The Dissertation Committee for Hayley LeBlanc
certifies that this is the approved version of the following dissertation:

Robust Storage Systems for Persistent Memory

Committee:

Vijay Chidambaram, Supervisor

James Bornholt

Rajeev Joshi

Jacob R. Lorch

Dixin Tang

Emmett Witchel

Robust Storage Systems for Persistent Memory

by

Hayley LeBlanc

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May 2025

Dedication

To Paul.

Acknowledgments

There are quite a few people without whom this dissertation would not have been possible. I first want to thank my advisor, Vijay Chidambaram. Vijay is an exceptional mentor in all areas of research. He taught me how to plan out and see the big picture behind a project, and how to stay on track and not get distracted by minutia. From Vijay, I learned to always dig deeper until I really understood how a system worked or why our results looked a certain way. I have always had a tendency to work alone, but Vijay showed me that the best research comes from working with others in the community, and he always encouraged me to work hard at establishing those connections. Vijay is also an excellent speaker and writer, and I learned a lot from him about how to share my research with the community. He taught me that each paper and talk should tell a story, that the high-level insights are more important than the low-level details, and that every sentence must help tell that story. I could not have asked for a better advisor, and I am incredibly lucky to have worked with Vijay over the past five years.

I have had many other academic mentors over the course of my PhD. My internship mentors, Rajeev Joshi at Amazon Web Services and Jay Lorch at Microsoft Research, both had an immense impact on my work and who I am as a researcher. Somehow, I managed to convince both of them to hire me in the spring, quite late in the internship application cycle, and I am very, very lucky that they did. Rajeev introduced me to the idea that formal methods and correctness are not all-or-nothing and taught me Rust, two things that played a significant role in the trajectory of my research. He perfectly balanced the amount of hands-on feedback that I needed to make progress early in my PhD with the freedom to learn how to solve problems independently. Rajeev's constant support and encouragement strengthened my confidence in myself as a researcher, which I especially needed in the first few years of my PhD. I'd also like to thank my other mentors at AWS for their advice and support:

Rustan Leino, Andy Warfield, Xi Wang, Luke Nelson, and the members of the S3 Automated Reasoning Group.

Jay introduced me to the world (and community) of systems verification, and I could not have asked for a better mentor and collaborator to learn about it from. His expertise was absolutely invaluable in achieving the goal of building a verified PM storage system as the last piece of my PhD. He created a research environment in which I felt comfortable being wrong, which I believe is critical for doing good research but something I have always struggled with. I'd also like to thank our other collaborators on the PoWER project — Chris Hawblitzel, Cheng Huang, Yiheng Tao, and Nickolai Zeldovich. Chris' feedback, advice, and insights about systems verification were crucial for the PoWER project. Cheng and Yiheng provided an invaluable production-side perspective to the project, which helped shape the design of CapybaraKV and taught me a lot about the practical considerations of building new systems. Nickolai contributed an astonishing amount to the project, despite joining only a few months before we submitted the paper, and I am very grateful for his help in strengthening the work. I am also grateful to Shan Lu, Ed Nightingale, and everyone in the MSR Systems Research Group for their support during my internships and during my job search — I'm very excited to join you all!

I am also grateful to James Bornholt, who was a mentor both at UT and during my internships at AWS. I learned much of what I know about formal methods from research, classes, and reading groups with James. His questions and feedback consistently guided my research in the right directions, and I am especially grateful to him for helping me realize that the ideas that eventually turned into SquirrelFS were worth pursuing.

I'd like to thank the remaining members of my PhD committee, Dixin Tang and Emmett Witchel, for their questions and support. Their feedback, both on this dissertation and on earlier papers, has both improved the work itself and has helped me grow into a stronger researcher.

I also want to thank my other collaborators at UT. I had the privilege of working with Isil Dillig earlier in my PhD, and her feedback and questions always helped me think deeper about research and how to understand the systems and techniques I was using better. I am also very grateful to her for inviting me to her lab's meetings and events, which helped me feel welcome at UT in the weird first few years of the COVID-19 pandemic. I'd like to thank the other UT students I collaborated with directly: Shankara Pailoor, Om Saran K R E, and Nathan Taylor. Shankara was the first other graduate student I worked with, and he was an excellent role model for how to communicate about projects and present results in meetings. I am also very grateful to Om for his excellent work on adding SplitFS support to Chipmunk; it wouldn't have been completed without him. Nathan is a well of systems knowledge and an excellent speaker, and I am very lucky to have worked with him on SquirrelFS. His fearless leadership of the systems and PL reading group also helped me become more confident in my ability to discuss research papers.

2020 was a very weird time to start a PhD, and I wish I had been able to spend more time with everyone at UT. I am incredibly grateful to my labmates in the UT SaSLab: Soujanya Ponnapalli, Rohan Kadekodi, Aashaka Shah, Sekwon Lee, Yeonju Ro, Sangamithra Goutham, and Om. I feel very lucky to have had joined a lab of folks who care so much for one another, even after graduation. Soujanya, thank you for always knowing when I needed support (even when I didn't know it) and being there without me having to ask. Rohan, thank you for your help with everything persistent-memory related and for being such an excellent instructor for Virtualization. To everyone in the lab, I'm grateful for the time we managed to find to spend together despite the pandemic, and I'm looking forward to seeing each other for many more years at work and at conferences. I have many others at UT to thank, for fun outings and research discussions and good conversation at GWGMC events — Akhil Jalan, Newton Ni, Aditya Tewari, Yingchen Wang, Charlotte LeMay, and Josh Hoffman.

I had many incredible professors and mentors before UT who helped me get

where I am today. At Denison, Thomas Bressoud introduced me to systems and helped me realize it was what I wanted to do for my PhD. May Mei taught me that I do enjoy math, taught me much of what I know about public speaking, and always gives very useful advice. Peter Grandbois, one of the most selfless people I know, showed me how to be a good leader. Back in high school, Hollee McNamee taught the first math class I ever enjoyed and gave me confidence to pursue it in college. Last but certainly not least, Jason Galbraith was my first computer science teacher and always encouraged me to keep going. I may have never gone into computer science if he had not suggested I take his Java programming course.

I'm immensely grateful to friends who helped me stay sane throughout my PhD, even though we mostly rarely saw each other in person. To Mattia Carbonaro, for going on five years of weekly movie night and zoo trips whenever I was in town. To Kyra Zagorski, for being the best roommate anyone could ask for and for many walks and conversations and delicious meals; I'm so glad we overlapped again in Seattle. To Alena Friedrich, for always visiting when you were in Texas and lots of fun hangouts and excellent food in Seattle. To James Horan, for Blaseball, movie night, and many more visits and holidays together.

My family has been incredibly supportive throughout this entire journey, and it would not have happened without them. I want to thank my extended family — Grandma and Grandpa, Grammy and Grampy, all LeBlancs and Jourises — for their love and the support and pride they showed about my accomplishments. To Sally and Michael, thank you for always welcoming me and for many wonderful holidays and visits. I'm pretty sure I've done some of my best research from your house. My sister Hannah has stuck by me since the beginning, and despite being in completely different disciplines, our PhD research has been similar enough to have deep, interesting conversations. I hope we eventually manage to pull off the collaboration we've been talking about for years! It is hard to express in words how grateful I am to my parents, Jaime and Darrell. They have unconditionally loved and supported me

throughout this entire process and have always made sure to be there for major milestones. Although both of them studied computer science, they never pushed me into it, but I ended up here anyway (and it's pretty nice to be able to talk about research with them!).

Finally, I have to thank my partner, Paul Bass. I truly could not have done this without him. Paul is my biggest cheerleader, my best friend, and one of the best computer scientists I have ever met. I cannot put into words how grateful I am to Paul for his support (including following me, twice, for my career) and love over the past nine years.

Abstract

Robust Storage Systems for Persistent Memory

Hayley LeBlanc, PhD
The University of Texas at Austin, 2025

SUPERVISOR: Vijay Chidambaram

Protecting the integrity of stored data is the main responsibility of storage systems. However, it is challenging to ensure that data is always kept safe in the event of crashes or data corruption, and there is no one-size-fits-all approach to building robust systems. The developers of different systems have different goals, requirements, and resources, and thus have different priorities when it comes to how they gain confidence in the correctness and robustness of their systems. In this dissertation, we present a set of new techniques for building crash-consistent and corruption-resistant systems. Each technique occupies a different point in the trade-off space between complexity and the level of confidence it offers, providing developers with a toolbox of approaches that are useful in a variety of settings.

The techniques presented in this dissertation focus on the problem of ensuring robustness in storage systems built for persistent memory (PM). Persistent memory (PM) technologies, such as Intel Optane DC Persistent Memory and battery-backed DRAM, promise low-latency, byte-addressable access to dense storage media. These characteristics present an opportunity to build new file and storage systems that provide both high performance and strong crash-consistency guarantees, but also introduce new challenges when it comes to building robust systems. Developers of PM storage systems must contend with a complex low-level interface and the need

to develop new designs to take advantage of new performance characteristics. Furthermore, PM systems differ in many fundamental ways from traditional systems, so many previously-developed tools and techniques are not compatible.

In this dissertation, we present three new techniques for ensuring that PM storage systems are robust in the face of crashes and corruption. We first present Chipmunk, a testing tool for PM file systems for crash consistency bugs. We analyze 23 bugs in five PM file systems found by Chipmunk and learn important lessons about how to design these systems and prevent such bugs. We next present SquirrelFS, a PM file system that utilizes the Rust compiler to statically check ordering-related crash-consistency properties. SquirrelFS checks these properties in a new crash-consistency mechanism we call Synchronous Soft Updates using an API design pattern called the typestate pattern. Finally, we present PoWER (Preconditions on Writes Enforcing Recoverability), a new approach to formally verifying crash-consistency, and use it to build CapybaraKV, a verified PM key-value store. Unlike prior work on verifying crash consistency, PoWER relies only on standard verifier features and requires minimal additional libraries and infrastructure. We also introduce several new techniques that make building verified PM storage systems easier, including a new primitive for atomic checksum updates and a Rust crate to help developers implement fast, provably-safe durable updates.

This dissertation advances the state of the art by presenting new results that inform how we should build robust storage systems and new techniques to help developers achieve this goal. While the tools and techniques we discuss were developed with PM in mind, many are applicable to traditional storage systems as well. All the work presented in this dissertation is open-source.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Massive amounts of data are generated every day [86], and much of this data is entrusted to cloud storage services. It is estimated that, as of 2025, cloud providers store hundreds of zettabytes of data [70, 202, 253]. These providers are responsible for a wide range of data, from personal files of individual users [68, 92, 196] to critical financial [76, 91, 195, 235, 242] and government records [7, 26, 93, 193]. All of these users depend on storage services to keep their data safe. Most providers promise at least 11 nines of annual durability [6, 90, 194], meaning that in a given year, they expect to lose at most approximately one gigabyte out of every 100 exabytes stored (although most services quantify durability in terms of objects).

Achieving this level of durability is challenging, in large part because system crashes or power loss events can interrupt operations on stored data at any time. Power outages can take entire datacenters offline [57, 218, 246], and bugs in kernel or device driver code can cause system crashes that require system reboots. Such interruptions to storage system operations can cause data loss or corruption [42, 97]. Furthermore, data can become also corrupted during regular execution due to software bugs, media errors, hardware defects or cosmic rays [14, 15, 71]. Modern storage systems use crash-consistency techniques such as journaling [217] to handle unexpected power cycles and checksums [14] to detect corruption. However, these approaches are challenging to implement correctly in practice. They are complicated to build and difficult to test, but bugs in their design or implementation can cause severe data loss in the event of a crash or corruption [201, 270, 271].

In this dissertation, we attempt to answer the question: *how can developers gain confidence in the robustness of their storage systems* using different approaches like testing, lightweight static checking, and verification? Although developers would generally agree that they want their systems to tolerate rare events like crashes and corruption, they do not all share the same priorities or resources that can be put

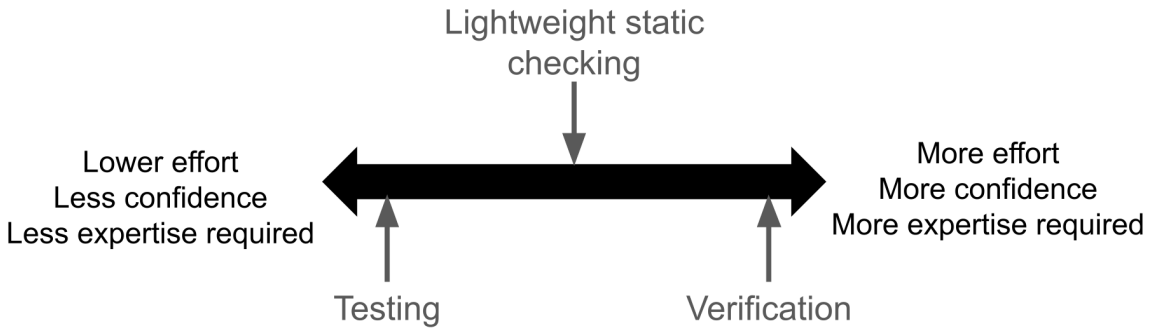

Figure 1.1: **Spectrum of techniques.** The figure shows a spectrum of crash-consistency approaches, labeled with the three techniques (testing, lightweight static checking, and verification) discussed in this dissertation.

towards achieving this goal. There is no one-size-fits-all approach to building good storage systems. Each system differs in terms of its durability and performance SLOs, the crash-consistency guarantees it should provide, the frequency and kind of data corruption it is expected to experience, the amount of developer time available, among many other factors. In this dissertation, we focus on the trade-off between the level of developer effort and expertise and the amount of confidence that a developer gains in the robustness of their system.

Figure 1.1 shows a simple spectrum representing this trade-off, labeled with three points for which we provide new techniques. The most common approach is testing for bugs, but no amount of testing can guarantee that a system is resilient to failures and/or corruption. A system that requires extremely high assurance of correctness may instead be formally verified, but this requires specialized developer expertise and significant additional development time. A system may also fall somewhere in the middle, if, e.g., its developers want stronger guarantees than testing can provide but cannot commit to verification.

In this dissertation, we focus on these problems in the context of *persistent memory* (PM), a type of storage-class memory that provides low-latency, byte-addressable access to durable storage. Using PM for durable data storage, either in place of or alongside slower devices like SSDs or HDDs, has been a topic of research for

decades [48, 199, 260, 261] but has recently become more viable with the introduction of Intel's Optane DC Persistent Memory Module [51] and improving battery technology for battery-backed DRAM [140]. The PM programming model has also solidified in recent years with the introduction of PM-specific instructions and features into Intel's x86 ISA [229]. This has made the development of realistic PM storage systems more feasible, resulting in a flurry of research on this topic over the past 10 years or so [8, 66, 67, 132, 133, 155, 266, 267]. Although Optane has since been discontinued, other companies including Samsung [251] and a variety of startups [72, 145] have continued to develop new PM technologies. Battery-backed DRAM can also be used to support fast, byte-addressable access to durable storage, and has been adopted by cloud services like Azure Storage [153]. Microsoft recently proposed Managed-Retention Memory [169], which obtains improved bandwidth, write endurance, and density in exchange for shorter retention times (e.g., days instead of years), which we expect would have many of the same considerations as longer-term PM systems.

Developers have to tackle new challenges to build robust PM storage systems because PM differs from traditional storage devices in a number of ways. PM is accessed directly via memory loads and stores, rather than block-level requests that are passed down to hardware through several layers of software. Taking advantage of PM's unique performance characteristics requires novel system designs, including hybrid user-space/kernel-space architectures and new crash-consistency mechanisms. Even the PM systems that are most similar to traditional storage systems do not use previously ubiquitous techniques, such as in-memory page caches. PM storage systems also face a very different set of trade-offs when it comes to providing crash-consistency guarantees thanks to the hardware's low access latency. These differences mean that we need new ways to help developers build correct and resilient storage systems for PM.

In this dissertation, we present three different techniques for ensuring that PM storage systems are robust. Although these techniques were developed for PM, many of the ideas presented here are also useful for traditional storage systems, as the

fundamental challenges of building crash consistent systems remain largely the same. We also discuss insights and lessons learned by working on both systems testing and verification, which are generally studied by two separate research communities.

All the works presented in this dissertation is open source. See Appendix A for links to each codebase.

1.1 CHIPMUNK: Testing crash consistency

In order to develop effective tools to help developers gain confidence in the correctness and reliability of their PM systems, we first need to understand how bugs arise in these systems. Although recent research has produced many new PM storage systems, prior work on crash-consistency testing for PM has focused primarily only on bugs with specific root causes in certain libraries and database applications [63, 82, 96, 121, 181–183, 210]. Most PM file systems had not undergone any systematic crash-consistency testing, as they are incompatible with existing testing tools for block-based file systems [143, 201, 271].

We present CHIPMUNK [166], a crash-consistency testing tool for PM file systems, and use it to investigate bugs in these systems. CHIPMUNK is based on CrashMonkey [201], a record-and-replay tool for systematically testing traditional file systems, but introduces new techniques tailored to PM systems. For example, CrashMonkey's recording step uses the Linux kernel block layer to intercept and log I/O, but PM file systems do not use this layer and access storage via memory loads and stores. To record durable updates, CHIPMUNK instead leverages the observation that PM file systems all use a small set of *centralized persistence functions* to perform durable updates, and automatically instruments these functions to record I/O.

PM file systems also differ from traditional systems in terms of the crash-consistency guarantees they provide. Traditional systems buffer updates in volatile memory and require users to invoke system calls like `fsync` to ensure durability. In contrast, PM systems write updates directly to storage and provide synchronous

system calls in which updates are all durable by the time the call returns. Finding crash-consistency bugs in PM file systems thus requires checking stronger properties about additional crash states. To do this, Chipmunk records ordering points (i.e., `sfence` instructions), which ensure that all previous updates become durable before subsequent updates, and injects a crash before each one. For each simulated crash, it then replays subsets of in-flight (i.e., made between the prior ordering point and the crash point) updates and checks each resulting state for consistency. Chipmunk replays updates at cache-line granularity; in practice, we find that the number of such updates between any two ordering points is small, making brute-force checking of all states feasible.

We tested seven PM file systems with Chipmunk and found 23 unique bugs in five of them. Most bugs have been acknowledged by maintainers, and many have been fixed. By analyzing this corpus of bugs, we found several surprising patterns about how bugs arise in these systems. For example, although prior work on testing PM storage systems has primarily focused on finding what we call *PM programming errors* (low-level issues missing cache line flushes or store fences), most bugs were *logic bugs* in the design or implementation of the system and were often tied to certain PM-specific design patterns. This observation suggests that more holistic techniques will be required to build robust PM storage systems and motivates the rest of this dissertation.

1.2 SquirrelFS: Statically checking crash consistency

Although Chipmunk successfully found many bugs, it cannot make strong guarantees about the absence of crash-consistency bugs. We next explore the middle of the spectrum in Figure 1.1, to obtain stronger confidence in our systems without requiring proofs or specialized verification tools. To do so, we present SquirrelFS, a new PM file system with statically-checked ordering-related crash-consistency guarantees using the Rust programming language.

In this chapter, we focus on the idea that crash consistency relies upon the order in which updates become durable and that many bugs are caused by unexpected reorderings [42,81]. A development methodology that checks that updates are ordered correctly at compile time can thus rule out a large class of bugs. Using SquirrelFS, we present such a methodology by exploiting several recent developments in storage and programming languages.

First, the Rust programming language has a strong type system that supports strong compile-time safety checks; for example, ensuring that there are no memory leaks or race conditions. Rust is growing rapidly in popularity has already been shown to be useful as a tool for development of realistic and crash-safe storage systems [24, 113]. Rust's type system can statically check that certain operations are carried out in a given partial order using a design pattern called the *typestate pattern*. In this work, we observe that this can be used to statically check ordering-related crash-consistency guarantees by encoding ordering-based update invariants in Rust types.

Although ordering is a crucial aspect of nearly all crash-consistency mechanisms, static ordering checks will be most valuable in a system that derives crash consistency *entirely* from ordering. Soft updates [187], a technique used in BSD FFS, fits this mold but is notoriously complicated and rarely used [12,81]. We observe that much of soft updates' complexity stems from *asynchrony*; it is difficult to determine the correct order of multiple asynchronous updates issued by different system calls. A *synchronous* implementation, in which the effects of a system call are durable by the time it returns, would eliminate a significant amount of this complexity. The second development we leverage is persistent memory's support for fast, synchronous storage systems, which makes such an implementation feasible. We call this new variant of soft updates Synchronous Soft Updates (SSU).

SquirrelFS is implemented in Rust and uses typestate-checked SSU to obtain ordering-related crash-consistency guarantees. Ordering rules are encoded in typestates such that an implementation that allows incorrectly-ordered updates will

fail typechecking during compilation. The dependencies that form these ordering rules are encoded in the signatures of methods on persistent objects and provide the only interface to modify durable state. To gain confidence that these rules are correct, we checked the design of SquirrelFS in the model checker Alloy [4]. Although the model and implementation are separate, a system written with the typestate pattern encodes a state machine and is straightforward to represent as a transition system model in Alloy.

In SquirrelFS, all system calls are synchronously durable, and metadata-related system calls are crash-atomic. We develop a new technique to make the `rename` system call fully atomic, which prior soft updates implementations have not supported. SquirrelFS is written in unmodified Rust and requires no proofs or additional verification infrastructure. We evaluate SquirrelFS on Intel Optane DC PMM and find that it achieves competitive performance with other PM file systems on many common research benchmarks. We also test SquirrelFS with Chipmunk and find no ordering-related crash-consistency bugs, indicating that the typestate pattern is effective at preventing them.

1.3 CapybaraKV and PoWER: Verifying crash consistency and corruption detection

Chipmunk and SquirrelFS present powerful and effective techniques for finding and preventing crash-consistency bugs, but some developers may want even stronger assurances. On the far end of the spectrum in Figure 1.1, they prove that their system is correct and robust to failures and corruption using formal verification. We explore using verification to guarantee the crash consistency and corruption detection capabilities of PM storage systems in this chapter. We present several new techniques and systems, including PoWER (Preconditions on Writes Enforcing Recoverability), a new approach to crash-consistency verification, and CapybaraKV, a verified PM key-value store.

Crash consistency is a particularly difficult property to verify because Hoare logic [77, 111], which is widely used as the basis for automated verification tools, does not provide a straightforward way to reason about possible crash states. To verify a program using Hoare logic, a developer annotates each function with preconditions, which must hold when the function is called, and postconditions that must hold when it returns. The system verifies if it can be proven that these conditions always hold. As long as the postcondtion is always established when the function finishes, its internal behavior is unconstrained. This makes it challenging to reason about crashes, which may occur at any time, including while a function is executing. Prior work has introduced extensions to Hoare logic to tackle this challenge [30, 40, 104], but these extensions increase the complexity of verification and tie techniques to the specific tools that support them.

We introduce a new crash-consistency verification technique, PoWER (Preconditions on Writes Enforcing Recoverability), which only requires basic verifier features. PoWER is based on the observation that all crash states that may result from a durable update can be exhaustively described without new forms of reasoning. We can add a precondition to write methods stating that all such states must be legal according to a specification of consistency using standard Hoare logic and quantifiers, which are widely supported. In order to update durable state, developers must satisfy this precondition, thereby proving that the update does not introduce any potential crash consistency issues. PoWER only relies on a few common verification constructs and is thus not tied to any specific verification tool or framework.

We use PoWER to build several systems in different verification languages. In this dissertation, we focus on CapybaraKV, a verified PM KV store written in the Rust verification framework Verus [164] using PoWER. In addition to crash consistency, CapybaraKV has verified corruption-detection capabilities based on a new model of bit corruption that, unlike prior work, is applicable to any data layout and storage device. We also introduce several novel techniques to solve PM-specific verification challenges we encountered building CapybaraKV. First, we observe that

it is difficult to maintain checksums for corruption detection in a crash-safe way (and even more difficult to verify this). We introduce the corruption-detecting Boolean (CDB), a new primitive for crash-consistent checksum management, and verify its use in CapybaraKV. Second, we find that a common pattern in PM storage systems — copying bytes directly between DRAM and PM — is potentially unsafe but cannot be handled by the verification tools we use. To tackle this, we build a new Rust crate that helps developers combine the power of the Rust compiler and the Verus verifier to prove that these operations are safe.

We evaluate CapybaraKV on Intel Optane PM and find that it achieves competitive performance with two other PM KV stores, pmem-RocksDB [120] and pmem-Redis [119]. We also run several experiments on battery-backed DRAM, where it also obtains good performance. CapybaraKV is currently single-threaded, but we show that it can be effectively parallelized using sharding.

1.4 Contributions

This dissertation makes the following contributions:

- We design Chipmunk, a record-and-replay testing tool for PM file systems that presents novel techniques to locate crash-consistency bugs in these systems.
- We present a corpus of 27 new crash-consistency bugs across six PM file systems and an analysis of their root cause. We present a set of observations about these bugs and distill them into a set of lessons for future developers of storage systems and testing tools for PM.
- We design SquirrelFS, a PM file system with statically-checked ordering-related crash-consistency properties.
- We introduce Synchronous Soft Updates, a new mechanism based on the traditional soft updates approach, together with a way to obtain ordering guarantees

about a Rust implementation of SSU at compile time.

- We design CapybaraKV, a PM key-value store with verified crash-consistency and corruption-detection properties.
- We introduce PoWER, a new, tool-agnostic approach to verifying crash consistency in storage systems.
- We introduce several PM-specific techniques for writing verified storage systems in the presence of data corruption.
- We present a discussion of new insights and lessons from studying both storage system testing and verification.

1.5 Overview

The rest of this dissertation is structured as follows. Chapter 2 provides background information required to understand the following chapters. We describe persistent memory, crash consistency, data corruption, the Rust programming language, and several techniques from formal methods literature used in this dissertation. Chapters 3, 4, and 5 introduce Chipmunk, SquirrelFS, and PoWER/CapybaraKV, respectively. Each of these chapters motivates the development of the new system, describes the techniques we developed to build it, and presents a performance evaluation of the system on Intel Optane PM. Chapter 6 discusses related work on testing storage systems, lightweight language-based static checking, and formal verification. Chapter 7 provides additional discussion about the systems presented in the prior chapters. We compare the guarantees and assumptions made in each of the systems in detail, discuss the applicability of our proposed techniques to non-PM storage systems, and describe our experiences learning systems verification when building CapybaraKV. Chapter 9 describes ideas for future work on storage system correctness and persistent memory and concludes the dissertation.

Chapter 2: Background

This chapter provides background information about various aspects of this dissertation. In §2.1, we introduce persistent memory (PM) and discuss its performance and reliability characteristics. In §2.2, we discuss the problem of crash consistency more broadly and describe existing crash consistency mechanisms. In §2.3, we cover the issue of bit corruption in stored data and describe techniques for detecting corrupted data. In §2.4, we discuss the Rust programming language and describe the aspects of the language that are relevant to this dissertation. In §2.5, we provide an overview of formal methods techniques used to this dissertation, such as program verification and model checking.

2.1 Persistent memory

Persistent memory (PM), also known as non-volatile main memory (NVMM), is a type of storage-class memory that provides byte-addressable durable storage with low access latency. Technologies such as Phase-Change Memory (PCM) [168], Spin-Torque Transfer RAM (STT-RAM) [156], resistive RAM (ReRAM) [64], 3D XPoint [2], battery-backed DRAM [58,114,115,239], and memory-semantic SSDs [236] have been explored for potential use in PM hardware, but few have been made commercially available. Only battery-backed DRAM and 3D XPoint, available as Intel Optane DC Persistent Memory Module devices from 2019 through 2022, have been available for purchase.

In this dissertation, we assume a PM programming model, failure model, and performance profile based on Intel Optane PM, which we discuss in more detail next. Most recent systems for PM have been tested on Optane PM and are designed based on its performance characteristics. We expect that future PM offerings will have different performance characteristics from Optane PM, but retain a similar program-

ming and failure model. Since the techniques presented in this dissertation primarily focus on reliability and data integrity in PM storage systems, they are not restricted to Optane PM and could be used with other types of PM hardware.

2.1.1 Intel Optane DC Persistent Memory

Intel Optane DC Persistent Memory Module is a type of non-volatile DIMM (NVDIMM) that provides access to PM over the memory bus [51]. We now provide an overview of its performance and how it is used by PM-aware applications and storage systems.

2.1.1.1 Performance characteristics

Optane PM provides much lower latency access to durable data than traditional (i.e., solid state drive and hard disk drive) devices, but is slower than DRAM. Optane PM read latency is 2–3× higher than DRAM [124]. Sequential reads are faster than random reads, which experimental data has shown is most likely due to internal 256-byte batching [124, 216]. Writes to Optane PM have similar latency to DRAM [124].

The maximum write bandwidth of a single Optane PM NVDIMM (2.3 GB/s) is approximately 1/3 that of DRAM; maximum read bandwidth (6.6GB/s) is 1/6 that of DRAM [124]. Real bandwidth utilization depends on factors like whether the PM is interleaved, access sizes and patterns, and the number of readers and writers accessing PM. A common setup is to interleave multiple NVDIMMs on one NUMA node, exposing them as a single device to software and accessing 4KB blocks of individual DIMMs in a round-robin pattern [216].

Optane PM's 3D XPoint is denser than DRAM, but less dense than SSDs [46]. When on the market, it was cheaper per gigabyte than DRAM [105]. Recent research has thus investigated it as a way to increase the memory capacity of individual servers and as a durable storage device, either instead of or alongside traditional block-based

media.

2.1.1.2 Accessing PM

Optane PM can be operated in either *Memory Mode* or *App Direct Mode*. In Memory Mode, the PM is used to increase memory capacity and does not guarantee durability of data [116]. DRAM is used as an L4 cache, and PM is used as main memory. In App Direct Mode, a PM device is presented as a storage device to software and guarantees durability [116]. Devices in App Direct Mode are supported in Linux by DAX (Direct Access), a feature that gives software access to PM via memory mapping. The systems discussed in this paper all use PM in App Direct Mode.

There are two DAX modes: `fsdax`, which supports accessing PM via an in-kernel file system; and `devdax`, which exposes the device as a character device that can be accessed without a PM-aware file system. All systems discussed in this paper, and most related work, use the `fsdax` mode. Some user-space PM file systems (e.g. Strata [155] and Assise [8]) use `devdax` to manage a full PM device without an in-kernel file system. Both modes allow an application to map PM into its address space and directly access the underlying storage via memory loads and stores without interference from a file system. This differs significantly from memory-mapped access to files in traditional file systems, which stage updates in DRAM and do not give applications direct access to the underlying storage device.

2.1.1.3 Programming model

Under the current x86 PM programming model, there are two primary ways to write data to PM. First, standard memory store instructions (e.g., `mov`) can be used to store data at persistent addresses. Data stored in this way is first cached in CPU caches before it is written to PM. Second, non-temporal (NT) store instructions (e.g., `movnti`) may be used to bypass the CPU caches. Data written with NT

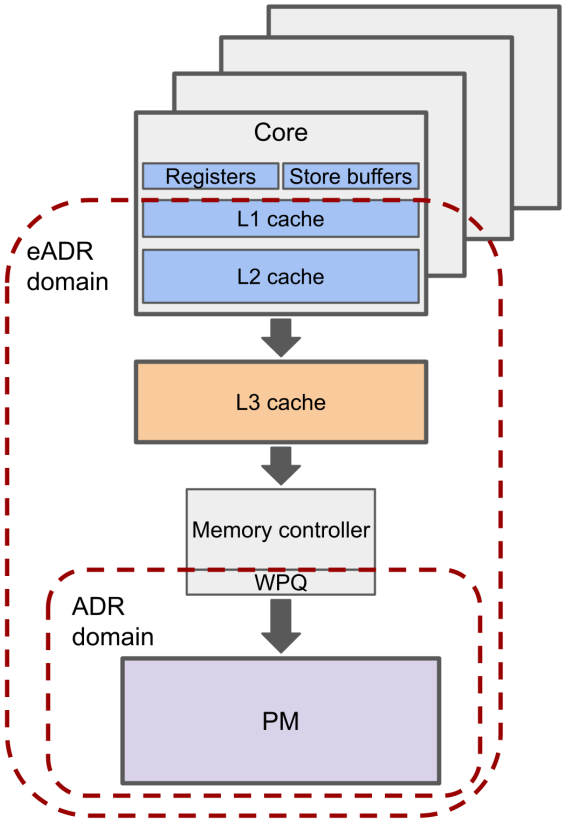

Figure 2.1: **PM power-fail protected domains..** The figure shows the power-fail protected domains for ADR and eADR systems.

stores is still temporarily stored in volatile CPU buffers separate from the main cache hierarchy [214].

In both cases, additional instructions are required to ensure that writes become durable in the correct order. Data must reach the device's *power-fail protected domain* in order to be guaranteed persistent in the event of a crash or power loss [229]. Figure 2.1 shows an overview of current and proposed power-fail protected domains on Intel hardware. In current systems, the power-fail protected domain contains the device's write-pending queue (WPQ) but does not contain CPU caches or store buffers, so data that is not flushed from these locations before a crash will be lost. To support this model, PM systems require a hardware feature called *asynchronous*

DRAM refresh (ADR), which ensures data in the memory's WPQ will be flushed on power loss [122, 229].

To guarantee persistence of cache-resident data, one of several cache line flush and writeback instructions (`clflush`, `clflushopt`, or `clwb`) must be used [229]. `clflush` is supported by legacy CPU versions and serializes flushes. `clflushopt` is an optimized version of `clflush` that does not serialize flushes to achieve better performance. `clwb` is also not serialized and performs a cache write-back, which unlike a flush does not invalidate the target cache line, potentially improving performance. Since `clflushopt` and `clwb` may be reordered, developers must use memory or store fences (`mfence` or `sfence`) to ensure that durable updates are ordered correctly [229]. In this dissertation, we refer to data that has been written but not ordered with respect to subsequent writes as *in-flight* or *outstanding* interchangeably.

Intel proposed extended asynchronous DRAM refresh (eADR) in 2021 [122], which extended the power-fail protected domain to include the CPU caches. eADR removes the need for explicit CPU flushes or write-backs for crash consistency, but developers still need to use store fences for ordering [117, 205]. eADR was supported by one generation of Optane PM-compatible processors [122], but it has not been supported by subsequent generations [205, 206]. ADR, however, continues to be supported.

2.1.2 Battery-backed DRAM

Battery-backed DRAM has been available for years and is currently deployed in production cloud services such as Azure Storage [153]. To use battery-backed DRAM, standard DRAM is connected to an additional power sources such as a lithium-ion battery [140] or a supercapacitor [192]. In the event of power loss, the DRAM is powered for long enough to flush its data to a hard drive or SSD [140].

Although battery-backed DRAM is currently somewhat more accessible than other PM technologies, there are still obstacles to its widespread deployment. First,

DRAM capacity has improved much more quickly than battery capacity [140], making it difficult to provide sufficient battery power. Second, batteries heat up during use, introducing additional thermal management considerations that will grow with increased battery capacity and density [140]. Third, battery capacity fluctuates over time due to factors like humidity, temperature, and age, so batteries must be overprovisioned to ensure sufficient power is available when needed [140]. Fourth, batteries are expensive throughout their life cycle: acquisition and maintenance introduce additional operating costs [140], and they are generally considered hazardous waste and require special facilities for safe disposal when removed from use [255]. Finally, battery manufacturing has significant environmental costs, both from mining materials and manufacturing [53], which is likely to become increasingly important to cloud providers.

