the removal of audio from either end of a recording, resulting in a new file; raw recordings are never overwritten. Depending on the duration of audio we wish to keep (after the trim), the bytes to be copied increase. Stereo recordings at a 44.1 KHz sampling rate produce approximately 5MB of data for 30 seconds of audio, and 1 GB for 2 hours of audio—both valid workloads depending on the project.

The WAV audio format is a lossless, uncompressed format that organizes audio data into frames or chunks. It stores this data sequentially on disk, making it easy to read and write, and uses the Resource Interchange File Format (RIFF) structure for organization. It begins with a main header (the RIFF header) followed by chunk headers and their associated data. For WAV files, it is typically a single chunk header that contains audio metadata, followed by the audio bits in the chunk data portion.

To perform audio trimming such as "removing the first N seconds", the application needs to calculate how many bytes to skip from the chunk data portion. To do so, it needs three key pieces of information: the sampling

rate, the bits per sample, and the number of channels (mono or stereo)—all present in the chunk header. The product of these parameters with `N` provides the number of bits to skip. The application can then create a new file with a new RIFF and chunk header (as chunk metadata reflects a different chunk data size), and then begin copying chunk data from the calculated offset.

Applications that perform WAV Trimming

We use three applications to perform WAV trimming: ffmpeg, hsl_unaligned, and hsl_aligned. The first, ffmpeg, is a well-known swiss-army knife for multimedia processing. We use it *unmodified*, both as a performance comparison and to ensure the implementations we build generate the correct output.

The second, hsl_unaligned, is a custom C++ application which implements WAV trimming as described in the background, using HSL's `COPY` verb. However, depending on the `N` seconds to skip from the beginning, the offset in the chunk data (and the whole file) may or may not be aligned to the underlying file-system's block boundary (typically 4KB).

The third, hsl_aligned, is an improvement over the unaligned version. Although commonly 4KB, we create and use a HSL API which internally uses `stat()` to find the blocksize. The WAV format supports adding junk chunks [97, 125] which were originally meant to align chunks to certain boundaries for CD-ROMs and ignored by media players. We first write a junk chunk to the destination file such that the junk chunk, headers, and audio data before the next aligned block sum up to two blocks. For example, given a 4K block size, if the starting offset for the audio was the 96^{th} byte in a block, we ensure that the junk chunk, headers, and first 4000 bytes of audio data occupy 8KB. We then proceed with copying the remaining data.

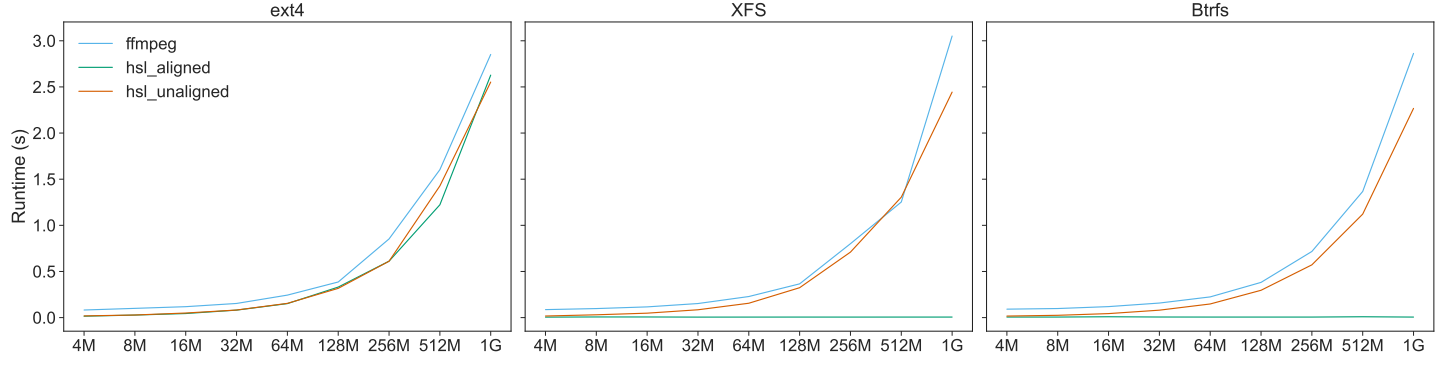

Figure 6.12: **Evaluating audio trimming WAV files:** The figure shows three graphs for experiments on three different file systems: ext4, XFS, and Btrfs. For each graph, the three different colored lines indicate different implementations: ffmpeg, hsl_aligned, and hsl_unaligned. The y-axis reports the runtime to trim audio files of different sizes (x-axis in log scale). The figure shows the time taken in seconds (y-axis) to trim WAV audio files of different sizes (x-axis). Trimming WAV files on XFS and btrfs with hsl_aligned outperforms the rest due to the metadata extent manipulation instead of a data copy.

Workload

We run a workload that only trims from the beginning of the file; audio editors routinely remove audio before the "take"[5]. Trimming the end of the file reduces the number of bytes to copy but does not have any impact on the alignment characteristic we wish to evaluate.

Findings

We run the audio trimming workload for all three applications, removing the first 5 seconds of audio for uncached files of varying sizes (4MB to 1GB). Figure 6.12 documents our findings on three different file systems (ext4, XFS, and Btrfs).

On ext4, all three applications have similar performance, growing with increasing file size as the amount of data to copy increases. Both our applications perform slightly better than ffmpeg as they are only geared towards WAV trimming unlike ffmpeg that has layers of abstractions to handle many other multimedia editing workloads.

On XFS and Btrfs, ffmpeg and hsl_unaligned perform similarly as described for ext4. However, hsl_aligned completes significantly faster (8ms) even for large files due to the extent sharing through `copy_file_range()`. Although `copy_file_range()` is used by HSL in both applications, the file system only performs extent sharing when both source and destination file offsets begin at extent boundaries. As hsl_unaligned does not ensure copies start at the extent boundary, XFS and Btrfs internally use the slower data copy path.

To conclude, there are significant performance differences in copying aligned and unaligned data. Unfortunately, as seen in hsl_unaligned, HSL cannot provide these performance benefits directly as forcing alignment transparently changes application semantics. Instead, applications that can adjust alignment, can query HSL to find the extent boundaries.

[5]The segment of the recording where the actual content begins, following commands like "take" and "action" is often referred to as the Slate or Clapperboard moment.

6.2.3.3 Merkle Proofs

Here, we cover a real world application of uncached scattered reads—merkle proofs, where the application knows exactly which pages (tree nodes) to read from a file. While Section 6.2.2.5 showed significant speedups (7x) for uncached scattered reads as part of microbenchmarking, we cover a realistic workload that depends on the height of merkle trees here.

We ask the question: How much of a speedup can HSL provide for a realistic merkle proof workload? We begin with a brief background on merkle proofs, followed by experiment details and our findings.

Background on Merkle Proofs

A merkle proof is used to validate membership in a set. It utilizes merkle trees where leaf nodes are hashes of the set members. While they are used in ZFS, BitTorrent, and IPFS, we focus on their use in Blockchains [56].

Merchants or vendors on the blockchain often use lightweight clients to verify whether payments have been made (transaction membership in a block). Lightweight clients, lacking the complete blockchain except for the headers, query full nodes for the necessary merkle proofs. The full node cannot simply return a binary response as untrusted nodes can participate in the blockchain; the lightweight client needs a stronger proof to trust the full node's response.

To provide a strong proof of transaction inclusion, the full node uses a merkle tree, where each leaf node represents a transaction hash (the set members) and each non-leaf node is a hash of its children. The node constructs the merkle proof by sending the transaction hash and a set of sibling hashes needed to reconstruct the path to the merkle root.

The lightweight client uses these hashes to verify the transaction by hashing them together and comparing the result to the trusted merkle root in the block header. If the computed root matches the block header's root, the transaction's inclusion is verified. A malicious node cannot fake transaction inclusion, as generating the proof without the correct set of

hashes would result in a different merkle root, and attempting to create a valid merkle root without the actual data is computationally infeasible.

As most transactions are verified once (for confirmation before rendering service), it is highly unlikely to be a previously seen transaction (unlikely to be cached). However, full nodes do maintain indices on transactions to quickly identify their location in the blockchain on disk. A key observation in merkle proofs is that the path is predetermined if you have the leaf location of a transaction and the number of transactions, both of which are available to full nodes that index transactions, making the operation an uncached scattered read.

Applications and Workloads

We build two applications, implementing the merkle proof component performed by full nodes as described above, both with and without HSL. As the height of a merkle tree depends on the number of elements in the set (the leaf nodes), the height of a merkle tree in a blockchain grows with the number of transactions in a block. Depending on the size of a transaction, the number of active transactions happening on the block chain, and the maximum size of a block, the number of transactions in each block can change. We vary the tree height 3 to 20 accounting for 4 to ~500K transactions within a block.

Findings

We run both merkle proof applications on an uncached file containing the merkle tree of transactions on three different file systems (ext4, XFS, and Btrfs) and vary the tree height as described above. With increasing tree height, more non-leaf nodes (from leaf to root) must be read from disk to generate the proof. Figure 6.13 documents our findings.

Without the use of HSL (the unmodified application), the time taken to generate the proof grows with tree height on all three file systems. This is due to the use of traditional `preadv()` system calls, where the required non-leaf nodes to compute the hash are read one-by-one in a blocking

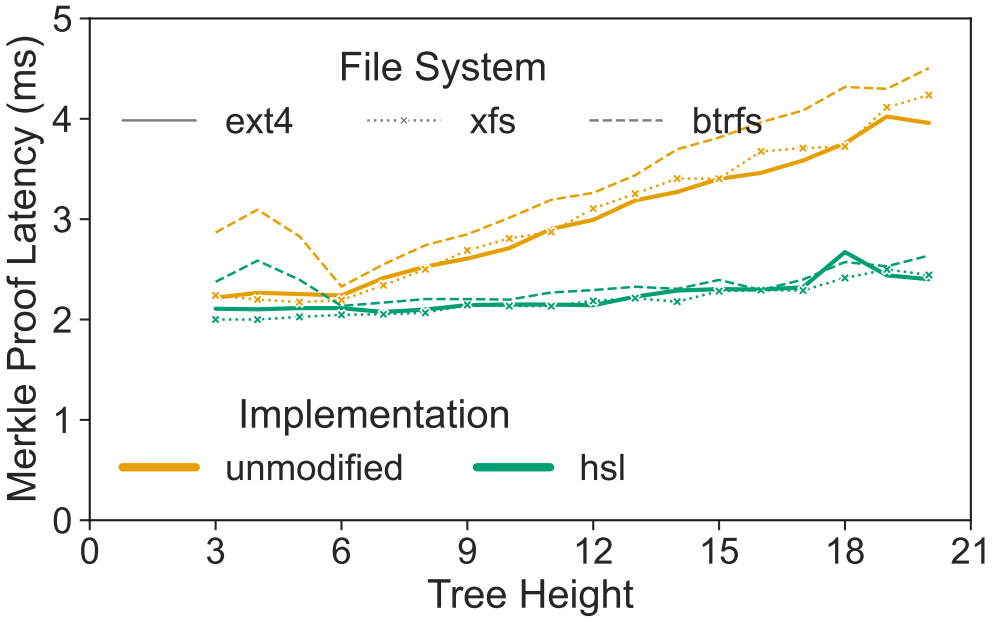

Figure 6.13: **Evaluating merkle proofs:** The figure shows the time taken in milliseconds (y-axis) to validate a merkle proof on merkle trees of different depths (x-axis). We compare two implementations, with and without HSL differentiated by color. We run the workload on ext4, XFS, and btrfs differentiated by line styles.

fashion. The version that uses HSL outperforms the unmodified version as the io_uring transport exploits device i/o parallelism for scattered reads (§6.2.2.5), growing at a slower rate compared to the unmodified version. For a merkle tree height of 20, the unmodified version takes ~4ms while HSL takes ~2ms.

While HSL can provide a 2x speedup on merkle proofs and maybe even more for larger merkle trees, it is impractical to expect 500K transactions in a block. While popular blockchains can have 500K active transactions per second, they limit the size of each block. On Bitcoin, blocks are limited to a 1MB size, allowing a maximum of 4K transactions (provided all transactions are small). Even so, proofs for merkle trees corresponding to 4K transactions (a height of 12) are still faster with HSL (2ms) than

without (3.2ms).

Although the absolute difference in performance between the two implementations is small (1.2ms), a full node receives requests from multiple lightweight clients as there are more lightweight clients than there are full nodes. Faster merkle-proof generation by a full node leads to faster transaction confirmations by lightweight clients and prevents queue buildup. As blockchain full nodes run on many different systems, HSL can help by performing scattered reads more efficiently on those that support io_uring.

6.2.3.4 LMDB

We now cover a real world application of scattered writes—updating an LMDB database. The Lightning Memory-Mapped Database Manager (LMDB [18]) is an embedded key-value store that uses B-Tree data structures whose nodes reside in a single file—the database. Here, we ask the question: can LMDB benefit from using HSL? and if so, we quantify those benefits. We begin by describing our modifications to LMDB, followed by experiment details and findings.

Modifying LMDB

We begin with an unmodified version of LMDB v0.9.31. As the entire database is exposed as a memory-mapped file, fetching data from the underlying file system through `read()` and its variants (or even through HSL) is unnecessary. Instead, we modify the write path.

While LMDB does have the option of also writing to the memory map (MDB_WRITEMAP), applications lose the ability to perform nested transactions and protection from bugs such as wild pointer writes. Therefore, we modify the write path when MDB_WRITEMAP is not used—where LMDB uses `pwritev()`.

When a new key is inserted or an existing key-value pair is updated, LMDB modifies one of its B-Trees using a copy-on-write approach. The B-Tree node (page) containing the new or existing key and all its ancestors

up to the root are copied and modified to create a new version of the B-Tree (due to the new root). LMDB keeps track of these modified ("dirty") pages as a linked list, sorted by page number (which is a one-to-one mapping to the on-disk file offset). On transaction commit (which can include multiple updates like the one described here), the linked list of dirty pages is flushed to disk.

Modifying LMDB to work with HSL was a simple modification of the `mdb.c` file, replacing the loop that used `pwritev()` over the dirty pages to construct a collection that is executed with the `WRITE` verb. We then modify the `Makefile` to link it with the HSL library.

Workload

As our modifications affect the write path, we perform a 100% update operation on an LMDB database containing ~1.8 Million entries of 32B keys and 256B values. We update records randomly, and vary the number of keys updated in a transaction as larger transactions can have more dirty pages. The workload continuously selects and updates key-value pairs for the given transaction size and reports the throughput per second. We run the workload with the `MDB_NOSYNC` flag to evaluate applications that use such an optimization to avoid flushing file system buffers (through `fsync()`) on transaction commit; they allow loss of recent transactions but avoid corruptions. Not using the flag would force disk i/o for every transaction, leading to equivalent results in both implementations as time spent in i/o would dominate any optimizations (§6.2.2.5).

Findings

We run the workload on both the unmodified and HSL versions on three different file systems (ext4, XFS, and Btrfs), vary the number of updates in each transaction, and report our findings in Figure 6.14.

We observe no significant difference in throughput between the two different implementations. This is due to having few dirty pages to be written out per transaction. While increasing transaction size increases

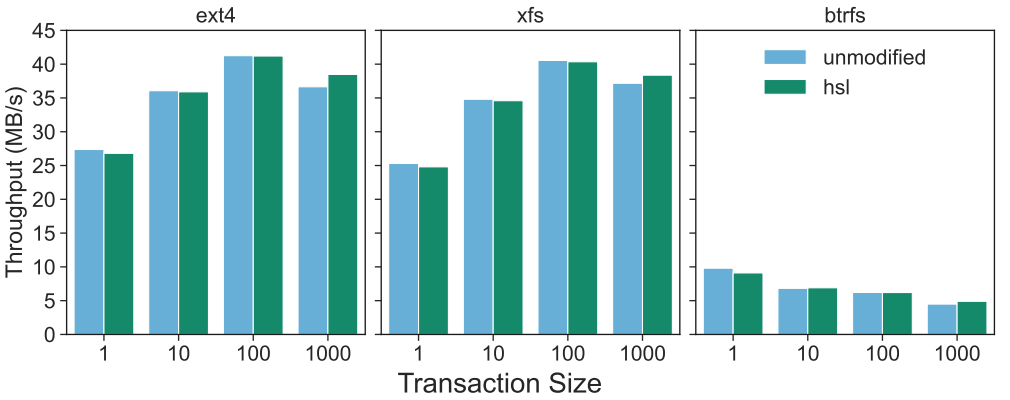

Figure 6.14: **Evaluating update operations to LMDB:** The figure shows the throughput measured in MB/s (y-axis) for a 100% update workload. We vary the number of update operations within a transaction (x-axis). We compare two implementations: LMDB with HSL, and an unmodified LDMB differentiated by color. We run LMDB on three file systems: ext4, xfs, and btrfs as indicated above each graph.

the number of dirty pages to be written out, even with a transaction size of 1000, the number of dirty pages do not exceed 2000. As each key-value pair is less than a single page size, even the worst case of each key in a different page would lead to a 1000 dirty pages for the actual leaf nodes containing the key-value entries. The remaining 1000 dirty pages are intermediate nodes that may be shared across the leaf nodes being modified.

On ext4, HSL avoids io_uring altogether, using a similar implementation (`pwritev()`) as the unmodified version. On Btrfs, HSL avoids io_uring as there are too few pages to be written out; Btrfs requires atleast 16K scattered writes (§6.2.2.2). On XFS, HSL does use io_uring but the performance (although not degraded as it would if used with Btrfs) is still similar to `pwritev()`.

To conclude, while easy to modify LMDB to use HSL for scattered writes, realistic workloads that perform small sized transactions do not gain any performance benefits by using HSL. However, as HSL matches the unmodified application's performance, LMDB can still benefit from

future optimizations. As io_uring is relatively new, kernel development has been focused on getting it to work with other subsystems, not getting it to work *well*. Future updates to the kernel can change the performance crossover points, and if that happens, HSL can take advantage of it without any further changes to LMDB.

6.2.3.5 SQLite

In our final application study, we revisit scattered reads with a relational database—SQLite, an embedded RDBMS that uses B-Tree data structures for its tables and indices, all within a single file [117].

Unlike Merkle Proofs (6.2.3.3), SQLite does not know all the pages it needs to read in advance. And the pages it intends to read may be in the page cache. Additionally, SQLite is more complex than the other applications we studied—it generates query plan that determines when and how rows of the database are read.

Here, we ask the question: How easy is it to integrate SQLite with HSL, and are there any benefits to doing so? We focus on the `SELECT` clause—most frequently used to query structured data. As modifying SQLite to use HSL was not trivial, we first describe relevant SQLite internals and our changes before proceeding with the workload and findings.

SQLite Internals

Internally, a SQL query is first translated into bytecode that is executed by the SQLite Virtual DataBase Engine (vdbe) [118]. The `SELECT` query is converted into bytecode to perform a scan over the table[6]. The bytecode generated is executed by SQLite's virtual machine which opens a B-Tree cursor on the table, iterates over all rows, and picks those that match any `WHERE` criteria either to be used in aggregation or as output records.

[6]Users can view bytecode by prefixing queries with `EXPLAIN`

The presence of indices can transform the above scan operation into fewer random reads of the necessary rows, providing the same output efficiently. When SQLite's query planner identifies an index associated with the table, it generates bytecode to open another B-Tree cursor for the index and iterates on the index to find the id of the rows matching the criteria. For each row id found, it walks the B-Tree of the table (from root to the page containing the row), and adds the row to its aggregation or as an output record. The iteration is a sequence of bytecode operations that resembles the following:

1. Move index cursor to the first match location.
2. Go to step 7 if index is at a non matching location.
3. Move table cursor to the row index cursor points to.
4. Read required columns from row the table cursor points to.
5. Generate output record or internally update the aggregate[7] value.
6. Increment index cursor to next location and go back to step 2.
7. Return any requested aggregate and stop.

In step 4, SQLite generates bytecode to read matching rows *one-by-one* inside the loop. Assuming the cursor is not already at the required location, the table B-Tree is walked from root to the leaf page containing the required row.

SQLite Modifications

As the read operation in SQLite is meant to read one row at a time, HSL's collections are of little use. Before introducing HSL, we first modify SQLite to perform the leaf page reads in bulk.

To do so, we alter the generated bytecode to perform steps 2 and 3, and store the required row ids in a buffer. Unfortunately, identifying the page holding a row (for one of the row ids) requires walking the table B-Tree

[7]For queries such as SELECT sum(column) FROM table.

from the root—a pointer chasing workload. We repeatedly walk the B-Tree and identify the leaf pages, incurring immediate reads for intermediate pages. However, we stop at the penultimate node, i.e., the leaf parent, and do not read the leaf page. Instead, we store the page number of each leaf page.

Once all leaf page numbers are collected, we can then perform a one-to-one mapping from page number to offset. As multiple rows can exist in the same leaf page, we remove duplicates from the collection. We then read all the pages into SQLite's application page cache using the same primitives SQLite internally uses to read the page. Finally, we execute the above original steps, with step 4 reading the columns from a page that is in SQLite's cache.

We perform the alterations to the bytecode *after* it has been generated. We focus on rewriting bytecode for `SELECT` queries with `WHERE` clauses, which are frequently used by SQL users. Expanding scope to more complex queries such as `GROUP BY` would require time consuming modifications to the query planner to generate the desired bytecode (instead of altering it), a task better suited for SQLite's authors/maintainers.

After ensuring that this modified version works correctly, we use HSL instead of SQLite's primitives to read the pages. Instead of a loop reading pages one-by-one, we construct a collection of the leaf pages and issue a `READ` to HSL.

Workload

We generate a database for the TPC-H suite and run query 6 which operates on the `lineitem` table—the largest table in the database, to which we add an index for this experiment. The query performs an aggregate over two columns `sum(columnX * columnY)` over rows that match the `WHERE` clause.

As some pages may be present in the page cache, we first run the workload to identify all pages that would be accessed. We clear the page cache and then read some pages from those identified, to account for cases

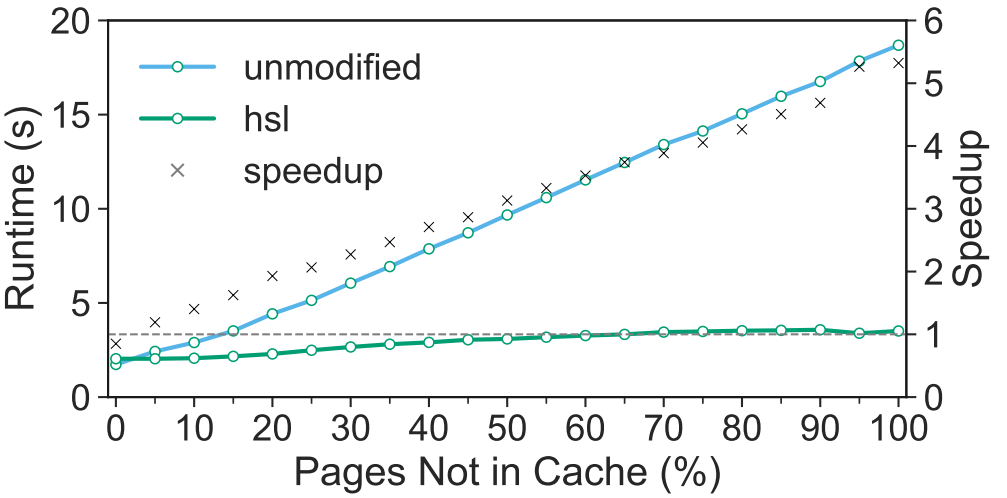

Figure 6.15: **Evaluating TPC-H query 6 with SQLite on ext4:** The figure shows the time taken in seconds (left y-axis) to execute the query on a database that has some pages in the cache (x-axis). We compare two implementations: unmodified SQLite and SQLite with HSL, differentiated by color. The right y-axis shows the speedup obtained when using SQLite with HSL. At x=0, all pages are in the cache, and the two implementations are nearly identical; the unmodified version is slightly faster. However, with even 5% uncached pages, HSL outperforms the unmodified version.

where a portion is in the kernel's page cache. Finally, we execute the query and measure the time taken to obtain the aggregate result.

Findings

We run the workload on an unmodified SQLite, and on SQLite with HSL, on databases scaled to different sizes (.1x, 1x, and 10x), and on different file systems. We also vary the proportion of pages in the kernel's page cache, and document our findings for ext4 using the original size (1x) in Figure 6.15; we report both runtime and the speedup obtained by using SQLite with HSL.

When all pages are in the page cache (x=0), both versions perform similarly. As all pages are in the page cache, the difference in scattered read performance is in the order of milliseconds (6.2.2.5). The unmodified

version is faster than SQLite with HSL (the speedup is below the 1x dashed line, 0.89x).