2.1.3 Compute Express Link (CXL)

CXL is an open standard defining a set of interconnect protocols over PCIe [56]. It supports a wide range of devices, including GPUs and DDR memory devices traditionally accessible over the memory bus. CXL supports persistent memory over its CXL.io and CXL.mem protocols as a Type 3 device [56, 230].

PM over CXL will adhere to the current PM programming model [230] and existing applications will continue to work properly on CXL-attached PM. The CXL specification includes a feature called global persistent flush (GPF), which is analogous to ADR [230] and defines the power-fail persistence domain for CXL-attached persistent memory.

This dissertation does not directly focus on systems built for CXL-attached PM, but the techniques we present are compatible with such systems due to their shared programming model and crash behavior. CXL-attached memory is expected to have different performance and capacity characteristics, which may result in new design patterns for PM storage systems.

2.1.4 PM applications

A variety of storage systems, applications, and libraries have been built to take advantage of PM. We briefly describe several types of these PM-aware offerings, and discuss specific systems relevant to this dissertation later. The applications discussed in this section use PM in App Direct mode for durability.

PM libraries and packages. Intel and various research groups have built libraries to help developers build PM-based systems [118, 259]. These libraries include durability primitives to manage cache lines and ordering, persistent allocators, and variants of other common data structures designed for compatibility with PM. PMDK [118] is one of the most widely-used PM libraries.

PM file systems. Recent research has led to the development of many PM-aware file systems with a diverse set of architectures and designs [8, 48, 66, 69, 132, 133, 155, 167, 266, 267, 277]. We discuss these designs in more detail in §2.1.5 and Chapter 3. These systems are mostly compliant with POSIX [250] and implement the same file system operations as traditional file systems. Most PM file systems are thus drop-in compatible with legacy applications and can be used to speed up their access to storage.

PM databases and key-value stores. PM's low-latency and byte-addressable interface has made it an interesting target for both the development of new databases and key-value stores as well as ports of existing systems. Both in-memory systems like Redis [221] and memcached [191] and durable databases like RocksDB [224] have been ported to PM [119, 120, 172, 278], and recent research has looked at building new systems [18, 41, 274]. Most of these applications memory-map PM into their address spaces using a PM-aware file system, then access durable structures directly. These systems must be ported or designed for PM to ensure that they manage durable data structures in PM in a crash-consistent way.

2.1.5 Differences from traditional storage systems

Storage systems for PM have several key differences from systems designed for slower, block-based storage media, which we outline here.

No page cache. Traditional file systems keep recently-accessed durable data in DRAM in a structured called the page cache to hide the high access latency of the storage device. Updates to files and file metadata are first reflected in the cache and later flushed to storage, either explicitly by a user via a system call like `fsync` or internally by the file system after a delay. Due to the low access latency of PM, caching data and staging updates in DRAM is unnecessary and can hurt performance. PM file systems thus write updates directly to the durable media without maintaining an intermediate volatile cache in DRAM.

DRAM-resident data structures. It is difficult to correctly maintain complex, frequently updated data structures, so many PM storage systems move components like allocators and indexes into DRAM. These data structures are reconstructed when the system is mounted, either from a checkpoint made during a clean unmount or during a crash recovery process. Traditional storage systems generally keep these structures on disk, as it is easier to keep them consistent with a larger atomic write size and rebuilding these data structures from a slow disk could be prohibitively expensive.

Direct access to storage media. There are several layers of software between a disk-based file system and the storage device itself [201]. For example, the Linux kernel's block layer provides a common interface between file systems and devices and can reorder or batch requests to underlying device drivers [27]. Although each layer adds additional software overhead, this is eclipsed by the latency of the storage device itself, and optimizations like batching writes to consecutive blocks can improve performance. PM storage systems do not use these intermediate layers; instead, they map the device into their address space and update it directly via memory loads and

stores. The additional layers of software are unnecessary and, as PM is so much faster than traditional storage, would negatively impact performance. Some PM file systems go further to eliminate software overheads introduced by the kernel by moving part or all of the file system into user space [8, 133, 155].

Strong crash-consistency guarantees. Because traditional storage systems temporarily keep updates in volatile storage, these systems cannot promise much about the state of the system after a crash. A user must invoke `fsync` or a similar system call to guarantee that updates to a file become durable at a specific time. Prior work [215] and real-world incidents [254] have shown that it is difficult to build crash-safe applications based on these weak guarantees. This is exacerbated by the fact that different file systems provide slightly different guarantees [201]. PM storage systems can, in contrast, support synchronous operations that provide much stronger guarantees that tend to be more consistent across systems. Since updates are written directly to the media and not staged in DRAM, much less data is at risk of being lost in a crash. In many PM file systems, `fsync` is a no-op and all effects of a system call are durable by the time the call returns.

2.2 Crash consistency

A storage system is *crash consistent* if it can recover to a valid, internally consistent, and functional state after a system crash or power loss event. Specific crash-consistency guarantees vary from system to system, and there is an inherent trade-off between performance and consistency. In this section, we define the crash failure model assumed in this dissertation, describe mechanisms systems use to achieve crash consistency, and discuss the trade-offs between performance and correctness. While this dissertation primarily focuses on crash consistency for PM systems, the techniques and concepts described here are generally applicable to all storage media.

2.2.1 Failure model

In this dissertation, we assume a fail-stop model for crashes. We assume that the system has a defined power-fail protected domain, and that updates that are not in that domain will be lost in the event of a crash. For example, in a PM system with ADR, updates that have not been pushed to the write-pending queue prior to a crash will be lost. However, depending on the guarantees and semantics of the underlying storage device, partial updates may be nondeterministically written to storage and potentially reordered by software or hardware. For example, cache eviction policies may result in partial writes appearing on a PM device after a crash.

Our crash-consistency failure model does not include partial disk failures or data corruption. We discuss corruption of stored data due to causes other than crashes in §2.3.

2.2.2 Crash-consistency mechanisms

Storage systems need mechanisms to ensure that the integrity of data and metadata is maintained in the event of an unexpected power cycle. This is challenging for several reasons. First, storage systems all use some amount of caching in volatile memory (DRAM, CPU caches, and/or store buffers) to hide the high latency of the underlying storage device and to coalesce updates to improve bandwidth and reduce write amplification. Thus, individual updates issued to the storage device by software are generally asynchronous, even in PM-based systems. Updates may be evicted or flushed from such caches in an unexpected order, which can lead to inconsistencies if a crash between reordered updates causes the system to end up in an incorrect state. Second, even if full synchrony were feasible, most operations require multiple durable updates that cannot be performed atomically by the storage device, so a crash could still leave the system in an inconsistent state.

These systems need software mechanisms to ensure that they stay consistent in the event of crashes. We now outline mechanisms used by both research and

production storage systems. Note that these approaches are not mutually exclusive, and many systems combine elements of multiple mechanisms.

File-system check. A file-system checker, more commonly known as `fsck`, scans the contents of a storage device to detect and potentially repair inconsistencies such as orphaned inodes or dangling directory entries [188]. In some cases, `fsck` can fix some inconsistencies; e.g., many tools can handle orphaned inodes by moving them to a lost-and-found directory, but a user must manually move or delete the recovered inodes afterward. Each `fsck` implementation is specific to the system it is built for and is not compatible with other systems. Furthermore, since `fsck` must scan the entire system after each crash, it is slower than other techniques and incurs prohibitive post-crash overhead for many real world production systems.

Logging. Logging is a general technique in which data is organized in an append-only data structure [101,141]. This can be used for crash consistency if appends are crash-atomic. Sprite LFS [227] introduced the idea of a fully log-structured file system, in which all data and metadata updates are appended to a single log. Managing entirely log-structured data and metadata requires additional indexing structures to keep track of the location of each file. For example, Sprite LFS uses a checkpoint region with a fixed location to keep track of the location of structures like inodes. Over time, old data in the log must be garbage-collected and live data compacted to ensure the system does not run out of space for new appends.

Journaling. Journaling is a technique based on write-ahead logs [200] in which records about pending updates are appended to a log and later copied elsewhere on the storage device. There are two main variants: a redo journal stores pending operations to be applied later, and an undo journal records the current state of the target data so that an interrupted operation can be rolled back. Journaling is widely used by file systems like ext4 [74] and XFS [263]. Most modern systems do not journal all updates, as this can impact performance and incur significant write amplification.

A common approach is to journal metadata updates but write data updates in place, which provides metadata but not data atomicity in the event of a crash.

Copy-on-write. Copy-on-write (COW) uses shadow paging [37, 38] to obtain crash consistency. Rather than updating data or metadata structures in place, updates are always written to unused regions, which can require copying unmodified bytes on the same block along with the update. Once the updated region is durable, the pointer to the old version will be updated to point to the new version to atomically change the view of the system from the old state to the new one. COW has been used in systems including ZFS [80] and btrfs [28]. COW can be used to obtain strong crash-consistency guarantees, but can also cause significant write amplification. It can also introduce complexity, as one update to the file system tree may result in cascading COW updates to other parts of the tree.

Soft updates. Soft updates is a technique in which dependencies between durable updates are tracked and enforced when the in-place updates are flushed to storage. By adhering to the ordering rules established by these dependencies, the system ensures all potential crash states are consistent. Soft updates systems generally use some variation on following invariants from [187]:

- Never point to a structure before it is initialized;
- Never reuse a resource before all existing pointers to it have been cleared;
- Never reset the current pointer to a resource before setting the new one.

Soft updates incurs significantly less write amplification and other storage overheads than other mechanisms, since it does not use any external durable data structures and does not make any copies of data. It has been used in the BSD FFS file system [187], but has not been widely adopted due to its complexity. A soft updates system must have a way to track update dependencies and enforce them, which is complicated by the presence of cyclic dependencies between blocks. Soft updates systems are also

limited in the consistency guarantees they provide without an additional post-crash recovery mechanism, as dependency tracking cannot work around the fact that most operations require multiple non-atomic updates to different parts of the system. Most systems allow for resource and space leaks in the event of a crash, which can be fixed using an `fsck` tool or a recovery procedure, since the system will continue to function properly even if the leaks are not fixed unless it runs out of space.

Other techniques. Many systems use a combination of these mechanisms to ensure consistency. For example, many Linux-based file systems use the jbd2 journaling layer [62] to journal metadata and/or data, while also providing an `fsck` tool to fix the system if inconsistencies occur due to software bugs or media errors. Researchers have also proposed other crash-consistency techniques, although these have not seen wide adoption. For example, backpointer-based consistency [44] is a technique that avoids the need for ordering primitives (which are required by most other approaches) by storing both forward and reverse pointers for each file system object. For example, a directory will contain pointers to its files, each of which also includes a pointer back to that same directory. A consistent view of the system is built by examining all of these pointers. Optimistic crash consistency [43] builds on journaling and removes some ordering points to improve performance by using checksums to detect inconsistencies and proposing that disks notify software when an update has become durable.

2.2.3 Crash-consistency trade-offs

There are crucial trade-offs between crash consistency, performance, and complexity in storage systems. At one end, a fully-synchronous storage system, in which every update is immediately written to durable storage in program order, provides strong, clearly-defined crash-consistency guarantees but is prohibitively slow. On the other, an asynchronous system with no crash consistency mechanism at all incurs no crash-consistency related overheads, but also cannot provide any guarantees about the state of the system in the event of a crash.

Note that these trade-offs are not tied to specific crash-consistency mechanisms. The general mechanisms can be modified and customized to meet a system or application's specific needs. For example, many systems do not journal data updates by default for performance, but the user often has the ability to enable data journaling for stronger consistency. The techniques listed here were all originally developed for asynchronous systems running on slow disks, but most have also been used in synchronous systems for faster storage that obtain stronger crash-consistency guarantees. The high-level ideas behind each technique can be implemented and applied in different ways depending on the requirements of the system.

Strong consistency incurs additional overheads. Most crash-consistency techniques incur additional storage overhead and write amplification, which increase with stronger crash-consistency guarantees. For example, a system can obtain very strong consistency by journaling every durable update, but doing so causes significant write amplification (since every update is written twice) and requires additional ordering points. As a result, many file systems do not journal data updates by default, since the system can maintain internal consistency by only journaling metadata.

Weaker consistency complicate guarantees. Storage systems are complicated, with many interconnected operations and durable data structures. When disk operations can be reordered to a greater degree, it becomes more difficult to clearly capture the exact legal states that may occur in the event of a crash. For example, it can be difficult to determine the exact effect of `fsync` after operations involving multiple files in POSIX systems, and different systems may provide different guarantees, which may not be officially documented [50, 201].

Peformant mechanisms with good consistency guarantees give up simplicity. Mechanisms like soft updates and backpointer-based consistency achieve good performance with crash-consistency guarantees only slightly weaker than those provided by techniques like journaling and COW. However, in doing so, they introduce additional complexity that makes using these mechanisms difficult to use in practice.

Developers generally find this complexity to not be worth the performance benefits, especially since they do not strengthen crash-consistency guarantees [12].

2.3 Data corruption

The contents of a storage system become corrupted over time, which can result in loss of system functionality or user data. Storage and memory devices use built-in error correcting codes (ECC) to detect such corruption, but these do not guarantee that all instance of corruption will be caught and/or fixed before returning data to the user [14, 83, 238]. As a result, many software systems provide built-in or optional corruption detection and recovery mechanisms.

The causes and effects of data corruption on traditional block-based storage media are relatively well studied. There are two primary types of data corruption that impact users: latent sector errors and silent corruption [14]. Latent sector errors occur when a sector is read and its ECC indicates an issue that cannot be corrected (e.g., a manufacturing defect or physical damage to that sector) [15]. These errors are communicated to device drivers, which typically pass them up the software stack. Silent corruption is corruption that internal device ECCs do not detect [14]. This may happen, for example, due to a checksum collision or a bug somewhere in device firmware or storage software stack.

Data corruption on persistent memory is less well studied, as PM is not widely deployed and has not been the subject of publicly-available large scale corruption studies. Different types of PM hardware may experience different levels and patterns of data corruption. Prior work on detecting corruption on PM assumes the presence of the same ECC-based checks as DRAM [267] and that an uncorrectable media error will be returned to the user as a machine check exception. PM is also susceptible to silent data corruption [267]. For example, since each device is generally mapped into either the kernel's address space or that of a user-level application and is thus easily accessible to software, "scribbles" due to software bugs may unexpectedly modify the

contents of the device.

Software systems generally address silent data corruption by storing checksums, often cyclic redundancy checks (CRCs), for key data structures and/or user data [20, 74, 152, 262, 267]. CRCs are not guaranteed to find every instance of data corruption. Depending on the amount of data being stored and size of the checksum, two distinct sequences of bytes may have identical checksums due to the pigeonhole principle. Most CRC algorithms are designed to guarantee detection of a certain number of flipped bits, but cannot provide provable guarantees beyond this [152]. The chance of a CRC collision in practice, however, is very low. CRCs are not cryptographic hashes and can be reverse engineered [73].

2.4 Rust

Rust is a systems programming language focused on safety. A key goal of Rust is to prevent critical bugs like segmentation faults and data races that are common in memory-unsafe languages like C and C++ while providing similar performance to these languages. This section provides an overview of Rust features relevant to this dissertation: its ownership-based type system (§2.4.1), generics (§2.4.2), traits (§2.4.3), and macros (§2.4.4). This section draws on the official Rust documentation [146] and "Rust for Rustaceans" by Jon Gjengset [88].

2.4.1 Ownership in Rust

Rust has a unique type system based on *ownership* that enables it to provide strong compile-time guarantees. Ownership is similar to linear type systems, in which each value may only be used once, but is better suited to low-level systems programming in which frequently copying values in order to mutate them is undesirable.

In Rust, each value in memory has one owner. Generally, a value is owned by the variable bound to it. The value's owner is responsible for managing access to it, including deallocating the value when it goes out of scope. Values can be moved

(i.e., change owners) by reassigning them to a new variable. The old variable no longer owns the value, and the value cannot be accessed via this variable, although the location of the value in memory has not changed.

For example, on line 1 of the following listing, we create a Rust `String` on the heap and bind it to variable `x`. On line 2, we bind the same string to variable `y`. If this were a C string, this would create a new alias for the string data, and the same string would be accessible via both `x` and `y`. In Rust, however, this moves ownership of the string to `y`, so an attempt to access it via `x` (line 4) will result in the compiler error shown in red.

```
1 let x = String::from("hello, world!");
2 let y = x;
3         - value moved here
4 println!("{}", x);
5                ^ value borrowed here after move
```

Accessing every value through only a single owner is overly restrictive for most use cases, so Rust allows values to be *borrowed* without moving them via references. There are restrictions on how borrowed values can be used to ensure Rust's safety properties always hold, which are also enforced by the compiler. Each value has a lifetime that defines when it may be borrowed, and attempting to borrow a value after its lifetime ends will result in a compiler error. There may be any number of shared references to a value, but while any of those references are live, the value may not be changed (either by the owner or a borrower). Alternatively, there may be one mutable reference to a value, via which the value can be modified. When a mutable reference is taken, the owner temporarily gives up its ability to mutate the value until the reference's lifetime ends. These rules ensure that there are never multiple mutable aliases to the same value.

For example, in the following listing, we create a `String` on line 1 and take a mutable reference to it on line 2. `x` still owns the string, but `y` temporarily has the ability to mutate it. On line 4, we attempt to mutate the string via `x`, but because it

has been borrowed, this results in the compiler error shown in red.

```
let mut x = String::from("hello, world!");
let y = &mut x;
        ------ first mutable borrow occurs here
x.push('!');
^ second mutable borrow occurs here
y.push('!');
- first borrow later used here
```

Although ownership imposes restrictions on developers and is a large part of why Rust is thought of as having a steep learning curve, it brings significant benefits as well. For example, it helps prevent data races, since each value can only be owned and modified by one thread at a time. Since Rust can track the lifetime of each value via its ownership and borrowing, it can automatically insert calls to free values into compiled binaries. Rust's type system also allows for forms of static program analysis that are difficult in other languages, which we discuss further in §2.5 and Chapter 4.

2.4.2 Generics

Rust generics are similar to C++ templates and Java generics in that they allow for definitions to be used with different concrete data types. Functions, methods, struct and enum definitions, and traits can all be generic.

For instance, a linked list storing values of type `T` can be defined as follows. In this example, each node owns the next node in the list, and `LinkedList` instance owns its own head node. The `Option` type defines an optional value (since the tail node will not have a `next` value), and is itself generic over the type of that value.

```
struct Node<T> {
  value: T,
  next: Option<Node<T>>,
}
struct LinkedList<T> {
  head: Option<Node<T>>,
  len: usize,
}
```

Generic items in Rust are *monomorphized* during compilation. During monomorphization, an instance of the generic item is generated for each concrete type it is used with. When, e.g., a generic function is called on a concrete type, the compiler fills in the monomorphized version of the function for that type. The version of the function to be called in each location is known statically and does not have to be looked up at runtime. While this increases the size of the compiled binary, it ensures that Rust generics introduce no runtime performance overhead.

2.4.3 Rust traits

Rust traits are used to define shared behavior between different types. They are similar to interfaces in Java and abstract classes in C++. Traits can define required and optional associated functions and methods, as well as associated types and constants. A type can provide an implementation of a trait to support the behavior it defines. Many common operations provided by Rust types — for example, explicitly copying or hashing a value — are implemented as methods of traits that user-defined types can implement. Trait implementations for user-defined types can often be generated automatically using macros (§2.4.4). Rust does not have a formal notion of inheritance, but a trait can be specified as a subtrait of another, which requires any type implementing the subtrait to also implement the supertrait.

For example, we use a trait similar to the one in the following listing in our specification of PM in CapybaraKV to describe the key methods, `read` and `write`, we want a region of PM to support. This makes CapybaraKV portable; for example, it can run on both Linux and Windows, using different backends that both implement this trait. Note that the trait shown here, while not pseudocode, is significantly simplified from the version used in CapybaraKV, and we have removed Verus-specific syntax from the definition.

```
1 pub trait PersistentMemory {
2   fn read(&self, addr: u64, len: u64) -> Result<Vec<u8>, PmError>;
```

```
3   fn write(&mut self, addr: u64, bytes: &[u8]) -> Result<(),
      PmError>;
4 }
```

Trait bounds. Generic can be restricted via trait bounds, which specify a set of traits that the given type must implement. This can be used to ensure the type has certain properties or supports certain methods that will be called on instances of the type. For example, types used as the keys in a standard library `HashMap` must implement a `Hash` trait that defines how to hash them. (This is a slight oversimplification; the trait bound specifies that references to keys must be hashable, not the keys themselves, but the distinction is usually unimportant).

Marker traits. Although the primary function of Rust traits is to define shared behavior between types, they are also frequently used to specify whether a certain property holds for a given type without defining any associated functions or types. Such traits are called marker traits and are defined with an empty body. Marker traits are frequently used as trait bounds and are often unsafe to implement directly, to ensure that developers do not erroneously implement them for types they do not hold for. For example, the standard library `Send` and `Sync` traits, which are used to specify whether a type is safe to send or share between threads respectively, are marker traits. Most marker traits are either automatically generated for user-defined types with the property they represent, or have restrictions on when they can be safely implemented.

2.4.4 Rust macros

Rust has a powerful macro system that allows for sophisticated automated code generation. It supports two types of macros: declarative macros, which use a limited grammar to define relatively straightforward code generation rules; and procedural macros, which support more complex parsing and code generation. Macros are evaluated and applied during compilation before most checks, including type and

borrow checking, are run. We briefly describe the two types of macros here, and go into more detail about how we use procedural macros in CapybaraKV in §5.4.1.1.

2.4.4.1 Declarative macros

Declarative macros define how to translate a given syntactically-valid string into another. They are declarative in the sense that they specify what translation should occur without specifying how to perform the translation. The definitions of declarative macros use different syntax from standard Rust because they are pattern matching on Rust syntax rather than Rust values. They are generally somewhat difficult to read as a result, and are primarily used for straightforward translations of small amounts of code and generating boilerplate. For example, Rust provides a declarative macro that allows vectors to be created using the same syntax as arrays.

2.4.4.2 Procedural macros

Procedural macros are more powerful than declarative macros because they allow one to implement custom operations on Rust syntax. This functionality is implemented in standard Rust functions that take input parsed into a sequence of tokens and returns a generated output sequence. The macro must output valid Rust tokens but can otherwise perform any required operations on the input.

There are three types of procedural macros. We briefly describe function-like macros and attribute macros and go into more depth about `derive` macros, which are discussed further in Chapter 5. Procedural macros must be defined in a separate crate from the project they will be used in, unlike declarative macros that can be defined anywhere.

Function-like macros. Function-like macros are similar to declarative macros and are invoked in the same way, but allow arbitrary operations on the input tokens. Function-like macros replace the code that invokes them.

Attribute macros. Attribute macros are invoked as annotations on items, such as structure or function definitions. They take two inputs: tokens that are passed within the attribute itself and the entire token tree of the item it annotates. Commonly-used attribute macros provided by the Rust standard library let users label functions as unit tests for a test harness or specify the in-memory layout of a data structure. Attribute macros replace the item they annotate.

`derive` macros. `derive` macros are invoked using a `#[derive(T)]` annotation on a structure and automatically generate an implementation of trait `T` for that type. They take a trait to implement and the definition of the implementing structure as inputs. Unlike function-like and attribute macros, `derive` macros do not replace the item they are invoked on, instead inserting the derived implementation after the deriving type's definition. Many standard library types have `derive` macros that provide default implementations of these traits, but custom implementations can be provided for most of these traits as well. For example, a commonly derived trait is `Debug`, which implements a `fmt` method that converts a structure into a human-readable string to print out. Note that derive macros are not limited solely to deriving the specified traits, as we discuss in Chapter 5; they can insert any arbitrary syntactically-valid code.

2.5 Formal methods

In this section, we provide background on several techniques from formal methods used in this dissertation. We discuss these techniques generally here, and go into more detail about their usage with storage systems in later sections. We compare them and discuss their theoretical limitations in Chapter 7.

2.5.1 Model checking

Model checking is a technique for checking correctness of the design of a system. It can be done *in situ*, as in eXplode [271], to check the correctness of the

implementation of a system, but is more commonly used *ex situ* on a separate model of the system. Model checking tools like TLA+ [157], Alloy [4], and Spin [112] are growing in popularity primarily to check properties about production systems [212], and have successfully been used to identify design flaws in complex distributed protocols [273].

To model check a system, a developer first writes a model describing the behavior of the system. Models are structured as abstract state machines in which transitions are atomic operations between system states. Transitions generally have a guard condition specifying when they may be taken. A model checking tool explores possible execution traces on the model from a starting state, usually up to a user-specified bound. Unbounded model checkers exist, but often impose restrictions on the model. For example, Alloy supports unbounded checking, but only for models that do not use integers. The tool checks that user-specified correctness invariants hold throughout all explored traces; if it finds a reachable state in which one does not hold, it returns an error together with the counterexample trace that resulted in that state.

2.5.2 The typestate pattern

The typestate pattern is a design pattern that encodes an object's runtime state into its type. This state information can be updated with each operation performed on the object and checked at compile time. Whereas the type of a mutable value does not change over its lifetime, its runtime state generally does. While an object's type defines the set of functions that can be called on it, it does not restrict *when* they may be called, even though they may be invalid depending on the state of the object. The typestate pattern provides a way to further restrict the set of legal operations on a value to only those that are valid in its current state.

To build intuition about the typestate pattern, we will use a standard example: keeping track of the current state of a file via a file handle API. Figure 2.2 shows an

```
1  struct FileHandle<State> { path: String, ...}
2
3  impl FileHandle<Closed> {
4    fn new() -> Self {...}
5    fn open(self) -> FileHandle<Open> {...}
6  }
7
8  impl FileHandle<Open> {
9    fn close(self) -> FileHandle<Closed> {...}
10   fn read(&self, addr: usize, len: usize) -> String {...}
11   fn write(&self, addr: usize, bytes: &[u8]) {...}
12 }
13
14 fn typestate_check_fail() {
15   let file_handle = FileHandle::new(foo.txt); // FileHandle<Closed>
16   let bytes = file_handle.read(0, 64);
17                            ^^^^ method not found in
18                                 `FileHandle<Closed>`
19 }
```

Figure 2.2: **Typestate example.** The listing contains a Rust implementation of a file handle API with typestate. A file handle's typestate represents whether it is open or closed. A file can only be read or written via an open handle.

abbreviated Rust definition of a `FileHandle` type and several methods. Note that this example is describing how a userspace program might access a file, not how the file is managed internally by a file system. We represent the file handle's state using a generic type parameter (§2.4.2). Handles can only be created in the closed state, and the only operation defined for a closed handle is to open it. When the handle is open, a user may read or write to the file or close the handle. The function `typestate_check_fail` on line 14 shows an attempt to read a file handle while it is closed. The file handle is initially created in the closed state, and the `read` method is not defined for the type `FileHandle<Closed>`, so this code does not compile. Note that Rust can infer all types in this code snippet, including the value of typestate parameters.

The typestate pattern is most useful in languages that impose restrictions on

aliasing. Specifically, the language must ensure that there is at most one mutable alias for each value at a time [243]. If we try to track typestate for a value that may be modified via multiple mutable aliases, an operation on one alias will not update the typestate of the others. Thus, the value's typestate will not be guaranteed to reflect its real runtime state, so compile-time checks will not provide much value to the developer. We discuss challenges and limitations of the typestate pattern in more depth in Chapter 7.

2.5.3 Formal verification

A formally verified system is one that has been proven correct with regard to a specification. To verify a system, a developer writes an implementation, a high-level specification of correctness, and a set of proofs that the implementation matches the spec. There are two broad categories of tools used to verify programs: proof assistants (also known as interactive theorem provers) and verification-aware programming languages. We now provide an overview of these two tool types.

2.5.3.1 Types of verification tools

Proof assistants. Proof assistants such as Rocq [226] (formerly known as Coq), Lean [203], and Isabelle/HOL [123] are highly-interactive tools for constructing and checking logical and mathematical proofs. These tools present languages and interfaces that allow a user to guide the system through the process of searching for a proof [106]. They are based on type theories in which proofs can be verified entirely via type checking [13, 225]. Proof assistants have been used to verify software systems, including OS kernels [5, 40, 100, 147, 173], as well as to formalize and check mathematical theorems like the Four-Color Theorem [89].

Proof assistants provide a high degree of flexibility and expressivity, which enable users to reason about a wide variety of arbitrary features and semantics using custom logics [29]. This comes at the cost of automation and simplicity; most proof

assistants are known to have a steep learning curve and writing proofs requires a significant amount of manual effort. For example, it took 11 person-years to write proofs for the seL4 kernel [148]. Obtaining an executable from a program verified by a proof assistant is generally not straightforward. Building seL4 involved writing a formal semantics for a subset of C so that the implementation could be reasoned about in Isabelle [148]. Some systems, such as the verified file system FSCQ [40], are written entirely in a proof-assistant-provided language and then extracted to a functional language (e.g., Haskell for FSCQ). Recent work on projects like Goose [31], which translates Go code into Coq for verification, have made bridging this gap easier.

Verification-aware programming languages. Verification-aware programming languages like Dafny [171], F* [245], and Verus [163,164] support formal specifications and proofs of executable implementations in a single language. Programs written in these languages can be compiled, either to another general-purpose language like Java or C, or directly to an executable using an existing language's compiler. These tools are usually based on Hoare logic [77,111], in which each function is annotated with a precondition that must hold when the function is called and a postcondition that must hold when it returns. The verifier checks that the preconditions and postconditions of each function in the program always hold.

These languages verify programs by automatically translating code into queries to an underlying solver, often an SMT (satisfiability modulo theories) solver such as Z3 [59] or CVC5 [17]. These solvers generalize the Boolean satisfiability problem to determine whether more complex formulas including, e.g., integer arithmetic and operations on arrays, are satisfiable [59]. Whether these queries are satisfiable (or if the solver can determine satisfiability at all) is used to prove or disprove program correctness.

Verification-aware languages generally provide a higher degree of automation than proof assistants, at the cost of reduced flexibility and expressivity. They are limited to reasoning using built-in logics and the language's own semantics. How-

ever, they still require developers to write proof code, often $6-10\times$ the amount of implementation code, to help the underlying solver prove correctness [5, 40, 104, 148]. Since these languages can be compiled to other low-level languages (or are built on top of existing languages [10, 61, 164]), they can be easier to use in practice to ensure the correctness of critical systems without sacrificing performance. One of the most mature verification-aware languages, Dafny [171], is used in production at Amazon Web Services [35, 36, 190].

2.5.3.2 Rust verifiers

There has been a flurry of recent research on formally verifying Rust programs. This work has targeted Rust because its strong static type system can simplify verification for both developers of verified programs and verification tools. We now provide a brief overview of recent Rust verification tools.

Verus. Verus [163, 164] is the primary Rust verification tool discussed in this dissertation. It uses a set of macros to add a specification language to Rust and uses an SMT solver backend to dispatch verification conditions. Verus relies on the Rust compiler's ownership and borrow checking to facilitate heap reasoning, and extends these checks to ghost code. Verus is aimed primarily at verifying low-level systems and supports verification of concurrent programs, additional forms of automation, and custom support for common systems idioms like bit manipulation to achieve this goal. Recent work has shown that Verus outperforms many other verifiers, including Rust-based tools, on several verification-time benchmarks [163]. It has also been used to build verified systems that achieve competitive performance with unverified systems [163]. CapybaraKV, introduced in Chapter 5, is built in Verus. An early version of its log component is discussed as a case study by Lattuada *et al.* [163].

Creusot. Creusot [61] is an SMT-solver based verifier for Rust programs. Like Verus, it relies on the Rust compiler to check ownership and aliasing properties that are difficult to verify. It also uses procedural macros to build on Rust with a specifi-

cation language for expressing, e.g., preconditions and postconditions. Unlike Verus, it does not support ownership and borrow checking in ghost code and cannot verify concurrent programs.

Prusti. Prusti [10] is another SMT-based Rust verifier. It is based on the Viper [204] verification tool, which provides a general interface and intermediate representation between multiple front-end languages and backend solvers. Unlike Verus and Creusot, Prusti does not offload reasoning to the Rust compiler; instead, it re-verifies ownership properties using Viper's separation logic support.

Aeneas. Aeneas [1, 110] is a tool that translates programs written in a subset of Rust into a functional language for verification in a proof assistant like Rocq or Lean. Developers write the original program in Rust and verify it in a different language. Aeneas takes advantage of the observation that programs in their targeted subset of Rust can be translated into a pure functional equivalent thanks to Rust's static type system. It supports backends to translate Rust code into one of several proof assistant-supported functional languages in order to verify it. It is unable to verify code involving unsafe Rust or interior mutabililty (a common pattern in which ownership is managed at runtime).

RustBelt. RustBelt [128] aims to model part of the Rust language in order to prove that it is sound. It focuses on verifying *unsafe* Rust, a part of the language in which some restrictions are lifted at the cost of some safety guarantees. Much of Rust's standard library uses unsafe code internally, and the language-provided safety properties rely on this code being correct. Most external crates (Rust's term for libraries) use little to no unsafe code [11], but those that do also depend on it being correct to provide safety guarantees. The developers of RustBelt formally modeled part of Rust in Rocq and proved that several components of the standard library and popular external libraries are safe according to their model.

Kani. Kani [137] is a model-checking tool for Rust programs. Like RustBelt, Kani

focuses on checking safety properties in unsafe Rust, but can also check more general user-specified properties via assertions. Kani is a crate that can be imported as a dependency and used by other Rust projects. For example, it was used to check particularly critical properties in Firecracker [154]. Since Kani uses bounded model checking, it cannot guarantee full verification in many cases.

2.6 Summary

In this chapter, we introduce key concepts for understanding the rest of this dissertation. We first discuss persistent memory (PM), the storage technology at the center of this dissertation, and provide an overview of the performance characteristics and programming model of existing hardware. We also describe different types of applications that use PM and compare them to traditional storage systems. We next discuss the problem of crash consistency and describe techniques used in current storage systems and related research to build crash-safe systems. We introduce data corruption as another problem storage systems must solve in order to be robust, and discuss how current storage systems detect it. Next, we describe key features of the Rust programming language relevant to this dissertation. Finally, we discuss techniques from formal methods used in this dissertation, including model checking and formal verification. We also provide an overview of recent work on Rust-based verification tools.

Chapter 3: Crash consistency in PM file systems

Work on PM file systems has produced a diverse set of systems with new crash-consistency specifications and architectures, including in-kernel [69, 132, 266, 267], kernel-bypass [8, 66, 155], and hybrid systems [133]. These systems have introduced new architectures that differ significantly from the standard design practices of traditional storage systems, as discussed in §2.1.5. In order to build robust PM storage systems, we first need to understand the key causes of crash-consistency bugs in these systems. This chapter presents Chipmunk, a record-and-replay framework for testing the crash consistency of PM file systems. Using Chipmunk, we found 23 new crash-consistency bugs in five PM file systems. We analyze these bugs to obtain useful insights about PM file system design and efficient crash-consistency testing of these systems, which inspired the other projects described in this dissertation.

In §3.1, we motivate the need for a new tool to test PM file systems. In §3.2, we describe the challenges addressed by Chipmunk, the design and implementation of the tool, and our methods for workload generation. In §3.3, we describe how we tested PM file systems using Chipmunk and present the bugs that were found. In §3.4, we distill these bugs into a set of observations about how PM crash-consistency bugs arise and lessons for future testing and system design work.

This chapter is based on the paper *Chipmunk: Investigating Crash-Consistency in Persistent-Memory File Systems* published in EuroSys 2023 [166].