As page cache occupancy decreases, the number of pages requiring i/o increases, leading to increased runtime for the unmodified version. While our version of SQLite with HSL must also perform i/o, it issues a scattered read in bulk, exploiting device i/o parallelism. This leads to speedups very early on, even if just 5% of the pages (1.13x).At the extreme when no page is in the page cache, SQLite with HSL performs 5.4x faster.We observe similar results for XFS and Btrfs, and on smaller (0.1x) and larger (10x) scales.

To conclude, integrating HSL into SQLite was difficult even when opting for the easy way of rewriting bytecode instead of modifying the logic that generates it. SQLite did not have an easy existing abstraction to perform vector operations.

However, using SQLite with HSL is worth the effort given the speedups (up to 5.4x). The one instance where unmodified SQLite outperforms SQLite with HSL is when all pages are already cached. But the absolute difference in that situation is 300ms, compared to the speedups HSL provides in all other situations (~15s); these numbers scale approximately on smaller or larger databases.

7 Related Work

In this chapter, we discuss how prior work relates to this dissertation. In line with the organization of our chapters, we start with works that involve crash testing and fault injection (§7.1). Next, we mention works that study system calls made by applications (§7.2). Finally, we discuss works that aim to improve application performance by minimizing system calls (§7.3) and using domain specific languages (§7.4).

7.1 Crash Testing and Fault Injection Studies

As Chapter 3 involved studying file-system reactions to `fsync()` failures and their impact on applications, this section discusses how our work builds upon and differs from past studies in key ways. We include works that study file systems through fault injection, error handling in file systems, and the impact of file-system faults on applications.

Our study on how file systems react to failures is related to work done by Prabhakaran et al. with IRON file systems [89] and a more recent study conducted by Jaffer et al. [54]. Other works study specific file systems such as NTFS [11] and ZFS [130]. All these studies inject failures beneath the file system and analyze if and how file systems detect and recover from them. These studies use system-call workloads (e.g., writes and reads) that make the file system interact with the underlying device.

While prior studies do exercise some portions of the `fsync` path through single system-call operations, they do not exercise the checkpoint path. More importantly, in contrast to these past efforts, our work focuses specifically on the *in-memory* state of a file system and the effects of *future operations* on a file system that has encountered a write fault. Specifically, in our work, we choose workloads that continue after a fault has been introduced. Such workloads help in understanding the after-effects of failures during `fsync` such as masking of errors by future operations, fixing the fault, or exacerbating it.

CrashMonkey and Ace [75] utilize the bounded black-box crash testing approach to exhaustively generate workloads and discover many crash-consistency bugs by simulating power failures at different persistence points. In the context of non-volatile memory (NVM) file systems, Vinter [55] traces NVM accesses through dynamic binary translation and generates crash states to detect crash-consistency violations. Our work focuses on transient failures that may not necessarily cause a file system to crash and the effect on applications even though a file system may be consistent. Additionally, we inject faults in the middle of an `fsync` as opposed to after a successful `fsync` (persistence point).

Gunawi et al. describe the problem of failed intentions [48] in journaling file systems and suggest chained transactions to handle such faults during checkpointing. Another work develops a static-analysis technique named Error Detection and Propagation [49] and conclude that file systems neglect many write errors. Even though the Linux kernel has improved its block-layer error handling [71], file systems may still neglect write errors. Verma et al. highlight the problem of consistent modification of application durable data (CMADD) [122]—evolving application data that needs to be persisted atomically in the presence of failures when file systems only guarantee crash-consistency, not application data consistency. Verma et al. [122] add failure-atomic updates to HP's Advanced File

System, and introduce `syncv` to atomically update multiple files, similar to Windows Vista TxF [79] and TxOS [83]. Our fsync-study results are purely based on injecting errors in bio requests that the file system can detect. We circumvent the impact of failed intentions on applications by forcing a checkpoint operation before proceeding (§5.2.2). Additionally, as the implementation choice is delayed to run time, redo log and manifest implementations can utilize the failure-atomic update interfaces on the appropriate systems.

As the file systems themselves can have bugs in their failure-atomic update mechanisms, EnvyFS [12] utilizes N-version programming [6] to multiplex file-system operations across multiple file systems, and determines a majority result. The existence of a majority implies there may be a minority with a different result, in line with our findings of non-uniform file system error handling characteristics. SubSIST, a component in EnvyFS to reduce time and space overheads coalesces data blocks. Therefore, an implementation of EnvyFS with data-block writes to the journal would also be written to eventual locations in other file systems (ext4 ordered mode). When multiplexing on a majority of non-data-journaling file systems, error reporting would be timely, allowing applications to guard against ext4 data mode failed intentions. However, as the implementation was on different file systems than the ones we studied, we cannot know for sure.

SibylFS [96], motivated by varied behavior of file systems, is a mathematically rigorous model that specifies the range of allowed behaviors of file systems. It is useful to identify POSIX violations and platform convention violations. However, they do not cover the `fsync` system call. More importantly, none of the file systems we studied are in violation of POSIX's definition of `fsync`; they are different realizations of a standard that is silent on state after failure.

Vondra describes how certain assumptions about `fsync` behavior led

to data loss in PostgreSQL [124]. The data loss behavior was reproduced using a device mapper with the dm-error target which inspired us to build our own fault injector (dm-loki [25]) atop the device mapper, similar to dm-inject [54]. Additionally, the FSQA suite (xfstests) [127] emulates write errors using the dm-flakey target [24]. While dm-flakey is useful for fault-injection testing, faults are injected based on current time; the device is available for x seconds and then exhibits unreliable behavior for y seconds (x and y being configurable). Furthermore, any change in configuration requires suspending the device. To increase determinism and avoid relying on time, dm-loki injects faults based on access patterns (e.g., fail the 2^{nd} and 4^{th} write to block 20) and is capable of accepting configuration changes without device suspension.

LazyFS [92] is a fault-injection tool, similar to CuttleFS, to debug and reproduce durability bugs in application storage systems. Like CuttleFS, LazyFS maintains its own page cache. However, LazyFS injects different faults (lost or torn writes) and does not emulate different post-failure file-system characteristics (such as differing in page cache content).

Recent work has shifted the focus to study the effects of file-system faults in distributed storage systems [40] and high-performance parallel systems [13]. Similarly, our work focuses on understanding how file systems and applications running on top of them behave in the presence of failures.

7.2 System Call Studies

In Chapter 4, we performed a survey of applications to study how they commonly interact with the underlying file system. In this section, we discuss prior related studies.

Bagherzadeh et al. [8] conduct an empirical study of system-call related source-code changes in the Linux kernel and identify the type of

changes such as bug fixes, new additions, security enhancements, and more. Tsai et al. [121] study Linux API usage across all applications and libraries in the Ubuntu Linux 15.04 distribution, suggesting metrics for the importance of different APIs (including system calls), offering insights into the most important system calls to support. Dubeyko et al. [27] explore dependencies between system calls and hardware under everyday use-cases (compilation, installation, web browsing) and publish reports on system call frequency distribution noting a strong performance dependency on CPU architecture. Kodirov et al. [62] highlight the existence of system-call bloat in gedit using a combination of source code analysis and dynamic tracing with DTrace on Solaris. Harter et al. [51] focus specifically on the i/o behavior of home-user applications in Apple software: iWork(Pages, Numbers, Keynote) and iLife(iPhoto, iTunes, and iMovie), concluding that future storage systems should bridge the gap between application needs and existing low-level features. Didona et al. [21] study modern storage APIs such as io_uring and SPDK, concluding that the Linux Kernel's io_uring interface can compete with SPDK but only with careful understanding and tuning.

Libtrack [5] performs dynamic and static analysis of POSIX use in applications, finding that applications tend to not use POSIX directly but through platform-specific frameworks and libraries. Additionally, the authors mention that new abstractions are arising but are not converging, leading to non-standard accesses on different operating systems. Libtrack focuses on abstractions needed for graphics and networking workloads. File systems are not covered at the same depth. Instead, the authors note that applications use higher-level storage abstractions through SQLite.

Our work focuses on applications that interact with storage, relying on the underlying file system. The tools we use (Ikhnaie 4.1.2) are not meant to automatically discover patterns but to direct our attention to source code responsible for repeated system calls. We manually inspect source

code and documents to construct a set of *intentions*—high level operations to be executed using the best low-level features. Like Ikhnaie, DIO [29] intercepts system calls, enriches the traces with additional kernel context (process name, operations on files vs directories), and provides visualizations to simplify exploration of traces. Unlike Ikhnaie, DIO does not trace user-space functions responsible for those system calls. However, Ikhnaie could benefit from the syscall analysis algorithms and visualizations used in DIO.

7.3 System Call Performance Efforts

While prior works create more efficient system calls or reduce the overheads in existing ones, none are motivated by the need to choose the right system call for a given underlying file system. However, HSL's design allows it to be effectively combined with the rest, incorporating all their benefits. We categorize prior works in two ways: those requiring active integration by the developer and those that are transparent/seamless; HSL belongs to the former. We cover both categories below.

7.3.1 Performance through Seamless Integration

Among works that require no change by application developers, Cassyopia [91] applies compiler optimization techniques to reduce the number of kernel boundary crossings in system calls. Through runtime profiling, system calls are clustered and the binary is rewritten to use a multi-call[90], a mechanism also found in hypervisors. Userspace Bypass (UB) [132] transparently pulls user-space code into the kernel, reducing boundary crossings for consecutive system calls. FlexSC [108] converts conventional system calls transparently into their exception-less counterparts, a new mechanism for interacting with the OS that is most effective on multi-core processors. HSL can utilize compiler optimization techniques in its

middle-end and use multi-call and FlexSC as backend transports; UB may not be necessary if the backend creates a single multi-call.

CFFS [129] attempts to speed up reads to multiple files, by combining them into a composite file transparently. Composite file creation is performed either on an entire directory, based on file references within file contents, or by analyzing accesses. Because composite files contain metadata for the entire set within a single file, applications benefit from fewer accesses to random blocks containing metadata for different files. Depending on the layout of composite files, read accesses can also benefit from native prefetching. HSL intention implementations that deal with multiple files can instruct the exact membership and layout of composite files to CCFS.

KML [2] is a prototype Machine Learning (ML) architecture within the OS aimed to replace manual heuristics used to optimize storage systems. KML applied to file-system readahead logic and NFS's rsize (which decides the chunk size of data transferred by client or server) shows promising results on mixed workloads. As HSL also has its own heuristics and values calculated through profiling, HSL's execution may not be effective on a KML system as the heuristics captured during profiling may not hold true due to KML's dynamic behavior. However, as KML runs identically in user or kernel mode, HSL can utilize KML for workload-dependent heuristics (e.g., how much to read with the HSL's `.expect` hint).

7.3.2 Performance through Active Integration

Many works have introduced new primitives that developers can use instead of existing system calls, some step-based and some semantic. Step-based primitives are combinations of frequently seen system call sequences and named similarly. Semantic primitives have a higher-level name to describe the operation. In both cases, middle-end passes in HSL can replace patterns with user-defined verbs.

Step-based: Vincente et al. introduce compound system calls [123] that developers must actively use instead of common system call sequences. FusionFS [128] aggregates I/O operations into CISCOps but requires the use of a library file system.

Semantic: Kokoszka et al. implement `search()` [63] to find path names within a list of directories matching a pattern (file metadata). Search on file data has been accelerated through Dynamic Sets [111] where the OS chooses the order of processing of files, prioritizing those that are cached or faster to retrieve. vNFS [14]—a client for NFSv4 which supports compounding—exposes a vector API to minimize round trips.

Some works identify `fsync` performing dual roles: ordering and durability. OptFS [16] decouples `fsync` with new primitives `osync` and `dsync`. Featherstitch patches [35] in user space allow applications to specify their precise consistency requirements to the underlying file system which can then reorder writes safely. HSL can be modified to include directives that (if unable to infer) indicate the same.

Lastly, XRP [131] allows user-defined BPF programs to be offloaded to the storage driver, speeding up pointer-chasing workloads by issuing new I/O requests without jumping back to user space. XRP endpoints can be defined in HSL similar to the AWS Lambda endpoints used for filtering.

7.4 Domain Specific Languages

We believe that HSL (chapters 5 and 6) is the first work to apply a declarative language as the indirection layer for file-system related system calls. However, there have been Domain Specific Languages (DSLs) in other domains, as well as in the context of filestores—structured collections of data files housed in a conventional file system. We cover them below.

Triton [119], like HSL, is a DSL to efficiently use the GPU instead of the file system. Like file systems, some GPU's may have efficient operations but using them requires expert knowledge and is often not portable. Instead, developers can express computation in Triton which internally performs optimization passes and instruction selection to generate efficient GPU kernels.

Ziria [112] is a DSL in the wireless systems programming space. While the implementations of wireless protocols on existing software-defined radio (SDR) platforms involves the use of optimized digital signal processing (DSP) algorithms, they also require efficient and correct expression of reconfiguration when system state changes. Ziria provides domain-specific abstractions for programming wireless SDRs with reconfigurable pipelines, aggressively performs optimization transformations, and utilizes the most efficient DSP algorithms for the task.

In the storage domain, PADS [32] is a declarative data description language for ad hoc data—non-structured data in multiple text and binary formats. PADS generates C libraries and tools to manipulate the data, eliminating the need to write parsing and serialization routines. Forest [31] is an embedded Haskell DSL for describing and managing filestores (similar to ad hoc data)—collections of data files housed in conventional file systems. Forest highlights the challenges in using filestores (lack of documentation, incorrect data loading or storing, and incorrect error detection) and explores solutions using principles of typed programming languages; TxForest [22] builds on Forest and addresses limitations in concurrent accesses to filestores. While all the above are related to storage, they are concerned with correctly and conveniently manipulating or accessing the contents within filestores. HSL operates at a lower level, addressing durability correctly (as file systems can behave differently) and choosing the most efficient system calls; the above works can utilize HSL internally.

8 Conclusions and Future Work

In this chapter, we summarize our work and what we learned from each part individually (§8.1). Then, we discuss lessons learned across the dissertation as a whole (§8.2). We then offer directions on how each part can be extended in the future (§8.3).

8.1 Summary

This dissertation is comprised of three parts. In the first part, we performed an in-depth study of `fsync()` failures from the perspective of file systems and applications. With fault injection, we showed that neither file systems nor applications handle failures uniformly, and current strategies are insufficient to guard against data loss. In the second part, we performed an explorative study of how applications interact with the file system. We found multiple instances of fine-grained system calls being used towards higher-level tasks—intentions. In the third part, we designed, implemented, and evaluated HSL: a declarative language that uses runtime information to execute a correct and efficient intention. We now summarize each of these parts individually.

8.1.1 Understanding `fsync()` Failures

We first performed a study on how three file systems (ext4, XFS, and Btrfs) react to disk failures that result in `fsync()` failures. We considered disk failures that can fail a single sector write while allowing writes to other sectors (a fail-partial failure model), and transient failures where a failed sector may accept writes later. We built `dm-loki`, a loadable kernel module device-mapper target to perform deterministic fault injection using the above failure model and log all writes to the device. We built `blockviz` to visualize experiments with `dm-loki`, that generated traces to help analyze file-system behavior. We asked and answered 11 questions (§3.1.1.5) pertaining to `fsync` failure behavior. While all file systems marked the dirty pages clean, ext4 and XFS retain the latest content in the page cache while Btrfs reverted to match on-disk content. We found additional problems with ext4: directory data block failures could cause a silent loss of directory entries, and `fsync` failures in ext4 data mode are not reported immediately.

We then shifted focus to applications and their `fsync()` error-handling strategies. We built CuttleFS: a FUSE file system that can inject failures deterministically at the file level rather than the block level (dm-loki), and emulate the reactions of previously studied file systems. Additionally, we emulated environmental behavior such as system restarts and page-cache evictions. We studied five applications: Redis, LMDB, LevelDB, SQLite, and PostgreSQL, and found invalid outcomes. In addition to data loss and invalid outcomes, we discovered and termed another class of invalid outcomes: false failures—instances where a failure is reported but the application state is identical to success; a problem for non-idempotent operations. With the exeception of Redis (which did not check the return code of `fsync`), all other applications perform some error handling by either reverting state or crashing immediately. While none of them retry `fsync`—as the issues with doing so became well known—none of their other strategies worked well across all file systems either.

While neither file systems nor applications follow a uniform error-handling strategy, there are valuable insights to be gained from their varying behaviors. Copy-on-write techniques have shown promising results in mitigating `fsync` failures, both at the file system level (e.g., Btrfs) and the application level (e.g., LMDB). However, applications like SQLite and PostgreSQL, which use redo logging for durability, often seek faster write throughput—something that can be challenging on copy-on-write file systems like Btrfs.

If an application requires more performance than copy-on-write file systems can offer, it will need to account for the specific failure characteristics of the underlying file system. Moreover, for portable applications, error handling should not be treated uniformly across file systems, as they exhibit different behaviors. Error handling must be tailored to each file system, rather than relying on a single approach or assuming uniformity across operating systems.

8.1.2 Discovering and Categorizing Intentions

In the second part of this dissertation, we explored application and file-system interactions through system calls. We built Ikhnaie—a tool to trace and visualize those interactions. Using Ikhnaie, documentation, and manual inspection of source code, we explored applications across multiple domains and classified commonly seen interactions after identifying what the applications were trying to do; we termed these interactions as *intentions*. We focused on read and write intentions.

Among single-file operations, applications can perform read operations either in a content-dependent or content-independent manner; with one of four access patterns for the latter. We identified filtered reads as a common task where data that does not match the filter criteria is copied unnecessarily.

For write operations, most applications append new data sequentially,

while overwriting occurs in a scattered access pattern. Applications concerned with atomicity or durability use either custom update protocols (e.g., LMDB, Redis) or general approaches like copy-modify-replace or physical redo logging. The copy-modify-replace technique, used in applications like vim and LevelDB, is similar to copy-on-write: effective, but less efficient, and therefore typically avoided in an application's *hot path*. Physical redo logging, as seen in SQLite, is more commonly used for durability in the *hot path*, though its implementation is not flawless (§3).

We divided multi-file operations into homogenous (all reads or all writes) and heterogeneous operations (such as copies), identifying two distinct types of copy workloads. Some applications perform a few large, contiguous copies (e.g., cp, tar), while others handle multiple small, scattered copies (e.g., checkpointing redo logs). Similar to single-file writes, applications concerned with atomicity or durability across multiple files use physical redo logging (e.g., PostgreSQL) to track changes to existing files, and manifests (e.g., LevelDB, RocksDB) to track changes to an immutable set of files.

We discussed the challenges developers face when implementing these operations. Often, they adopt a lowest-common-denominator approach for correctness, designing to defend against the weakest file system. This can lead to redundant and costly protections on more robust file systems. On the other hand, trying to account for each file system's unique behaviors adds a significant cognitive burden. From a performance perspective, we highlighted the key considerations developers must evaluate when choosing an efficient implementation for these operations: selecting the right primitives, determining when to provide the file system with additional context, and deciding whether to use modern interfaces like io_uring. We also identified other operations—such as prepends, inserts, and moves—that complete our classification but are rarely used in practice due to their well-known limitations.

In many cases, applications make repeated system calls to achieve a higher-level goal, or intention, due to the fine-grained nature of these calls. Identifying and understanding these intentions can lead to improved performance, a trend Linux has followed over the years by introducing new system calls. Offering reference implementations for common intentions like atomicity and durability, such as redo logs, can help developers prevent data loss when building newer applications.

8.1.3 Declarative System Calls with HSL

In the final part of this dissertation, we presented HSL: The High Level System Language, a declarative language for file-system interaction. HSL replaced system calls with verbs—words that convey intent rather than implementation—and introduced dedicated verbs for physical redo logs and manifests. We incorporated collections to allow verbs to handle multiple arguments (e.g., reading multiple scattered locations) and added hints to provide extra context to the underlying system, such as `fadvise`.

Additionally, we introduced a filter directive to offer a uniform interface for filtering, facilitating easy switching of implementations based on whether a file system is local or remote. With HSL's middle-end, we demonstrated that peephole optimization passes could leverage niche efficient interfaces in certain file systems, such as the Google Cloud Storage `compose` API for file concatenation. The HSL backend utilizes runtime information—including the file system type, mount options, page cache proportions, and collection sizes—to determine the optimal execution strategy for specific intentions. Furthermore, we introduced the concept of transports, which separates system call functionality from invocation methods, including `io_uring` as an alternative transport mechanism.

We conducted fault-injection experiments on applications utilizing the redo log and manifest verbs, revealing no invalid outcomes. HSL effectively managed file-system-specific errors without imposing addi-

tional cognitive burdens on developers. Performance evaluations of HSL's features demonstrated significant improvements in operations such as copying, concatenation, and using `io_uring` for scattered reads. Specifically, the `.filter_endpoint` directive excelled in scenarios involving rare matches (e.g., searching for errors in a log file) by consistently minimizing network costs. Additionally, the `.expect` hint was effective in avoiding unnecessary `fadvise` system calls, which can degrade performance during small strided reads. However, we observed no significant performance enhancements when using HSL for scattered writes.

We performed case studies on various applications to assess the ease of integrating HSL and the associated performance benefits. Integration with HSL was straightforward for all applications except SQLite. Performance improvements were observed across all case studies except for LMDB, which utilizes scattered writes—a scenario where HSL did not yield benefits. In the context of audio trimming, HSL achieved better performance only after modifying the application design to ensure alignment with the file system's block boundary (4KB), highlighting the necessity for certain application-level adjustments to fully leverage HSL's capabilities.

Our implementation of HSL successfully achieved all its design goals (§5.1), with the exception of `Standalone Sufficiency`—currently, applications must still utilize system calls since HSL does not yet support all existing system calls. Through various case studies, we demonstrated that:
Incremental Integration: HSL remains effective even when used partially.
Portable Efficiency: HSL selects the appropriate interface based on the underlying file system.
Reduce Cognitive Burden: Utilizing the redo log and manifest verbs eliminates the need for developers to manage file system-specific error handling strategies.
Extensibility: Although not a dedicated verb, HSL allows the addition of implementations for specific tasks, such as concatenation, for file systems

that offer dedicated APIs.

Despite its advantages, HSL does not universally guarantee performance improvements. For instance, in the audio trimming evaluation, HSL was unable to align data for the application without compromising semantic guarantees. Although data alignment can boost performance, it necessitates that applications are designed accordingly. Furthermore, integrating HSL is not always straightforward, as evidenced by challenges encountered with SQLite. Applications lacking batching or vector operation support may require significant refactoring to fully utilize HSL.

HSL may offer limited advantages in data center environments where workloads are well-defined and the systems running applications are controlled or standardized. In contrast, HSL is particularly beneficial for developers deploying applications across diverse systems with varying environments. In such scenarios, where developers may not have full control over the deployment environment, HSL helps ensure both efficiency and correctness without requiring environment-specific optimizations.

Currently, HSL is in its early developmental stages. It does not yet possess the maturity of modern compilers, which benefit from extensive optimization capabilities. Additionally, HSL lacks a feature to lock in specific implementation paths, complicating efforts to ensure that the code tested during development precisely matches what will execute in production environments.

8.2 Lessons Learned

Here, we present a few important lessons we learned while working on this dissertation.

#1: Characterize, then construct

We began this dissertation by characterizing the behavior of `fsync()` across both applications and file systems. From there, we expanded our analysis

to other interactions between applications and file systems. Once we identified areas for improvement, we proceeded to construct HSL. Even during the construction of HSL, we adhered to the same method in terms of performance: "Measure, Then Build" [3]. Building HSL required a thorough understanding of the performance characteristics of existing interfaces. Only after measuring the performance of `sendfile()` and `copy_file_range()` under various workloads and environments could we design the most efficient copy operation. This two-step approach—first characterizing existing systems, then building new tools or making incremental improvements—has proven to be an effective strategy.

#2: Inject failures below your abstraction layer

In this dissertation, we have evaluated real-world applications that struggle with durability, even when that is their intended goal. Injecting failures at the device level, using tools like dm-loki, is a standard practice for file systems in testing, and we extended this approach to examine the state further. Since the kernel supports multiple file systems, performing tests at a lower layer allows us to evaluate durability across them all. However, the same is not true for applications, which do not coexist in the same way and are not developed to share infrastructure.

When it comes to fault injection, there are multiple levels to consider: at the same level as the application, or at lower levels. Injecting faults at the same level through mocking only captures the application's response to the failure and doesn't reflect the true state of the underlying system. This can give a false sense of reliability, as retries or crash-recovery mechanisms may appear to resolve the issue during tests but could still lead to data loss in production.

We advocate for injecting faults at the immediate lower level from the application. This approach allows us to enforce deterministic failures, ensuring more accurate and repeatable testing. While using dm-loki would allow failure injection at the block device level, the inherent non-

determinism of file-system block allocation makes it difficult to reproduce specific failure scenarios reliably. Modifying multiple file systems to inject failures is also impractical. Instead, we chose to emulate a well-defined state using a FUSE file system (CuttleFS), a layer directly below the application. This allows us to think critically about the system state, while still maintaining deterministic control over failure injection.