3.1 Motivation

Research on PM file systems has produced a variety of new systems that take advantage of PM's unique characteristics. PMFS [69], NOVA [266], NOVA-Fortis [267], and WineFS [132] are implemented in the kernel. Strata [155], Assise [8], and SplitFS [133] are implemented as kernel-bypass systems. Strata and Assise are

implemented entirely in user space, while SplitFS handles file data in user space and passes metadata operations to a kernel component. As described in §2.1.5, the design of these PM file systems differs significantly from that of traditional file systems. In particular, these systems do not require `fsync` for durability, so we say that they have *strong* crash-consistency guarantees. Several systems (ext4-DAX and XFS-DAX [177]) are based on existing disk-based file systems. These systems share much of their code with their original implementations and have the same *weak* crash-consistency guarantees.

Why current tools are not enough. Existing work on crash-consistency testing is insufficient for today's PM file systems for four reasons. First, prior work on testing disk-based file systems cannot record writes to PM. CrashMonkey [201] and Hydra [143], two state-of-the-art tools for testing traditional file systems, rely on the kernel block layer to record disk I/O. Since PM file systems do not use the block layer, these tools are incapable of intercepting writes made by these systems. Second, these tools do not check all necessary crash states. CrashMonkey and Hydra only insert crashes after `fsync`-related system calls. Injecting crashes during system calls is crucial for exposing bugs in the complex and untested crash-consistency mechanisms of PM file systems. Furthermore, the consistency checkers for these tools would need to be rewritten to properly check these crash states.

Third, tools for testing PM file systems do not scale well. Yat [161], PMTest [183], and Vinter [135] record individual PM I/O instructions, resulting in a high number of instrumentation points. Vinter uses PANDA [65] for dynamic binary instrumentation, which introduces significant overhead. Yat records individual memory stores using a modified hypervisor and has limited optimizations to focus on interesting crash states. Its authors report that it would take over 5 years to fully check one of their three test workloads.

Fourth, prior work on testing general PM applications cannot test high-level crash-consistency properties of file systems. These tools focus on *PM programming*

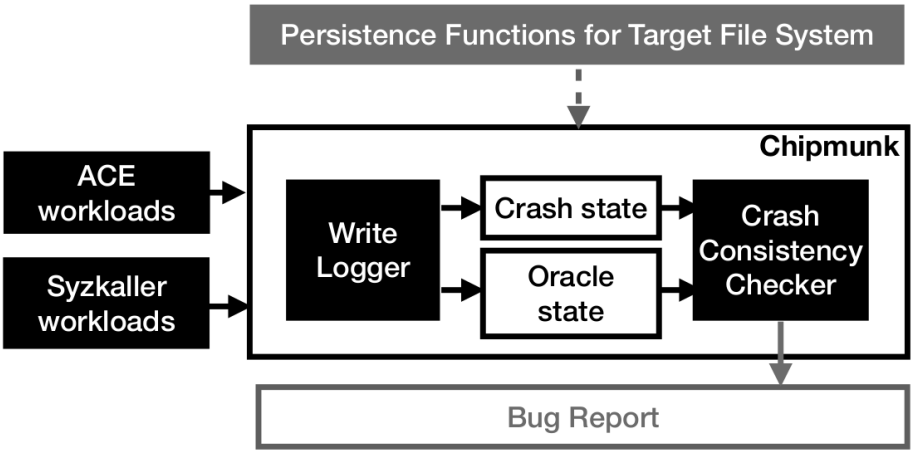

Figure 3.1: **CHIPMUNK architecture.** Given a target file system and its persistence functions, CHIPMUNK uses workloads from both ACE and Syzkaller to test the file system. CHIPMUNK produces bug reports with enough detail to reproduce the bug.

errors, like missing or unnecessary cache-line flushes and store fences [63, 96, 121, 181–183]. Several tools [82, 210] can detect narrow classes of logic bugs — for example, that certain fine-grained updates are atomic — with hard-coded checks or developer-provided oracles. We are interested in checking higher-level crash consistency guarantees without requiring specifications from developers, so these tools are not sufficient.

3.2 CHIPMUNK

In this section, we describe CHIPMUNK, a framework to find crash consistency bugs in PM file systems. CHIPMUNK tackles the challenges outlined in §3.1 with function-level interception and a new testing strategy tailored to PM file systems. CHIPMUNK can test all PM file systems implementing POSIX, and requires no modification of file system code. We have run CHIPMUNK on file systems in both user and kernel space.

3.2.1 Overview

CHIPMUNK is a record-and-replay framework. It first runs a given workload (a sequence of file-system operations) and records the writes made by the file system. Workloads are run sequentially, so there is only one system call running on the file system at any given time. It then replays recorded writes to create crash images, which represent the state of the system if it had crashed at different points during the workload. CHIPMUNK mounts the target file system on the crash image, lets it recover, and then checks whether it has recovered to a consistent state.

We use two tools to generate workloads for CHIPMUNK to test. The ACE workload generator is based on a hypothesis from the CrashMonkey work [201] that testing small workloads on a newly-created file system is effective at finding crash-consistency bugs. To determine if this hypothesis holds for PM file systems, we also use the Syzkaller [95] gray-box fuzzer to generate long, complicated workloads.

We used CHIPMUNK to test seven file systems: six in-kernel systems and one hybrid system with both user and kernel components. There were no publicly-available user-space PM file systems that supported recovery from arbitrary crashes when this work was originally published (§3.3.1).

3.2.2 Challenges

In order to effectively test PM file systems, CHIPMUNK must overcome three key challenges: how to intercept writes, how to deal with new crash-consistency semantics, and how to deal with very large sets of crash states. We describe each challenge and outline the empirical observations and design decisions that allow CHIPMUNK to discover many bugs in PM file systems.

Intercepting writes. Prior work on testing file-system crash consistency has taken a black-box approach to recording writes to storage media. Yat [161] uses a modified hypervisor that triggers a VM exit on stores, flushes, and fences to PM, PMTest [183]

uses the tracking mechanism provided by WHISPER [208] to trace these instructions, and Vinter [135] uses dynamic binary instrumentation in PANDA [65]. These approaches introduce overheads associated with the instrumentation tools and a high number of instrumentation points.

CrashMonkey [201] and Hydra [143], two state-of-the-art tools for testing crash-consistency in traditional file systems, also use a black-box approach based on the Linux kernel's block layer. The file systems they target issue all writes to storage via this layer, so it provides a natural interception point. However, since PM file systems do not use this layer, we cannot use this approach to log writes in Chipmunk.

Instead, Chipmunk uses gray-box *function-level instrumentation* to intercept writes to PM. Each PM file system we examined uses a small set of *centralized persistence functions* to perform I/O. These abstractions simplify reasoning about PM semantics and potentially enable portability to new architectures. All tested systems implement functions for some subset of the following: non-temporal `memcpy`, non-temporal `memset`, flushing cache lines associated with a buffer, and issuing store fences. Each of these operations handles a single, contiguous non-temporal store, a contiguous set of cache line flushes, or enforces store ordering. Chipmunk is not limited to recording just this set of functions; a system's logger can be written to handle other types of persistence functions, if, for example, the system is designed for another persistence model.

Chipmunk requires developers to provide the names of centralized persistence functions; it then instruments these functions at runtime using the Kprobes [178] and Uprobes [179] debugging mechanisms in the Linux kernel. This gray-box approach to recording writes has multiple benefits. It makes logging feasible without requiring source code modification, and it makes Chipmunk portable to new PM architectures since the semantics of x86 PM primitives are not built into the recording code. It also enables Chipmunk to encode information about the context in which a write

was made and use it during consistency checking.

New crash-consistency semantics. In traditional file systems, if a user wants to ensure that a specific file or set of files is persistent on disk, they must call an `fsync`-related system call to flush updates from the volatile page cache to the storage media. Since crash-consistency guarantees are not well-defined if the system crashes prior to such system calls, and the journaling mechanisms that handle incomplete updates are very mature, systems like CrashMonkey and Hydra only insert crash points after `fsync`-related calls.

However, PM file systems with strong crash consistency specs clearly define the correct state of each file at every point during execution, not just after `fsync`. These systems guarantee that most operations are both synchronous and atomic. To test the novel, complex, and untested crash-consistency mechanisms of PM file systems, Chipmunk must inject crashes during system calls (and not just after `fsync`). We developed a new testing strategy and set of consistency checks (§3.2.3) to handle these new crash points for systems with strong guarantees. Chipmunk coalesces logically-related non-temporal stores and flushes (e.g., those all associated with the same file data write) and replays them in different combinations to focus on interesting crash states. It uses an oracle-based checker that compares the post-crash state of each file to a set of possible legal states.

Increased number of crash states. PM file systems with strong crash consistency make fine-grained writes to storage media in the critical path of system calls. As a result, a workload can result in significantly more crash states that are interesting to test than in systems with weaker guarantees. Specifically, if the number of in-flight writes between each store fence is too high, the number of possible crash states to check will explode to an intractable number.

We studied five systems with strong guarantees and made two observations that we use to overcome this challenge. First, when PM file systems perform metadata operations, they issue a small number of small writes to PM with frequent store fences.

We found that the average number of in-flight writes for metadata operations is three and the maximum is 10 in the tested systems. This means that at any given crash point in metadata-related system calls, the number of crash states is small enough to test exhaustively. Second, although file data operations often involve many in-flight writes with few store fences, checking every possible crash state is unlikely to expose new bugs. We want to check how the file system recovers when *some* file data is lost in a crash, but it is unnecessary to check every possible state. Chipmunk coalesces data associated with the same file data update into a single write, and checks only a small subset of states with missing data. Metadata about the size of buffers and how they are written guides this heuristic; for example, non-temporal `memcpy` on a large buffer usually indicates a file data write.

3.2.3 Chipmunk Architecture

Chipmunk is built on top of the CrashMonkey framework [201]. CrashMonkey consists of a set of user space utilities, a block device wrapper kernel module that intercepts writes, and a copy-on-write device to facilitate constructing file system snapshots. We adapt CrashMonkey's user space utilities to target PM file systems and build new kernel modules based on Kprobes and Uprobes, two Linux kernel debugging utilities, to record writes to PM.

Given a workload and a target file system, Chipmunk proceeds in four steps (Figure 3.2): (1) run the workload and log the writes made by the file system; (2) construct crash states; (3) check each crash state; and (4) generate a bug report if required. We now describe these steps in detail.

Logging writes. Chipmunk uses two dynamic debugging and tracing tools, Kprobes [178] (for in-kernel file system components) and Uprobes [179] (for user space components), to automatically instrument centralized persistence functions at runtime. Our logging modules only require the name (for kernel space components) or offset (for user space object files) of these functions in order to instrument them. Kprobes

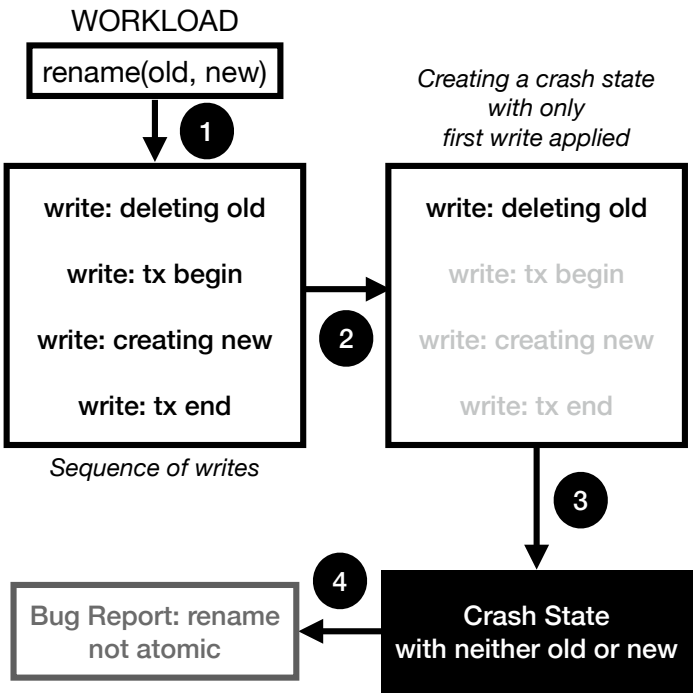

Figure 3.2: **Chipmunk workflow.** The figure shows how crash consistency is tested using a `rename()` workload. In this example, the old file being deleted is updated in-place, while the new file creation happens inside a transaction. 1) Chipmunk runs the workload and logs a sequence of PM operations. For simplicity, the operation has been reduced to a sequence of four logical writes. 2) Chipmunk creates a crash state where only the old file is deleted; the other writes are lost. 3) The consistency checker finds that both the old and new files are missing. 4) Chipmunk creates a bug report. Chipmunk discovered this bug in NOVA (bug 4).

and Uprobes are used together in the same logging module to test SplitFS.

Kprobes and Uprobes make a copy of a probed instruction and replace the first byte(s) with a breakpoint instruction. When the breakpoint is hit, control is passed to a handler function. In Chipmunk's loggers, these handlers record the probed operation and its arguments. The destination of each update is translated from a virtual to physical address within the logging module to facilitate later replay. The user-space test harness also inserts markers into this log to record the start and end of each system call, which Chipmunk uses to determine which writes to PM are associated with each system call. This approach requires no code changes to the file-system implementation other than to prevent the compiler from inlining the persistence functions. In our experience, identifying these functions was simple, and we expect it to be even simpler for file-system developers since these functions are used extremely frequently.

Constructing crash states. Given a workload and a file system to test, Chipmunk selects crash points based on the crash consistency guarantees of the file system and simulates crashes at these points. For ext4-DAX and XFS-DAX, crash points are placed after `fsync`, `sync`, and `fdatasync` calls. For the other systems, crash points are injected after each store fence invoked by the file system. A single system call may perform multiple store fences, resulting in multiple crash points per system call. We construct and create possible crash states out of the in-flight writes that follow each store fence. This process checks crash states that occur both during and between system calls.

Chipmunk replays a workload by walking through the log of flushes, non-temporal stores, and fences. When it encounters a cache line flush or non-temporal store, it adds a structure containing the type, contents, and destination address of the flush/store to an in-flight vector. When it encounters a store fence, it first constructs and checks crash states based on the in-flight vector, then flushes the contents of the vector to the replay device. Each crash state is constructed by replaying a subset of

the contents of the in-flight vector on top of all updates that precede the most recent store fence, which are guaranteed to be persistent. We replay the updates in each subset in program order. For n in-flight writes, there will be $2^n - 1$ crash states to check. As noted in §4.3.3, we have observed that n is small in practice for metadata-related calls, allowing CHIPMUNK to apply this exhaustive testing strategy. File data writes do not adhere to this pattern but can be coalesced into a small set of large writes. Since a small number of in-flight writes is not a guarantee, CHIPMUNK can place a configurable cap on the number of writes to replay. We find that in practice, even a cap of two writes is sufficient to reveal many bugs (§3.4.1).

Testing crash states. To check file-system consistency, CHIPMUNK first mounts the target file system on each crash state, which is itself a useful consistency check. Once successfully mounted, CHIPMUNK compares the file-system state against an oracle representing valid post-crash state(s). The oracle runs the original workload on a fresh file system instance in parallel with log replay. When CHIPMUNK encounters the beginning of a new system call in the log, it records the current oracle state of file(s) that will be modified, then executes that system call on the oracle file system. Files in a crash state are compared against an oracle version of the file by checking whether metadata provided by `stat` differs between the two versions. For regular files, CHIPMUNK also compares the contents of each version. For directories, CHIPMUNK compares the directory entries of each version. Several crash states may be compared to the same few oracle states, so CHIPMUNK caches the metadata and contents for each oracle file version in memory.

Most system calls in file systems with strong guarantees are intended to be atomic. The main exception is `write`, although many systems provide the option to make `write` atomic. Further, all system calls are *synchronous*, in that modifications to persistent data are made durable by the time each system call completes. CHIPMUNK focuses on checking these atomicity and synchrony properties.

When a crash is injected in the middle of a system call, CHIPMUNK checks

that the operation is atomic by comparing modified files in the crash state to the current and previous oracle versions. If the operation modifies multiple files, the files must all match the same version. When a crash is injected after a system call, CHIPMUNK checks that the system call is synchronous by comparing the crash state against the current oracle state. CHIPMUNK also confirms that files that should be unmodified by the current system call match the current oracle state in each crash state. These checks validate properties implied by POSIX or widely expected by users in practice [25, 215]. Finally, to validate that the file system is in a usable state, CHIPMUNK creates files in all directories, then deletes all files. If any of these checks or operations fail, CHIPMUNK outputs a bug report describing the inconsistency and the corresponding crash state.

Because the consistency checks mutate the crash state, we reuse our logging infrastructure to record an undo log for these mutations and roll back the changes when advancing to the next crash state.

3.2.4 Workload Generation

Given a workload, CHIPMUNK generates crash states and tests them for consistency. An orthogonal challenge is generating workloads for CHIPMUNK to test. The CrashMonkey work [201] introduced the hypothesis that small workloads on new file systems are useful in finding crash-consistency bugs. While this hypothesis was true on traditional file systems, we aim to test whether it holds on PM file systems. To this end, we modify CrashMonkey's Automated Crash Explorer (ACE), which systematically explores workloads of a given size, to work with CHIPMUNK. We also modify the Syzkaller [95] gray-box fuzzer to work with CHIPMUNK. Syzkaller generates long, complex, randomized workloads while aiming to improve code coverage.

3.2.4.1 Automatic Crash Explorer

We used a modified version of ACE [201] to systematically generate workloads. ACE was designed to exhaustively generate workloads of a pre-defined structure to test traditional file systems. Given a sequence length n, ACE generates workloads with n core file-system operations over a small, predetermined set of files, then satisfies dependencies by opening and closing files and adds `fsync`, `fdatasync`, or `sync` operations. A workload with n core system calls is called a "seq-n" workload.

We use a slightly modified version of ACE's default mode, which inserts at least one `fsync`-family operation in each workload, for ext4-DAX and XFS-DAX. We also added a mode that does not insert these calls for the other systems.

We test all seq-1 and seq-2 workloads, as well as the subset of seq-3 workloads containing only `pwrite`, `link`, `unlink`, and `rename` calls (i.e. "seq-3 metadata" workloads [201]) to make testing tractable. For PM file systems with strong consistency, we generate 56 seq-1 tests, 3136 seq-2 tests, and 50650 seq-3 metadata tests. The default mode generates 419 seq-1 tests and 432462 seq-2 tests; we did not run seq-3 metadata tests on ext4-DAX and XFS-DAX.

3.2.4.2 Syzkaller

We modify Syzkaller [95], a state-of-the-art gray-box kernel fuzzer, to generate workloads for Chipmunk.[1] As is standard in gray-box fuzzing, our fuzzer starts with an initial set of test cases (seeds) and uses genetic programming to generate new tests for Chipmunk from those seeds. As Chipmunk executes each workload, Syzkaller collects code coverage information by recording coverage points inserted by the compiler. If the workload covered new parts of the kernel, the fuzzer adds it to its set of seeds and generates new workloads from it.

Syzkaller generates workloads by randomly selecting sequences of system calls

[1] Shankara Pailoor contributed to the integration of Syzkaller with Chipmunk.

and argument values. It generates syntactically and semantically valid workloads by using a detailed template for each system call that specifies more precise *qualified type* information [45] for the call's arguments. For example, the template for `write` specifies that its first argument is a valid file descriptor in use by the program, rather than an arbitrary integer.

To adapt Syzkaller to our setting, we restrict it to only generate workloads that contain file-system operations, and replace its workload executor with a custom one. Our executor invokes CHIPMUNK on each workload and records code coverage both before the crash and during recovery. For PM file systems with strong consistency, we add crash points between each system call and in the middle of the final non-failing system call in the workload. For ext4-DAX and XFS-DAX, we include `fsync`, `sync`, and `fdatasync` in workloads and check crash states after each call to one of these system calls. We add a `sync` at the end of each workload to make sure we check at least one crash state. Since Syzkaller is a kernel fuzzer, and we are primarily interested in collecting coverage on SplitFS's user space component, we use GCC's sanitizer coverage instrumentation to collect code coverage [79] and modify Syzkaller to use this coverage information. This required adding some code to SplitFS to log the basic blocks covered during fuzzing, but did not require modification of any existing code. We do not collect code coverage of SplitFS's kernel component.

Like many fuzzers, Syzkaller can quickly generate many bug reports that are duplicates. In our setting, this duplication also arises when multiple crash states trigger the same bug. To address this problem, we extended Syzkaller to automatically triage bug reports generated by CHIPMUNK during fuzzing. We use a simple triaging procedure that clusters bug reports by lexical similarity. We also updated Syzkaller to display these bug report clusters in its UI dashboard to make them easier for users to debug.

3.2.5 Implementation

Although CHIPMUNK was originally based on CrashMonkey, the two systems diverged early in development and most of CHIPMUNK's core code is new. CHIPMUNK has twice as much code dedicated to constructing and checking crash states as CrashMonkey. The increase in test harness size primarily comes from the complexity of PM write semantics when constructing crash states, the need to associate each logged write operation with the system call that issued it, and a much more complex set of consistency checks that account for the semantics of each tested system call in different file systems. We also wrote new code to track oracle state for the consistency checks, as CrashMonkey's was insufficient for checks outside of `fsync`-related calls. Running SplitFS also required further modification to be made to the test harness, since it requires its object files to be dynamically linked to the test harness at runtime.[2] About 2000 LOC in CHIPMUNK's core testing infrastructure comes from a Syzkaller-specific test harness that executes fuzzer-generated tests. The only parts of CrashMonkey that remained largely unchanged were the code that loads and runs ACE-generated tests and some functions related to recording the system calls in a workload.

CHIPMUNK's core testing infrastructure consists of about 9000 lines of C++ code as reported by `sloccount`. Five system-specific logger modules add about 1000 lines of C code each (several similar systems share modules). We also added about 1000 lines of Go to Syzkaller to handle crash consistency tests and to collect coverage when remounting crash states.

3.2.6 Discussion

Limitations. We made certain design decisions to make testing PM file systems for crash consistency tractable. However, it is possible that these choices may cause

[2] Om Saran K R E helped add support for SplitFS to CHIPMUNK.

CHIPMUNK to miss some bugs or limit its ability to test some file systems. First, CHIPMUNK cannot test every workload, and does not test all possible crash states for each workload. Second, CHIPMUNK assumes that the PM file system has centralized persistence functions. A PM file system that uses in-line assembly or macros to update PM would not be compatible with CHIPMUNK. Third, CHIPMUNK does not currently support checking concurrent workloads. Testing crash consistency for concurrent workloads is a hard problem [78, 207] that prior tools do not support. While supporting concurrent workloads could expose more bugs, it is out of the scope of this paper, and we leave it as future work.

Despite these limitations, we believe that CHIPMUNK is a useful addition to the set of tools for building robust PM file systems. In particular, the level of automation provided by CHIPMUNK allows developers to test new or in-development PM file systems efficiently.

Persistence models. CHIPMUNK is implemented for x86's epoch-based persistence model, which at the time of writing was available on Intel's Optane DC Persistent Memory Module. Since then, Intel has cancelled their Optane project. However, other hardware vendors have announced similar byte-addressable persistent storage devices [72, 109, 145, 251], which are expected to use the same persistence model [230].

Because CHIPMUNK logs writes at the function level and runs checks on real file-system images, its techniques are not tied to any persistence model. Adding support for a new model would involve implementing its semantics in CHIPMUNK's replay logic and logger modules, but the rest of the framework would remain the same. Since the high-level operations governed by primitives in different persistence models are broadly similar, these changes should be straightforward. We expect that file systems for new models will use centralized persistence functions as the preferred abstraction for writing data to durable storage for portability and simplicity.

Furthermore, our bug analysis shows that many bugs are caused by logic errors rather than PM programming errors. Logic bugs will continue to occur across

persistence models, and we expect CHIPMUNK would be a valuable tool for testing file systems built for a variety of persistence models.

Code coverage. Although CHIPMUNK is not intended to be complete and achieving high code coverage metrics was not a goal of this work, our results indicate that using ACE and Syzkaller to generate workloads enables thorough testing of important file system features. Our workload generation strategy, which focuses on testing common file system operations, was effective at exposing many new crash-consistency bugs in the tested systems. For long-running tests, CHIPMUNK could be paired with tools like OSS-Fuzz [94] to focus on high code coverage.

3.3 Testing PM File Systems

In this section, we evaluate CHIPMUNK's effectiveness at finding bugs across different PM file systems.

3.3.1 Methodology

File systems. We ran CHIPMUNK with seven open-source PM file systems: NOVA [266], NOVA-Fortis [267], PMFS [69], WineFS [132], SplitFS [133], and ext4-DAX and XFS-DAX [177]. NOVA, NOVA-Fortis, PMFS, WineFS, and SplitFS in strict mode have strong crash-consistency guarantees, so CHIPMUNK inserts crash points both during and after system calls when testing these systems. Ext4-DAX and XFS-DAX have weak guarantees, so CHIPMUNK only inserts crash points after `fsync`-related system calls when testing them. CHIPMUNK is compatible with Strata and Assise as well, but we learned after communication with authors that neither system's current artifact supports recovery from arbitrary crashes, so we were unable to proceed with evaluation on these systems. We have also tested SQUIRRELFS using CHIPMUNK and discuss the results in Chapter 4.

System calls. We select a set of system calls to test based on what is supported by

each file system and what crash consistency guarantees they provide. We focused on ten key operations: `creat`, `mkdir`, `fallocate`, `write`, `link`, `unlink`, `remove`, `rename`, `truncate`, and `rmdir`. Tests run on ext4-DAX and XFS-DAX also include `setxattr` and `removexattr`, which are not supported by the other systems we tested. All tests also include `open` and `close` as necessary, and tests on ext4-DAX and XFS-DAX use at least one of `fsync`, `fdatasync`, or `sync` to ensure that data becomes persistent. We did not test `mmap` with CHIPMUNK, as modifications to memory-mapped regions are not handled via centralized persistence functions and a number of other tools have been built to target the crash-consistency of memory-mapped data (§6.2.3).

3.3.2 Experimental setup

Test infrastructure. All experiments described in this paper were run on QEMU/KVM virtual machines running Debian Stretch. Each VM is allocated one CPU (except for those testing WineFS, which requires four CPUs) and 8 GB of RAM (except for those testing SplitFS, which requires 32GB). Each VM also has two 128 MB emulated PM devices, which are used to execute the workload, construct the oracle file system, and check crash states.

We run ACE-generated workloads on a single Amazon EC2 m5d.metal instance with 96 vCPUs, 384 GB memory, and four 900 GB NVMe SSDs. We use these resources to check multiple file systems using workloads of multiple sequence lengths in parallel. For the systems with strong crash consistency, we ran seq-1 and seq-2 tests on individual VMs, as the number of tests to run was relatively small. We split seq-3 metadata workloads across 10 VMs and ran them in parallel. At the time we ran these experiments, WineFS and SplitFS both had bugs that prevented many seq-3 tests from running. The number of in-flight writes at any time during ACE tests is consistently low, so we do not place a cap on the number of crash states for ACE. For ext4-DAX and XFS-DAX, we ran seq-1 tests on an individual VM and split seq-2 tests across 20 VMs. We were unable to run seq-3 metadata tests on these systems

due to time constraints.

To evaluate CHIPMUNK with Syzkaller, we ran seven Chameleon Cloud [142] bare metal instances, which have two Intel Xeon Gold 6240R CPUs each with 24 cores and 48 threads, as well as 192 GB RAM, and 480 GB storage. Each host fuzzed a different file system using 15 virtual machines. Each fuzzer starts with an empty set of seeds. Syzkaller-generated tests can be long and generate many crash states, so to avoid the fuzzer getting stuck, we run CHIPMUNK with a cap of two writes per crash state; as §3.4.1.2 observes, this cap does not affect its ability to find bugs in practice.

3.3.3 Evaluation

ACE tests. For each file system under test, CHIPMUNK took less than 1 hour to run seq-1 workloads on a single VM. Running these tests on NOVA/NOVA-Fortis, PMFS, and WineFS takes less than 15 minutes. Seq-2 tests take 7–20 hours on the PM file systems with strong consistency. It takes about 30 hours for all seq-2 tests to finish running on ext4-DAX and XFS-DAX when using 20 VMs in parallel. For systems tested on seq-3 workloads, it took 16–26 hours to run them in parallel on 10 VMs. The number of crash states to check on each workload varies as much as 3× between file systems, with PMFS generally checking the most and WineFS checking the fewest. Overall, CHIPMUNK found 19 bugs using ACE tests across five of the tested systems.

Syzkaller. We ran CHIPMUNK with Syzkaller for 18 hours on 15 VMs, for a total of 270 CPU hours spent fuzzing each system. During this time, CHIPMUNK checked over 40 million crash states across all tested systems, finding 23 unique bugs. Four of these bugs cannot be found with ACE-generated workloads.

Comparison. We ran Syzkaller and ACE on each file system and recorded the cumulative CPU time taken to find all bugs when using each workload generator. Figure 3.3 shows the result of this experiment. ACE finds the first 19 out of 23 bugs in less than three CPU hours total, but is unable to find the final four bugs.

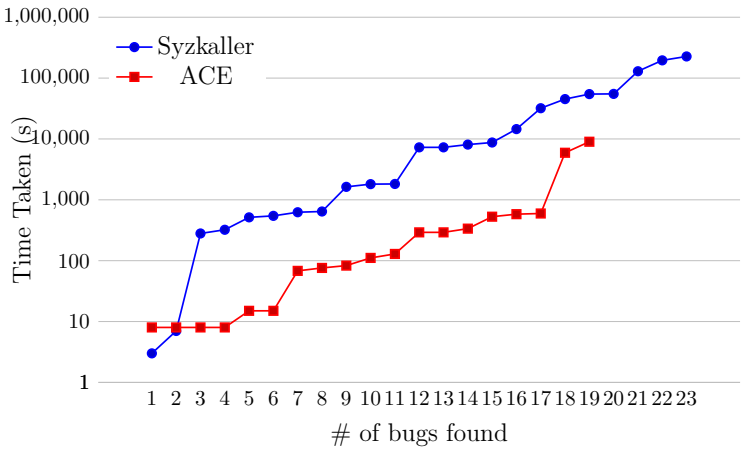

Figure 3.3: **File system testing.** The figure shows cumulative time taken to find crash-consistency bugs by ACE and Syzkaller.

Aside from a couple bugs that Syzkaller and ACE workloads both trigger almost immediately, Syzkaller takes almost 20× more CPU time than ACE to find the first 12 bugs and almost 6× more CPU time to find all bugs exposed by ACE. However, when we let Syzkaller run for an additional 47 CPU hours, it is able to find four bugs that are not detected by ACE. ACE misses these bugs because they do not conform to the patterns that it uses to generate workloads. For example, two of these bugs create two open file descriptors to the same file and modify the file's contents through both file descriptors. ACE workloads do not open multiple file descriptors for the same file and thus cannot trigger these bugs.

While the results of this experiment indicate that Syzkaller has greater overall bug finding capability than ACE, the ACE tests are considerably more resource efficient. This suggests that the ACE tests can be run locally to find bugs during file-system development, whereas Syzkaller should be run for a long time in an environment with ample compute resources for more comprehensive crash-consistency testing.

Bug #	File System	Consequence	Affected system calls	Type
1	NOVA	File system unmountable	All	Logic
2	NOVA	File is unreadable and undeletable	`mkdir, creat`	PM
3	NOVA	File system unmountable	`write, pwrite, link, unlink, rename`	Logic
4	NOVA	Rename atomicity broken (file disappears)	`rename`	Logic
5	NOVA	Rename atomicity broken (old file still present)	`rename`	Logic
6	NOVA	Link count incremented before new file appears	`link`	Logic
7	NOVA	File data lost	`truncate`	Logic
8	NOVA	File data lost	`fallocate`	Logic
9	NOVA-Fortis	Unreadable directory or file data loss	`unlink, rmdir, truncate`	PM
10	NOVA-Fortis	File is undeletable	`write, pwrite, link, rename`	Logic
11	NOVA-Fortis	FS attempts to deallocate free blocks	`truncate`	Logic
12	NOVA-Fortis	File is unreadable	`truncate`	Logic
13	PMFS	File system unmountable	`truncate, unlink, rmdir, rename`	Logic
14 & 15	PMFS, WineFS	Write is not synchronous	`write, pwrite`	PM
16	PMFS	Out-of-bounds memory access	All	Logic
17 & 18	PMFS, WineFS	File data lost	`write, pwrite`	PM
19	WineFS	File is unreadable and undeletable	All	Logic
20	WineFS	Data write is not atomic in strict mode	`write, pwrite`	Logic
21	SplitFS	Operation is not synchronous	All metadata	Logic
22	SplitFS	File data lost	`write, pwrite`	Logic
23	SplitFS	File data lost	`write, pwrite`	Logic
24	SplitFS	Operation is not synchronous	All	Logic
25	SplitFS	Rename atomicity broken (old file still present)	`rename`	Logic

Table 3.1: **Crash-consistency bugs.** The table lists bugs found by CHIPMUNK, their consequences, and the system calls that they affect.

3.3.4 Results

Crash-consistency bugs. Using ACE and Syzkaller generated tests, CHIPMUNK finds 23 new unique crash-consistency bugs across the tested file systems. The number of bugs is based on the number of unique fixes required to patch all the bugs, not different user-visible consequences. Two bugs are found in both WineFS and PMFS for a total of 25 bugs.

Table 3.1 describes the consequences of each bug and the system calls they affect. The bugs are classified as either logic or PM errors (§3.4.1). CHIPMUNK found eight bugs in NOVA, five bugs in SplitFS, four bugs in NOVA-Fortis, two bugs in PMFS, two bugs in WineFS, and two bugs in both PMFS and WineFS. Many of these bugs have serious consequences: three prevent the file system from being mounted entirely, and three impact the atomicity of `rename`, which many applications rely on [215]. Many others cause data loss or prevent a user from accessing files entirely.

Bugs 4, 5, and 13 in Table 3.1 were found independently by both Vinter [135] and CHIPMUNK. Vinter's authors also reported a bug related to an optimization in the non-temporal store function used by NOVA and NOVA-Fortis, which CHIPMUNK can reproduce. CHIPMUNK found a related bug impacting PMFS and WineFS (17 and 20).

CHIPMUNK did not find any bugs in ext4-DAX or XFS-DAX. We attribute this to the maturity of the base file systems. Most code in ext4-DAX and XFS-DAX is shared with their non-DAX versions, which are very well tested. CrashMonkey also found no new bugs in either system, and Hydra found only one new crash-consistency bug in ext4.

Non-crash-consistency bugs. While working with CHIPMUNK, we also found eight non-crash-consistency bugs not included in Table 3.1. We were able to find these bugs because they caused KASAN errors, segmentation faults, or incorrect behavior that our consistency checks could detect. For example, using the fuzzer, we discovered that

Observation	Associated bugs
Many bugs are logic/design issues, not PM programming errors.	1, 3–8, 10–13, 16, 19, 20, 21–25
The complexity of performing in-place updates leads to bugs.	4–7, 14, 15
Recovery related to rebuilding in-DRAM state is a significant source of bugs.	1, 3, 7, 11, 13, 16, 19, 24, 25
Complex features for increasing resilience can introduce crash consistency bugs.	2, 9–12
Many can only be exposed by simulating crashes during system calls.	3–6, 9–13, 19, 20
Short workloads were sufficient to expose many crash consistency bugs.	1–6, 9–20, 21–25
Many bugs are exposed by replaying a few small writes onto previously persistent state.	3–6, 9–13, 19, 20

Table 3.2: **Bug observations.** The table lists observations about PM file systems and the bugs associated with them.