#3: Standards ensure interoperability, not uniform behavior

While studying different system calls, we observed that although various file systems accept the same arguments and generally produce the same results (when successful), they are not identical. This allows for innovation, as different file systems are free to experiment with diverse designs and optimize for different workloads. However, applications require portability. Standards ensure that applications can run on these systems and, in most cases, function correctly. Yet, differences in implementation and design lead to varied behavior across file systems.

We observed this variability with `fsync`, especially when failures occur. This is primarily because standards do not mandate specific actions in the event of a failure. Another example is the `fadvise` system call, where providing unnecessary advice can negatively affect performance for small strided reads on some systems, though this may not be the case on others.

Even within an operating system like Linux, system calls can give the impression of uniformity, though they are not always standardized. For instance, `sendfile` and `copy_file_range` can be used for copying data. While not POSIX-compliant, the expectation is that they will work correctly on Linux across all file systems—and they do. However, they do not always exhibit the same performance characteristics.

It is important to remember that standards only guarantee that your application will work, but not necessarily that it will perform well. And when a standard is silent on failure behavior, extra caution is warranted.

#4: Granular system calls fall short, plan for transformations

In our exploration of system calls and our implementation of HSL, we examined many system calls and their historical context. Newer system calls were introduced as Linux responded to the needs of applications. For instance, the `pread` and `pwrite` system calls were introduced in Linux 2.1 to avoid the need for `lseek`. Later, in Linux 2.6, the scatter-gather interfaces `readv`, `writev`, `preadv`, and `pwritev` were added. More recently, Linux introduced `preadv2` and `pwritev2`, which allow flags to modify behavior on a per-call basis, without requiring changes to the file descriptor itself. These additions are driven by application demand, but Linux cannot remove the older system calls due to the need for backward compatibility. As a result, applications must consciously decide when to adopt the newer interfaces.

History suggests that we may never have a perfect set of system calls. There will likely always be cases where patterns emerge that could be optimized if the underlying system had more context. For example, Linux introduced `sendfile` and `copy_file_range` to accelerate common copy operations. Similarly, if an `unlink` often follows a `copy`, this could be optimized into a single move operation. Thus, we believe that relying solely on a static set of system calls is insufficient. An intermediate layer that can rewrite and transform sequences, adapting to new patterns and performance optimizations, is a better approach. The kernel can only offer an ever-expanding list of system calls, but it cannot dynamically make these transformations for us.

#5: System calls are not "local" anymore

System calls were originally designed to request functionality from the operating system, which traditionally resided locally on the same device. However, the landscape has changed. Today, system calls can be routed elsewhere than handled entirely within the local operating system. This shift can occur with remote or FUSE file systems, where the OS may

reroute the system call to a different subsystem outside its direct control. For example, a local FUSE file system might interact with cloud storage, or a local file system may interact with a network block device.

A system call that is fast on a local device can take significantly longer when executed over a network. Additionally, because these remote systems operate outside the traditional OS boundaries, they may offer additional interfaces not constrained by the OS's system call interface. While they provide an interface for interoperability, they might also offer more efficient methods of communication or data transfer.

Developers must carefully consider the environments in which their applications will run. If the application operates in an architecture where system calls are not always local, it becomes crucial to abstract high-level intentions to ensure optimal performance. In such cases, using a tool like HSL enables the system to select the best path, optimizing for the specific constraints of the environment.

8.3 Future Work

In this section, we discuss ways to extend the work done in this dissertation.

8.3.1 Extending the `fsync()` Study

While using `dm-loki` and `Cuttlefs` was beneficial in performing failure injection, we noticed their drawbacks when revisiting durability errors on redo logs and manifests during the HSL development process.

At the time of creating these tools, the latest Linux kernel was version 5.2. By the time HSL was being evaluated, the kernel had reached version 6.5. The `dm-loki` kernel module was not compatible with the later kernel version, presenting challenges in evaluating whether the same durability issues persisted. Although `FUSE` also underwent changes, updating `CuttleFS` was relatively straightforward.

It would be beneficial to develop tools that can automatically characterize the `fsync()` behavior across different file systems, or even across different versions of the same file system. Such tools would provide valuable documentation for developers that is not readily available from POSIX. While the kernel maintains backward compatibility for applications, allowing them to run without breaking user space, anything internal to the kernel—especially kernel modules outside of the mainline kernel—can break between versions. One alternative strategy would be to perform fault injection using a network block device. This would still operate at the kernel layer, injecting faults, but the network protocol would remain stable over time, avoiding the challenges of evolving kernel versions.

The `fsync()` system call is one where the file system must update its internal state, and failures during this process require proper error handling and state cleanup. Future work could extend this by identifying other system calls that modify internal state and exhibit non-uniform failure handling. A parallel approach could involve a detailed examination of POSIX itself to identify potential deficiencies in its specifications. Are there other interfaces in POSIX that allow for significant differences in implementation behavior? Understanding these gaps could inform more consistent and reliable system call implementations in future versions of file systems.

8.3.2 Expanding and Detecting Application Intentions

In this work, many application intentions were discovered, i.e., what an application truly aims to accomplish when it makes multiple system calls to the underlying file system, by studying the interactions between applications and the file system. There are three directions in which this work can be extended.

First, now that these intentions have been identified and are better understood, tools could be created to automatically detect them. Such

tools could inform developers that these intentions can be abstracted, and a language like HSL could be used to streamline their implementation.

Second, the study primarily focused on read and write intentions. However, another class of intentions—metadata intentions—was observed. Applications like `Make` rely heavily on querying metadata, such as the creation or modification times of files. The underlying requirement of these applications is to obtain the subset of files that have been modified since a specific point in time. Further work could explore how applications use and modify metadata, and the reasons behind those actions. This could provide opportunities to optimize systems for metadata operations, as current systems primarily optimize the data path. There might even be room for specialized storage systems tailored to build tools that depend heavily on metadata.

Third, there are intentions that were not observed but which logically complete the set of potential intentions. These were absent because their limitations are well known to application developers. For example, prepending data is generally avoided due to the inefficiencies caused by shifting the remainder of the data. However, if such operations were made more efficient, it could simplify the development of certain types of applications. Additionally, exploring whether there are applications that would benefit from such functionality, but have been intentionally designed otherwise due to current limitations, could open new avenues for optimization.

8.3.3 Extending HSL: Overcoming Limitations and Optimizing Performance

In this section, we discuss how to extend HSL, a declarative language designed to understand and execute intentions in the most efficient manner across different systems.

The first way to extend HSL is to address its current limitations. Borrowing from compiler design, additional optimization passes could be added that go beyond basic peephole optimizations. For example, dead code elimination could be applied by avoiding issuing `fsync()` on directories when the mount options already handle that. Over time, these transformations would allow novice users to write code that interacts with the system much like user-space code does, while still generating the most efficient set of system calls. Another limitation to address is the need for production systems to ensure that the paths tested during development are identical to those taken in production. Just as compilers generate code tailored to a specific platform, HSL needs the capability to take input, such as the target platform and assumed workloads, ensuring it adheres to a consistent strategy. This would prevent unexpected behavior in production environments compared to the test scenarios.

The second way to extend HSL is to broaden the scope of runtime information. Further studies are needed to determine what additional runtime information could be useful and how it can be exposed to HSL. For instance, our current implementation can only gather cache statistics over large contiguous regions because finer granularity incurs significant costs. Exposing this information at the level of individual reads, such as revealing how much of a read was served from the cache, could optimize future operations in HSL. Similarly, exposing device-specific characteristics could provide further optimizations. Some devices support offloading specific compute tasks to smart devices, which can accelerate filtering operations, while specifications like NVMe's SGL Bit Bucket Descriptor allow for skipping blocks in a single I/O request, potentially making strided reads more efficient.

The third extension involves designing dynamic switching of intention implementations. Currently, HSL switches certain implementations based on runtime information such as collection size and file system characteris-

tics. However, for redo logs and manifests, decisions are made solely based on file system characteristics. A key observation is that once a checkpoint or commit operation is complete, it may no longer be necessary to stay with a particular implementation. The on-disk structure already contains a magic number corresponding to the implementation in use, allowing for flexibility. By leveraging runtime statistics, different implementations could be selected for better performance. For instance, if all blocks in a redo log are block-aligned, file systems like XFS and Btrfs could offer faster checkpointing due to quicker copying to the base file. Currently, only one correct implementation is provided, but as with system calls, multiple implementations may exist, with some being more efficient on different file systems.

8.4 Closing Words

Most applications do not have unrestricted access to hardware; they rely on operating systems and file systems to manage reading and writing to storage. These applications are typically designed to run across a variety of systems, allowing developers to write portable code without knowing the exact environment in which it will be executed.

Designing applications for portability, however, presents significant challenges in achieving both correctness and performance. This complexity also makes it more difficult for systems engineers to contribute effectively. Our work aimed to reduce these challenges, much like how compilers have simplified development for user-space applications.

We posed the question: Can a declarative language for file-system interactions improve the correctness and performance of portable applications? Our research showed that achieving correctness is difficult without a deep understanding of the systems on which the application will run. Additionally, systems are often underutilized due to the absence of a

clear performance hierarchy among standardized interfaces for certain intentions.

By introducing a declarative interface like HSL, we addressed these concerns, providing a more efficient way to handle system interactions. Although HSL represents an early step, it has already shown promising results in improving both durability and performance.

A
The HSL Specification

Here, we describe HSL's specification for each verb. Note that the specification differs slightly from the design (§5.1) to reflect the current implementation.

A note on return values: Currently, when `.execute()` on the script completes, an integer is returned—the return code. A return code of 0 indicates success while 1 indicates a failure. The specification needs to be improved to handle accurate identification and bubbling up of errors with respect to collections. One solution is to have two layers of return codes for errors. The first layer is a single return code for quick checks of success. The second layer is return codes for each collection item which can be accessed either by binding a vector to HSL, or querying HSL to report more information (such as the errno) for a particular index in the collection. On failure, the first layer would return a non-zero error code (not simply 1), indicating the first element in the collection that faced an error. Additionally, to handle the case of incomplete reads or writes (which may occur even in the absence of failures), users can query HSL for the bytes read or written for a collection. HSL will treat incomplete reads or writes as errors with a `HSL::EINCOMPLETE` error code.

A.1 READ

The `READ` verb is used to read data from one or more files using file descriptors; for multiple files, the file descriptor argument must be a collection. The data to be read may be specified as a single contiguous range with a starting offset and a length, or a collection of offsets.

(1) `READ fd, buf, size;`
Reads from a file descriptor `fd`, using the internal offset within the kernel for the file descriptor, similar to `read()`. The above reads a single contiguous range of `size` bytes into memory buffer `buf`. All three identifiers require *binding* at runtime.

(2) `READ fd, buf, 4096;`
Unlike fd and buf, a constant size (e.g., 4096) can be written as is without requiring runtime binding.

(3) `READ fd, buf, size, offset;`
Similar to (1) and (2), reads data but from a specific offset (like `pread()`). Additionally, like size, offset can be a constant.

(4) `READ fd, [buf, size]@records;`
Reads a contiguous range from disk into multiple buffers, similar to `readv()`; a gathered read. As the internal offset is used (like (1)), each read must happen in order for the application to know which read is in which buffer; i.e., only ordered collections are supported.

A note on collections: the name "records" is only an identifier used for binding. Any other name can be used in its place. During binding, HSL requires that `records` binds to an integer indicating the size of the collection, `records.buf` binds to a vector of buffers (pointers), and `records.size` binds to a vector of integers if `size` is not a constant like 4096. The size of `records.buf` and `records.size` must be at least the size of the collection, and each entry in `records.buf` must be a valid allocated memory

buffer. Moreover, `records.buf[0]` must have enough memory to hold `records.size[0]` bytes.

(5) `READ fd, [buf, size]@records, offset;`
Similar to (4), but starts from the given offset instead of the internal one (like `preadv()`).

(6) `READ fd, {buf, size, offset}@pages;`
Reads arbitrary ranges from disk into multiple buffers; a scattered read which would require a looping with `preadv()`. We use an unordered collection as the user specifies the exact offsets for each buffer and the size of each buffer. Like (4), the collection named "pages" requires `pages.buf` and `pages.size`. Unlike (4), it also requires `pages.offset` where data at `pages.offset[i]` of length `pages.size[i]` will be read into `pages.buf[i]`. Unlike the size identifier, `pages.offset` must be a vector of integers (contrast to (3) where both size and offset can be constants).

(7) `READ fd, [buf, size, offset]@records;`
Similar to (6) but forces an ordered read operation. We don't have a genuine use case for it but support it regardless.

(8) `READ {fd, buf, size, offset}@fileset;`
Perform a multi-file scattered read operation given a collection named "fileset". In addition to collection details mentioned in (6), `fileset.fd` is a vector of file descriptors where `fileset.fd[i]` is the descriptor to be used for the operation.

A.1.1 Using `.expect` Hints

An `.expect` hint must always be preceded by a `READ`.

(1) `.expect SEQ;`
Issues an `fadvise()` system call for sequential access on the file descriptor of the preceding `READ`.

(2) `.expect RSEQ;`
Performs a readahead (either through `readahead()` or `fadvise()` with `FADV_WILLNEED`) on the previous chunk for the file descriptor of the preceding `READ`. The chunk size is the same as the bytes to be read by the preceding `READ`.

(3) `.expect STRIDE length;`
Performs a readahead (similar to (2)) on the chunk after the given stride length.

(4) `.expect next_offset;`
Performs a readahead (similar to (2)) on the given offset.

A.1.2 Using `.filter` Directives

Like `.expect`, a filter directive must also be preceded by a `READ`; specifically, single contiguous reads.

(1) `.filter \n, keyword, callback_fn, ctx, SEQ|RSEQ;`
Consider a single contiguous `READ` that stores the data in a single buffer named `BUF`. In the absence of the filter directive, a user that wishes to filter records in `BUF` must inspect each record. Records are differentiated based on a delimiter (e.g., line records use the `\n` delimiter). The user then loops over records in `BUF` and checks if the record contains a keyword. Every matched record can then undergo further processing. The process is then repeated on new data that is read into `BUF`, which can be the next (sequential) or previous (reverse sequential) chunk.

With the `.filter` directive, the user specifies the delimiter as the first argument followed by the keyword. Internally, HSL performs the filtering over `BUF`. To support the use case of "further processing" of matched records, the user specifies a callback function (`callback_fn`) which is invoked on every matched record. Currently, the callback function has the

signature `bool fn(char *buf, size_t len, void *ctx);` which accepts the matched record and the size of the record. The context variable `ctx` is user-specified context which is also bound to HSL as seen in fourth parameter. Finally, the fifth parameter determines the direction in which the process repeats (`SEQ` or `RSEQ`). If the callback function returns `true`, HSL repeats the process; on `false` HSL stops execution. If a callback function always returns `true`, HSL runs until the end (or beginning) of the file depending on access direction.

(2) `.filter_endpoint s3fs http://url/to/aws/lambda`
In addition to (1), a user may also specify a filter endpoint so that if the filter workload is running on the file system matching the first argument (s3fs here), the endpoint can be invoked. For s3fs, we assume a http endpoint to a lambda function but the HSL backend can be extended to run any custom function that takes the second argument as a parameter. The lambda must be configured to accept the delimitter and keyword similar to (1). Unlike (1), it must also accept a starting offset and the size of `BUF` (the amount of data to transfer). As the lambda endpoint returns all matching records up until `BUF` is full and discards any non-matching records, it also returns the next offset that it should resume reading from. HSL, with `BUF` full of matching records, calls the callback function on each record.

A.2 WRITE

HSL's `WRITE` verb follows the same format as `READ` except that there are no hints or directives currently associated with writes.

A.3 APPEND

Similar to `WRITE`, HSL also contains a dedicated `APPEND` verb.

(1) `APPEND fd, [buf, sz]@records;`
As appends follow a particular order, they are only allowed with ordered collections.

A.4 FSYNC

(1) `FSYNC fd;`
Invokes `fsync()` on the file descriptor. When running on file systems that do not have a safe file flush (e.g., fsync on a newly created file does not flush directory entries), the backend must also call `fsync` on the parent directory.

(2) `FSYNC [fd]@fileset;`
Invokes `fsync` on multiple file descriptors in the given order as specified by the ordered collection. Each `fsync` is handled similar to (1).

(3) `FSYNC {fd}@fileset;`
Invokes `fsync` on multiple file descriptors without concern for ordering. Each `fsync` is handled similar to (1).

A.5 COPY

(1) `COPY src_fd, dst_fd;`
Performs a full file copy from a given source to destination; a single large contiguous range.

(2) `COPY src_fd, dst_fd, src_offset, dst_offset, size;`
Copies `size` bytes from the source file at the source offset (src_offset) to the destination file at its destination offset (dst_offset); a single large contiguous range.

(3) `COPY src_fd, dst_fd, {src_offset, dst_offset, size}@ranges;`
Performs many scattered copies from source file to destination file given the unordered collection named "ranges". HSL copies `ranges.size[i]` bytes from `ranges.src_offset[i]` in the source file to `ranges.dst_offset[i]` in the destination file.

A.6 REDOLOG

Our implementation of redo logs involves two files, a redo log and the base file on which multi-chunk updates must be performed atomically. As our implementation of redo logs is for physical redo logging, existing applications that use physical redo logging need only change the way they append, read, and checkpoint their write-ahead logs without any complex changes to on-disk layout.

We describe each verb related to redo logs and how they may change when dealing with multiple basefiles (for multi-file and multi-chunk atomic updates).

(1) `REDOLOG_OPEN rfd, basefile, chunksize;`
If a redo log does not already exist, HSL creates a new redo log and writes metadata headers to remember which basefile it is associated with and the size of the chunks. HSL stores a "redolog file descriptor" in `rfd`, which is not a conventional Linux file descriptor but an opaque HSL id that represents the redo log abstraction.

To handle multiple files, a collection of basefiles and basefile ids can be passed; we still assume a fixed chunk size. The basefile ids are required so that future operations on chunks use chunk ids and basefile ids to identify the file and location within the file instead of using the basefile path. The internal metadata headers must be modified to remember all basefiles and their ids. HSL will still return a single opaque HSL id which is stored in `rfd`.

(2) `REDOLOG_READ rfd, {id, buffer}@chunks;`
Reads an unordered collection of chunks, preferring latest stable data from the redo log. The data for `chunks.id[i]` will be stored in `chunks.buffer[i]`.

For multiple files, the collection will also include a file id.

(3) `REDOLOG_APPEND rfd, {id, buffer}@chunks;`
Appends an unordered collection of chunks to the redo log. The collection *can* be ordered but it is unnecessary due to the atomicity guarantees HSL provides over the entire collection.

Similar to (2), the collection will also include a file id for multiple files.

(4) `REDOLOG_SYNC rfd;`
Attempts to make all appended chunks stable using an update protocol that involves `fsync()`. On success, future `REDOLOG_READ`s must **always** return the latest content. On failure, future `REDOLOG_READ`s must **never** return any content that was part of the appends before the sync and must continue to return the stable content before the sync.

The redo log internally knows the basefile or basefiles it refers to. No changes are necessary to support multiple basefiles here.

(5) `REDOLOG_APPLY rfd;`
Performs the equivalent of checkpointing for the redo log, writing the latest stable chunks to their location in the basefiles. Regardless of success or failure, this operation must ensure the guarantees of (4) still hold.

Similar to (4), no changes are necessary to support multiple basefiles as the metadata contains the required information.

A.7 MANIFEST

Unlike redo logs, our implementation of manifests cannot be drop-in replacements in applications that use manifests. Applications like LevelDB

and RocksDB have custom on-disk formats for manifest files. LevelDB manifest files map a level to an ordered list of `sst` files.

As HSL manifests use a generic map container of string keys and string values, a LevelDB manifest with 3 files in a level would have 3 keys for that level in HSL. For example, instead of `Level 3:a.sst,b.sst,c.sst`, LevelDB would need to insert 3 key-value pairs: `L3i0:a.sst`, `L3i1:b.sst`, and `L3i2:c.sst`.

(1) `MANIFEST_OPEN mfd, path;`
Opens or creates a new manifest at the location `path`. Similar to redo logs, stores an opaque HSL id in `mfd`, the manifest file descriptor. Like LevelDB's CURRENT file, path is the root pointer to a manifest file (an on-disk mapping of keys to values). Unlike LevelDB, it can have any name the user wishes to provide.

(2) `MANIFEST_ADD mfd, key, path;`
Looks up the given `mfd` and updates the manifest's map with the given key and path. The path need not exist just yet but must exist on commit.

(3) `MANIFEST_REMOVE mfd, key;`
Similar to (2), but removes the key from the map and therefore does not require a path.

(4) `MANIFEST_COMMIT mfd;`
Performs the update protocol to safely persist a modified in-memory manifest map to disk. On success, future readers must **always** use the new manifest map. On failure, future readers must **never** read any of the content from the new manifest and must continue to access the previous content before the commit.

(5) `MANIFEST_KEYS mfd, num_keys, keys;`
Stores the number of keys present in the manifest in `num_keys`, and the keys themselves in `keys` which must be a vector of strings.

(6) `MANIFEST_READ mfd, key, fd;`

Stores a read-only file descriptor for the file associated with the given `key` into `fd`. Users can then use HSL or their own custom logic to read data from `fd`.

B
Evolving and Extending HSL

The current implementation of HSL is approximately 4500 lines of C/C++ code excluding tests and build tool configurations. HSL currently focuses on the characteristics of three local file systems (ext4, XFS, and Btrfs) and two remote file systems (s3fs and gcsfs), and provides a single implementation for physical redo logging and manifest files. Additionally, the performance crossover points for different interfaces may differ when HSL is run on different hardware, operating systems, or versions of the same operating system.

Here, we discuss the steps to take in evolving HSL for different hardware and versions, and also in extending HSL to support new file systems or custom intentions.

B.1 Evolving HSL

The primary concern when running HSL on unsupported file systems is the confidence in redo logging or manifests adhering to their correctness semantics. The implementations of REDOLOG and MANIFEST issue warnings when running on unsupported file systems so users are aware of the risks. However, they do fall back to regular conventional methods so users can still run them as they do today with the risk of data loss. For

example, we do not issue a double `fsync` as we do not know if the file system requires it; maybe it requires three `fsync` calls.

A secondary concern when running HSL on supported file systems but on different OS versions or hardware is whether portable efficiency is still guaranteed. The current performance crossover points were measured on Linux v6.5.7 on a Cloudlab c220g2 machine using an SSD. During installation, HSL has some defaults from the above configuration as a base. However, we plan to provide another tool `HSL-profile` which can be run at any time post installation to identify properties of the current system. The same experiments we run to identify the performance crossover points will be packaged as workloads to be profiled as part of `HSL-profile`.

Additionally, similar to profile-guided optimization, the HSL Runtime in each HSL-enabled application can log activity which can be used by `HSL-profile` to determine which file systems and underlying devices need to be profiled. While not yet implemented, the results can be stored as part of a configuration that HSL reads when the runtime within an application is initialized.

B.2 Extending HSL for New File Systems

While HSL officially supports the three file systems we studied, the default system call backend will work for existing POSIX compliant or compatible file systems. However, like with applications that do not use HSL, using unsupported file systems can cause data loss or limit performance. Supporting a new file system requires adding a backend implementation for each existing verb minimally in the syscall transport.

Similar to the Linux Kernel's virtual file system layer (vfs), a developer (who is hopefully an owner or maintainer of the file system) also writes a HSL backend. An example file system (exfs) would have a file `exfs-hsl-syscall.cc` that internally registers the handlers for each verb,

similar to how vfs has handlers for certain operations; any verb that does not have a registered handler will use the default POSIX compliant system call. If the developer wishes to support the iouring transport, another file—`exfs-hsl-iouring.cc`—must be provided. Similarly, kernel-bypass file systems that use shared memory can create `exfs-hsl-shm.cc`.

When initializing a transport, the developer has the option of making it private or public. For example, a public iouring transport can be used when verbs operate on other file systems within the application. However, a private shm transport is specific to a file system (not just in type but the instance mount point).

Note that the developer does not need to decide between multiple existing transports, the HSL profiler will discover performance related switches if multiple are available. However, the developer must decide on which is the best system calls to execute the verb and its arguments. As the backend is executed at runtime, the code can query properties from HSL runtime as well as the size of collections or cache properties to decide the right system calls. While not yet implemented, we plan to provide a library with helper functions to access HSL runtime information.

While the above addresses portable efficiency, it does not handle correctness. To avoid re-implementations of redo logs and manifest files, each file system must provide "traits" at run time similar to the behavior inferences studied in Chapter 3; e.g., whether a page is clean or dirty after fsync failure, and whether error reporting is immediate or delayed. HSL will query traits at run time for a given file descriptor as the same file system may have different behavior depending on its mount options (e.g., ext4 ordered and data mode). The implementations of complex intentions such as redo logs and manifest files will use traits to decide post-failure behavior. If a file system does not expose the traits required by a verb, and if an application uses that verb on the file system, HSL will issue warnings so users are aware of the risks.