NOVA does not properly handle `write` calls where the number of bytes to write is extremely large; it will allocate all remaining space for the file, causing most subsequent operations to fail.

3.4 Bug Analysis

This section presents an analysis of the 23 crash-consistency bugs found by Chipmunk (Table 3.1). To the best of our knowledge, this is the largest corpus of crash-consistency bugs in PM file systems.

3.4.1 Observations

We first present observations about the nature of the crash-consistency bugs found by Chipmunk, and then present observations about crash-consistency testing.

3.4.1.1 Nature of crash-consistency bugs

Observation 1: Most of the observed bugs are logic issues rather than PM programming errors. Prior work on crash-testing PM applications focuses on bugs related to subtleties in the PM programming model, like CPU store reordering. However, the majority of bugs we found—19 of 23—are actually due to higher-level logic bugs rather than mistakes in managing PM. The "type" column in Table 3.1 classifies bugs into logic bugs or PM bugs. Logic bugs are issues that cannot be fixed by adding cache line flushes or store fences. These results suggest that it is not sufficient for a file-system crash-consistency testing tool to focus on exploring the persistence behavior of individual writes and reorderings; it must also check higher-level consistency properties that cannot be validated at the level of individual writes. We note that all bugs in found in SplitFS are logic bugs. Our results suggest that SplitFS's use of ext4-DAX to handle metadata operations reduces risk of PM programming errors, but does not eliminate logical bugs that impact crash consistency. All the bugs CHIPMUNK found in SplitFS are related to its optimized logging approach, which SplitFS uses to provide stronger crash-consistency guarantees than ext4-DAX.

Observation 2: In-place update optimizations are a common source of crash consistency bugs. One of the allures of PM is that programs can access it as memory, performing fine-grained reads and writes directly rather than coalescing them into larger block-sized I/O operations. This design makes it possible in principle to reduce the overheads of traditional consistency mechanisms like journaling by manipulating on-disk data structures directly. Most of the systems we tested use a journal for crash consistency, but have performance optimizations to bypass the journal in certain circumstances. For example, NOVA updates the link count of a file by updating a per-inode log. Appending to this log is usually done via a journaled transaction, but if the previous operation on the file also updated its link count, NOVA may modify that log entry in place.

We found these optimizations to be particularly error-prone: six of 23 bugs in

Table 3.1 are caused by in-place updates. For example, in bug 4, NOVA's `rename` implementation removes the directory entry from the parent inode in-place but journals the other metadata changes, allowing the file to be lost in a crash before the journal transaction commits.

Fixing these bugs often requires journaling more data. To quantify the impact of fixing such bugs, we compared the performance of NOVA before and after fixing two rename atomicity bugs (4 and 5). We tested both versions on Intel Optane DC Persistent Memory media. In a microbenchmark that repeatedly overwrites a file using `rename`, the fixed version is 25% slower. A more real-world metadata-intensive benchmark (checking out different stable versions in the Linux kernel git repository) shows negligible overhead (<1%). In some cases, journaling can even be better than in-place updates. The fix for bug 6 replaces an in-place update in `link` with extra logging, but makes a microbenchmark that repeatedly creates links to a file 7% faster, likely because checking whether the in-place update is safe requires an extra read from the media.

Observation 3: Rebuilding volatile state during crash recovery is error-prone. In a traditional file system, crash recovery scans on-disk structures like journals and updates the durable state to match. In contrast, PM file systems often keep metadata like free page lists in DRAM as a performance and write endurance optimization and rebuild them at mount. This rebuilding code is subtle because it must account for potential inconsistencies or partial states after a crash, and we found that nine of the 23 bugs in Table 3.1 were in such code. For example, bug 13 can be caused by a crash during a `truncate` system call on PMFS. This operation first stores information about the truncation in a "truncate list"; if the system crashes before the truncation is complete, the truncate list can be replayed to finish the operation. However, replaying truncations requires accessing the free page list, which is kept in DRAM and thus lost in the crash. Attempts to replay truncations therefore cause a null pointer dereference.

Rebuilding volatile state is more complex in PM file systems that maintain *per-CPU* volatile state to improve scalability. For example, in bug 19, WineFS failed to properly index into an array of per-CPU journals that were read during crash recovery, preventing journaled updates from being accessed after a crash.

Observation 4: Resilience mechanisms to recover from media failures can introduce new crash-consistency bugs. NOVA-Fortis [267] is an extension of NOVA that adds fault detection and tolerance for media errors and software bugs by (among other techniques) replicating and/or checksumming inodes, logs, and file data. While NOVA-Fortis is not explicitly designed to increase crash resilience, we tested it to determine if it is more tolerant of crashes than NOVA.

NOVA-Fortis has all the same crash-consistency bugs we found in the original version of NOVA, and in addition has four new bugs caused by the added complexity of maintaining redundant state and checksums. A common theme in these bugs is that data and metadata modifications are often not atomic with checksum and replica updates, allowing checksum validation to fail (and render a file inaccessible) even if the file system is consistent and data intact.

3.4.1.2 Crash-consistency testing in PM file systems

Observation 5: Many observed bugs require simulating crashes during system calls. Current crash-consistency testing tools for traditional file systems, like CrashMonkey [201] and Hydra [143], insert crashes only after `fsync`-related system calls. This heuristic exploits the fact that most POSIX APIs only make crash-consistency guarantees after persistence operations, so intermediate states are unlikely to violate the specification. It allows these tools to scale to test larger workloads, and does not appear to cause them to miss bugs: CrashMonkey has a mode to insert crashes during system calls, but it did not find any additional bugs.

We found that this same heuristic does not work for PM file systems. 11 of the 23 bugs in Table 3.1 require a crash to occur during a system call. This is a

corollary of our observation that most PM file systems implement most system calls synchronously, making their effects persistent by the end of the system call. For example, the rename atomicity bugs in NOVA (bugs 4 and 5) arise when a crash during the system call leaves only some writes persisted. Waiting until the system call completes would hide these bugs, as NOVA flushes all writes by the end of the operation.

Observation 6: Short workloads suffice to expose many crash-consistency bugs. We use ACE [201] to exhaustively generate small test workloads. ACE's design is based on an empirical study of historical crash consistency bugs in traditional file systems that showed that most bugs could be reproduced with at most three core operations. It was unclear whether this would hold for PM file systems. However, 19 of the 23 bugs we found in PM file systems can be found using ACE, suggesting that this same *small-scope hypothesis* [126] holds for PM file systems. We also run Chipmunk using the Syzkaller gray-box fuzzer, which can generate much longer workloads but without the exhaustiveness guarantees of ACE (§3.2.4). Syzkaller found four bugs that ACE did not. However, all four bugs were found on short workloads: three would be considered seq-2 and one seq-3. ACE missed them not because of size but because of complexities that ACE omits to make exhaustive enumeration tractable, such as testing non-8-byte-aligned writes.

Observation 7: Most of the observed buggy crash states involve few writes to PM. Chipmunk generates crash states by snapshotting known-persistent disk states between store fences, and then replaying all subsets of the in-flight writes between each store fence (§3.2). For a system call with n in-flight writes before a fence, this means Chipmunk should consider all $2^n - 1$ possible crash states. However, we found that most bugs found by Chipmunk involve crash states that include *small* subsets of the in-flight writes. Of the 11 bugs in Table 3.1 that involve a crash in the middle of a system call, 10 can be exposed by a crash state that replays only a *single* write onto the last known-persistent state; the final bug requires two writes.

This observation suggests a profitable heuristic would be to only test small subsets of in-flight writes. CHIPMUNK exploits this observation by enumerating crash states in increasing order of subset size, allowing it to find most crash-consistency bugs quickly. In our experiments, we often cap the number of writes that are replayed to build each crash state, primarily to prevent Syzkaller from spending many hours checking a single outlier test with a high in-flight write count. The highest in-flight write count we observed, 20 writes in some PMFS `write` calls, would take about 30 hours to check exhaustively using CHIPMUNK. A cap of two is enough to find all bugs presented in this paper; a cap of five is sufficient to check all crash states for most system calls in the PM file systems we tested.

3.4.2 Lessons Learned

Based on our observations above, we have distilled three lessons for developers of PM file systems and for building the testing tools that support them.

Lesson 1: Synchronous crash consistency on PM file systems simplifies the user experience, but complicates implementation and testing. Crash-consistency guarantees in modern file systems are something of a vicious cycle. File-system developers argue that relaxed guarantees are required to extract reasonable performance [176], but these weak guarantees are a pain point for application developers and have caused severe data loss in popular applications [21,50,215], so file-system developers implement workarounds to "fix" common mistaken application patterns and make the intended guarantees even less clear. The fine write granularity and low latency of PM finally offers a path to strengthen file-system crash-consistency models, making resilient applications easier to build and validate. PM file system developers have taken advantage of this opportunity by making all system calls synchronous and durable.

While this end result is exciting, implementing it correctly carries new risks for PM file-system developers compared to traditional file systems. We found that many

PM file-system bugs come from complex optimizations to realize high-performance synchronous crash-consistency — combining in-place updates with other consistency mechanisms, replacing persistent state with reconstructible volatile state, or introducing new logging protocols — that are uncommon techniques on slower storage media. This is a rich new design space for storage systems, and identifying the right primitives for these optimizations will be good future work. These optimizations also create complexity for testing and validation of PM file systems, which we found requires driving the file system into exercising deeper data structure manipulations and recovery mechanisms than existing crash-consistency tools are capable of.

Lesson 2: Diverse testing mechanisms and checkers help invalidate assumptions about crash-consistency patterns. Most crash-consistency testing tools build on heuristics and patterns in historic bugs to select the workloads they test. We expected to bring those patterns across to PM file systems, focusing on short workloads and a small set of potential crash points. However, we found instead that most assumptions about file-system crash consistency do not carry across to PM, where the consistency mechanisms and guarantees are significantly different. Finding crash consistency bugs in PM file systems requires exploring many more crash states than other file systems, including crashes in the middle of system calls; we had to develop new techniques to make this search tractable. We also found that fuzzing was an effective way to invalidate assumptions from prior file systems experience, such as the significance of unaligned writes and exercising per-CPU code paths.

Another assumption we carried into this work was that the difficulty of building a PM file system lies in correctly applying the PM programming model. We intended to focus on exhaustively testing the precise persistency behavior of PM file system code. However, we found instead that most PM file system bugs were logic errors. Existing tools that focus on detecting specific PM programming error patterns [63, 82, 96, 121, 181–183, 210] would miss many of these bugs. Writing general-purpose consistency checks and applying gray-box fuzzing to generate workloads helped to

invalidate these assumptions.

Lesson 3: Lightweight testing offers a scalable approach to detecting many crash-consistency bugs. CHIPMUNK is, in principle, a bounded exhaustive testing [201] tool for PM file systems: given enough time, it can check every possible crash behavior of every possible workload up to some bounds on its size and inputs. Of course, it is not tractable to exhaust this search space even with very small bounds. However, we found that CHIPMUNK is an effective *lightweight* testing tool, in that it can quickly and automatically find many bugs by checking small workloads and few crash states, and then run for longer to find more corner-case issues. CHIPMUNK runs the ACE seq-1 workloads in less than 15 minutes on most tested systems. On the other hand, the fuzzer frontend to CHIPMUNK takes 1–2 orders of magnitude longer to run but finds four more bugs than ACE. These two frontends are complementary. They enable a lightweight approach that helps developers iterate quickly on new code, while offering stronger confidence as the code gets "closer to production" [24].

3.5 Summary

This chapter presents CHIPMUNK, a new record-and-replay framework for testing the crash consistency of PM file systems. We use CHIPMUNK with the ACE workload generator and the Syzkaller gray-box fuzzer and find 23 unique bugs across five PM file systems. To the best of our knowledge, this is the largest corpus of crash-consistency bugs on PM file systems. Our study of these bugs provides insights into how crash-consistency bugs arise in PM file systems and what types of tools are needed to test these systems.

Chapter 4: Statically checking crash consistency using Rust

In this chapter, we introduce SquirrelFS, a PM file system that uses the Rust programming language to obtain some statically-checked crash-consistency guarantees. We motivated the need for new techniques and tools to help developers build robust PM storage systems in Chapter 3, and we now discuss one such technique that provides a lightweight way to gain confidence in the crash consistency of system. Our static checking approach provides stronger guarantees of robustness than testing with tools like Chipmunk. These checks do not obtain the guarantees that formal verification offers, but they require no proofs or specialized language support, making them more accessible to engineers.

In §4.1, we motivate the development of SquirrelFS by describing the limitations and challenges of other approaches used to ensure crash consistency. We also describe the key insights about PM, soft updates, and the Rust programming language that make SquirrelFS possible. In §4.2, we describe the design and implementation of SquirrelFS and how we checked its design using the model checker Alloy [125]. We also provide detailed information on the durable update dependencies checked by the Rust compiler in SquirrelFS. In §4.3, we discuss our experience developing SquirrelFS and its Alloy model. §4.4 presents our evaluation of SquirrelFS' performance and correctness. In §4.5, we discuss the theoretical limitations and guarantees of the typestate pattern. We also compare Rust's typestate support to that of several other typestate-oriented languages.

This chapter is based on the paper "SquirrelFS: using the Rust compiler to check file-system crash consistency" [167] published at OSDI 2023. It also contains content from an extended version of this paper accepted to ACM Transactions on Storage in 2025.

4.1 Motivation

We first motivate the need for a crash-consistency approach that provides some static guarantees without proofs or verification. We also describe in more detail the key observations that SquirrelFS is based on.

4.1.1 Crash consistency

As discussed in §2.2, there exist many mechanisms that can be used to protect the integrity of data and metadata in a storage system in the event of a crash. However, ensuring that the design and implementation of a mechanism achieves crash consistency is challenging. There are currently two primary approaches. First, as discussed in Chapter 3, we can test systems for crash consistency using specialized tools like Chipmunk. While such testing tools can find many bugs, they cannot prove overall correctness or the absence of crash-consistency bugs. Second, we can verify the correctness and crash-consistency of the system. Verification is stronger than testing in that it can prove that the system is bug-free, but it also requires developers to write proofs that the implementation matches a specification of correctness, which is difficult and time-consuming. Prior work on verifying crash consistency required 7–13 lines of proof for every line of code. We discuss this approach further in Chapter 5.

Recent work has explored a middle ground between testing and verification. Corundum [113] is a Rust crate for PM systems that, like SquirrelFS, uses the Rust type system to enforce certain low-level PM safety properties at compile time. For example, Corundum ensures that every update to PM occurs in a logged transaction, and prevents the storage of pointers to volatile memory in durable structures. SquirrelFS was inspired by Corundum and aims to enforce higher-level properties like file-system crash consistency with Rust.

```
fn new_file() {
  // Dentry<Free>
  let d = Dentry::get_free_dentry();
  // Inode<Free>
  let i = Inode::get_free_inode();
  // Dentry<Init>
  let d = d.set_name("foo");
  let d = d.commit_dentry(i);
                ^ expected `Inode<Init>`, found `Inode<Free>`
}
```

Figure 4.1: **Invalid typestate example.** The listing shows the typecheck process throwing an error when an uninitialized inode is passed to a function that expects an initialized inode.

4.1.2 The opportunity: Rust and PM

We observe an opportunity to ensure file-system crash consistency in a cheap manner. First, we note that the Rust programming language can **statically enforce a specific order on operations** via its support for the *typestate pattern* [88, 233], described in §2.5.2. We can enforce crash-consistency properties at compile time by encoding pertinent information in the types of durable objects.

For example, one consistency rule enforced by soft updates is that a directory entry should never point to an uninitialized inode. Figure 4.1 shows how typestate is used to enforce this rule. To create a new file, we first obtain a free directory entry and inode. Initially, both objects have typestate `Free`. Then, we initialize the directory entry, transitioning its type to `Dentry<Init>`. The listing then has a bug in which the directory entry's inode number is set by `commit_dentry()` before the inode is initialized, breaking the consistency rule. The Rust compiler catches this bug because the inode's current typestate `Free` does not match the typestate `Init` expected by the function.

Since soft updates is entirely built on ordering updates to file-system objects, we can translate the required partial order into a set of types and use Rust's type

checking to enforce the order. Thus, the invariants we want to maintain are translated into something the type system and compiler can enforce. We note that we are able to do this with an *unmodified* Rust compiler; the new types introduced are no different to the compiler from existing types in the codebase.

However, implementing soft updates correctly remains challenging even with typestate support. With soft updates, file-system updates are applied to the page cache in DRAM, and then later written to storage in the right order. Determining the right order requires tracking complex dependencies across asynchronous operations. When a single file-system metadata object (such as an inode or a bitmap) is updated multiple times, it can lead to cyclic dependencies.

This leads to our second observation: persistent memory (PM) file systems support synchronous operations thanks to the low latency of the storage media [265, 269]. These file systems write updates directly to storage without first caching them in DRAM [69, 132, 133, 155, 266]. A **synchronous implementation** of soft updates for persistent memory **eliminates** the complexities of asynchronous dependency management, greatly simplifying the mechanism and allowing the relevant invariants to be encoded in Rust's type system.

4.2 SquirrelFS

We now present the design and implementation of SquirrelFS, a novel file system that uses the unmodified Rust compiler to check its crash consistency. If the compilation is successful, it indicates that the ordering-based invariants hold throughout the file system: in other words, the checking is complete. If compilation fails, the error reported by Rust is useful in figuring out which operations are out of order. Compilation takes only seconds, offering quick feedback to developers.

SquirrelFS is built on two key ideas:

- A novel crash-consistency mechanism, Synchronous Soft Updates, that achieves

crash consistency purely via ordering file-system operations (§4.2.1)

- Using the Rust typestate pattern to encode ordering invariants into the Rust type system (§4.2.2)

It is important to note that we are not modifying the Rust compiler in any way. To the Rust compiler, it is no different from type-checking any other code base; we are merely using the type checking to ensure that crash consistency holds in the file system.

We now describe the key ideas in more detail.

4.2.1 Synchronous Soft Updates

We develop *Synchronous Soft Updates* (SSU), a novel crash-consistency mechanism. SSU is based on the traditional soft updates approach, but differs in two key aspects. First, soft updates was designed for asynchronous settings, but all operations are synchronous in SSU. Second, soft updates does not provide atomic rename; a crash during a rename of `src` to `dst` can result in both being present after a crash. SSU fixes this flaw; renames are atomic, and a crash during rename will result in either `src` or `dst` after recovery.

We now discuss why we developed SSU, its key aspects, and how renames are atomic in SSU.

Why a new mechanism? To go with our overall approach of encoding ordering-based invariants into the Rust type system, we needed a mechanism that achieves crash consistency purely via ordering file-system updates. This rules out mechanisms such as journaling and copy-on-write that use writes to a log or an extra copy to obtain atomicity. Soft updates [187] obtains crash consistency by enforcing ordering on in-place persistent updates to file-system objects; thus, it was a good match. However, traditional soft updates suffered from two problems that we needed to tackle. The first challenge was that soft updates had significant complexity arising from needing

to track dependencies between asynchronous file-system operations; the presence of cyclic dependencies also requires complex roll-back and roll-forward logic. The second challenge is that soft updates does not provide atomic operations, particularly rename; atomic rename is a crucial primitive for a number of POSIX applications [215]. Thus, we need to modify soft updates to tackle both its high complexity and lack of atomic operations.

Synchronous operations. We observe that the root of complexity in soft updates (such as cyclic dependencies and structures for tracking dependencies) is *asynchrony*. A *synchronous* implementation of soft updates neatly avoids these complexities. All updates would be made durable by the end of each system call, which would eliminate the need to track cross-operation dependencies. Cyclic dependencies would no longer arise because there are no pending updates that can conflict with each other. The SoupFS [67] soft updates file system for persistent memory eliminated cyclic dependencies using fine-grained updates, but still required asynchronous dependency tracking. A synchronous implementation is necessary to overcome both sources of complexity.

A synchronous version of soft updates was not feasible until now, as running this on magnetic hard drives or even solid state drives would be prohibitively slow. However, synchronous soft updates is a good match for persistent memory (PM) due to its low latency; system calls in many existing PM file systems are already synchronous [69, 132, 133, 266].

Similar to traditional soft updates, SSU maintains crash consistency by enforcing ordering among updates to file-system objects. SSU implements the original soft updates rules [84]:

1. Never point to a structure before it has been initialized;

2. Never re-use a resource before nullifying all previous pointers to it;

3. Never reset the old pointer to a live resource before the new pointer has been set.

These rules are significantly easier to enforce in a synchronous setting, as there is no need to track dependencies across asynchronous operations. Like soft updates, SSU focuses on the integrity of file system metadata and cannot guarantee that operations on file data are atomic. SSU could be combined with journaling or copy-on-write to obtain stronger data guarantees.

Atomic rename in SSU. SSU ensures renames are atomic by cleaning up file-system state after a crash. In traditional soft updates, if there is a rename from `src` to `dst`, it is impossible to tell after a crash whether `src` or `dst` should be removed. To resolve this, SSU adds an extra field, called the *rename pointer*, to directory entries in order to persistently save enough information to complete the rename operation after a crash. The rename pointer in the destination directory entry points to the physical location of the source directory entry. The rename pointer allows the file system to follow soft updates rule 3 (never reset the old pointer before the new one has been set) while also retaining the ability to distinguish between `src` and `dst` after a crash.

Note that this is similar to what journaling-based file systems do; they write a log entry specifying `src` and `dst` so that the right clean-up action can be performed. In SSU, the information in this log entry is distributed over the source and destination inodes; taken together, they provide enough information to the file system.

Figure 4.2 illustrates the process. Step ❶ shows an example system state prior to the `rename` operation. In ❷, `dst`'s rename pointer (dotted line) is set to `src`. `dst` is invalid, and `src` is still valid. In ❸, we make `dst` valid; this also logically invalidates `src`. This is an atomic point; after this step, the file system will always complete the rename operation. If the file system crashes prior to this step, the rename pointer is cleared on recovery. In ❹, we physically mark `src` as invalid. In ❺, the rename pointer is cleared, and in ❻ `src` is fully deallocated. Each step

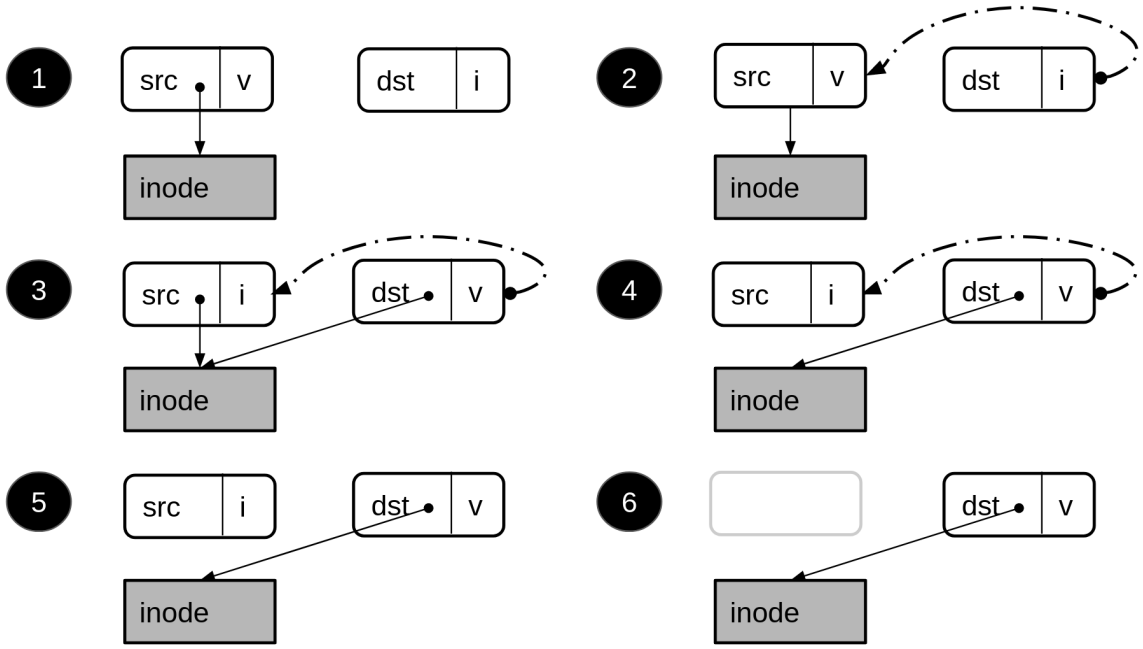

Figure 4.2: **Atomic `rename`.** The figure shows the steps in atomic soft updates `rename`. The dotted lines represent rename pointers and the solid lines represent inode pointers. `src` and `dst` are directory entries. The labels "v" and "i" indicate whether a directory entry is valid or invalid.

either modifies metadata that is invisible to the user (e.g., deallocating an orphaned directory entry) or atomically modifies a single 8-byte value. All modifications must be durable before proceeding to the next step.

Rename recovery. A question that arises is how the file system finds `src` and `dst`. This is an example of how SSU is tailored for PM file systems. In PM file systems, it is common for the file system to scan persistent objects to construct indexes in DRAM; we add the rename-recovery procedure into this scan. Thus, when building volatile indexes after a crash, the file system also looks for and completes any partially completed rename operations.

A pseudocode Rust implementation of the rename recovery procedure is shown in Listing 4.1. For simplicity, this code is written without typestates. We describe the typestates used in `rename` and its recovery in §4.2.5. During both standard and post-crash remount, SquirrelFS first scans inodes and page descriptors to create an index mapping live inodes to their pages. After a crash, SquirrelFS passes this index to `rename_recover` (line 1 of Listing 4.1) and scans all directory entries in live directory pages (lines 2 and 3). For each directory entry, we first check if its rename pointer is set (line 4). If it is not set, then the directory entry was not the destination in an interrupted and incomplete `rename`. It may still require cleanup if it is the source for an interrupted `rename`, in which case it will be handled when we scan the corresponding destination. If the rename pointer is set, we know that this directory entry was the destination of a `rename` that crashed in step ❷, ❸, or ❹. We can now use the inode pointers of the source and destination directory entries to determine what steps are necessary for cleanup. If they do not point to the same inode, we either crashed in Step ❷ or ❹. Crash recovery is the same in both cases and only requires clearing the rename pointer (line 10). If we had crashed in Step ❷, this will effectively roll back to before the operation; if it was Step ❹, this will transition the system to Step ❺. Otherwise, if the two directory entries point to the same inode, we must have crashed in Step ❸. In this case, we recover by picking up where the

```
1  fn rename_recover(pages: &PageIndex) -> Result<()> {
2    for page in pages.filter(|p| p.get_type() == DIR) {
3      for d in pages.dentries() {
4        if d.rename_ptr().is_set() {
5          // d was a dst dentry in an interrupted rename and
6          // we crashed at step 2, 3, or 4.
7          let src = d.get_src_from_rename_pointer();
8          if d.get_ino() != src.get_ino() {
9            // we crashed at step 2 or 4
10           d.clear_rename_pointer().flush().fence();
11         } else {
12           // we crashed at step 3
13           src.clear_ino().flush().fence();
14           d.clear_rename_pointer().flush().fence();
15           src.dealloc_dentry().flush().fence();
16         }
17       } // else, no cleanup needed for now
18   }}}
```

Listing 4.1: The listing shows the recovery procedure to clean up interrupted `rename` operations.

`rename` operation left off and complete it using the same functions used for this part of `rename` during normal operation (lines 13–15). This process may leave orphaned structures, which will be cleaned up later in recovery.

4.2.2 Using Rust to enforce ordering

Rust's typestate pattern can be used to ensure that *a set of functions are always called in certain partial order*. A total order is not necessary, as many operations involve independent updates that can be safely reordered. As we discussed previously (§4.1), an object's typestate is encoded in generic type parameters in its definition, and the partial order is encoded in the function signatures of its associated functions.

We encode two states (as different type parameters) in the types of persistent objects:

```
impl Inode<Clean, Free> {
  fn init_inode(self, ino: u64, ...) -> Inode<Dirty, Init> {...}
}
impl Dentry<Clean, Alloc> {
  fn commit_dentry(self, inode: Inode<Clean, Init>)
    -> Dentry<Dirty, Committed> {...}
}
impl<S> Inode<Dirty, S> {
  fn flush(self) -> Inode<InFlight, S> {...}
}
impl<S> Inode<InFlight, S> {
  fn fence(self) -> Inode<Clean, S> {...}
}
```

Listing 4.2: Pseudocode implementations of file system objects with persistence and operational typestate. Typestate arguments are shown in bold.

- *Persistence* typestate is a representation of whether an object's most recent update(s) have been made durable. We use three persistence typestates: `Dirty`, `InFlight`, and `Clean`.
- *Operational* typestate represents the operations that have been performed on an object and is used to determine what operations can happen next.

Persistence and operational typestate are separate to capture the fact that most storage devices do not synchronously flush updates. For example, in persistent memory, updates go to the CPU caches first, and must be explicitly flushed to the persistent media.

Listing 4.2 shows implementations of several methods of persistent `Inode` and `Dentry` types with persistence and operational typestate as generic type parameters. The methods `flush` and `fence` invoke a cache line write back and store fence respectively and are generic with respect to operational typestate. These methods must be used to ensure updates are persistent before continuing; for example, `commit_dentry()` requires an `Inode<Clean, Init>` to ensure the inode's initialization cannot

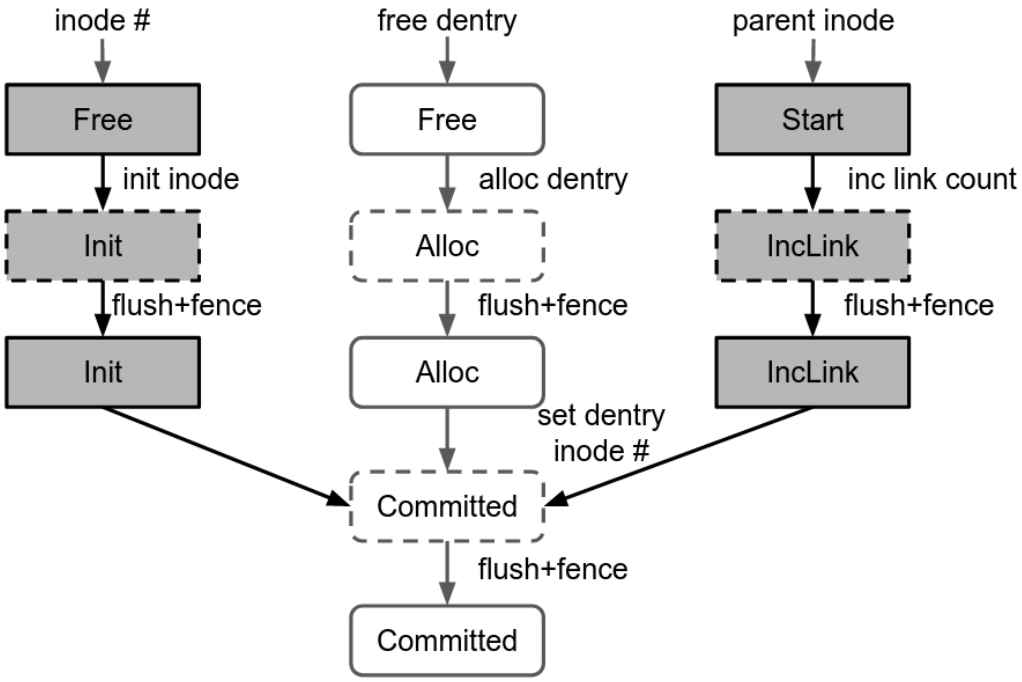

Figure 4.3: `mkdir` **dependencies.** The figure shows persistent updates and corresponding dependencies made during `mkdir`. Inodes are dark gray and directory entries are white. Each object is labeled with its operational typestate and its outline indicates whether it is clean (solid) or dirty (dotted).

be transparently reordered with the directory entry updates.

This formulation of persistence typestate has several performance benefits. First, because the `flush` and `fence` methods can only be called on an object whose typestate indicates it is not yet persistent, typechecking will prevent redundant persistence operations (thereby improving performance). Second, developers can wait to flush updates until it is strictly necessary and can write additional transitions to enable multiple updates to share a single fence.

Why Rust? In order to obtain useful compiler-checked guarantees from the typestate pattern, each object must have exactly one typestate [243]. Languages like C and C++ cannot enforce this restriction, but Rust can, via its ownership-based type system. See subsection 2.5.2 for more detail.

4.2.3 Examples

To illustrate the typestates and dependency rules used in SSU, we describe two example operations, `mkdir` and `unlink`. The typestates shown in these examples are described in more detail in Table 4.1.

Example 1: `mkdir`. We first describe the dependency rules in an SSU implementation of `mkdir`, shown in Figure 4.3. To be crash consistent, an SSU implementation of `mkdir` must ensure (1) that a structure never points to an uninitialized resource, and (2) that each inode's link count is greater than or equal to its actual number of links. Both rules prevent dangling links in the event of a crash.

During `mkdir`, three file-system objects are modified: an inode for the new directory, a directory entry for the new directory, and the inode of the parent directory. In Figure 4.3, inodes are represented by dark gray boxes and directory entries are represented by white boxes. Each durable object is labeled with a typestate representing its current status at each point during the operation. The full set of typestates and dependencies for each durable structure are shown in Figure 4.9 for inodes and Figure 4.10 for directory entries. The initial updates to each data structure are independent, so there are no dependencies between them, and they can share a single store fence to ensure durability before setting the directory entry's inode number (not shown). SquirrelFS uses volatile allocation structures that are not persistent during `mkdir`.

The system first finds the parent inode and obtains a free directory entry in one of the parent's pages as well as a free inode. The inode is then initialized (i.e., setting its inode number, link count, timestamps), the directory entry's name is set, and the parent inode's link count is incremented.

Next, we commit the directory entry by setting its inode number. This makes the directory entry valid and connects the inode to the file system tree. Directory entry commit is dependent on inode and directory entry initialization and parent link increment. Committing the directory entry before initializing the inode can result in

a directory entry pointing to a garbage inode; committing before incrementing the parent's link count can lead to dangling links.

Example 2: `unlink`. To show how SSU handles multiple links to files and operations involving deallocations, we now describe how `unlink` works. Figure 4.4 shows the dependencies between each step in this operation. As before, each node represents a durable object at each step in the operation, with inodes in dark gray, directory entries in white, and data pages in gray. Figures 4.8 (inodes), 4.10 (directory entries), and 4.11 (data pages) show the full dependency relationships for each of these structures.

The `unlink` operation is initially passed a volatile directory entry structure maintained by the Virtual File System layer (VFS). We use this to look up the durable inode and directory entry to unlink, both of which are initially in the `Start` typestate. First, we clear the directory entry's inode, which makes it invalid for future lookups. The inode's link count is now greater than the number of directory entries pointing to it, but this cannot cause dangling links in the event of a crash and is thus safe. We can now durably decrement the inode's link count, which requires passing in an immutable reference to the directory entry (shown in Figure 4.4 with a dotted arrow) to ensure the link count is not decremented before a linked directory entry has been cleared. The directory entry can now be deallocated. If the freed directory entry was the last live entry in a page, that page may now be safely deallocated. We omit these steps for brevity, but they are similar to those taken to deallocate data pages later in this example.

The next step depends on whether the inode's link count is now zero. This is checked at runtime by a function that returns the inode in the `Complete` state if the link count is greater than zero, and returns both the inode in `UnmapPages` state and a list of pages in `ToUnmap` state if the link count is zero. If the inode's link count is greater than zero, there is still at least one directory entry pointing to it, so the inode and associated pages should not be freed, and the operation is complete. If the link

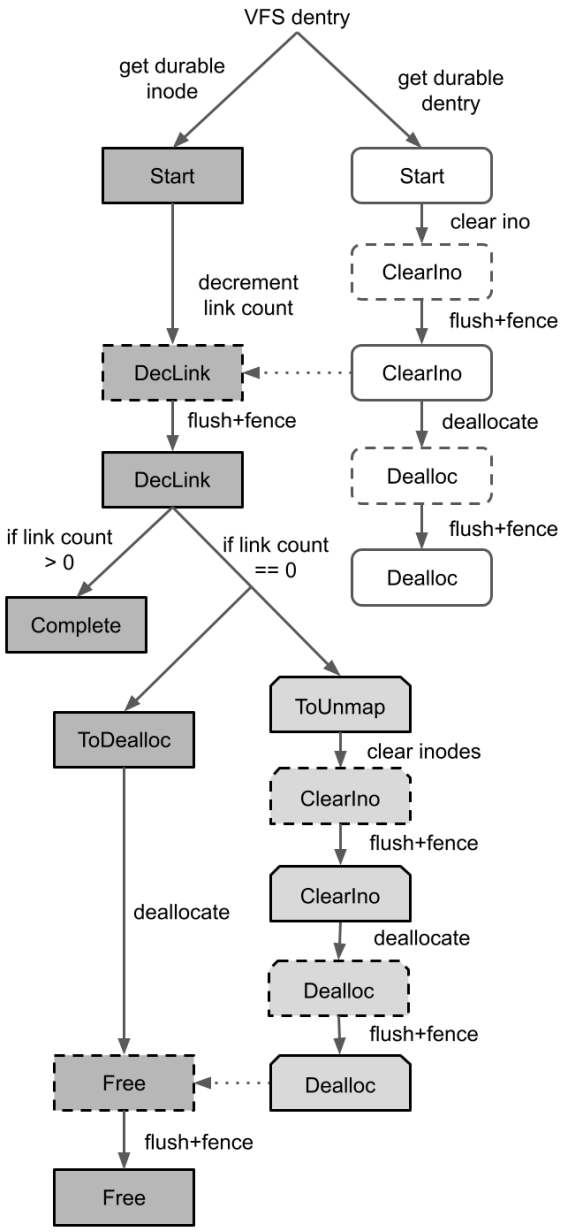

Figure 4.4: `unlink` **dependencies.** The figure shows the persistent updates and corresponding dependencies made during `unlink`. Inodes are dark gray, directory entries are white, and page lists are light gray. Each object is labeled with its operational typestate and its outline indicates whether it is clean (solid) or dirty (dotted). A dotted arrow indicates that the typestate transition requires a reference to an object in a particular typestate but does not modify that object's typestate.

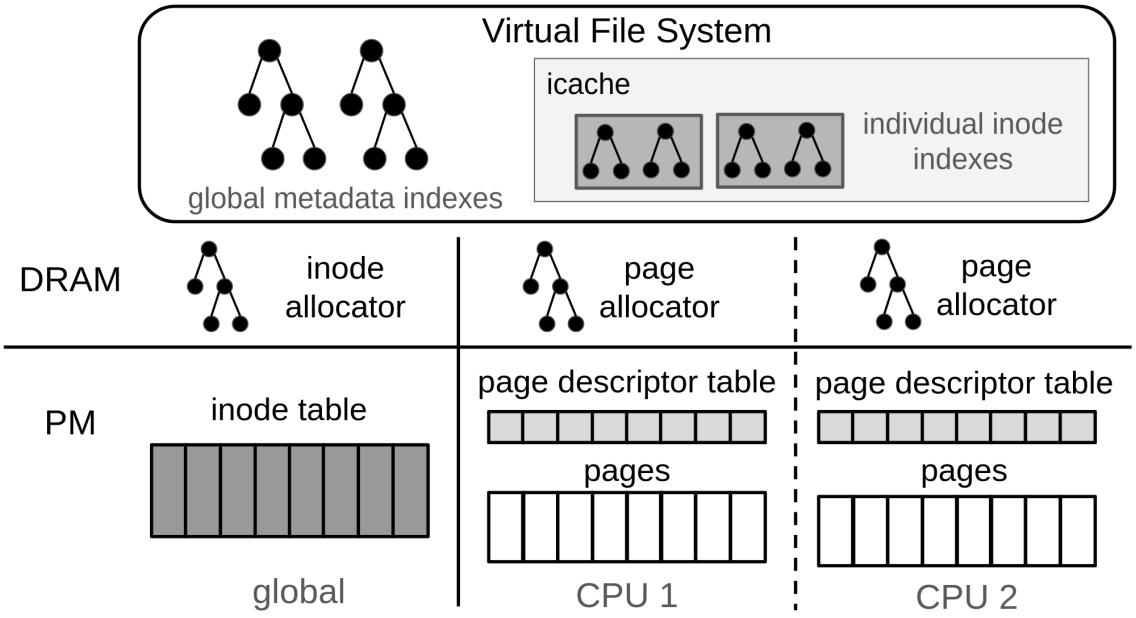

Figure 4.5: **SquirrelFS overview.** The figure shows the main components of SquirrelFS. Each CPU has its own pool of pages and private page allocator. The inode allocator is shared between all CPUs. Volatile indexes are stored in VFS data structures.

count is zero, there are no more links to the inode, so the inode and its pages can be deallocated. SquirrelFS uses a backpointer-based page management approach (§4.2.4) in which each page points to the inode that owns it, so the pages must be deallocated before the inode to prevent dangling pointers in the event of a crash. If, instead, the inode pointed to a structure containing the locations of its pages, the inode would need to be freed first.

4.2.4 Implementation

We implemented SquirrelFS in Rust with 7500 LOC. It uses bindings from the Rust for Linux project [232] to connect to the Linux Virtual File System (VFS) layer. Figure 4.5 shows SquirrelFS's architecture. We also built a model of SquirrelFS in the model-checking language Alloy [125] to check its design for crash consistency issues. We describe our experience developing SquirrelFS in §3.4.2.

Overview. The design of SquirrelFS combines aspects of FreeBSD's FFS [189] and PM file systems such as NOVA [266] and WineFS [133]. Like FFS, it has a simple on-storage layout, and uses soft updates. Like other PM file systems, SquirrelFS uses volatile index structures that are built when the file system is mounted.

SquirrelFS's design was primarily influenced by two factors. First, we wanted to keep dependencies as simple as possible and avoid nested persistent structures that are difficult to represent in typestate. Second, we assume the x86 PM persistence model in which only aligned updates of 8 bytes (or smaller) are crash atomic. We also assume volatile CPU caches. In this model, persistent addresses can be accessed via regular memory stores, but the corresponding cache line must be flushed or written back before updates become persistent. A memory barrier like a store fence must also be invoked to correctly order stores [229]. Durable structures may also be updated via cache-bypassing non-temporal store instructions, which still require a store fence for persistence ordering. This programming model influences the structure of persistent objects and restricts the set of legal orderings. SquirrelFS also supports systems with durable CPU caches (e.g., with eADR), which eliminate the need for explicit cache line write backs. Typestate checking is also useful for these systems, as store fences are still required to order memory stores and the high-level update dependencies are the same as with volatile CPU caches.

All system calls in SquirrelFS are synchronous, meaning that updates to durable structures made by each system call are durable by the time the system call returns. As such, `fsync` is a no-op in SquirrelFS. Metadata-related operations are also crash-atomic. Data-related operations are not atomic in the current SquirrelFS prototype, which matches the default behavior of other PM file systems like NOVA [266]. These operations could be made atomic by using copy-on-write to update file contents.

Persistent layout. SquirrelFS uses a simple layout to reduce the complexity of update dependencies. At mount time, the entire PM device is mapped into

SquirrelFS's address space using DAX (direct access) management functions provided by the kernel [177]. SquirrelFS divides this mapped region into four sections: the superblock, the inode table, the page descriptor table, and the data pages. The inode table is an array of all the inodes in the system. SquirrelFS reserves enough space for approximately one inode for every 32KiB of data (eight pages).

The page descriptor array contains page metadata. Rather than having inodes point to the pages they own, each page descriptor contains a backpointer to its owner (similar to NoFS [44]) and stores its own metadata (e.g., its offset in the file). This approach simplifies dependency rules for updates involving page allocation and deallocation. All remaining space after the page descriptor table is used for data and/or directory pages.

All PM allocations are managed internally by SquirrelFS. As we discuss next, these allocators are kept only in volatile memory, so there is no allocator-related metadata stored on PM.

Volatile structures. SquirrelFS's persistent layout simplifies typestate and update dependency rules, but it is not amenable to fast lookups. Therefore, SquirrelFS uses indexes in DRAM to speed up lookup and read operations. Each inode in the VFS inode cache has a private index for the resources it owns; index data for uncached nodes is stored in the VFS superblock.

Like many other PM file systems, SquirrelFS uses volatile allocators: allocation information is not stored in a persistent manner, but rather rebuilt each time the file system is mounted. It uses a per-CPU page allocator and a single shared inode allocator (which could be converted to a per-CPU allocator to improve scalability). The allocators use free lists backed by kernel RB-trees.

SquirrelFS's indexes and allocators are rebuilt by scanning the file system when SquirrelFS is mounted. Any persistent data structure that is zeroed out is considered free during the rebuild scan and is added to the corresponding free list. During the scan, SquirrelFS keeps track of allocated objects and uses them to

traverse the file system tree and build the in-memory indexes. Data structures with any non-zero bytes are counted as allocated. After a crash, it is possible for an object to be allocated but invalid, e.g., a directory entry with a name but no inode pointer, or an inode that is not reachable from the root. These objects are zeroed out and added to the free lists during recovery. Thus, only updates that change an object's validity or update user-visible state in place need to be crash-atomic, since any invalid objects will be cleaned up after a crash.

Synchronous Soft Updates. SquirrelFS uses an implementation of SSU for crash consistency. As shown in Figure 4.3, operations that involve creation of new objects must first durably allocate and initialize resources before linking them into the file system (setting the directory entry's inode in the example) to enforce rule 1 (never point to a structure before it has been initialized). Deallocation proceeds in reverse; links are first cleared, then the object itself is deallocated by zeroing all of its bytes. SquirrelFS enforces rule 2 of soft updates (never re-use a resource before nullifying all previous pointers to it) by treating durable objects that are not completely zeroed out as allocated and by ensuring via typestate that pointers to the object are cleared before the object can be zeroed.

Typestate transition functions. SquirrelFS updates the typestate of objects via *typestate transition functions*. These functions take ownership of the original object, modify it, and return it to the caller with the new typestate. These functions are defined only on certain typestates to ensure they are called in a safe order. For example, the typestate transition function `commit_dentry()`, shown in Listing 4.2, is only defined for directory entries with type `Dentry<Clean, Alloc>`, and also takes ownership of an inode of type `Inode<Clean, Init>`. Calling `commit_dentry()` out of order — e.g., on a directory entry that has not yet been persistently allocated — is a potential crash-consistency bug and results in a compiler error.

Concurrency. SquirrelFS supports concurrent file-system operations. It relies on VFS-level locking on durable resources like inodes. This locking, together with Rust's

type system, ensures that each resource has only one owner — and only one type — at any time, enabling strong typestate-based compile-time checking. SquirrelFS uses internal locks to protect its allocators and indexes.

Building a model with Alloy. While the typestate pattern can enforce a given operation order, it cannot verify that this order is crash consistent. To gain more confidence that SquirrelFS's design is crash consistent, we built a model of SquirrelFS in the Alloy model checking language [125].

Alloy provides a language for specifying transition systems and a model checker to explore possible sequences of states (traces) of these systems. Alloy's implementation is based on a logic of relations; each system is composed of a set of constraints that define a set of structures and the relations between them, and the model checker uses constraint solving to find traces.

In SquirrelFS, there is roughly a one-to-one mapping between typestate transitions in the Rust implementation and the next-state predicates in the Alloy model. Each next-state predicate specifies the states in which the transition may occur and the changes it makes to the model's state. The model includes next-state predicates for typestate transitions and persistent updates. It also includes transitions that model crashes and recovery, which let us check SquirrelFS's design for crash-consistency bugs.

Each persistent structure in SquirrelFS is represented by a corresponding structure, also called a signature, in Alloy. The model also includes a `Volatile` signature that is used to model volatile aspects of the file system like its indexes. Each typestate is represented by a signature, and instances of persistent structures are mapped to their current typestate. Each file system operation is also represented by a signature, and relations map system calls to instances of persistent objects they are operating on as well as other volatile state (e.g., the locks held by that operation). We use this to model concurrent file-system operations.

```
pred inc_link_count_mkdir [i: DirInode, op: Mkdir] {
  // *Guards* specify when this transition can occur
  initialized[i] and op in i.i_rwsem_excl and
  lt[i.link_count, max] and isFalse[Volatile.recovering]

  // *Frame conditions* specify what parts of system
  // state are unchanged by the transition
  inode_values_unchanged_except_lc
  pointers_unchanged
  dentry_names_unchanged
  (all i0: Inode - i | unchanged[i0.link_count]
    and unchanged[i0.prev_link_count])
  (all o: PMObj - i | unchanged[o.typestate])
  locks_unchanged
  ops_unchanged_except_mkdir[op]
  unchanged[op.inode_typestate]
  unchanged[op.dentry_typestate]
  page_values_unchanged
  Volatile.recovering' = Volatile.recovering
  Volatile.allocated' = Volatile.allocated

  // *Effects* specify how the transition changes system state
  make_dirty[i]
  i.link_count' = add[i.link_count, 1]
  i.prev_link_count' = i.prev_link_count + i.link_count
  i.typestate' = IncLink
  op.parent_inode_typestate' = IncLink
}
```

Figure 4.6: **Alloy example.** The listing shows a transition predicate in Alloy that increments a directory inode's link count during a `mkdir` operation.

Alloy model example. Figure 4.6 contains the source code of a transition that increments an inode's link count during a `mkdir` operation. Transitions in Alloy models are defined as predicates that are true in steps where the transition is applied and false at all other times. The predicate is evaluated based on the conjunction of each line in its body. The next value of a variable is referenced with `'` syntax; for example, `i.typestate' = IncLink` is true if `i`'s typestate in the next step of the trace is `IncLink` (Alloy uses single-equals for equality).

This predicate takes two arguments: the inode to modify and a representation of the `mkdir` operation. Transition predicates Alloy models are generally organized into guards, frame conditions, and effects. The guards specify what must be true in a given state for the transition to occur. In this predicate, the operation must hold the inode's lock, the inode's link count must be less than the maximum, the system must not be recovering from a crash, and an `initialized` predicate must hold. Incrementing an inode's link count is always safe in soft updates, so this operation has relatively weak guards. A transition's guards describe the typestate dependencies of the operation as well as aspects like which locks that are held during the operation and runtime error handling.

The frame conditions describe what parts of the system state may not change in this transition and must include all mutable parts of system state that should not be modified. Frame conditions do not directly correspond to any part of typestate transitions written in Rust, but are necessary in Alloy as the model checker is free to arbitrarily change aspects that are not captured by the frame conditions. Our model contains a set of macros and predicates to make frame conditions more concise; for example, `locks_unchanged` specifies that no locks are acquired or released in this transition.

The effects specify what parts of the system state do change during this transition. Some effects correspond directly with a durable or typestate update in the implementation of the corresponding transition function in SquirrelFS; for exam-

ple, line 22 of Figure 4.6 specifies that the inode's persistence typestate becomes `Dirty`, and line 23 specifies that its link count is incremented. The model also tracks previous link count values between the last flush/fence and this update so that it can roll back to previous values in the event of a crash. Line 24 specifies that the current link count is added to the set of previous link counts for this inode. The model also tracks the current typestate of this inode expected by the `mkdir` operation (line 26), which helps maintain the association between the `op` argument and the persistent objects it modifies across multiple transitions.

4.2.5 Typestates

Table 4.1 describes the three persistence and 22 operational typestates used in SquirrelFS to represent Synchronous Soft Updates dependencies. SquirrelFS additionally uses traits to define sets of typestates for which a single operation can safely be performed on an object with any of the included typestates. For example, the `AddLink` trait is implemented by `Alloc` and `IncLink`, two inode typestates in which it is safe to add a link from a new directory entry to that inode.

Recovery typestates. Several typestates are used only during recovery: `Recovery`, `TooManyLinks`, and `Recovering`. Most durable operations during crash recovery can use existing functionality and typestates, as most of these updates involve deallocating orphaned structures that are not visible from the file system root.

`Recovery` is a special typestate used when reading a persistent object after a crash. Functions that return an object with typestate `Recovery` do not perform the same validity checks as functions that return an object with typestate `Start` and can thus be used in scenarios where, e.g., an object was incompletely initialized before a crash.

`TooManyLinks` is used to represent an inode whose link count is higher than its actual number of links. Like traditional soft updates systems, SquirrelFS allows an inode's link count to be incremented even if doing so may cause a link count leak

Type	Associated structures	State	Meaning
Persistence	All	`Dirty`	One or more updates to this object have not been flushed from CPU caches.
		`InFlight`	All outstanding updates to this object have been flushed but not followed by a store fence.
		`Clean`	All outstanding updates to this object have been flushed and fenced.
Operational	All	`Start`	Object has been read from PM in an initialized state.
		`Free`	Object has been freed, or read from PM in an uninitialized state.
		`Alloc`	Object has just been allocated.
		`Recovery`	Indicates that an object has been read during recovery and may be in an invalid state.
	Inodes	`IncLink`	Object's link count has been incremented.
		`DecLink`	Object's link count has been decremented.
		`IncSize`	Inode's size has been increased.
		`DecSize`	Inode's size has been decreased.
		`TooManyLinks`	Indicates that an inode read during recovery has an incorrect link count and needs to be repaired.
		`UnmapPages`	Indicates that an inode's pages can be unmapped safely; once they are, the inode can be deallocated.
	Page desc.	`Zeroed`	All bytes in the page have been zeroed.
		`Writable`	Page's backpointer is set; page can now be written to.
		`Written`	Page has had data written to it
		`ToUnmap`	Indicates that a page descriptor's inode number field can be safely cleared.
	Dentries	`SetRenamePointer`	Destination dentry's rename pointer has been set to point to source dentry. Only used with destination directory entry.
		`InitRenamePointer`	Destination dentry's inode number has been set to target inode. Only used with destination directory entry.
		`Renaming`	Rename pointer to source dentry has been set. Only used with source directory entry.
		`Renamed`	Destination dentry's inode number has been set to target inode. Only used with source directory entry.
		`Recovering`	Represents a directory entry whose inode number does not need to be modified during post-crash rename cleanup. Only used with source directory entry.
	Dentries & page desc.	`ClearIno`	Object's inode number field has been cleared.
		`Dealloc`	Object has been persistently deallocated.
	Inodes & dentries	`Complete`	The current operation has finished updating this object.

Table 4.1: **Typestates used in SquirrelFS..**

in the event of a crash. Unlike other systems, SquirrelFS detects such leaks and fixes them during crash recovery by counting the number of directory entries that refer to each inode and comparing the count to the inode's real link count. The only legal operation on a `TooManyLinks` inode is a special `recovery_dec_link` function that durably sets the inode's link count to the correct value.

`Recovering` is used only to represent a source directory entry in a rename operation that is interrupted by a crash in Step ❷ or ❹ in Figure 4.2. In both steps, the destination directory entry's rename pointer must be cleared, either to roll the operation back (Step ❷) or forward (Step ❹), but the source directory does not need to be modified yet. Rename pointer fixes run before orphaned resource cleanup; if the source directory entry is orphaned after the rename is completed or rolled back, it will be deallocated later.

Metadata updates. Operations that modify file metadata like link count, size, or permissions need to update an existing inode in place. Each of these fields fits within 8 bytes and can be updated atomically. SquirrelFS does not currently support atomic updates to multiple separate file attributes. Some metadata operations, such as changing link count or file size, are either dependent on or are depended on by other operations. For example, as shown in Figure 4.3, the parent inode's link count must be durably incremented before setting the new directory entry's inode number in `mkdir` operations. These operations have special typestates used to enforce these dependencies. Others, such as setting permissions or changing file ownership, are completely independent of other durable operations and can be performed at any time. Updating these attributes is permitted in any typestate and does not change the inode's operational typestate, but does set its persistence typestate to `Dirty`.

Typestate dependencies. Figures 4.7, 4.8, 4.9, 4.10, and 4.11 illustrate the dependency relationships between typestates in SquirrelFS. Each typestate is represented by a gray box connected by solid arrows representing typestate transitions. The arrows are labeled with the operation or condition associated with that tran-

sition. Dotted arrows represent *implicit* typestate transitions. These are usually transitions from `Complete`, a typestate indicating that an operation has completed all updates to an object that is still in use, to `Start`, the starting state of initialized persistent objects in system call handlers. This transition is not associated with an explicit function invoked by SquirrelFS; rather, it occurs implicitly between when the typestate wrapper around a persistent reference is dropped and when the same persistent value is read again.

Most typestates are used with multiple types (see Table 4.1), but the dependencies and transition functions differ between types of persistent objects. For simplicity, these diagrams do not reflect all dependency relationships in SquirrelFS. They do not include recovery typestates and do not show full dependencies for operations that interact with multiple persistent objects. Typestate transitions that have a dependency on the operational typestate of another object that is not shown are marked with an asterisk. While these figures present persistence and operational typestate separately, they are used together in SquirrelFS to determine whether a transition is safe; for an example, see Figure 4.3. An object must be `Clean` before an operation that updates its operational typestate can be performed. However, independent transitions may change the typestate of some objects while others are `Dirty` or `InFlight`.

Figure 4.7 shows the dependencies between the three persistence typestates used in SquirrelFS. Persistence typestates are maintained separately from operational typestates and apply to all types of persistent objects. Each typestate wrapper type implements `flush` and `fence` functions to transition from `Dirty` to `InFlight` and `InFlight` to `Clean`, respectively. Any function that modifies a `Clean` persistent object acts as a transition from `Clean` to `Dirty`.

Figure 4.8 shows dependencies between typestates associated with regular inodes in SquirrelFS. Regular and directory inodes are distinguished using a type parameter and are thus generally treated as different types with some shared func-

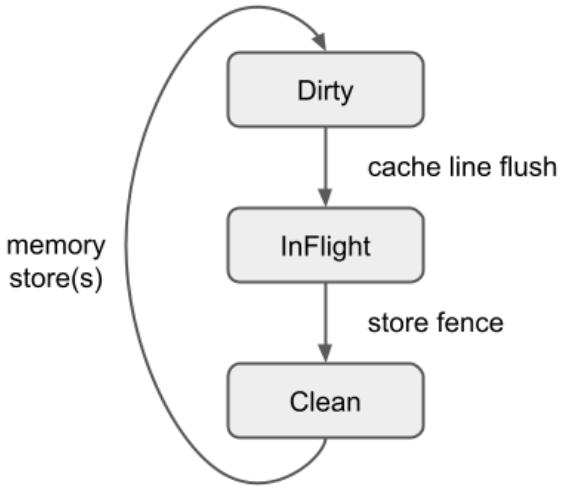

Figure 4.7: **SquirrelFS persistence typestates.** The figure shows dependencies between persistence typestates in SquirrelFS.

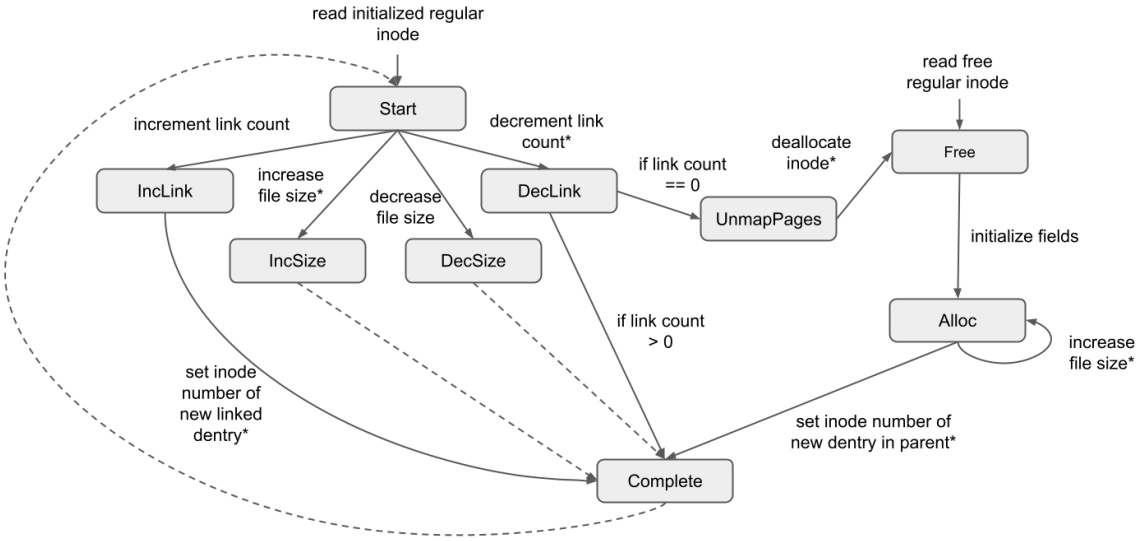

Figure 4.8: **SquirrelFS regular inode typestates.** The figure shows dependencies between operational typestates for regular inodes in SquirrelFS.

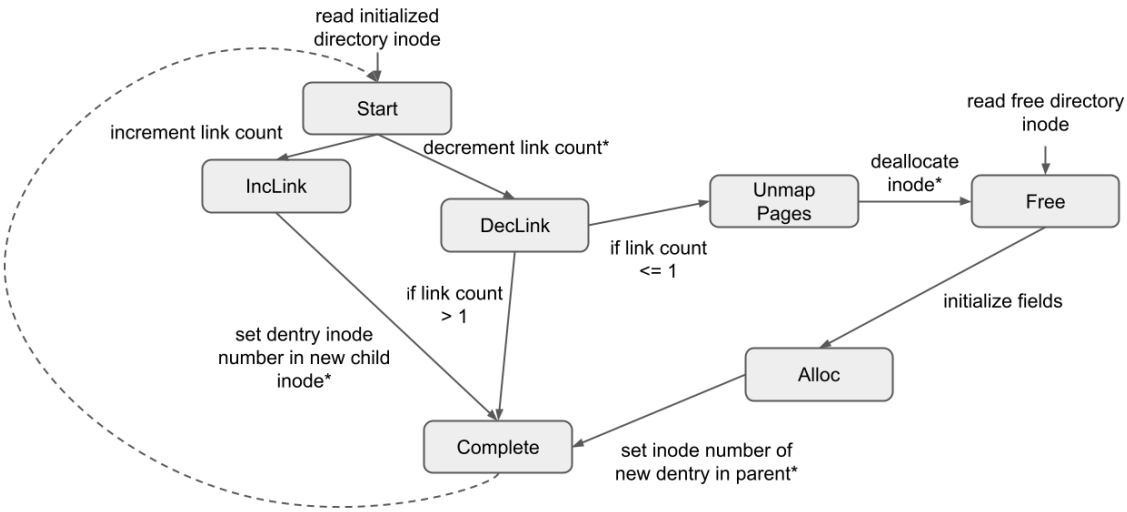

Figure 4.9: **SquirrelFS directory inode typestates.** The figure shows dependencies between operational typestates for directory inodes in SquirrelFS.

tionality, so they have separate typestate dependency diagrams. Increasing a regular inode's size also requires a page descriptor structure with a typestate indicating that we have already written some additional data to this file to ensure that a crash cannot accidentally expose garbage data. SquirrelFS maintains as an invariant that an inode's link count is always greater than or equal to the true number of links, so decreasing the inode's link count also requires a directory entry whose inode number has been cleared. The transitions from `IncLink` and `Alloc` to `Complete` do not make any durable modifications to this inode; rather, they are made by a function that updates a directory entry's inode number. The transition to `Complete`, which has no explicit transition functions to other states, ensures that an `IncLink` or `Alloc` inode can only be used with a single directory entry update.

Figure 4.9 shows dependencies between typestates associated with directory inodes. Directory inode dependencies are similar to those of regular inodes except that the conditions for transitions associated with link counts are slightly different and directory inodes do not use the {`Inc`,`Dec`}`Link` typestates. The specific directory entry operation associated with the transitions from `IncLink` to `Complete` is also

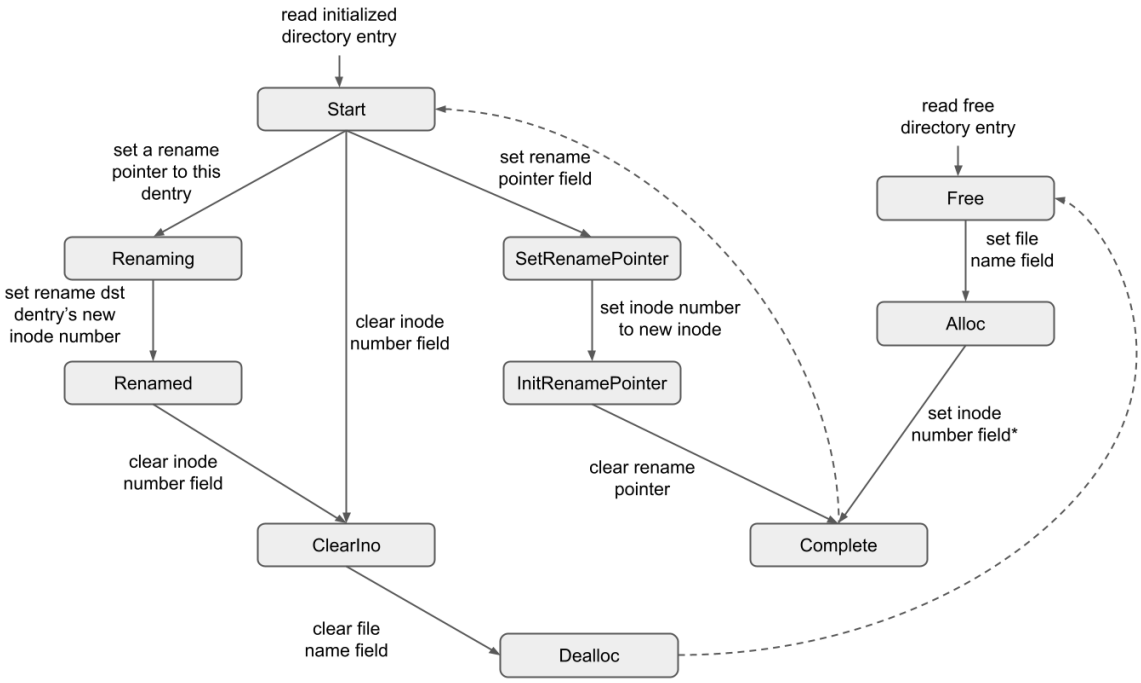

Figure 4.10: **SquirrelFS directory entry dependencies.** Directory entry update dependencies in SquirrelFS.

different, since a directory inode's link count is increased only when a new child directory entry is created.

Figure 4.10 shows dependencies between typestates associated with directory entries. Along with similar allocation/deallocation transitions to inodes, SquirrelFS uses several directory-entry-only typestates in rename operations to provide a fully crash-atomic implementation of the `rename` system call. `Renaming` and `Renamed` represent the state of the source directory entry after the rename pointer has been set and after the destination directory entry's inode number has been updated, respectively (steps ❷ and ❸ in Figure 4.2). `SetRenamePointer` and `InitRenamePointer` represent the state of the destination directory entry as its rename pointer and new inode number, respectively, are set. The same functions simultaneously update the typestates of both directory entries involved in the `rename` operation in both cases.

Figure 4.11 shows dependencies between typestates associated with page de-

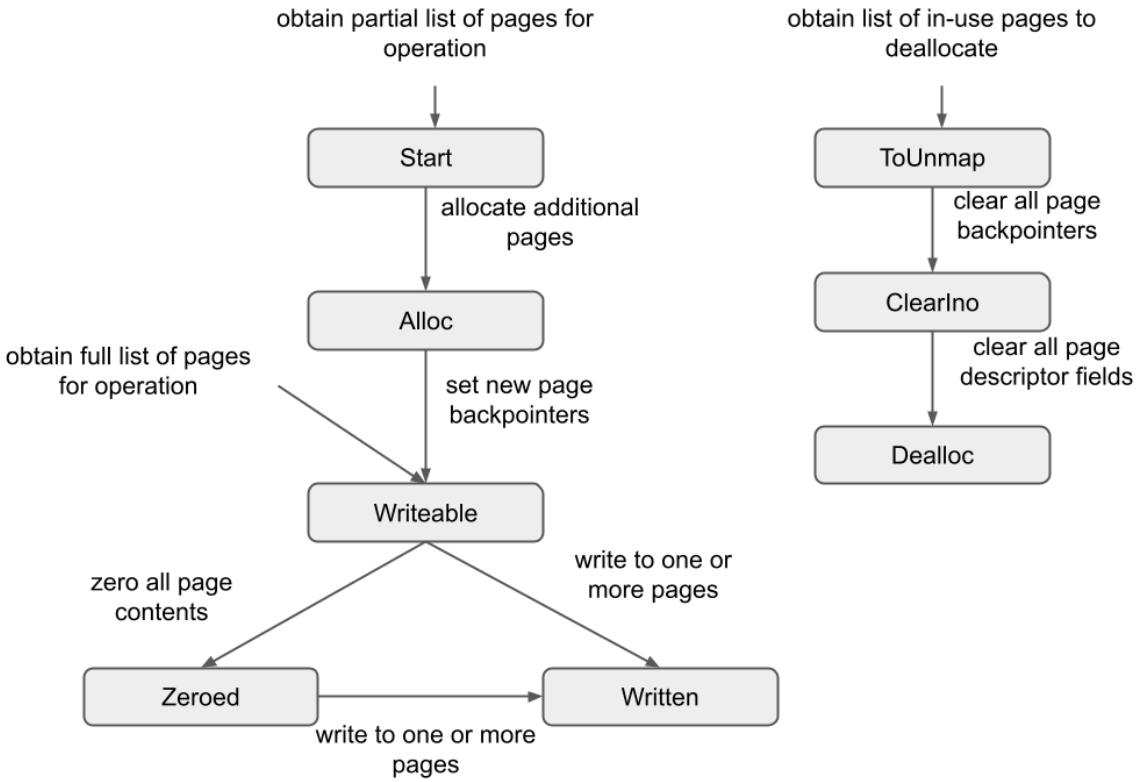

Figure 4.11: **SquirrelFS page typestates.** The figure shows dependencies between operational typestates for pages in SquirrelFS.

scriptors and page contents. For simplicity, SquirrelFS uses a single wrapper structure to represent modifications to both page descriptors and the contents of their associated pages. The typestate of these structures describes the state of a list of logically-sequential pages in a file, and typestate transitions on these structures update the state of all pages in the list. SquirrelFS tracks the typestate of collections of pages, rather than per-page typestate as with other types of persistent objects, due to language limitations and the inherent challenge of tracking typestate for an arbitrary number of objects. For more detail, see §4.3.3 and §4.5. This diagram is not cyclic because the lists used in each operation are constructed on-demand with the pages required for that operation, so there is no implicit transition from, e.g., `Complete` to `Start` as there is for other structures.

SquirrelFS may obtain an incomplete list of pages if the required pages have not already been allocated, e.g., during an append operation. The caller passes the wrapper constructor the range of desired pages; if all pages in that range exist, the constructor returns a list in the `Writeable` state. Otherwise, it returns a list in the `Start` state, which cannot be written to or zeroed out until the missing pages have been allocated.