In addition to backend implementations for HSL verbs, the developer must also register handlers for certain bookkeeping operations such as "checkpointing" so that HSL can trigger a checkpoint if required (e.g., for manifest files if directory entries can disappear). As more intentions are added, there may be more bookkeeping operations to be registered as handlers. If a verb requires a handler that is missing and is required for correctness, HSL will issue warnings so users are aware of the risks.

B.3 Extending HSL for Custom Operations

One of the benefits of HSL as a language instead of a library is the ability to identify intentions retrospectively and add new operations without modifying existing application code. The choice of making a new verb public or ephemeral (only within HSL's middle-end and backend) is left to the developer or organization. A general guideline is to avoid making new verbs public, unless they apply to multiple file systems or is a new correctness related update protocol. If a verb is required for a very specific performance optimization, it does not need to pollute the frontend requiring developers to learn about a new "keyword".

A public verb has an additional initial step which we cover first before describing with the remaining common procedure. To introduce a new verb, the HSL frontend language must be updated to detect the letters as a verb. Similar to registering a backend, the verb is registered with the HSL frontend using `HSL::Frontend::RegisterVerb`. The function takes the new verb as a string as well as a callback function and stores them in a map. Internally, when HSL detects a non-standard verb, it checks the map and calls the callback function. The callback function must accept a single parameter that represents a parsed argument list. Internally, the callback function can query the number of arguments and the type of each argument and return a success code if everything is as intended. A failure

return code fails the frontend parsing of a HSL script, similar to compiler syntax errors. In addition to the success code, the function also returns the handler functions to execute the verb in the backend (similar to file systems registering handlers for verbs).

Even if a verb is public, unmodified applications can benefit from it if the developer writes a middle-end pass that transforms a sequence of existing verbs. The developer can register a new pass that operates on basic blocks within HSL scripts. If a sequence of HSL statements (verbs and their arguments) match certain criteria, the developer can replace them with a public verb or an ephemeral one. For ephemeral verbs, the handler functions should be provided here; public verbs would have done so in the frontend. As the transformation is performed using HSL helper functions, HSL remembers the existing statements that comprise the new verb which can be used as a fallback if a backend implementation does not exist. Alternatively, the developer can inject runtime predicates (e.g., if file system for fd matches a specific file system), and then apply the transformation, leading to two new basic blocks.

The final part in introducing a verb is to provide an implementation for it which is passed to HSL either at the frontend or middle-end as mentioned above. If no implementation is provided, ephemeral verbs fall back to the pre-transformed version. Note that HSL does not have to revert transformations. As passes and implementations are registered during initialization before a script is executed, HSL simply does not apply the transformation.

To avoid having all other file systems also support the custom verb in the backend, the developer should provide a generic implementation. Internally, by querying runtime information, the implementation can write file-system specific code to use special custom interfaces when possible.

Bibliography

[1] Adobe. Best audio format file types. `https://www.adobe.com/cl/creativecloud/video/discover/best-audio-format.html`. Accessed: 2024-07-30.

[2] Ibrahim Umit Akgun, Ali Selman Aydin, Andrew Burford, Michael McNeill, Michael Arkhangelskiy, and Erez Zadok. Improving storage systems using machine learning. *ACM Trans. Storage*, 19(1), January 2023.

[3] Remzi Arpaci-Dusseau. Measure, then build. `https://www.usenix.org/conference/atc19/presentation/keynote`, jul 2019. Accessed: 2024-07-30.

[4] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 1.00 edition, August 2018.

[5] Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh. Posix abstractions in modern operating systems: the old, the new, and the missing. In *Proceedings of the EuroSys Conference* (*EuroSys '16*), London, United Kingdom, April 2016.

[6] A. Avizienis. The n-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, Dec 1985.

[7] Jens Axboe. Efficient IO with io_uring. `https://kernel.dk/io_uring.pdf`. Accessed: 2024-07-30.

[8] Mojtaba Bagherzadeh, Nafiseh Kahani, Cor-Paul Bezemer, Ahmed E. Hassan, Juergen Dingel, and James R. Cordy. Analyzing a decade of linux system calls. *Empirical Software Engineering*, 23(3):1519–1551, Jun 2018.

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

[10] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *Proceedings of the 2007 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (*SIGMETRICS '07*), pages 289–300, San Diego, CA, June 2007.

[11] Lakshmi N. Bairavasundaram, Meenali Rungta, Nitin Agrawal, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Analyzing the Effects of Disk-Pointer Corruption. In *Proceedings of the International Conference on Dependable Systems and Networks* (*DSN '08*), pages 502–511, Anchorage, Alaska, June 2008.

[12] Lakshmi N. Bairavasundaram, Swaminathan Sundararaman, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Tolerating File-System Mistakes with EnvyFS. In *To appear in the Proceedings*

of the Usenix Annual Technical Conference (*USENIX '09*), San Diego, California, June 2009.

[13] Jinrui Cao, Om Rameshwar Gatla, Mai Zheng, Dong Dai, Vidya Eswarappa, Yan Mu, and Yong Chen. PFault: A General Framework for Analyzing the Reliability of High-Performance Parallel File Systems. In *Proceedings of the 2018 International Conference on Supercomputing*, pages 1–11, Beijing, China, June 2018.

[14] Ming Chen, Dean Hildebrand, Henry Nelson, Jasmit Saluja, Ashok Sankar Harihara Subramony, and Erez Zadok. vNFS: Maximizing NFS Performance with Compounds and Vectorized I/O. In *15th USENIX Conference on File and Storage Technologies* (*FAST 17*), pages 301–314, 2017.

[15] 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*), pages 228–243, Farmington, PA, November 2013.

[16] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 228–243, New York, NY, USA, November 2013. Association for Computing Machinery.

[17] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Consistency Without Ordering. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies* (*FAST '12*), pages 101–116, San Jose, CA, February 2012.

[18] Howard Chu. LMDB: Lightning Memory-Mapped Database Manager. http://www.lmdb.tech/doc/. Accessed: 2024-07-30.

[19] Copy_file_range(2) - Linux manual page. https://man7.org/linux/man-pages/man2/copy_file_range.2.html. Accessed: 2024-07-30.

[20] FUSE file system to emulate different file-system failure reactions: CuttleFS. https://github.com/WiscADSL/cuttlefs, 2020. Accessed: 2024-07-30.

[21] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. Understanding modern storage apis: A systematic study of libaio, spdk, and io_uring. In *Proceedings of the 15th ACM International Conference on Systems and Storage*, SYSTOR '22, pages 120–127, New York, NY, USA, 2022. Association for Computing Machinery.

[22] DiLorenzo, Jonathan and Mancini, Katie and Fisher, Kathleen and Foster, Nate. TxForest: A DSL for Concurrent Filestores. In Lin, Anthony Widjaja, editor, *Programming Languages and Systems*, APLAS '19, pages 332–354, Cham, 2019. Springer International Publishing.

[23] Why does ext4 clear the dirty bit on I/O error? https://www.postgresql.org/message-id/edc2e4d5-5446-e0db-25da-66db6c020cc3%40commandprompt.com, 2020. Accessed: 2024-07-30.

[24] Device Mapper: dm-flakey. https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/dm-flakey.html, 2020. Accessed: 2024-07-30.

[25] Custom Fault Injection Device Mapper Target: dm-loki. https://github.com/WiscADSL/dm-loki, 2020. Accessed: 2024-07-30.

[26] Man Pages: dmsetup. `https://man7.org/linux/man-pages/man8/dmsetup.8.html`, 2020. Accessed: 2024-07-30.

[27] Viacheslav Dubeyko, Om Rameshwar Gatla, and Mai Zheng. Nature of system calls in cpu-centric computing paradigm. *CoRR*, abs/1903.04075, 2019.

[28] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The Design and Operation of CloudLab. In *2019 USENIX Annual Technical Conference* (*USENIX ATC 19*), pages 1–14, Renton, WA, July 2019.

[29] Tânia Esteves, Ricardo Macedo, Rui Oliveira, and João Paulo. Diagnosing applications' i/o behavior through system call observability. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops* (*DSN-W*), pages 1–8, Porto, Portugal, 2023.

[30] Is data=journal safer for Ext4 as opposed to data=ordered? `https://unix.stackexchange.com/q/127235`, 2020. Accessed: 2024-07-30.

[31] Kathleen Fisher, Nate Foster, David Walker, and Kenny Q. Zhu. Forest: a language and toolkit for programming with filestores. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, page 292–306, New York, NY, USA, 2011. Association for Computing Machinery.

[32] Kathleen Fisher and Robert Gruber. Pads: a domain-specific language for processing ad hoc data. In *Proceedings of the 2005 ACM*

SIGPLAN Conference on Programming Language Design and Implementation, PLDI '05, page 295–304, New York, NY, USA, 2005. Association for Computing Machinery.

[33] Christian Forfang. Evaluation of High Performance Key-Value Stores. Master's thesis, Norwegian University of Science and Technology, June 2014.

[34] FreeBSD VFS Layer re-dirties pages after failed block write. `https://github.com/freebsd/freebsd/blob/0209fe3398be56e5e042c422a96a4fbc654247f4/sys/kern/vfs_bio.c#L2646`, 2020. Accessed: 2024-07-30.

[35] Christopher Frost, Mike Mammarella, Eddie Kohler, Andrew de los Reyes, Shant Hovsepian, Andrew Matsuoka, and Lei Zhang. Generalized file system dependencies. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 307–320, New York, NY, USA, October 2007. Association for Computing Machinery.

[36] fsync(2) - Linux Programmer's Manual. `http://man7.org/linux/man-pages/man2/fdatasync.2.html`, 2020. Accessed: 2024-07-30.

[37] Explicit volatile write back cache control — The Linux Kernel documentation. `https://docs.kernel.org/block/writeback_cache_control.html#explicit-cache-flushes`. Accessed: 2024-07-30.

[38] PostgreSQL's handling of fsync() errors is unsafe and risks data loss at least on XFS . `https://www.postgresql.org/message-id/flat/CAMsr%2BYHh%2B5Oq4xziwwoEfhoTZgr07vdGG%2Bhu%3D1adXx59aTeaoQ%40mail.gmail.com`, 2020. Accessed: 2024-07-30.

[39] Fsync Errors - PostgreSQL wiki. `https://wiki.postgresql.org/wiki/Fsync_Errors`, 2020. Accessed: 2024-07-30.

[40] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies* (*FAST '17*), pages 149–165, Santa Clara, CA, February 2017.

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

[42] Gcc front ends. `https://gcc.gnu.org/frontends.html`, 2022.

[43] Gcc plugins. `https://gcc.gnu.org/wiki/plugins`, 2018.

[44] Thread-local storage. `https://gcc.gnu.org/onlinedocs/gcc/Thread-Local.html`, 2022.

[45] Google Cloud Storage FUSE. `https://cloud.google.com/storage/docs/gcsfuse-install`. Accessed: 2024-07-30.

[46] Google Cloud Storage. `https://cloud.google.com/storage/docs`. Accessed: 2024-07-30.

[47] Alan Grosskurth and Michael W Godfrey. Architecture and evolution of the modern web browser. *University of Waterloo in Canada*, 24, 2006.

[48] Haryadi S. Gunawi, Vijayan Prabhakaran, Swetha Krishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving File System Reliability with I/O Shepherding. In *Proceedings of the*

21st ACM Symposium on Operating Systems Principles (*SOSP '07*), pages 293–306, Stevenson, WA, October 2007.

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

[50] Robert Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles* (*SOSP '87*), pages 155–162, Austin, Texas, November 1987.

[51] 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. *ACM Trans. Comput. Syst.*, 30(3), aug 2012.

[52] Amazon Media Hotline. Amazon Web Services Launches S3. `https://press.aboutamazon.com/2006/3/amazon-web-services-launches`, March 2006. Accessed: 2024-07-30.

[53] FUSE (Filesystem in Userspace). The reference implementation of the Linux FUSE (Filesystem in Userspace) interface. `https://github.com/libfuse/libfuse`, 2020. Accessed: 2024-07-30.

[54] Shehbaz Jaffer, Stathis Maneas, Andy Hwang, and Bianca Schroeder. Evaluating File System Reliability on Solid State Drives. In *2019 USENIX Annual Technical Conference* (*USENIX ATC 19*), pages 783–797, Renton, WA, July 2019.

[55] Samuel Kalbfleisch, Lukas Werling, and Frank Bellosa. Vinter: Automatic Non-Volatile memory crash consistency testing for full systems. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 933–950, Carlsbad, CA, July 2022.

[56] Teemu Kanstrén. Merkle Trees: Concepts and Use Cases. `https://medium.com/coinmonks/merkle-trees-concepts-and-use-cases-5da873702318`, February 2021. Accessed: 2024-07-30.

[57] Hannu H. Kari. *Latent Sector Faults and Reliability of Disk Arrays*. PhD thesis, Helsinki University of Technology, September 1997.

[58] The kernel development community. File systems in the linux kernel. `https://docs.kernel.org/filesystems/index.html`. Accessed: 2024-07-30.

[59] The kernel development community. syscall(2) - linux manual page. `https://man7.org/linux/man-pages/man2/syscall.2.html`. Accessed: 2024-07-30.

[60] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Summer. One-level Storage System. *IRE Transactions on Electronic Computers*, EC-11:223–235, April 1962.

[61] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, and Jupyter development team. Jupyter notebooks - a publishing format for reproducible computational workflows. In Fernando Loizides and Birgit Scmidt, editors, *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87–90, Netherlands, 2016. IOS Press.

[62] Nodir Kodirov and RJ Sumi. Study on the file system calls of desktop applications. Technical report, University of British Columbia, 2021.

[63] Brenden Kokoszka, Patrick Donnelly, and Douglas Thain. Search Should Be a System Call. Technical report, University of Notre Dame, 2013.

[64] Philip Koopman and John DeVale. Comparing the Robustness of POSIX Operating Systems. In *Proceedings of the 29th International Symposium on Fault-Tolerant Computing (FTCS-29)*, Madison, Wisconsin, June 1999.

[65] Andrew Krioukov, Lakshmi N. Bairavasundaram, Garth R. Goodson, Kiran Srinivasan, Randy Thelen, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Parity Lost and Parity Regained. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, pages 127–141, San Jose, CA, February 2008.

[66] Configure lambda function timeout. `https://docs.aws.amazon.com/lambda/latest/dg/configuration-timeout.html`, 2024. Accessed: 2024-07-30.

[67] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, March 2004. IEEE Computer Society.

[68] LevelDB. `https://github.com/google/leveldb`, 2020. Accessed: 2024-07-30.

[69] Lightning Memory-Mapped Database Manager (LMDB). `http://www.lmdb.tech/doc/`, 2020. Accessed: 2024-07-30.

[70] Man Pages: losetup. https://man7.org/linux/man-pages/man8/losetup.8.html, 2020. Accessed: 2024-07-30.

[71] Improved block-layer error handling. https://lwn.net/Articles/724307/, 2020. Accessed: 2024-07-30.

[72] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, volume 2, pages 21–33. Citeseer, 2007.

[73] Avantika Mathur, Mingming Cao, and Andreas Dilger. Ext4: The Next Generation of the Ext3 File System. *Usenix Association*, 32(3):25–30, June 2007.

[74] Jeffrey C. Mogul. A Better Update Policy. In *Proceedings of the USENIX Summer Technical Conference* (*USENIX Summer '94*), pages 99–111, Boston, MA, June 1994.

[75] 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 Symposium on Operating Systems Design and Implementation* (*OSDI '18*), pages 33–50, Carlsbad, CA, October 2018.

[76] WT-4045 Don't retry fsync calls after EIO failure. https://github.com/wiredtiger/wiredtiger/commit/ae8bccce3d8a8248afa0e4e0cf67674a43dede96, 2020. Accessed: 2024-07-30.

[77] MySQL 8.4 C API Developer Guide. https://dev.mysql.com/doc/c-api/8.4/en/c-api-function-reference.html. Accessed: 2024-07-30.

[78] Bug-27805553 HARD ERROR SHOULD BE REPORTED WHEN FSYNC() RETURN EIO. https://github.com/mysql/mysql-server/commit/8590c8e12a3374eeccb547359750a9d2a128fa6a, 2020. Accessed: 2024-07-30.

[79] Microsoft Developer Network. Transactional NTFS (txf). https://learn.microsoft.com/en-us/windows/win32/fileio/transactional-ntfs-portal, 2022. Accessed: 2024-07-30.

[80] Bug-207729 Mounting EXT4 with data_err=abort does not abort journal on data block write failure. https://bugzilla.kernel.org/show_bug.cgi?id=207729, 2020. Accessed: 2024-07-30.

[81] Thanumalayan Sankaranarayana Pillai, Ramnatthan Alagappan, Lanyue Lu, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Application Crash Consistency and Performance with CCFS. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies* (*FAST '17*), pages 181–196, Santa Clara, CA, February 2017.

[82] 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*), pages 433–448, Broomfield, CO, October 2014.

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

[84] POSIX Specification for fsync. `https://pubs.opengroup.org/onlinepubs/9699919799/functions/fsync.html`, 2020. Accessed: 2024-07-30.

[85] PostgreSQL. `https://www.postgresql.org/`, 2020. Accessed: 2024-07-30.

[86] PostgreSQL: Write-Ahead Logging (WAL). `https://www.postgresql.org/docs/current/wal-intro.html`, 2020. Accessed: 2024-07-30.

[87] 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.

[88] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Model-Based Failure Analysis of Journaling File Systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '05)*, pages 802–811, Yokohama, Japan, June 2005.

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

[90] Mohan Rajagopalan, Saumya K Debray, Matti A Hiltunen, Richard D Schlichting, T Labs-Research, Park Avenue, and Florham Park. System Call Clustering: A Profile-Directed Optimization Technique. Technical report, The University of Arizona, 2002.

[91] Mohan Rajagopalan, Saumya K Debray, Matti A Hiltunen, Richard D Schlichting, T Labs-Research, Park Avenue, and Florham Park. Cassyopia: Compiler Assisted System Optimization. In *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, pages 103–108, Lihue, Hawaii, May 2003. USENIX Association.

[92] Maria Ramos, João Azevedo, Kyle Kingsbury, José Pereira, Tânia Esteves, Ricardo Macedo, and João Paulo. When amnesia strikes: Understanding and reproducing data loss bugs with fault injection. *Proc. VLDB Endow.*, 17(11):3017–3030, August 2024.

[93] Anthony Rebello, Yuvraj Patel, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Can applications recover from fsync failures? *ACM Transactions on Storage (TOS)*, 17(2):1–30, June 2021.

[94] Redis. `https://redis.io/`, 2020. Accessed: 2024-07-30.

[95] Redis Persistence. `https://redis.io/topics/persistence`, 2020. Accessed: 2024-07-30.

[96] 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 '13)*, page 38–53, Monterey, CA, October 2015.

[97] RIFF file format. `https://www.daubnet.com/en/file-format-riff#:~:text=Structure%20of%20the%20%27-JUNK-%27%20chunk`. Accessed: 2024-07-30.

[98] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-Tree Filesystem. *ACM Transactions on Storage*, 9(3):1–32, August 2013.

[99] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-Tree Filesystem. *ACM Transactions on Storage* (*TOS*), 9(3):1–32, August 2013.

[100] Stefan Roesch. [PATCH v2 00/13] Support sync buffered writes for io-uring. `https://lore.kernel.org/all/6310fba5-6bbf-8488-1096-0eb2d8574583@fb.com/T/#m220c3f1a3246243ed39b7724b6e13d9393d66ecc`. Accessed: 2024-07-30.

[101] Stefan Roesch. [PATCH v9 00/14] io-uring/xfs: Support async buffered writes. `https://lore.kernel.org/io-uring/20220616212221.2024518-1-shr@fb.com/`. Accessed: 2024-07-30.

[102] FUSE-based file system backed by Amazon S3. s3fs-fuse, June 2024. Accessed: 2024-07-30.

[103] Bianca Schroeder, Sotirios Damouras, and Phillipa Gill. Understanding Latent Sector Errors and How to Protect Against Them. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies* (*FAST '10*), pages 71–84, San Jose, CA, February 2010.

[104] Timos K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems*, 13(1):23–52, March 1988.

[105] Margo Seltzer, Peter Chen, and John Ousterhout. Disk Scheduling Revisited. In *Proceedings of the Winter 1990 USENIX Conference*, pages 313–323, Washington, D.C., January 1990.

[106] Sendfile(2) - Linux manual page. `https://man7.org/linux/man-pages/man2/sendfile.2.html`. Accessed: 2024-07-30.

[107] Chuck Silvers. UBC: An Efficient Unified I/O and Memory Caching Subsystem for NetBSD. In *Proceedings of FREENIX Track: 2000*

USENIX Annual Technical Conference, pages 285–290, San Diego, CA, June 2000.

[108] 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*), pages 33–46, Vancouver, Canada, December 2010.

[109] Atomic Commit In SQLite. `https://www.sqlite.org/atomiccommit.html`, 2020. Accessed: 2024-07-30.

[110] SQLite Write-Ahead Logging. `https://www.sqlite.org/wal.html`, 2020. Accessed: 2024-07-30.

[111] David C. Steere. Exploiting the Non-Determinism and Asynchrony of Set Iterators to Reduce Aggregate File I/O Latency. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 252–263, St. Malo, France, October 1997. ACM.

[112] Gordon Stewart, Mahanth Gowda, Geoffrey Mainland, Bozidar Radunovic, Dimitrios Vytiniotis, and Cristina Luengo Agullo. Ziria: A dsl for wireless systems programming. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems* (*ASPLOS '15*), page 415–428, Istanbul, Turkey, March 2015.

[113] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS file system. In *USENIX 1996 Annual Technical Conference* (*USENIX ATC 96*), ATEC '96, page 1, San Diego, CA, January 1996. USENIX Association.

[114] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS File System. In

Proceedings of the USENIX 1996 Annual Technical Conference, San Diego, CA, January 1996.

[115] SystemTap. `https://sourceware.org/systemtap/`, 2020. Accessed: 2024-07-30.

[116] SQLite Team. An Introduction To The SQLite C/C++ Interface. `https://www.sqlite.org/cintro.html`. Accessed: 2024-07-30.

[117] SQLite Team. SQLite. `https://www.sqlite.org/index.html`. Accessed: 2024-07-30.

[118] SQLite Team. The SQLite Bytecode Engine. `https://www.sqlite.org/opcode.html`. Accessed: 2024-07-30.

[119] Philippe Tillet, H. T. Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, page 10–19, New York, NY, USA, 2019. Association for Computing Machinery.

[120] Elf handling for thread-local storage. `https://www.akkadia.org/drepper/tls.pdf`, 2013.

[121] 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 Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[122] Rajat Verma, Anton Ajay Mendez, Stan Park, Sandya Srivilliputtur Mannarswamy, Terence P. Kelly, and Charles B. Morrey III. Failure-Atomic updates of application data in a linux file system. In *Proceedings of the 13th USENIX Symposium on File and Storage Technologies (FAST '15)*, pages 203–211, Santa Clara, CA, February 2015.

[123] Elder Vicente, Rivalino Matias, Lúcio Borges, and Autran Macêdo. Evaluation of compound system calls in the Linux kernel. *ACM SIGOPS Operating Systems Review*, 46(1):53–63, February 2012.

[124] Tomas Vondra. PostgreSQL vs. fsync. How is it possible that PostgreSQL used fsync incorrectly for 20 years, and what we'll do about it. In *Free and Open source Software Developers' European Meeting*, Brussels, Belgium, February 2019. `https://archive.fosdem.org/2019/schedule/event/postgresql_fsync/`.

[125] WAVE Audio File Format. `https://www.loc.gov/preservation/digital/formats/fdd/fdd000001.shtml`, April 2024. Accessed: 2024-07-30.

[126] Youjip Won, Jaemin Jung, Gyeongyeol Choi, Joontaek Oh, Seongbae Son, Jooyoung Hwang, and Sangyeun Cho. Barrier-Enabled IO Stack for Flash Storage. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*, pages 211–226, Oakland, CA, February 2018.

[127] FSQA (xfstests). `https://git.kernel.org/pub/scm/fs/xfs/xfstests-dev.git/about/`, 2020. Accessed: 2024-07-30.

[128] Jian Zhang, Yujie Ren, and Sudarsun Kannan. FusionFS: Fusing I/O Operations using CISCOps in Firmware File Systems. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 297–312, Santa Clara, CA, February 2022. USENIX Association.

[129] Shuanglong Zhang, Helen Catanese, and Andy An-I Wang. The composite-file file system: Decoupling the One-to-One mapping of files and metadata for better performance. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*, pages 15–22, Santa Clara, CA, February 2016.

[130] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and 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)*, pages 29–42, San Jose, CA, February 2010.

[131] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. XRP: In-Kernel Storage Functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 375–393, Carlsbad, CA, July 2022.

[132] Zhe Zhou, Yanxiang Bi, Junpeng Wan, Yangfan Zhou, and Zhou Li. Userspace Bypass: Accelerating Syscall-intensive Applications. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 33–49, Bostan, MA, July 2023.