Code example. Figure 4.12 shows a typestate transition function used by SquirrelFS as part of `mkdir` operations. The shown code is a slightly simplified version of the real function from SquirrelFS. This function completes a `mkdir` operation by setting the new directory entry's inode number to that of a newly-allocated directory inode. The function is implemented only for `DentryWrapper` objects with type parameters `Clean` and `Alloc`.

This function (and any other functions defined in this `impl` block) can only be called on `DentryWrapper` objects with the specified typestates. This function also takes, and updates the typestate of, two `Clean` directory inodes: one in the `Alloc` state and one in the `IncLink` state. Together with the `self` parameter, these three parameters specify the typestate dependencies for this operation and ensure that

the new directory entry cannot point to the new inode until the new inode is fully allocated and its parent's link count has been incremented.

Note that although `DentryWrapper` takes two type parameters, it does not store values of these types at runtime. However, the Rust compiler requires that a `struct` that is generic over a type `T` own a value of that type. The `PhantomData` type seen on lines 13 and 14 is a Rust standard library type that can be used to satisfy this requirement without using unnecessary space. A `struct` with a field of type `PhantomData<T>` appears to own a value of type `T` to the compiler, allowing this check to pass. However, `PhantomData<T>` is a *zero-sized* type, meaning that the field takes up no space at runtime. SquirrelFS types with typestate use `PhantomData` to meet the compiler requirement without using any runtime resources for typestate. The type argument to `PhantomData` can usually be inferred automatically, but we include it here for clarity.

4.2.6 Limitations of the approach

It is important to note that the typestate-based approach used in SquirrelFS is not as powerful as full verification. Fully-verified systems, such as the FSCQ file system [40], use theorem provers that can prove a wide variety of complex properties. For example, a developer could prove, if required, that the system only uses even-numbered inodes for files.

In contrast, our typestate-based approach can only check *ordering-based* invariants. Our approach could be used to verify that functions are called in a specific order; for example, our approach can ensure that a file is not linked into the file-system tree before it is allocated. However, it does not verify the implementation of each function that is called.

Thus, full verification is significantly more powerful and general, but it pays a cost in terms of complexity and development time. Our approach is more targeted and ordering-based, but allows quick feedback and incremental development.

```
impl DentryWrapper<Clean, Alloc> {
  pub fn set_dir_ino(
    self,
    new_inode: InodeWrapper<Clean, Alloc, DirInode>,
    parent_inode: InodeWrapper<Clean, IncLink, DirInode>,
  ) -> (
    DentryWrapper<Dirty, Complete>,
    InodeWrapper<Clean, Complete, DirInode>,
    InodeWrapper<Clean, Complete, DirInode>,
  ) {
    self.dentry.ino = new_inode.get_ino();
    (
      DentryWrapper {
        state: PhantomData::<Dirty>,
        op: PhantomData::<Complete>,
        dentry: self.dentry,
      },
      InodeWrapper::new(new_inode),
      InodeWrapper::new(parent_inode),
    )
  }
}
```

Figure 4.12: `mkdir` **typestate transition function.** The listing shows a typestate transition function used by `mkdir`. Typestates are shown in bold.

We believe this approach is a valuable addition to the repertoire of tools we have for building correct file systems. This approach should be used alongside runtime testing and model-checking approaches.

4.2.7 Relevance beyond PM

While we have designed SquirrelFS for persistent memory, SquirrelFS would be relevant for any storage technology with byte-addressability and low latency. The Compute Express Link [47] standard will support attached memory devices, including PM, via the Type 3 (CXL.mem) protocol. These CXL-attached PM devices will have the same interface and persistence semantics as current NVDIMMs, though performance will be lower [16].

SquirrelFS, and SSU file systems in general, could be used on CXL-attached memory. As SquirrelFS's mount performance and memory footprint are tied to the size of the device, they may worsen with significantly larger-capacity devices. Further work will be required to optimize file systems based on our approach for such devices.

4.3 Experience developing SquirrelFS

We now describe our experience with designing, developing, and testing SquirrelFS. We also discuss the challenges we faced during this process.

4.3.1 Development process

Designing SquirrelFS. Our initial design closely followed that of BSD FFS [187], but most aspects eventually diverged due to differences between storage hardware and typestate considerations. We found that some data structures and crash-consistency properties were better suited for use with the typestate pattern than others. For example, we chose SquirrelFS's backpointer-based page management approach be-

cause it simplifies update dependency rules when allocating or deallocating pages. With backpointers, these operations involve a constant number of persistent updates and involve no additional durable structures. In contrast, tree or log-based approaches need extra persistent metadata and may require additional updates to balance the tree or free log space, both of which complicate dependencies and typestate management.

An important design decision we had to make was how granular typestate would be. One option was to use specific typestates to represent each fine-grained operation; e.g., have one typestate for initializing an inode's link count, another for setting its flags, etc. Another was to make each typestate more general, with transition functions potentially performing multiple persistent updates. More general typestates may sacrifice some bug-finding power, but they make the system easier to understand and develop. In SquirrelFS, we aimed to strike a balance by representing only operations that require a specific ordering with typestate. For example, when initializing an inode in SquirrelFS, the order in which the values of most fields are set is not relevant to crash consistency, as the contents of the inode are not visible until it is linked into the file system tree. Therefore, SquirrelFS uses only a single typestate (`Init`) to represent inode initialization, and another (`Committed`) to indicate when it has been added to the tree.

Parallel model and system implementation. We developed the Alloy model alongside SquirrelFS. This created a useful feedback loop in which the model supported the Rust implementation, and questions or changes to the implementation could be quickly reflected and checked in the model. We used an incremental development process, incorporating feedback from the Rust compiler and the model immediately as we implemented the system. Many transitions in the model could be translated directly into Rust typestate transitions, making the model a valuable guide for implementing file system operations. When we made mistakes translating the model into Rust, typestate checking quickly caught these issues.

Alloy also includes a graphical user interface for visualizing traces of operations

on the model. This was useful for both investigating invariant violations and seeing the set of transitions that occur in a given file system operation, which could be translated directly into system call handler implementations. It also demonstrated locations where multiple updates could safely share a single store fence, which helped us avoid redundant fences.

4.3.2 Finding bugs

While developing SquirrelFS, we used a combination of typestate checking, model checking in Alloy, and dynamic testing to find bugs.

Typestate checking. Typestate checking in the implementation was successful at quickly catching both missing persistence primitives and higher-level ordering bugs; we provide an example of each.

- *Missing persistence primitives.* Our initial implementation of `write` was missing flush and fence calls after setting the backpointer of a newly-allocated page. This bug was immediately highlighted as an error by the compiler. Had this bug made it into the implementation, a crash could cause a file to have a size larger than the number of pages associated with it, causing errors when trying to read the file.

- *Incorrect ordering.* Our initial `rename` implementation mistakenly decremented an inode's link count before clearing the corresponding directory entry. A crash could result in a link count that is lower than the true number of links, leading to a dangling link if the inode is subsequently deleted.

Although we did not specifically check execution paths without crashes, the crash-consistency invariants encoded in typestate were general enough to detect some bugs in this code. For example, the compiler caught a bug where pages were not fully deallocated during `unlink`, which did not require a crash to manifest. Typestate-related compiler errors were relatively uncommon overall, since using the model as a

guide for implementation helped us get ordering right early. However, it provided a crucial safety net to prevent subtle bugs when we did make mistakes.

Model checking with Alloy. The Alloy model found several high-level issues in SquirrelFS's design that would have otherwise been difficult to detect and time-consuming to fix, including the following examples.

- We initially believed that crash recovery would not be needed other than to fix space leaks. Alloy found an instance of the model where a crash during rename followed by deallocation of the destination directory entry could cause an invalid directory entry to reappear. Fixing this required the addition of recovery transitions.
- Early designs for SquirrelFS stored . and .. directory entries durably. We discovered via model checking that our original dependency rules for handling these directory entries during more complicated operations like `rename` were not correct. Ultimately, we decided to not store these entries, since they can be constructed at runtime using indexed information.

Testing. Neither the typestate pattern nor the Alloy model eliminated the need to test SquirrelFS. Our primary goal was to check crash-consistency, and we did not check any invariants that only impact regular, non-crash execution. We used handwritten tests and the xfstests suite [264] to test these unchecked parts of the code.

All bugs found through testing were in parts of SquirrelFS that were not checked by typestate or directly modeled in Alloy. Most bugs were related to updating volatile indexes or VFS inodes, e.g., failing to remove a deallocated object from an index or setting the wrong value in the VFS inode. There were also bugs in the implementations of typestate transitions, which are not themselves verified; for example, the transition that wrote new file data to a page did not always calculate

the offset for non-aligned writes correctly. Implementing bug fixes was quick since we did not need to modify the typestate-restricted interface to objects and there were no proofs to update.

4.3.3 Challenges encountered

Challenges with typestate. It is easier to write typestate-checked code than it is to write verified code, but this comes at the cost of less powerful compile-time checking. For example, checking universally-quantified formulas (e.g., all pages in a file are allocated) is undecidable, and unlike verification-aware languages, the Rust compiler has no heuristics to attempt to solve them. As a result, we cannot ensure invariants such as "all objects in a set are in a certain typestate"; specifically, we can't encode this in typestate because the number of objects in the set is not known at compile time.

This became a problem when implementing file-system operations like `unlink`, where we would like to e.g., check that the backpointers of all pages belonging to the file are cleared before deallocating the inode. Such a check ensures that the system always follows soft updates rule 2 (never re-use a resource before nullifying all previous pointers to it); by clearing all the page backpointers before deleting the inode, we ensure that none remain when the inode is eventually reused. However, it is impossible to check this property on arbitrary sets of pages if each page has its own typestate. We experimented with several workarounds, including forcing `write` operations to update no more than one page at a time (which was prohibitively slow and did not solve the problem for `unlink`), and storing typestate in page structures at runtime and manually adding assertions (which also impacted performance and lost the benefit of static checking).

Ultimately, we decided to use a single piece of typestate to represent *ranges* of pages (e.g., all pages in a file or a contiguous subsection). As shown in Figure 4.11, each typestate transition performs an operation on a range of pages, which

is obtained on-demand. This solution moves some logic into the typestate transition functions, where it is not checked for crash consistency issues. This sacrifices some static checking power, but provides a simple interface for modifying and reasoning about collections of pages. For example, the transition from `Writeable` to `Zeroed` in Figure 4.11 iterates over each page in the list and zeroes it out before returning. If there is a bug in this implementation that causes it to skip some pages, it may be caught by standard testing but will not be identified by typestate checking. In practice, we found that operations on page ranges were longer and slightly more complex than other typestate transition functions, but they are not difficult to understand or test.

Challenges with Alloy. As SquirrelFS grew in complexity, it became harder to maintain the model and get useful feedback quickly. The model checker uses a SAT solver to check invariants, and the formulas representing a large model can take days or weeks to solve. We checked that traces with multiple concurrent operations were crash consistent, which increased the size of the problem further. To get faster feedback, we built a custom utility to run multiple independent instances of the model checker in parallel and split larger predicates into smaller, more concrete sub-checks.

It could also be difficult to determine whether a reported failure was a false positive. A particular challenge was dealing with *frame conditions*, predicates that specify what should not change in a given transition. Alloy is free to arbitrarily change any state that the current transition does not explicitly mention, so frame conditions are crucial to constrain the model to real traces. This behavior helps Alloy find corner-case bugs, but it also leads to false positives. To overcome this challenge, we built a syntax-based checker that parses the model using Alloy's API and checks that each transition explicitly mentions all mutable structures in the model. The current version of the checker cannot detect all issues, but it detected many missing conditions that would have otherwise taken hours to catch via model checking.

4.3.4 Typestate beyond SquirrelFS

Costs and benefits of typestate. We do not have equivalent verified or unverified systems to compare with SquirrelFS in terms of development and debugging effort; however, in the authors' experience, designing and implementing SquirrelFS required more effort than a typical unverified system, but far less effort than a verified storage system. We believe that debugging SquirrelFS was faster and easier than debugging an equivalent unverified system, as following the typestate-enforced ordering rules made it easier to implement the system correctly in the first place and reduced the number of bugs overall.

Using the typestate pattern for crash consistency represents a useful new point in the trade-off space between runtime testing and full verification. While it comes at the cost of additional development effort compared to unverified systems to determine correct ordering rules and does not gain the same correctness guarantees as verified systems, it does eliminate an entire class of crash consistency bugs that are otherwise difficult to find and fix [143, 166, 201]. Furthermore, as the pattern builds ordering rules directly into a system's implementation, the rules will stay up to date and continue to prevent crash-consistency bugs as the system is developed further [107, 212].

Broader applicability. As the typestate pattern is a general approach for statically checking the order of updates to data structures, it is useful in a broad variety of contexts, several of which are described below.

- Volatile data structures: SquirrelFS does not use typestate to manage updates to volatile data structures, but prior work on typestate verification has focused entirely on such use cases [3, 243].
- Other types of storage systems: The typestate pattern could be used to enforce ordering invariants on durable updates in other types of storage systems (e.g., key-value stores) with different crash-consistency mechanisms. We note that

crash-consistency mechanisms like journaling and copy-on-write do not achieve consistency entirely through ordering and would require auxiliary techniques to check properties like atomicity.

- Durable layout: SquirrelFS's on-storage layout is tailored to reduce the number of durable updates per file-system operation and to simplify ordering rules. Other layouts could also be used in typestate-checked storage systems, although the complexity of the ordering rules would increase.

- Asynchrony: The typestate pattern is compatible with asynchronous systems, although the ordering rules to enforce are much more complicated in such systems, as updates from different operations may be interleaved.

4.4 Evaluation

We seek to answer the following questions in our evaluation of SquirrelFS:

1. What is the latency of different file-system operations on SquirrelFS? (§4.4.2)
2. How does SquirrelFS perform on macrobenchmarks? (§4.4.3)
3. How does SquirrelFS perform on real applications? (§4.4.4)
4. How long does SquirrelFS take to mount and recover from crashes? (§4.4.5)
5. What compilation, memory, and CPU overheads does SquirrelFS incur? (§4.4.6)
6. Is SquirrelFS correct? (§4.4.7)
7. How could SquirrelFS' performance be improved? (§4.4.8)

4.4.1 Experimental setup

We evaluate SquirrelFS on a two-socket, 32-core machine with 128GB of memory and one 128GB Intel Optane DC Persistent Memory Module. The evaluation machine runs Debian Bookworm and Linux 6.3.

We compare SquirrelFS against ext4-DAX [177], NOVA [266], and WineFS [132]. We configure all three systems to provide metadata consistency but not data consistency to match SquirrelFS's guarantees. All reported results are the average of multiple trials. The red errors bars in Figure 4.13 indicate the minimum and maximum values recorded over all trials.

We cannot compare SquirrelFS to SoupFS [67], the only other soft updates PM file system, as it is not open source. We do not present a comparison of SquirrelFS against ArckFS [277], a PM file system that uses a soft-updates-inspired crash-consistency mechanism, because ArckFS runs entirely in userspace. Userspace file systems have significantly different performance characteristics from in-kernel systems due to the reduced kernel crossings and different resource management behaviors. Small tests on ArckFS indicate that while it has lower average latency than SquirrelFS on some system calls due to its lower software overhead, it must periodically perform costly in-kernel PM allocations to serve some system calls, incurring high tail latencies.

4.4.2 Microbenchmarks

We compare each system's latency by testing several file system operations: appending and reading 1KB and 16KB to a file, file creation, directory creation, renaming a directory, and unlinking a 16KB file. None of the tests call `fsync`.

The average latency over 10 trials of the tested operations are shown in Figure 4.13(a). The lowest latency file system in each test is either WineFS or SquirrelFS. Ext4-DAX has the highest latency on many operations because it interacts with the Linux kernel block layer for tasks like block allocation, which incurs

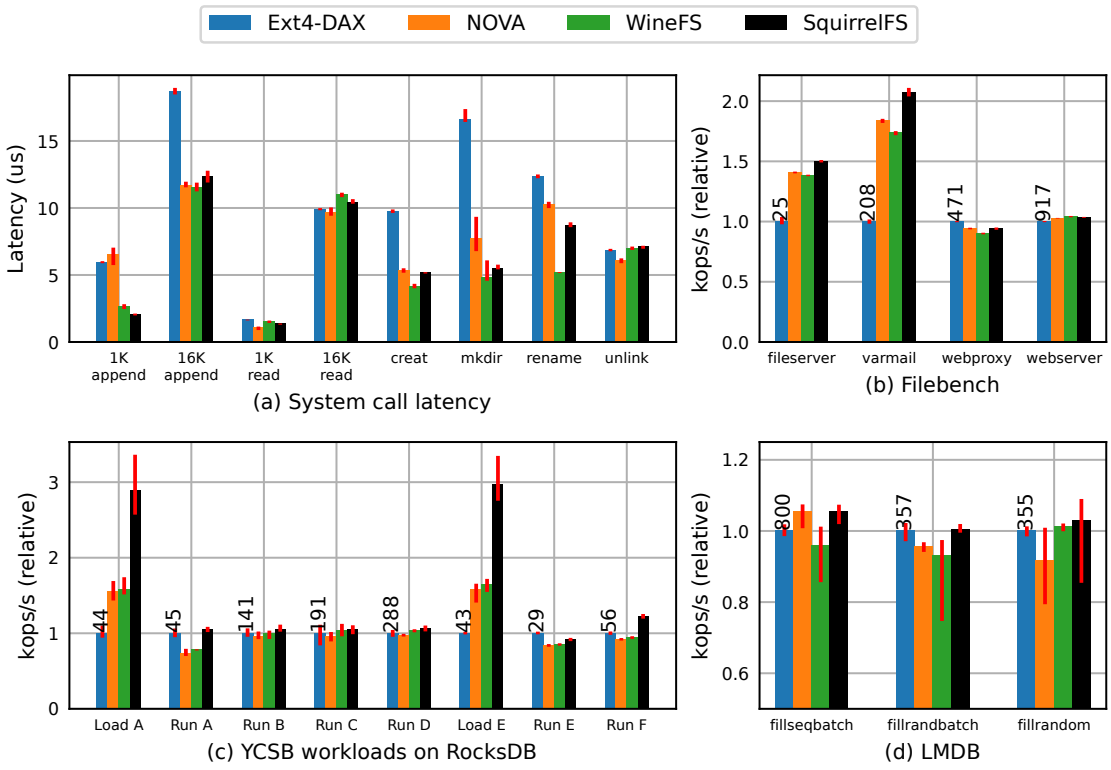

Figure 4.13: **SquirrelFS performance evaluation.** The figure shows the performance of the evaluated file systems on different benchmarks and applications. (a) shows absolute latency of different file system operations; (b), (c), and (d) show the relative throughput in kops/s of each system relative to Ext4-DAX on filebench, YCSB on RocksDB, and LMDB respectively.

additional software overhead. It achieves similar performance to the other systems on operations that do not go through the block layer (e.g., unlink). NOVA has higher latency on `mkdir` and `rename` than WineFS and SquirrelFS because operations that update multiple inodes require journaling in NOVA.

4.4.3 Macrobenchmarks

We evaluate SquirrelFS on the Filebench [248] storage benchmark suite. We run four workloads from the suite — fileserver, varmail, webserver, and webproxy — in their default configurations. Fileserver performs mostly writes with some whole

file reads; varmail is half appends and half reads; webproxy appends to each file and reads from it several times; and webserver reads and occasionally appends to a log file. Figure 4.13(b) shows the average throughput in kops/sec for each file system on each workload. SquirrelFS achieves slightly better throughput than the next fastest system on fileserver and varmail (8% and 13% better, respectively) and within 10% of the fastest system on both webserver and webproxy. Fileserver and varmail perform many small appends, which SquirrelFS performs well on due to its lack of journaling. Webserver and webproxy are more read-heavy, which all systems perform roughly equally on. Ext4-DAX does not go through the block layer on reads and it benefits from data contiguity awareness, making its performance similar or better than the other systems on these workloads.

4.4.4 Applications

YCSB on RocksDB. We evaluate the four systems on RocksDB [224] using YCSB workloads [49]. We run all workloads on a 25GB database with 25M records, 25M operations, and 8 threads. All workloads are run using standard workload configurations and the default settings of YCSB, which uses system calls for all operations. Figure 4.13(c) shows throughput in kops/second relative to Ext4-DAX on each tested workload.

SquirrelFS outperforms the other systems on Loads A and E, which are 100% small inserts. As seen on the other benchmarks, SquirrelFS performs particularly well on small appends due to its lack of journaling or logging. Writes that require page allocation are particularly expensive in the other systems, as journaling/logging the new metadata incurs an additional 2–3us in NOVA and WineFS and 3–4us in Ext4-DAX. Ext4-DAX and NOVA both also journal or log metadata on every append, spending roughly 30% of each non-allocating call (approx 1–1.5us) managing journals/logs.

All file systems are within 10% of Ext4-DAX's throughput on Runs B, C, and

D. All of these workloads are at least 95% small (4KB) reads, which all four systems achieve similar performance on.

SquirrelFS achieves the best throughput on Runs A and F, which are 50% reads and 50% updates (Run A) or read-modify-write operations (Run F). Ext4-DAX, NOVA, and WineFS all incur logging/journaling on these workloads; Ext4-DAX outperforms NOVA and WineFS because it has less journaling overhead for in-place updates and is more aware of data contiguity on reads.

Ext4-DAX achieves the best performance on Run E, which is 95% range scans and 5% inserts. Ext4-DAX's contiguity-awareness and better fragmentation-prevention mechanisms help it outperform the other systems on larger read operations.

LMDB. We also run LMDB [247], a memory-mapped database, using `db_bench`'s `fillseqbatch`, `fillrandbatch`, and `fillrand` workloads. Each experiment uses 100M keys on an empty file system. Figure 4.13(d) shows the throughput in kops/sec for each file system on each workload. Each file system has throughput with 12% of the other systems. Most updates are done to memory-mapped files, so differences in the performance of system calls and metadata management designs have a reduced impact.

Git. We also evaluate each system by using `git checkout` to switch between major Linux kernel versions from GitHub. We start at v2.6.12 (the oldest version of Linux available on GitHub) and successively check out versions 3.0, 4.0, 5.0, and 6.0. Table 4.2 shows the average time for each checkout over 10 iterations. The time to check out a given version in each file system is within 8% of the other systems.

4.4.5 Mount time

Table 4.3 shows how long it takes to mount SquirrelFS, compared to Ext4-DAX as a baseline, on a 128GiB PM device with different contents. We report the

System	Checkout time (s) v3.0	v4.0	v5.0	v6.0
Ext4-DAX	4.6	4.9	6.5	7.6
NOVA	4.6	4.9	6.5	7.2
WineFS	4.5	4.7	6.1	7.0
SquirrelFS	4.7	4.9	6.4	7.2

Table 4.2: **Linux checkout time comparison.** The table shows the time to `git checkout` each successive Linux kernel version.

	Mount time (s)				
		Normal		Recovery	
System	mkfs	Empty	Full	Empty	Full
Ext4-DAX	0.33	0.01	0.01	0.01	0.01
SquirrelFS	5.80	5.51	30.50	5.76	55.50

Table 4.3: **SquirrelFS mount times.** Time in seconds to mount Ext4-DAX and SquirrelFS in different states. Times in the "Normal" column come from mounting a cleanly-unmounted system, and those in the "Recovery" column are obtained by modifying the file system to run recovery when mounting a cleanly-unmounted instance.

average of 10 mount operations. We measure the time each system takes to run `mkfs` (i.e., to create an empty file system instance), to mount an empty system, and to mount a completely full instance. To create a full system, we create 128 directories and create 32KiB files in each directory in parallel until failure.

SquirrelFS takes longer than Ext4-DAX in all setups. During `mkfs`, it must zero out almost the entire device, since SquirrelFS interprets non-zeroed-out data structures as allocated, and construct the allocators. Mounting an empty system is similar, except that SquirrelFS scans the inode and page descriptor tables rather than zeroing them. Mounting a full system takes longer because SquirrelFS has to scan more data structures (e.g., it now must read each page of directory entries to determine the file system structure) and allocate space for each index.

Table 4.3 also reports the time taken by SquirrelFS and Ext4-DAX to perform recovery procedures on a cleanly-unmounted device in the "Recovery" column. We added a mount-time argument to each system that instructs it to run recovery-related code, regardless of whether a crash occurred.

SquirrelFS takes additional time to mount when running recovery even when no cleanup is needed. If SquirrelFS detects that it was not unmounted cleanly, it constructs additional structures to keep track of orphaned objects and the true link count of each inode. It fills in these structures during the regular rebuild scan and uses them to free orphans and correct link counts at the end of the mount process. SquirrelFS also checks each directory entry for non-null rename pointers and either rolls back or completes any interrupted renames. In this experiment, Ext4-DAX's mount time is unaffected by running recovery code.

SquirrelFS's mount time could be improved by parallelizing some of its rebuild and recovery logic. For example, the inode and page descriptor table scans are completely independent and could be done in parallel. The file system tree rebuild logic could also be distributed across multiple threads.

System	LOC	Compile time (s)
Ext4	45K	38
NOVA	16K	20
WineFS	9K	13
SquirrelFS	7.5K	10

Table 4.4: **Compilation time comparison.** The table shows the time to compile different PM file systems as loadable kernel modules. Ext4's line count includes interleaved DAX and non-DAX code.

	MiB used	
System	Empty	Full
Ext4-DAX	1	336
NOVA	1	64
WineFS	3	57
SquirrelFS	1104	3220

Table 4.5: **Memory usage comparison.** The table shows memory usage in empty and full instances of each evaluated system.

4.4.6 Resource usage

Compilation. SquirrelFS takes approximately 10 seconds to compile on our test machine, including typestate checking. This compares well to fully-verified systems; FSCQ [40] takes about 11 hours to verify, and VeriBetrKV [104] takes 1.8 hours (10 minutes when parallelized).

SquirrelFS also compiles faster than the other tested systems on the test machine. Table 4.4 shows the size of each system in lines of code and how long it takes to compile. SquirrelFS's more complicated typechecking does not noticeably impact its compilation time.

Memory. Table 4.5 shows the amount of memory used in empty and full instances of each evaluated system. We measured memory usage by checking the amount of available memory reported by `/proc/meminfo` before and after mounting and filling up the file system. The reported numbers are the average of five measurements.

SquirrelFS uses significantly more memory in both cases than the other evaluated systems.

In an empty system, most of this space is taken up by free lists. SquirrelFS uses free lists backed by red-black trees to allocate inodes and pages. Each inode and page number in the free list is 8 bytes, and each red-black tree node uses 24 additional bytes to store its color and child pointers. In this experiment, SquirrelFS has approximately 4 million inodes and 32.5 million data pages, which result in free lists that use about 1GiB of memory when full.

In a full system, most memory is used by SquirrelFS' indexes. The indexes use red-black trees to map inodes to information about their pages and/or directory entries, which is also stored in a second level of red-black trees. The exact memory usage depends on the number of directories and regular files in the system, as directories take more space to index since each directory's pages and dentries are indexed in memory. The dentry and data pages indexes are the most expensive, each occupying about 1GiB. In this experiment, all indexes remain in the VFS inode cache along with a lock around each index and some additional per-file metadata; altogether, this uses another 1GiB of DRAM.

CPU. SquirrelFS does not start new threads in any of its operations. We leave the use of more threads for operations like freeing pages, running crash recovery, etc. to future work.

4.4.7 Correctness

Model checking. We check that a correctness invariant always holds in all traces of our Alloy model. We bound traces to include two operations (which may be concurrent), 10 persistent objects, and up to 30 steps. The invariant includes both sanity checks on the model and file system consistency checks. The sanity checks ensure, for example, that objects will never end up with conflicting typestates. The consistency checks ensure that 1) objects always have a legal link count, 2) there

are no pointers to uninitialized objects, 3) freed objects do not contain pointers to other objects, and 4) there are no cycles of rename pointers and directory entries are pointed to by at most one rename pointer.

Testing. We test SquirrelFS using a set of handwritten tests and the xfstests [264] test suite. SquirrelFS currently passes all supported tests (67) from xfstests' `generic` test suite. The rest of the tests use system calls or arguments that are currently not supported by SquirrelFS.

Crash consistency. We used Chipmunk [166] (Chapter 3) to test SquirrelFS for crash-consistency bugs. We modified Chipmunk's test generators to remove several system calls that SquirrelFS does not currently support but otherwise ran its full suite of systematically-generated tests and fuzzed the system for approximately 24 hours. Chipmunk did not find any ordering-related crash-consistency bugs in SquirrelFS, providing evidence that typestate-checked SSU is an effective mechanism for preventing such bugs. Chipmunk did find four crash consistency bugs in unchecked parts of SquirrelFS code, three in its rebuilding of volatile data structures and one in the body of typestate transitions in which a cache line flush was issued to the wrong address. As these are not caused by incorrect update ordering, the typestate pattern did not catch them at compile time. We found that using the typestate pattern in SquirrelFS made locating and fixing these bugs faster and easier, as we could focus on the specific regions of code that are unchecked and are thus more likely to have bugs.

4.4.8 Limitations and improvements

As SquirrelFS is a research artifact focused on correctness rather than performance, its current design has several performance limitations. It uses more memory than the other evaluated systems and takes longer to mount. SquirrelFS uses more memory because it maintains an index of the entire file system in volatile memory. Its longer mount times stem from additional work during its scan of the entire system

and the need to allocate space for indexes and allocators as they are rebuilt. Recovering from a crash involves scanning more durable structures and keeping track of additional system information, which further increases post-crash mount times. We now discuss several ways that SquirrelFS's mount performance and memory usage could be improved.

Parallelizing mount. SquirrelFS runs all mount-time device scans and volatile data structure construction sequentially, and its mount time could be improved by dividing this work across separate threads. Parallelizing mount procedures is a well-known technique in PM file systems [132, 266]. During remount, SquirrelFS first scans the inode and page descriptor tables to determine which entries are allocated and which inode each allocated page points to. Since SquirrelFS uses per-CPU page descriptor tables, each table could be scanned in parallel. SquirrelFS' inode table is currently managed globally, but could also be divided among multiple threads to scan in parallel during mount.

After determining which inodes and pages have been allocated, SquirrelFS scans each directory entry page to determine which durable structures are reachable from the root of the system and build the global index. It performs a breadth-first search over the file system tree, starting at the root inode, by reading the valid directory entries in each directory page associated with the current inode. This scan could be performed by multiple threads in parallel, although it would require more synchronization than the previous step on shared data structures like the search queue and indexes. Finally, we could parallelize the construction of the inode and page free lists, which currently use an inefficient implementation that determines which table entries are free by iterating over lists of allocated entries.

There are also several recovery-specific operations that could also be parallelized. For example, checking for interrupted `rename` operations requires an additional directory entry scan before constructing the global index, which could also be divided among multiple threads. SquirrelFS also durably frees all orphaned data

structures after a crash; this could easily be parallelized but is unlikely to have a performance impact, since we expect there to be relatively few orphaned structures after a crash.

Durable indexes and/or allocators. SquirrelFS currently keeps all indexes and allocators in volatile memory. Along with increasing its memory footprint, this also hurts mount performance, since space for these structures must be allocated at mount time. SquirrelFS could improve in both dimensions by moving these structures (partially or entirely) to PM. Allocators could be straightforwardly stored on PM, e.g., using bitmaps as in the original soft updates implementations [187].

Reducing the memory footprint of SquirrelFS' indexes is more complex. We now consider two possible approaches. First, SquirrelFS could be modified to use a durable layout that is more amenable to lookups, eliminating the need for a global index. For example, SquirrelFS could use a simple direct/indirect block design to manage file contents as in prior soft updates implementations [187]. The dependencies for this approach are more complicated than SquirrelFS' current backpointer-based page management, so this would require nontrivial changes to its typestates as well as its durable layout.

Second, we could store separate durable index structures without otherwise changing the layout of SquirrelFS. For example, NOVA and WineFS checkpoint allocator structures during clean unmount so that they can be rebuilt without a full device scan [132, 266]. This improves regular-case mount performance, but it does not eliminate the need for an expensive post-crash device scan and would not reduce DRAM usage in SquirrelFS. Another option would be to store these structures in a dedicated region of PM to ensure updates become durable quickly without needing to modify the layout of the rest of the system. However, it would be difficult to keep these structures in sync with the main file system tree in the event of a crash. One potential solution would be to maintain per-file indexes and store some durable metadata (e.g., setting a bit or storing a checksum of the index contents) that can

be used to determine if we crashed while updating a given index. After a crash, we would rebuild only the indexes that may have been corrupted by the crash, which would require a shorter recovery scan. To simplify this scan, we would most likely also expand SquirrelFS' typestates and dependencies to include updates to these indexes.

Improved in-memory data structures. SquirrelFS' memory usage could also be improved by simply using more space-efficient data structures for volatile indexes. This would also improve mount times over the current implementation by reducing the amount of memory that needs to be allocated during mount. For example, SquirrelFS' allocators are currently implemented using the Linux kernel's built-in red-black tree library, but an in-memory bitmap or a list of free page ranges would be much more efficient. Its directory entry index uses full file names as keys, which could be hashed to save space, and the dentry metadata structures stored in this index could be further optimized. SquirrelFS also currently uses separate indexes to map directory inodes to their pages and child directory entries for simplicity, but these indexes could be combined to save additional space.

4.4.9 Summary

SquirrelFS provides comparable performance to other PM file systems, while providing strong guarantees about its crash consistency. Due to the innovative use of typestate checking, we were able to implement SSU and gain confidence in its correctness. SquirrelFS gains an advantage over other file systems in write-dominated workloads, since soft updates avoids writing to a log or to a second copy of the data. The design of SquirrelFS trades off good common-case performance for slightly longer mount times compared to other file systems; we believe this is acceptable since crashes are rare. SquirrelFS compiles at the same rate as other PM file systems, despite the strong type checking.

4.5 Typestate discussion

Prior work [60, 75, 160, 243] has established the formal guarantees that can be obtained from typestate checking, fundamental limitations of the approach, and the computational complexity of statically analyzing typestates in a program. In this section, we discuss these results and apply them to SquirrelFS and the Rust-based typestate checking used in its implementation.

Theoretical properties of typestate checking. Static program verification is computationally complex. Many problems in static analysis are at least NP-complete, if not undecidable or uncomputable. Frameworks for developing fully-verified programs, like interactive proof assistants and verification-aware programming languages, generally rely on sophisticated theorem provers such as Z3 [59] or other solvers [226]. To handle complex queries, these tools use heuristics and/or require a high degree of developer interaction based on a significant amount of work over many years in the verification community.

The computational complexity of tracking typestate in a program depends on the structure of the target program and the specific property being verified. Even conceptually-similar properties, such as ensuring that a file is not read after it is closed and ensuring that it is not read before it is opened, are in different complexity classes (here, P and PSPACE-complete, respectively). The presence of pointers and aliasing in a program significantly complicates typestate analysis; in the worst case, it is undecidable in a program with recursive data structures. For more discussion and full proofs of these results, see Field *et al.* [75].

Typestate checking in SquirrelFS. Although SquirrelFS uses typestate to check properties about data structures containing references to other structures (e.g., directory entries containing inode numbers), it only accesses a finite and statically-known number of typestate-ful objects in each operation. When a durable object is accessed, SquirrelFS checks its contents to determine its current typestate (generally `Start` or `Free`) and constructs a wrapper structure with the current typestate

that contains a reference to the durable object itself. Typestate transition functions act on the type of this wrapper, not the type of the durable object itself, which hides pointers from typestate checks and reduces complexity.

Comparison to typestate-oriented languages. Several programming languages have been designed partially or entirely around typestate-based static analysis. We focus here on two such languages: Vault [60], a C-like language for low-level systems used to build Windows 2000-compatible device drivers, and Plaid [3, 244], a general-purpose language that extends standard object-oriented programming with dynamically-checked typestates. These languages include specific features for reasoning about typestates, such as the ability to maintain multiple mutable aliases to a single value or to maintain typestate information about an arbitrary number of objects in a collection. We present a comparison of these languages to Rust to demonstrate other capabilities of typestate-related techniques not discussed or used in SquirrelFS, and to provide intuition about the limitations of the approach. Rust's typestate support is less powerful than that of these other languages, so properties that can be checked with typestate in a Rust program are a subset of those that could be checked in programs written in these typestate-oriented languages.

Vault. The Vault programming language is based on C and builds on the idea of typestate to support tracking states of aliased objects using prior theoretical frameworks [52]. Vault's key insight is that adding a level of indirection between an object's type and its compiler-tracked state allows for more flexible and powerful static checking. To do this, Vault tracks resources using a set of *keys*, which describe the current state of each object and can be checked in function pre- and post-conditions.

Unlike Rust, Vault allows multiple mutable aliases to individual objects, although developers are required to explicitly state which names alias to which objects. Rust does not support multiple mutable aliases to the same resource without runtime checking via the interior mutability pattern or use of unsafe Rust, both of which give up some strong guarantees from the compiler. We did not find the inability to check

properties about multiple mutable aliases problematic in SquirrelFS, as dealing with aliasing restrictions is a standard part of Rust development, and we generally did not need multiple aliases to the same durable structures. Vault also provides a way to statically reason about the state of arbitrary numbers of objects in a collection by storing key information alongside the corresponding resource. In Rust, typestates are part of an object's type, and objects of different types cannot be stored in a collection and still benefit from strong compile-time checks. This imposed a limitation on SquirrelFS's design that all objects with typestate had to be used only in fixed, statically-known numbers, which weakened the guarantees we could obtain in some operations.

Vault's main case study was a floppy disk driver compatible with Windows 2000 that tracked low-level states associated with, e.g., I/O request status, thread coordination operations, and interrupt levels. Some operations modeled in this case study — for example, passing a key from one thread to another to allow the second thread to access the associated resource, and a lock model in which the proper key must be held to access the resource — are similar to features of Rust's ownership-based type system and channels for sending data between threads, which are distinct from the concept of typestate in Rust.

Plaid. Plaid is a general-purpose object-oriented programming language that treats states similarly to classes. Plaid is dynamically typed and typestate violations are reported as runtime errors (unlike Rust and Vault, in which typestate-related checks run at compile time). As a result, Plaid avoids some challenges with typestate checking in Rust, such as the issue of tracking typestate of objects in collections, at the cost of static guarantees.

Plaid provides several ways to interact with aliased objects. Objects may be defined as `shared`, meaning that there may be multiple mutable aliases to that object but no client is allowed to change the object's state (i.e., only modifications that do not involve a state transition are legal). The authors also propose the use of dynamic

tests to check that an object's state has not been changed before some operations, which is more flexible than the former approach but introduces additional runtime overhead. Similar techniques could be achieved in Rust by using interior mutability, a pattern that provides a way to gain mutable access to an object while there exist immutable references to it. However, as in Plaid, this would give up the benefits of statically checking.

4.6 Summary

This chapter presents a methodology for crash-consistent file system development. We propose the use of the typestate pattern in Rust to statically check crash-consistency invariants with low proof burden. We also introduce a novel crash-consistency mechanism, synchronous soft updates, that is well-suited to enforcement with the typestate pattern and that eliminates many challenges associated with the original soft updates technique. We develop SquirrelFS, a new file system for persistent memory that uses statically-checked synchronous soft updates for crash consistency. SquirrelFS achieves comparable or better performance than other PM file systems and required no language modifications or verification expertise to build.

Chapter 5: Formally verifying PM storage systems

In Chapter 3 and Chapter 4, we discussed testing and lightweight language-based static checking as techniques for ensuring crash consistency. However, neither technique can guarantee that a system is correct with regard to a specification. To obtain stronger guarantees, we can formally verify the system using a proof assistant or verification-aware programming language. Verification is, however, difficult, and existing techniques for verifying crash consistency and corruption detection in particular are difficult to use in practice.

In this chapter, we present PoWER, a new approach to verifying crash consistency, and use it to verify CapybaraKV, a PM key-value store, and CapybaraNS, a PM notary service. Both systems also have verified corruption-detection capabilities based on a new model of data corruption, also discussed in this chapter. The key benefit of these techniques is that they are flexible and tool-agnostic. In particular, PoWER only uses standard constructs that are present in most verifiers and that are relatively easy to learn, making it more accessible than prior work that uses significant custom-built infrastructure or language features.

We first discuss prior work on verified storage systems to motivate the need for new approaches (§5.1). We next describe PoWER, our new crash-consistency verification approach, in detail (§5.2). We also describe the disk models we have used with PoWER, proofs that PoWER corresponds to other crash-consistency verification techniques, and a set of proof strategies we use to simply crash-consistency proofs with PoWER in this section. We describe our approach to detecting data corruption, including our novel model that allows developers to prove the absence of a certain number of bits of corruption, and a new primitive for crash-safe updates on persistent memory (§5.3). We next describe CapybaraKV (written in Verus) and CapybaraNS (written in Dafny), our verified systems built using PoWER and our corruption model (§5.4). This section also includes a discussion of `pmcopy`, a Rust

crate we developed for CapybaraKV to facilitate aspects of crash-consistency proofs that are not currently supported by Verus. Finally, we present an evaluation focusing on CapybaraKV and comparing its performance to unverified PM KV stores (§5.5).

This chapter is based on the paper "PoWER Never Corrupts: Tool-Agnostic Verification of Crash Consistency and Corruption Detection" [165], which will appear at OSDI 2025. We mention throughout the chapter the components to which co-authors made significant contributions. Much of the work described in this chapter was completed during two internships at Microsoft Research, during which I was mentored by Jay Lorch and Chris Hawblitzel and collaborated with Cheng Huang and Yiheng Tao in Azure Storage.

5.1 Motivation

In this section, we describe prior work on verified storage systems to motivate the need for a new, tool-agnostic technique.

5.1.1 Formal verification of crash consistency

Software verification tools are best suited for verifying properties encodable with Hoare logic [77, 111], and crash consistency does not seem to fit this mold. In Hoare logic, one writes specifications for functions in the form of preconditions, which must be true when the functions are invoked, and postconditions, which must be true when the functions complete. However, crashes may occur partway through function execution, and Hoare logic does not let a developer directly specify conditions that must hold throughout the body of each function. For this reason, there has been much research proposing specialized methodologies for reasoning about crash consistency. Table 5.1 compares these techniques with our proposed approach, PoWER, across several dimensions. We describe related work on verifying crash consistency in more detail in Chapter 6.

	Automation	Expressivity	System perf.	Tool support
CHL [40]	Low	High	Low	Low
Atomic inv. [32]	Low	High	Med	Low
State machine [104]	Med	High	Low	Med
Push-button [240]	High	Low	Low	Low
PoWER	**Med**	**High**	**High**	**High**

Table 5.1: **Verification technique comparison.** The table compares different crash-consistency verification approaches.

Crash Hoare logic. Crash Hoare logic (CHL) is used by FSCQ [40], an xv6-like file system implemented and verified in Rocq [226]. CHL extends traditional Hoare reasoning with *crash conditions* that describe system state in the event of a crash. CHL's support for crash conditions is based on separation logic [223], which enables describing disjoint resources (e.g., different parts of a disk) with separate predicates. CHL is currently only supported by Rocq-based verification tools, which have a steep learning curve. Rocq programs can only be run by first extracting them to Haskell or OCaml, which limits their performance. In contrast, PoWER can be used with any verifier that supports standard Hoare logic, including verifiers like Verus [163] that produce faster code.

Atomic invariants. GoJournal [32] and other systems using Perennial [31] make use of an advanced language feature called atomic invariants to reason about crash states. An atomic invariant is a predicate about a resource (e.g., a storage device); the only way to access the resource is to "open" the invariant, which allows execution of a single atomic instruction before re-establishing the predicate and "closing" the invariant. This approach is not tool-agnostic; support for atomic invariants is difficult to implement, so many popular verifiers (e.g., Dafny [171], Prusti [10], and Creusot [61]) do not support them. Even in those that do (e.g., Perennial, Verus), they are challenging to use and often lack documentation.

State machine refinement. State machine refinement was originally developed for

verification of distributed systems in the IronFleet project [107]. VeriBetrKV [104] applies the technique to verify a key-value store based on the intuition that storage software interacting with a disk is similar to unreliable nodes interacting in a distributed system. In these systems, each component (e.g., a single node, or a storage system journal) is modeled as a state machine and proven correct in isolation using Hoare logic assuming a synchronous and crash-free environment. Components are then proven correct via a series of state machine refinement proofs using TLA-style reasoning, which also prove properties related to asynchrony and crash consistency. IronFleet and VeriBetrKV are implemented and verified in Dafny with custom-built libraries for TLA-style reasoning, which is not natively supported. In contrast, our techniques do not require additional infrastructure or separate proofs of state machine refinement.

Push-button verification. Push-button verification [211] trades expressivity for the ability to verify code without writing any proofs. Crash refinement [240] is a technique for push-button crash consistency verification in which the developer specifies a (finite) set of all possible crash schedules and requires that all executions obey the specification. This significantly reduces annotation and proof burden but is limited to simple, bounded systems. Other highly-automated verification techniques, such as the approach taken in TPot [29], have not been applied to storage verification.

5.1.2 Formal verification of corruption detection

VeriBetrKV [104] formally reasons about the use of checksums for corruption detection. However, its corruption-detection axiom strongly dictates how data must be checksummed and where the checksum must be stored. VeriBetrKV does not allow checksumming data across blocks, and precludes storing the checksum separately from the data. As we discuss in §5.3.1, this is overly restrictive for the systems we target. In contrast, our proposed model imposes no limitations on what data is checksummed or on where the checksum and data are stored. Moreover, it is difficult to determine

what actual constraints VeriBetrKV's axiom implies for the underlying storage device, and whether it is sound, in contrast to our axioms that reason about the number of corrupted bits that CRC algorithms are designed to detect.

5.1.3 Summary

Properties like crash consistency and corruption detection are difficult to reason about using standard verification techniques like Hoare logic. To overcome this, existing techniques have restricted the tools and approaches available to developers who want to verify robust storage systems. Existing crash-consistency verification approaches rely on non-trivial additional infrastructure and/or specific language features. This restricts their applicability to future verified systems, especially because developers of these systems may want to use newer, more advanced verification languages that do not support these features in order to achieve good system performance. Furthermore, the only existing model of corruption detection in a verified storage system is not compatible with byte-addressable storage media. These two obstacles motivate the development of our new techniques for verifying crash consistency and corruption detection, which we discuss in the following sections.

5.2 Verifying crash consistency using PoWER

This section discusses our novel, tool-agnostic technique PoWER for verifying crash consistency in storage systems. Unlike prior work, it can be used to concisely verify crash consistency with only basic verifier features.

5.2.1 PoWER

Our main contribution is Preconditions on Writes Enforcing Recoverability (PoWER), a way to verify crash consistency based on standard Hoare logic. This is challenging because most verification tools are designed to reason about the state of the system before and after a method's execution. They do not provide a clear way

to specify what must be true *throughout* the method's execution, and since crashes may happen at any time, such a specification is necessary.

Overall idea. The key idea behind PoWER is to enforce at the API level that all durable updates must be provably crash safe. To do this, we add a single precondition to the API's write method, which requires that all new crash states introduced by the write are crash consistent.

This is possible because one can describe all new crash states introduced by a write *before* it is invoked, even though asynchronous partial completion of those writes can occur at any time. By adding a precondition to the write method that forces the developer to reason about all such resulting crash states, we can ensure the developer cannot introduce crash-consistency bugs.

Modifying the API this way ensures correctness with no performance cost because all annotations involved in ensuring preconditions on writes are erased at compile time. The compiler generates an executable equivalent to what would exist with the standard, non-PoWER API.

The PoWER API. A standard storage API includes three main methods: `read`, which returns the most recently written bytes at a given address; `write`, which starts an asynchronous write but does not necessarily make it immediately durable; and `flush`, which ensures prior writes are durable. Their standard preconditions check properties like addresses being in bounds.

Figure 5.1 shows a simplified specification of a `write` method in the PoWER API in Verus. The `requires` clause specifies preconditions and the `ensures` clause specifies postconditions. `old(x)` is the contents of mutable reference `x` upon method invocation. `x@` is shorthand for `x.view()` and represents an abstraction of `x` (e.g., the current state of a storage device including outstanding writes).

The new precondition introduced by PoWER is on lines 5 and 6; it requires that all newly introduced potential crash states are permitted. The `spec` function

```
pub exec fn write(&mut self, addr: u64,
  bytes: &[u8], perm: Tracked<&Perm>)
 requires
  addr + bytes@.len() <= old(self)@.len(),
  forall|s| can_result_from_partial_write(s, old(self).durable_state,
   addr as int, bytes@) ==> perm@.check_permission(s),
 ensures
  self@.can_result_from_write(old(self)@, addr as int, bytes@)
```

Figure 5.1: **PoWER write example.** The listing shows a simplified signature of an asynchronous `write` method used with PoWER in Verus.

`can_result_from_partial_write` (defined in Figure 5.2) evaluates to true if `s` is a possible crash state when writing `bytes` to `addr` in `old(self).durable_state`. In Verus, we express the set of permitted states with an unforgeable ghost (i.e., will be erased by the compiler) permission token `perm`. Thus, this precondition specifies that all storage states that may occur in the event of a crash during this write must be allowed by `perm`. The set of newly-introduced potential crash states is defined by the storage model, which accounts for device properties like atomic write granularity and alignment.

The postcondition of this function specifies that the state of the PM device is updated to reflect the write operation by the time it completes. We describe the exact meaning of `can_result_from_write` in the next section.

Storage model. PoWER is not tied to any specific device model or specification, and we used several different models while developing systems using PoWER. It can be used with any storage model that supports the standard operations described above and captures the behavior of the device in the event of a crash.

The PoWER systems described in this chapter initially used a storage model that we call the "natural model" as it is based on the standard intuition developers use when reasoning informally about storage systems. The model we currently use is based on the prophecy-based asynchronous disk model [249] used in Perennial [30,32].

Compared to the natural model, this model eases reasoning about crashes by letting proof code reason about possible future system states, akin to prophecy variables used in some refinement proofs [130, 159].

To explain the prophecy model, we first describe the natural model. The storage device is modeled as a sequence of bytes, which each have a durable value and optionally one or more "outstanding" values that may be lost in a crash. For reasoning about crash consistency, storage is divided into chunks with size equal to the atomic persistence granularity of the device. For instance, persistent memory uses a granularity of 8 bytes, while hard drives generally use 512 B or 4 KiB sectors. A read returns the most recent outstanding value for all bytes. A write updates the list of outstanding values at each modified byte. A flush completes all outstanding writes, replacing each byte's durable value with its most recent outstanding value. On a crash, each outstanding write operation is divided into chunk-granularity subwrites and some of the subwrites are durably performed. This model is tricky to reason about because the state includes both a current readable state and a set of outstanding writes. Developers must explicitly reason about all outstanding writes and their potential crash states each time the storage device is accessed, which complicates proofs. We initially simplified this model by allowing only one outstanding value at each byte at any time, but this imposed some unnecessary restrictions on legal operations and required us to prove the absence of outstanding values in the range of every durable update.

The prophecy model, given in Verus in Figure 5.2, is simpler in that the state consists of only two byte sequences: the read state and the durable state (lines 2 and 3). The read state reflects all writes performed so far, including outstanding updates that may be lost in a crash. The durable state reflects all subwrites performed so far that will *eventually* become durable, due to either a subsequent flush or a crash that will nondeterministically choose to render them durable.

Reads always return the contents of the read state. A write operation applies

```
pub struct PersistentMemoryRegionView {
 pub read_state: Seq<u8>,
 pub durable_state: Seq<u8>
}

pub open spec fn chunk_corresponds(s1: Seq<u8>, s2: Seq<u8>, chunk: int)
  -> bool
{
 forall|i: int| 0 <= i < s1.len() && i / chunk_size() == chunk ==>
  s1[i] == s2[i]
}

pub open spec fn can_result_from_partial_write(
 post: Seq<u8>, pre: Seq<u8>, addr: int, bytes: Seq<u8>) -> bool {
 post.len() == pre.len() && forall |chunk| {
   ||| chunk_corresponds(post, pre, chunk)
   ||| chunk_corresponds(post, update_bytes(pre, addr, bytes), chunk) }
}

impl PersistentMemoryRegionView {
 pub open spec fn can_result_from_write(self, pre: Self, addr: int,
   bytes: Seq<u8>) -> bool
 {
 &&& self.read_state == update_bytes(pre.read_state, addr, bytes)
 &&& can_result_from_partial_write(self.durable_state,
     pre.durable_state, addr, bytes)
 }
}
```

Figure 5.2: **Partial storage specification.** The listing shows part of the specification used by the prophecy-based asynchronous storage model in Verus.

the entire write to the read state and a nondeterministically chosen subset of chunk-granularity subwrites to the durable state. The function `can_result_from_write` in Figure 5.2 (lines 21–27) describes the effects of a given write. It specifies that the read state receives the full update (line 24), and that the durable state is updated to become a possible crash state (line 25–26) according to the storage model. The call to `can_result_from_partial_write` does not concretely choose a specific potential crash state; rather, it states that the resulting durable state is *some* crash state without specifying its exact contents. We say that this state is *prophesized* by the model. The postcondition of flush in the prophecy model asserts that the read state matches the durable state, but that neither of these is changed by the flush. That is, the durable state does not *become* the read state as a result of the flush. Rather, it asserts that the prophesized durable state must have already been equivalent to the read state. The intuition for why this is valid is that the flush narrows down the set of possible prophesized states to only the state with the full write, since we no longer need to consider possible branching timelines in which some subwrites were lost.

We switched from the natural model to the prophecy model partway through development of CapybaraKV and found that it made proving crash consistency much simpler. One must still reason, when calling a PoWER `write`, about all the possible new crash states the write can introduce. But after the write, thanks to the prophecy model, one need only reason about the single prophesized resulting durable state. The Perennial authors have proven, in Rocq, that any system proven correct using the prophecy model is also correct using the natural model, so this switch was safe to do. We have also formally proven the soundness of our prophecy-based model by building it as a library atop a similar natural non-prophecy model, leveraging support for prophecy variables in Verus.

Specifying crash-consistent states. PoWER, like all crash-consistency specifications [32, 40], requires that the specification writer formally specify the set of crash-consistent states. In theory, the developer may define this set however they like. We

```
pub exec fn log_append(&mut self, ps: &mut PoWERStorage, bytes: &[u8],
  perm: Tracked<&Perm>)
 requires forall|s| perm@.check_permission(s) <==> {
  ||| Self::rec(s) == Self::rec(old(ps)@)
  ||| Self::rec(s) == Self::rec(old(ps)@) + bytes@
 }
 ensures Self::rec(ps@) == Self::rec(old(ps)@) + bytes@,
```

Figure 5.3: **Verified log append signature.** The listing shows a signature for a log-append method that enforces crash consistency with PoWER. The `|||` syntax in Verus is similar to `||` and allows for "bulleted list" organization of disjunctions.

suggest the following approach, exemplified by Figure 5.3 which shows a simplified signature for a synchronous, crash-atomic `append` operation in an append-only log. Have the code define a recovery function `rec` that maps a sequence of bytes to an abstract state. Then, make the set of crash-consistent states be the union of two sets: (1) the set of states abstractly equivalent to the initial state, and (2) the set of states permitted by the postcondition. In other words, when crashing mid-operation, the code may either atomically execute that operation or do nothing.

5.2.2 Correspondence to other approaches

To demonstrate the soundness of the PoWER approach for specifying crash consistency, we use mechanically-checked proofs to show correspondence to two other approaches: CHL and atomic invariants.[1] These are among the current state of the art for verified storage systems, but require additional verifier features or code infrastructure beyond Hoare logic and are thus not tool-agnostic.

Correspondence to Crash Hoare logic. To validate the soundness of our PoWER specification approach, we produce a mechanically-checked Rocq proof of its correspondence to CHL. We prove that any code satisfying a PoWER specification satisfies a corresponding CHL specification, encoded as a *crash weakest precondition* (WPC)

[1]These proofs were contributed by Nickolai Zeldovich.

in Perennial, whose crash condition states that, if the system crashes, the storage will satisfy the recoverability predicate.

Since we have not implemented PoWER in Rocq, this proof is metalogical and depends on a trusted translation of PoWER semantics into Rocq. This translation is fairly natural, so we feel confident in the correctness of this proof.

Correspondence to atomic invariants. Crash consistency can also be specified using atomic invariants, an advanced feature of some verification tools including Verus but not Dafny, Prusti [10], or Creusot [61]. To demonstrate that satisfying a PoWER specification implies the satisfaction of an atomic invariant, we produce a Verus library that exposes the PoWER interface but enforces an atomic invariant about the storage state. Since both the PoWER interface we use for Verus code and this invariant-based specification are both in Verus, this proof, unlike the one for CHL, is entirely machine-checked.

5.2.3 Strategies for satisfying preconditions

We next discuss how a developer can prove their code matches a PoWER specification. The challenge is proving, immediately before each `write`, that all new crash states that can result from partial application of the write are permitted. In this subsection, we describe four design patterns that simplify this reasoning, and libraries we provide that make it easy to apply them. We classify durable updates into four categories—tentative, committing, recovery, and in-place—and provide strategies to prove the crash consistency of each category.

Tentative writes. We call a write *tentative* if it is intended to have no effect on the abstract system state until some subsequent write happens. Tentative writes generally modify data at addresses that are unreachable during recovery (e.g., inodes unreachable from a file system's root), and do not change system state regardless of which subwrites become durable. Their contents only become relevant after a subsequent, non-tentative write (e.g., storing a reachable direct or indirect pointer to

```
ghost predicate AddressesUnused<T>(s: seq<byte>, addrs: set<int>,
    rec: seq<byte> -> T)
{
  ∀ s2: seq<byte> : : |s2| == |s| ∧
    (∀ i: int : : 0 ≤ i < |s| ∧ i ∉ addrs ⟹ s[i] == s2[i]) ⟹
      rec(s2) == rec(s)
}

lemma Lemma_TentativeWritePermitted<T>(ps: PoWERStorage,
    addrs: set<int>, rec: seq<byte> -> T, bytes: seq<byte>,
    start: int)
  requires ∀ s : : rec(s) == rec(ps.View().durableState) ⟹
    s in ps.StatesPermitted()
  requires AddressesUnused(ps.View().durableState, addrs, rec)
  requires ∀ addr: int : : start ≤ addr < start + |bytes| ⟹
    addr in addrs
  ensures ∀ s : : CanResultFromPartialWrite(s,
    ps.View().durableState, start, bytes) ⟹
      s in ps.StatesPermitted()
```

Figure 5.4: **Dafny tentative write lemma.** The listing shows a Dafny library lemma for proving that a tentative write satisfies the PoWER write API.

the address).

The developer need not prove anything about the specific bytes in a tentative write to prove crash consistency. They must only prove that the addresses modified by the write are unreachable by the recovery function. We provide lemmas in our Verus and Dafny libraries (see Figure 5.4 for the Dafny version) that the developer can call to satisfy the precondition of the PoWER API.

Committing writes. We call a write *committing* if it changes the abstract state of the system using a single crash-atomic write. For example, such a write might update a pointer that causes a tree of objects on storage to become reachable by the recovery function. A committing write is typically done after a flush to ensure that a crash does not cause the state to be invalid due to lost tentative writes.

For committing writes, the developer only needs to reason about two possible crash states: the states that result from the committing write being dropped or applied. We provide, in our library, a lemma that ensures that for a committing write there are only these two possible crash states.

Recovery writes. *Recovery writes* are writes done as part of a recovery procedure (e.g., replaying a journal). Such writes are not tentative, as they can change the abstract view, and they are not committing, as they may not match the atomic write granularity. To prove that such writes are crash consistent, a developer must prove that they only modify bytes that will be written to by a completed recovery procedure, and that recovery is idempotent. If the system crashes while recovering from an earlier crash, all modifications made during the first recovery will be overwritten by the second recovery, ensuring that torn writes due to the second crash are fixed when recovery completes.

We have written a generic log component that can be used as an operation log in Verus. In CapybaraKV, we install logged operations on log commit or when replaying the log after a crash. Internally, this operation log must reason about intermediate crash states, but the user of the log component does not have to. They just log updates and eventually commit (or abort) the updates gathered in the log.

In-place writes. *In-place writes* non-atomically modify user-visible state and can change the abstract state of the system. They thus leave the system in a non-deterministic abstract state. This may be reasonable for systems that provide weak crash-consistency guarantees, e.g., a file system that lets a read see a write that is then lost by a crash. Such a specification would permit a large set of possible crash states, so in-place writes can be more easily proven to produce states in that set.

We have not yet had experience verifying storage systems with weakly crash-consistent semantics, so we currently have no support in our libraries for reasoning about in-place writes. The developer of such a system will have to directly prove the PoWER preconditions.

5.2.4 Extending PoWER for concurrency

Both of the verified systems we built using PoWER are single-threaded, and we achieve parallelism in CapybaraKV via sharding (§5.5.2). As an alternative, we speculate that PoWER could be extended in the future to support concurrency directly in verification tools that support it.[2] For example, a concurrent variant of PoWER for Verus could exploit Verus's ownership model, so that calls require full ownership of addresses they write to and partial ownership of addresses they read from. This way, we could ensure conflicting operations cannot occur and thus a reduction argument [180] makes it valid to consider there to be a total order of all operation invocations. (Note that operations themselves may not be totally ordered: An asynchronous write can still have post-invocation delayed effects on the durable state that do not obey the total order because chunk-granularity subwrites are each independently scheduled.) Once we have this, we can expect developers to be able to prove that their writes satisfy the preconditions required by a PoWER specification. They will likely need to use (in code outside the TCB) a global invariant about the state of the system to prove this. Such an invariant will let them reason, based on their local knowledge of the subset of the storage that is owned (or partially owned), about the possible crash states that can result from initiating their write. For instance, they may have partial ownership of the path from the recovery root to the state of their component. This partial ownership can be enough to conclude, e.g., that a certain part of the component's state is irrelevant to recovery and can thus be tentatively written to.

5.3 Provably detecting corruption

Stored data may become corrupted over time due to media errors, so checking the integrity of data using cyclic redundancy checks (CRCs) is standard in many

[2] This extension was proposed by Chris Hawblitzel, Jay Lorch, and Nickolai Zeldovich.

storage systems [20, 74, 262, 267]. A verified storage system should require that data read from the storage device is checked for corruption before it is used or returned to the user. In this section, we introduce a new model of media corruption, and a new corruption-resistant atomic primitive for persistent memory.

5.3.1 Modeling media corruption

We present a model of data corruption that allows developers to prove the absence of corruption under the assumption that no more than a certain number of bits will be corrupted.[3] Possible data corruption is modeled in the postcondition of the `read` method. In the event of corruption, the bytes returned by `read` may not match the last-written bytes to that location. We represent this using a predicate `maybe_corrupted`, shown on lines 1–7 in Figure 5.5, which we describe below. The developer must perform a CRC check to prove that the returned bytes are uncorrupted before using them.

There are several possible ways we expect PM may be corrupted. For example, random bit flips due to hardware errors or cosmic rays may corrupt the contents of memory and storage devices. Since PM is generally mapped into a file system and/or application's address space, "scribbles" from misbehaving code may also unexpectedly modify its contents [267]. This issue cannot be eliminated entirely via verification, as it may be caused by unverified components, other applications, or the kernel itself. The corruption model presented here takes advantage of the design of CRC algorithms to provably guarantee that a small number of flipped bits will always be detected, so we primarily target on the former. It does not make strong guarantees about detecting larger amounts of corruption, but since CRC collisions are rare in practice, we expect that it will be effective at detecting such corruption in general.

Our model of device corruption, as specified in Figure 5.5, is as follows. The storage model includes a ghost corruption mask with one bit per storage bit, which

[3] Nickolai Zeldovich contributed to the development of this model.

```
pub open spec fn maybe_corrupted(self, bytes: Seq<u8>,
  true_bytes: Seq<u8>, addrs: Seq<int>) -> bool
{
 &&& bytes.len() == true_bytes.len() == addrs.len()
 &&& forall |i: int| 0 <= i < bytes.len() ==>
      exists |mask: u8| {
       let masked_byte = mask & self.corruption[addrs[i]];
       bytes[i] == true_bytes[i] ^ masked_byte
      }
}
fn read(&self, addr: u64, num_bytes: u64) -> (bytes: Vec<u8>)
 requires self.inv(), addr + num_bytes <= self@.len(),
 ensures
  ({
   let true_bytes = self@.read_state.subrange(
    addr as int, addr + num_bytes);
   let addrs = Seq::<int>::new(true_bytes.len(), |i: int| i+addr);
   self.constants().maybe_corrupted(bytes@, true_bytes, addrs)
  })
```

Figure 5.5: **Read method specification.** The listing shows a read method specification describing possible corruption of returned bytes.

represents which bits may be corrupted. Where the bitmask is 0, reads return the correct data. Where the bitmask is 1, reads return arbitrary bits, not necessarily the same on each read. The population count of the bitmask (i.e., the number of 1s) is bounded by a constant c that is opaque to the verified code. The verified code has access to a trusted CRC library and an axiom stating that any two byte sequences with Hamming distance in $[1, c]$ have different CRCs. For the ECMA variant of CRC-64 that we use, $c = 1$ for arbitrary-length data (i.e., it guarantees to catch any single bit errors). This value can be higher for shorter lengths (e.g., $c = 3$ for approximately 1 megabyte) [151], but our implementation does not currently take advantage of this. This means that, assuming c or fewer device bits are corrupted, the result of a CRC check on a given buffer definitively proves whether the buffer has been corrupted.

Our model differs from VeriBetrKV's "corruption cannot produce a block with a valid checksum" [104]. Our model is more fundamental, describing the behavior of

the media at a lower level. It is also more flexible, allowing the contents protected by a checksum to be noncontiguous and to not be in the same block as the checksum. This flexibility is required when building PM systems like CapybaraKV and CapybaraNS, as we discuss next.

5.3.2 Checking for PM corruption

Persistent memory presents new challenges when it comes to maintaining CRCs for corruption detection. Traditional storage systems often store a CRC of each block's contents within the block itself [241]. However, PM's finer write granularity (8 aligned bytes [258]) makes this technique crash-unsafe, as the hardware does not guarantee that the CRC will be written atomically with any non-trivial amount of data. In this section, we introduce a new primitive for crash-atomic updates to arbitrary-sized data on PM.

To motivate the need for a new primitive, we first discuss existing work on crash consistency and corruption detection on PM. We tested NOVA-Fortis [267], the only corruption-resistant PM file system, with Chipmunk (Chapter 3) and found that its CRC management logic is prone to crash consistency bugs. Furthermore, we find that NOVA-Fortis's *Tick-Tock* algorithm for updating data and CRCs is based on a different set of assumptions and is not sufficient in either our model of corruption or VeriBetrKV's [104]. Tick-Tock maintains two copies (a primary and a replica) of each persistent data structure, each with its own CRC. To update the data structure, it updates the primary, then flushes, then updates the replica. On recovery, it uses whichever copy has a matching CRC, preferring the primary. However, CRC algorithms are designed to defend against media corruption in the form of a bounded number of bit flips, not to distinguish between different user-level values. CRC collisions between such values, which can differ arbitrarily, are plausible (and indeed likely in scenarios where adversaries can manipulate values).

For example, suppose a data structure currently has value D_0, and we begin

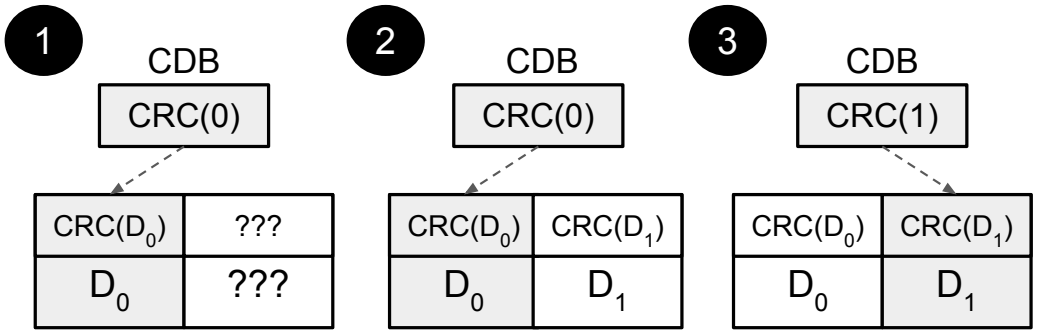

Figure 5.6: **CDB usage.** The figure shows one way to use a CDB to atomically update a data structure and its CRC.

updating it to D_1. Tick-Tock writes D_1 and $\mathsf{CRC}(D_1)$ to the primary; suppose a crash occurs after D_1 is written but before the CRC becomes durable, causing a CRC mismatch. Also suppose that the stored version of D_1 is corrupted into D_1' and that, by chance, $\mathsf{CRC}(D_0) = \mathsf{CRC}(D_1')$. Tick-Tock's primary CRC check will pass, and it will not detect the corruption. This is plausible in both our and VeriBetrKV's models, since D_1' and D_0 may have an arbitrary number of bit differences.

To address this challenge, we propose the following new primitive that enables atomic updates on PM. The *corruption-detecting Boolean* (CDB) is an 8-byte integer that can only take on two specific values, one representing `false` and one representing `true`. These two values should be chosen carefully such that neither is likely to be corrupted into the other; we use $\mathsf{CRC}(0)$ and $\mathsf{CRC}(1)$. Since a CDB is 8 bytes, it can be written to PM atomically with respect to crashes. Since its valid values are statically known, it can be checked for corruption without needing to maintain a separate CRC.

Here is one way to use a CDB to implement an atomically mutable data structure D. Reserve space for an 8-byte CDB and two versions of D plus their CRCs. The CDB indicates which version is considered valid at recovery time. To update, tentatively write a new version and its CRC to the invalid location, then flush, then use a committing write to flip the CDB, then flush. Figure 5.6 illustrates this for

the case where we start in step ❶ with D_0 as the valid version. In step ❷ we write the new version to D_1 and its CRC to $\mathsf{CRC}(D_1)$, then flush. In step ❸ we update the CDB to $\mathsf{CRC}(1)$ then flush. For the next update (not shown in the figure), we will go the other way: store the new version and CRC in D_0 and $\mathsf{CRC}(D_0)$, then flush, then update the CDB to $\mathsf{CRC}(0)$. Intuitively, this technique is similar to atomic pointer updates, which are commonly used in PM storage systems and concurrent code.

We find the CDB to be an extremely useful primitive, and use it in several places in CapybaraKV and CapybaraNS to facilitate atomic updates. For instance, we use it in CapybaraKV's log to atomically advance the log head or clear the log, two operations that form the basis of the system's atomic guarantees, using the steps in Figure 5.6. To our knowledge, ours is the first proven-correct algorithm for atomic updates on PM with corruption detection.

5.4 Verified systems

To demonstrate our support for multiple verification tools, we implement CapybaraKV, the first verified PM key-value (KV) store, in Verus and CapybaraNS, the first verified persistent notary service, in Dafny.[4]

5.4.1 CapybaraKV

CapybaraKV is an embedded PM key-value store with verified functional correctness, crash consistency, and corruption detection. It supports standard create, read, update, and delete operations on key-value pairs. It is parameterized by key and value types. CapybaraKV supports crash-atomic transactions in which operations are visible immediately but not durable until committed.

Figure 5.7 shows the workflow for verifying CapybaraKV and compiling it

[4]Jay Lorch, Chris Hawblitzel, Cheng Huang, and Yiheng Tao contributed to the design of CapybaraKV. Jay built CapybaraNS and several components of CapybaraKV.

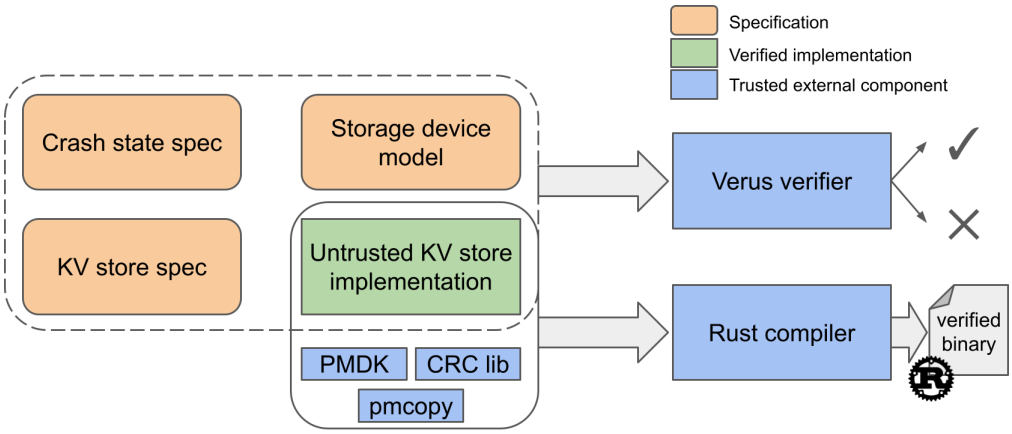

Figure 5.7: **CapybaraKV verification workflow.** The figure shows how different components of CapybaraKV are used to verify its implementation and compile it to an executable.

to a binary. CapybaraKV includes a specification of legal crash states, correct KV store behavior, and a model of the underlying storage device that are used only during verification. These specifications are erased when the implementation is compiled. The implementation has several dependencies on unverified external libraries. It uses PMDK [118] to invoke cache-line write backs and store fences and an external crate `crc64fast` [54] to calculate CRCs. It also uses `pmcopy` (§5.4.1.1), a crate we developed for use with CapybaraKV and future PM systems written in Verus, that helps developers prove crash consistency.

Specification. Its abstract state is two maps from keys to values, with one map representing what would result from an abort and the other from a commit. Its operations include create, read, update, delete, and commit. In the event of a crash, the commit operation may either abort or commit the current transaction; all other operations must abort.

Implementation. CapybaraKV has three main durable components: a main table, a value table, and an operation log. The main table and value table are durable arrays storing keys and values, respectively. Each entry in the main table contains a

key, the index of the corresponding value in the value table, a CRC over these fields, and a CDB (§5.3.2) indicating if the entry is valid. Each entry in the value table contains a value and its CRC. A value is considered valid if it is pointed to by a valid main table entry.

The operation log is a physical redo log. When a transaction is committed, we tentatively append a single CRC for all pending log entries, then commit the log using a committing write to a CDB as shown in Figure 5.6. Log entries are replayed using recovery writes. Once all entries have been installed, we issue a flush and clear the log.

New records are created in CapybaraKV by tentatively writing new main and value table entries and logging an update to the main table entry's CDB. Deletions only require logging the invalidation of the target's CDB. CapybaraKV uses copy-on-write to update existing records and logs an update to the value index of the record's main table entry. Log entries are small (¡64B) and never include keys or values.

CapybaraKV also has a volatile index, implemented using a Rust `HashMap` and verified using specifications from the Verus standard library, that maps all keys to their main table indexes. The main table and value table also maintain volatile free lists used as allocators. These volatile structures are rebuilt at startup after post-crash log replay.

5.4.1.1 Safe reads and writes

In PM systems like CapybaraKV, PM's low access latency makes minimizing overhead essential. In particular, I/O latency can easily be eclipsed by software overheads, so many systems `memcpy` data structures between DRAM and PM with no serialization. However, such low-level operations risk crash safety and can lead to undefined behavior. For example, structures that contain references (e.g., file handles or virtual addresses) cannot safely be stored on PM, as the reference may be invalid

```rust
#[repr(C)]
#[derive(PmCopy)]
pub struct PmCopyExample
{
  v1: u8,
  v2: u64,
  v3: i128,
  v4: bool
}
```

Figure 5.8: `PmCopy` **example.** The listing shows the definition of a durable data structure. The structure is annotated with `#[repr(C)]` to ensure Rust uses the C representation for its memory layout and `#[derive(PmCopy)]` to automatically generate safety checks and Verus-visible code about its layout.

after a crash. And, when reading stored data, we must ensure that data is placed in a properly laid-out buffer and checked for corruption before casting to a more useful data structure to avoid undefined behavior.

Unfortunately, these properties are difficult to verify because compiler-generated type and layout information is not available to verifiers. We tackle this issue by using the powerful Rust compiler to check properties that Verus cannot. We are inspired by Corundum [113], a Rust crate that uses various Rust language features to enforce safety properties in PM storage systems.

We have developed `pmcopy`, a trusted Rust crate that provides a macro to help developers check these crucial safety properties. It generates Verus ghost code to facilitate proofs that rely on type layout information and provides executable functions specified by this ghost code. It also adds compile-time assertions that are checked by the Rust compiler, not Verus, to check that axioms it synthesizes match compiler-generated information. CapybaraKV uses `pmcopy` to enforce safety properties about all durable data structures. We expect `pmcopy` will also be valuable in the development of other PM storage systems in Verus.

To use `pmcopy`, a developer need only include two annotations on the defini-

Trait	Description	Generated code
`PmSafe`	Ensures safety of copying to storage	No methods; trivial `PmSafe` trait bounds on field types
`PmSized`	Ensures size known at verification time	Spec and exec `size_of` and `align_of` methods, and static assertions that their output matches compiler-generated type layout
`Clone`	Implements explicit copy method	Specification that copy equals original
`PartialEq`	Implements equality operator	Specification that operator is consistent with Verus equality

Table 5.2: `PmCopy` **macro traits.** The table lists traits implemented by the `#[derive(PmCopy)]` macro in the `pmcopy` crate.

```
1 unsafe impl PmSafe for PmCopyExample
2   where u8: PmSafe, u64: PmSafe, i128: PmSafe, bool: PmSafe
3 {}
```

Figure 5.9: `PmSafe` **example.** The listing shows the implementation of `PmSafe` generated by the `PmCopy` macro on the structure in Figure 5.8.

tions of durable data structures, as show in Figure 5.8. The first annotation is the directive to use the C representation (`#[repr(C)]` on line 1), as the default Rust representation is intentionally under-specified and cannot be safely used for operations that rely on a known type layout [231]. The second annotation is `#[derive(PmCopy)]` (line 2), which causes `pmcopy` to automatically generate an implementation of the `PmCopy` trait and several supertraits. Routines that copy to and from PM require that their parameters implement this trait. `PmCopy` can be derived for `structs`, `enums`, and `unions`.

Table 5.2 summarizes four supertraits of `PmCopy` whose implementations are generated by deriving `PmCopy`. We describe these traits in more detail now.

`PmSafe`**.** This trait generates code that checks whether a structure is safe to store on PM. It has no methods and exists only to check this property at compile time.

Numeric types, characters, and Booleans are assumed to be `PmSafe`. Arrays of `PmSafe` types are also `PmSafe`. A user-defined structure is `PmSafe` only if all of its fields are `PmSafe`. Figure 5.9 shows the macro-generated implementation of `PmSafe` for the `PmCopyExample` structure defined in Figure 5.8. The implementation has trivial trait bounds (i.e., a bound that does not rely on type parameters) specifying that the type of each field must be `PmSafe`. If we attempted to derive `PmCopy` on a type that included a non-`PmSafe` type (e.g., a reference or raw pointer), this code would still be generated but would fail typechecking because the trait bound would not be met. `PmSafe` is unsafe to prevent users from providing their own, incorrect implementations. The only safe way to implement it is by deriving `PmCopy`.

`PmSized`. This trait (and a set of related helper traits generated by `PmCopy`) provides functions for reasoning about the layout of data structures in proofs. Although the Rust compiler statically generates type layout information, it is not available to Verus for use in proofs. This trait provides both executable and `spec` functions that manually calculate the size and alignment of each `PmCopy` type using the same algorithm used by the Rust compiler for `repr(C)` types. When used in verified code, these functions provide a way to formally reason about the size and alignment of types in proofs. Due to current limitations in Verus, we cannot prove that the executable and `spec` versions are equivalent, but we do check that the result of each of the executable functions matches the corresponding standard Rust version by generating static assertions of their equality.

`Clone` and `PartialEq`. These traits are part of the Rust standard library but do not have default specifications provided by Verus. `Clone` includes a method to explicitly duplicate an object, and `PartialEq` methods define what it means for two objects of the same type to be equal. Verus does not include a default specification of these traits for arbitrary user-defined types because the user may provide a conflicting trait implementation, which could introduce unsoundness. The lack of `PartialEq` specification is generally not an issue when dealing with concrete types, since Z3

supports equality comparisons, but can be problematic when dealing with generic types. Z3 has no built-in notion of a clone operation and user must always provide a spec for the behavior of `Clone`. This was particularly problematic in CapybaraKV's volatile index, which is generic over the key type and uses equality comparisons and clone operations in its executable code. To handle this, `PmCopy` generates an implementation and matching specification for both of these traits, which prevents the user from providing their own incorrect versions.

We found `pmcopy` very useful during development of CapybaraKV. Before writing it, we had hard-coded the layout and size of each data structure for use in proofs, but this is very risky. Whenever we defined a new structure or modified the contents of an existing one, we had to update this set of constants, which was error-prone. Adding `PmCopy` as a derivable trait made the addition and maintenance of user-defined types much easier and keeps the amount of trusted code related to type layouts and PM safety constant as the system grows. The static assertions about type layout also caught a change to Rust itself that would have impacted the soundness of our system. Partway through development of CapybaraKV, an update to the Rust compiler changed the layout of `u128` and `i128`, which caused an inconsistency between compiler- and `pmcopy`-generated layouts. Thanks to the static assertions `pmcopy` generated, this discrepancy was immediately flagged.

5.4.1.2 Discussion

CapybaraKV is designed for a particular use case, storing small keys and values on a small amount (10s of GiB) of dedicated PM, in a production cloud storage service. In targeting this use case, we made several simplifying design decisions that streamlined both implementation and verification but imposed limitations on functionality.

CapybaraKV requires users to allocate statically storage space and specify the number and size of keys and values at initialization. It does not currently support

dynamic resizing and will waste space if the number or size of records is smaller than initially specified. CapybaraKV uses a volatile index that keeps all keys in memory, which grows its memory footprint (especially if large keys are used) and must be rebuilt each time the system is started. CapybaraKV is single-threaded and does not require concurrency since parallelism can be achieved via sharding, as we discuss in §5.5.2. This parallelism is handled by a layer above CapybaraKV since PoWER does not currently support concurrency (see §5.2.4).

5.4.2 CapybaraNS

CapybaraNS is a notary service similar to the verified notary in Ironclad Apps [108]. It securely assigns logical timestamps to hashes so they can be conclusively ordered, and stores its state on persistent storage to permit persistence across crashes. We build and verify it in Dafny, with a trusted C# wrapper that provides external methods for CRCs, cryptography, and serialization.

Its abstract state consists of a current logical timestamp (a 64-bit unsigned integer) and a last hash. Its interface has two main operations: (1) `Advance` increments the timestamp and updates the last hash to one provided as input. (2) `Sign` uses the service's private key to sign a binding between the last hash and the timestamp. CapybaraNS uses the CDB algorithm from §5.3.2 to atomically update its storage state during an `Advance` operation.

5.5 Evaluation

This section addresses the following questions:

1. How much effort does it take to build and verify a new system with PoWER (§5.5.1)?
2. How does CapybaraKV compare to similar, but unverified, PM key-value stores (§5.5.2)?

	Trusted	Spec+Proof	Impl
CapybaraKV			
PoWER framework	971	2175	423
`pmcopy` crate	964	0	0
Base log	0	2173	590
KV store	1028	9815	3637
Total	2963	14163	4650
CapybaraNS			
PoWER framework	266	118	4
Notary server	148	545	274
Total	414	663	278

Table 5.3: **Verified lines of code.** The table lists the number of lines of code in each verified system.

5.5.1 Verification effort

Table 5.3 gives the number of lines of code in each of the major components of CapybaraKV and CapybaraNS, organized into trusted (i.e., unverified), specification/proof, and executable implementation code. We count the `pmcopy` crate towards CapybaraKV's trusted code, as it is unverified, but we do not count the lines of code it generates. The CapybaraKV PoWER trusted line count also includes a mock PM backend using a byte vector and backends for Windows and Linux. Both systems have a low proof-to-code ratio (6.0 for CapybaraKV and 2.4 for CapybaraNS).

Designing, implementing, and verifying CapybaraKV took approximately 1.5 years of work by a team consisting of both verification experts and newcomers. We built CapybaraNS when CapybaraKV was mostly complete, so its development benefitted from lessons learned when building CapybaraKV. It took less than one person-hour to port the PM specification to Dafny, about one hour to port the library supporting reasoning about tentative and committing writes, and about nine hours to implement and verify CapybaraNS in Dafny after writing its specification and C# wrapper.

The first part of CapybaraKV that we built was a durable log (the base

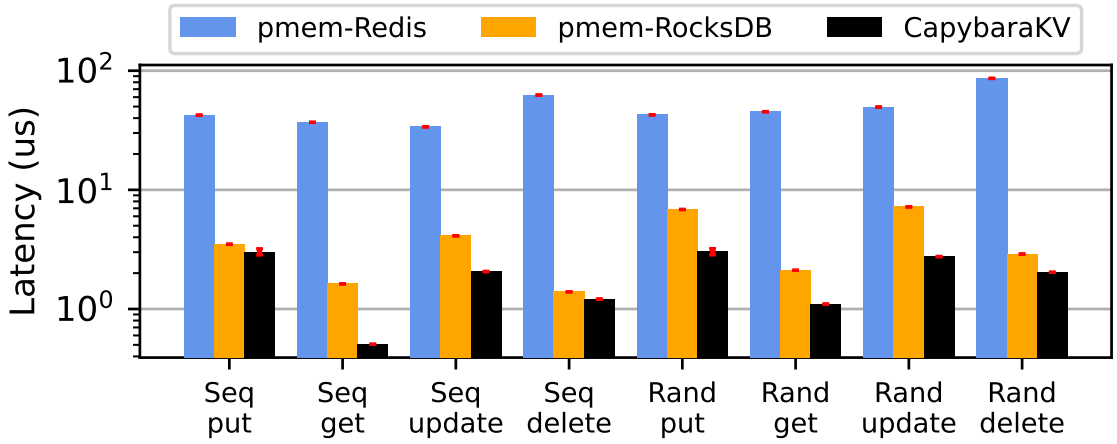

Figure 5.10: **CapybaraKV operation latency.** The figure shows average operation latency in microseconds. Note the log scale.

log in Table 5.3), and it went through several iterations before the final version was implemented and verified. The CapybaraKV-specific operation log was built on top of this log later on. Our initial version of the log provided synchronous atomic appends but did not check metadata for corruption. We subsequently added metadata CRCs, tentative appends with a commit operation, and experimented with different layouts and PM specifications. Thanks to Verus' fast verification performance and the fact that PoWER isolates crash-consistency reasoning to functions that perform updates, these modifications were generally straightforward; most difficulty came from designing new features rather than proving them correct.

Verification time. On one of our development machines (Linux v6.9.3, Intel Core i7–11850H CPU, 8 physical cores, 32GB memory), it took 56 seconds to verify CapybaraKV with 1 thread and 37 seconds with 8 threads. It takes 12 seconds to verify CapybaraNS with 1 thread; Dafny does not support multithreaded verification.

5.5.2 CapybaraKV performance

We evaluate CapybaraKV against two unverified PM key-value stores, pmem-Redis [119] and pmem-RocksDB [120]. CapybaraKV is the first verified KV store for persistent memory, so we could not compare to any prior verified systems. VeriBetrKV [104], the most similar verified system, is designed for block devices.

Experimental setup. Experiments were run on a two-socket machine with 32 physical cores, 128GB memory, and 128GB Intel Optane DC Persistent Memory. The evaluation machine runs Debian Trixie and Linux 6.7.12. We run the pmem-Redis server and client on the same machine, enable its pointer-based append-only file, and configure it to store the AOF and all values on PM. We configure pmem-RocksDB to memory-map files for reading and writing and to use non-temporal stores when appending to its write-ahead log.

Microbenchmarks. Figure 5.10 gives the average latency of put, get, delete, and update over 25M operations on records with 64B keys and 1KiB values in each evaluated system with both sequential and random access patterns (note the log scale). 95% confidence intervals are shown in red.

Pmem-Redis has the highest latency due to communication overhead between its client and server. CapybaraKV achieves similar or better latency to pmem-RocksDB on all measured operations primarily due to its operation log with tiny (¡64B) entries and its fast hash-map index. Pmem-RocksDB spends more time appending larger entries to its write-ahead log and managing its smaller but more complicated in-memory MemTable and cache structures. Key lookups in pmem-RocksDB involve searching the MemTable and potentially multiple durable files.

CapybaraKV's random get latency is approximately 2× its sequential get latency in this microbenchmark because sequential loads are faster than random loads on Optane PM [124]. The sequential get workload runs on records that were inserted sequentially, whereas random get accesses keys that were inserted in a different ran-

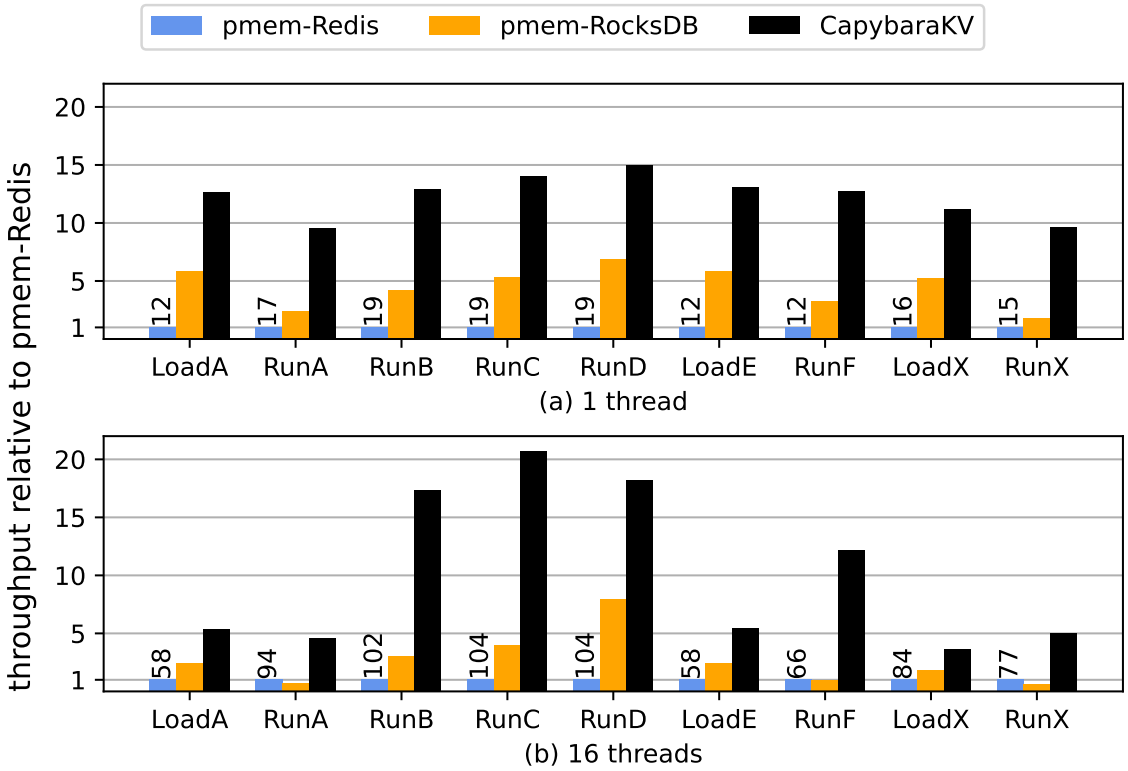

Figure 5.11: **YCSB performance.** The figures show YCSB throughput relative to pmem-Redis with (a) one thread and (b) 16 threads. Numbers above pmem-Redis bars show absolute throughput in kops/s.

dom order.

Battery-backed DRAM. We also evaluate CapybaraKV in a testing environment at a large cloud provider similar to the targeted production setup running Windows with 20GiB battery-backed DRAM. These experiments were run by Yiheng Tao. We run the microbenchmarks shown in Figure 5.10 on CapybaraKV in this environment and find that operations are up to 2× faster on battery-backed DRAM and follow similar performance patterns. We are unable to evaluate pmem-Redis and pmem-RocksDB in this environment as they do not support Windows.

Macrobenchmark: YCSB. We also measure each system's performance on several workloads from the widely-used YCSB benchmark suite [49]. CapybaraKV does

not currently support range queries, so we skip the YCSB workload (RunE) that includes them. We also introduce workload X, which is based on a trace of traffic to a production service similar to CapybaraKV at a large cloud provider and consists of 75% updates, 5% read-modify-write operations, and 20% reads with a uniform access distribution. All YCSB workloads use 15M keys and are executed 5 times on each system. The CapybaraKV instances in these experiments use 64B keys and 1024B values, both structured as byte arrays.

Single-threaded performance. Figure 5.11(a) gives the average throughput of each system using one thread relative to pmem-Redis. Unlike prior verified storage systems, CapybaraKV outperforms the unverified systems on these workloads. Pmem-Redis is unable to achieve high throughput on these operations due to its high per-operation latency. CapybaraKV performs better than pmem-RocksDB on these workloads for the following reasons. First, even in single-threaded workloads, pmem-RocksDB has background threads performing LSM tree compaction and flush operations, which can interfere with client thread performance; CapybaraKV uses no background threads. Second, CapybaraKV's in-memory hash-map index facilitates fast lookups for all keys, whereas pmem-RocksDB has to search its MemTable, and sometimes several durable files, to perform lookups. Pmem-RocksDB achieves its best performance on RunD, a read-heavy workload where the most recently-inserted keys are the most frequently accessed, because most lookups can be handled in the MemTable; in other workloads, lookups often require searching multiple durable files. CapybaraKV's index is also faster to modify than pmem-RocksDB's more complicated MemTable. Third, both CapybaraKV and pmem-RocksDB use CRCs to detect corruption, and pmem-RocksDB uses per-block CRCs that take longer to compute than CapybaraKV's CRCs on smaller data structures. Fourth, CapybaraKV takes less time writing to its log during insertions and updates because it uses tiny log entries that never include keys or values, whereas pmem-RocksDB's write-ahead log fully records all operations.

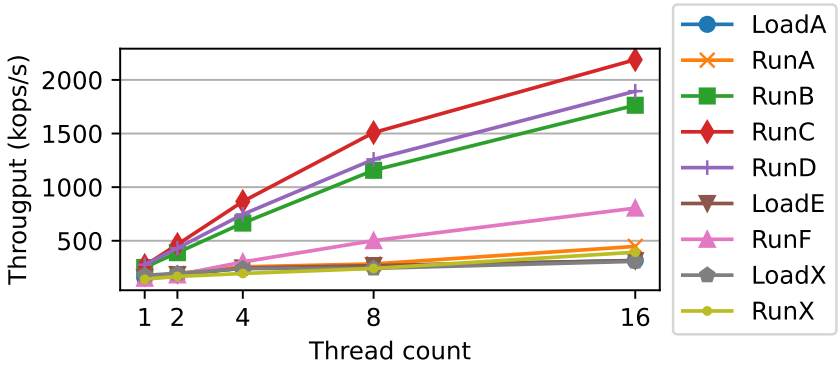

Figure 5.12: **Sharded CapybaraKV throughput.** The figure shows sharded CapybaraKV YCSB throughput with different thread counts.

Sharded performance. Figure 5.11(b) gives average throughput using 16 threads (the number of physical cores in each NUMA node on our test machine). CapybaraKV does not internally support concurrent access, but we can achieve parallelism via sharding, which we demonstrate here with the following simple setup. We build a new database interface layer in YCSB that, for a workload with k records and n threads, creates n separate CapybaraKV instances each with space for $k/n + 1$ records to act as shards. The shard placement of each record is determined using a hash of its key, and each shard is protected by a read-write lock to prevent concurrent mutations within a single shard. We do not shard the other two systems in this experiment, as they already support multiple concurrent clients. Figure 5.11(b) shows that with our simple sharding protocol, CapybaraKV scales similarly or better than the other systems with multiple client threads.

Figure 5.12 gives average throughput of sharded CapybaraKV on each YCSB workload with different thread counts. Read-heavy workloads like RunB (95% reads, 5% updates), RunC (100% reads), and RunD (95% reads, 5% inserts) scale well because the per-shard locks allow multiple concurrent readers. The Load workloads (100% inserts), RunA (50% updates, 50% reads), and RunX (75% updates, 5% read-modify-write, 20% reads) see some throughput improvement with more threads, but do not scale as well due to write lock contention. We find that our hash-based place-

	Startup time (ms)		Utilization (GiB)	
	Empty	Full	Memory	Storage
pmem-Redis	137	—	11.3	22
pmem-RocksDB	6	8	2.0	17
CapybaraKV	235	954	5.0	18

Table 5.4: **KV startup times.** The table lists startup times on empty and full instances, and memory and storage utilization on YCSB LoadA, on 128GiB Optane PM for each evaluated system.

ment scheme distributes keys reasonably evenly across shards, but it makes no effort to place hot keys in different shards or to temporally balance accesses. Some YCSB workloads perform partial value updates, which CapybaraKV does not support, so updates in our interface layer are implemented as read-modify-write operations. RunA and RunF (50% reads, 50% read-modify-writes) are thus very similar, but RunF scales better because the per-shard write lock is held for the entirety of an update operation and only during the write part of a read-modify-write.

Startup times. Table 5.4 compares how long it takes for each key-value store to start up on both an empty instance and a completely full instance on a 128GiB PM device. To measure full startup times, we insert records with 1KiB keys and 512KiB values into each system until it returns an out-of-space error, then repeatedly start and clean up each system on those records. Attempting to start the pmem-Redis server on a full instance fails with a memory allocation error. CapybaraKV's full startup time is about 4× its empty startup time because it initializes its in-memory index by scanning the entire KV store instance. For this same reason, it is slower than pmem-RocksDB on both empty and full startup.

Memory and storage utilization. Table 5.4 reports DRAM and PM utilization for the three evaluated systems on a 15M record instance set up by YCSB's Load A workload. In CapybaraKV, we use 64B arrays for keys and 1 KiB arrays for values. 18GiB is the approximate minimum space required by CapybaraKV to store 15M

records of this size; note that unlike pmem-RocksDB and pmem-Redis, this CapybaraKV instance cannot grow any further. In an optimally-provisioned instance, CapybaraKV has low storage space overheads due to its simple durable layout and does not use much more storage than pmem-RocksDB. CapybaraKV uses 2× more memory than pmem-RocksDB due to its in-memory index that contains every key in the instance. Pmem-Redis also keeps all keys in DRAM, but has higher per-key overhead than CapybaraKV (almost 700B as reported by the server). Pmem-Redis can be configured with a maximum memory limit after which it will evict records from DRAM, but this impacts throughput and is not set by default. To achieve durability guarantees similar to those provided by CapybaraKV and pmem-RocksDB, we configure pmem-Redis to store mappings from keys to durable values as well as the values themselves, which adds additional storage overhead.

5.6 Summary

This chapter presents techniques for proving crash consistency and corruption detection in storage systems. We introduce PoWER, a way to prove that a system is crash consistent using only basic verifier features like Hoare logic, and a new model of storage corruption that forces developers to properly perform corruption detection while giving them flexibility in how to do so. We develop a new primitive, the corruption-detecting Boolean, and show how to use it to implement a novel algorithm for atomic updates to PM data structures. To demonstrate that our approach is useful and tool-agnostic, we build two verified PM storage systems, CapybaraKV in Verus and CapybaraNS in Dafny. We evaluate CapybaraKV and find that its performance is competitive with unverified PM KV stores.

Chapter 6: Related work

This section provides an overview of work related to this dissertation. §6.1 discusses other persistent-memory storage systems. §6.2 covers prior work on testing crash consistency in both traditional and PM storage systems. §6.3 describes related work on lightweight formal methods, including techniques similar to those used in SquirrelFS. §6.4 discusses formal verification of storage systems.

6.1 Persistent memory storage systems

This section describes related work on storage systems that use PM as a storage medium. The systems described here all use DRAM for main memory and use PM either as the main storage device or as the top layer in a deeper storage hierarchy.

6.1.1 File systems

We first discuss file systems that support storing file data and metadata on PM. All the systems described in this section are written in C.

Modified block-based systems. Two widely-used Linux file systems, ext4 and XFS, have been modified to support PM. The PM versions, called ext4-DAX and XFS-DAX [177], take advantage of DAX support to bypass the page cache for data accesses. Otherwise, they share most code and functionality with their non-DAX modes. For example, metadata updates use the same block-based journaling code and the systems use the same durable layout for block- and byte-addressable media [131]. Ext4-DAX and XFS-DAX are currently the only PM file systems in the mainline Linux kernel.

PM-specific in-kernel file systems. There are a number of fully in-kernel file systems developed specifically for PM. These systems use a variety of architectures

that differ significantly from those of traditional storage systems and their ports to PM. BPFS [48] is a Windows file system that uses fine-grained copy-on-write for crash consistency. It predates the first announcements about Optane PM but correctly anticipated that PM hardware would support 8-byte atomic writes and ordering barriers. PMFS [69] is a POSIX-compliant PM file system built by Intel that uses undo journaling for crash consistency. While BPFS did not support memory-mapped files [48], PMFS used XIP (eXecute In Place), a predecessor of DAX, to provide applications with direct access to files. PMFS proposed two primitives for interacting with PM: a weakly-ordered version of `clflush`, and a `pm_wbarrier` operation to flush the contents of memory controller queues. Intel later added the `clflushopt` and `clwb` instructions, which support two optimizations for `clflush`. The addition of asynchronous DRAM refresh (ADR) as a requirement for PM support eliminated the need for `pm_wbarrier`, as it moved the write-pending queue into the power-fail protected domain.[1]

Subsequent systems use DAX, a feature in the Linux kernel that supports direct access to storage by file systems or user-space applications [177], to access PM. NOVA [266] is a PM file system that combines ideas from journaling and log-structured systems. Its successor, NOVA-Fortis [267], added support for detecting and recovering from data corruption using cyclic redundancy checks and replication. WineFS [132] is a hugepage-aware PM file system designed to prevent fragmentation over time, which can significantly improve performance for memory-mapped applications. It uses fine-grained per-CPU journaling to preserve data layout and prevent fragmentation. SoupFS [67] uses soft updates for crash consistency. Like SquirrelFS, it takes advantage of PM's byte-addressability to avoid complex block dependency management and rollback/forward logic, but unlike SquirrelFS it tracks dependencies for asynchronous writes at runtime.

[1]Before ADR was required for PM support, Intel proposed an instruction similar to *pm_wbarrier*, `pcommit`, to explicitly flush the write-pending queue. `pcommit` is deprecated and ADR ensures that the WPQ is automatically flushed when power is lost in current systems [228].

User-space and hybrid file systems. User-space file systems for PM can achieve significant performance improvements over in-kernel systems by removing the kernel from the storage software stack. The software overhead contributed by the kernel is negligible when compared to access latencies of slower block devices, but becomes a larger factor on PM [133, 155, 277]. However, user-space file systems are less compatible with legacy applications, as programs must be linked to a user-space library file system that intercepts system calls. It is also challenging to handle file permissions and concurrent accesses without mediation by the kernel, so many of these systems have both a user-space component and an in-kernel component [66]. We refer to these as "hybrid" file systems.

Strata is a file system that uses PM as the top layer in a storage hierarchy that can also include slower block media [155]. Assise, its successor, extends this idea to a distributed context [8]. Strata and Assise were both proposed as hybrid file systems but are currently implemented entirely in user space using `devdax` mode to manage PM directly from user space. CrossFS [222] also uses PM as a faster storage layer on top of a slower device and splits file system functionality between user, kernel, and firmware components to take advantage of device firmware features. SplitFS [133] uses a user-space library FS to handle data operations in memory-mapped files and offloads metadata operations to ext4-DAX. ZoFS [66] is designed to reduce interaction with the kernel by managing permissions for file system subtrees, rather than on a per-file basis, and uses hardware features to prevent corruption due to stray writes. The TRIO architecture [277] is a framework for building specialized user-space PM file systems tailored to particular use cases. TRIO-based systems implement most logic in user space but rely on the kernel to manage a small amount of core system state.

6.1.2 Key-value stores

Preexisting systems. Many existing key-value stores have been modified to support PM. This includes durable KV stores designed for block-based media and in-memory stores with no durability by default. These systems take advantage of PM's persistence, but their designs are not tailored to PM.

Pmem-RocksDB [120] is a PM KV store based on Meta's durable RocksDB system [224]. Pmem-Redis [119] is a version of the in-memory Redis [221] KV store that obtains durability by storing some data on PM. Like modified block-based file systems, these ported KV stores include some PM-specific optimizations but share most code with their original versions. As a result, these systems are often less performant than their PM-specific counterparts. In particular, they have been found to utilize very little of the available bandwidth of Optane PM [216].

PM-specific systems. Recent research has resulted in the development of several PM-specific KV stores. The majority of these systems use variations on log-structured merge (LSM) trees, which are common in block-based KV stores, and focus on improving throughput and reducing write amplification. ChameleonDB [274] uses a sharded structure for concurrency and an LSM implementation designed to reduce write amplification with Optane PM's 256B internal block size. SLM-DB [134] combines ideas from durable LSM trees and B+-tree indexes to simplify durable layout without sacrificing read or write performance. ListDB [144] combines LSM trees and skip lists to improve write performance and reduce remote NUMA accesses. Some systems start from existing KV stores but tailor them to specific PM use cases rather than simply porting existing functionality to PM. For instance, NoveLSM [138] is based on LevelDB [87] and uses a multi-level storage hierarchy with PM at the top. MatrixKV [272] is based on RocksDB and also uses PM as a storage layer above an SSD to more efficiently handle higher LSM tree levels.

Several systems have broken from the standard LSM-based design to further optimize performance on Optane PM in particular. Viper [18] uses an in-memory

hash map to map keys to durable blocks that are carefully aligned to optimize performance on interleaved NVDIMMs and allows applications to write directly to PM. FlatStore [41] also uses an in-memory index and batches small updates to PM in per-core logs. Skye [216] is based on an empirical study of KV operations on Optane PM and implements several bandwidth utilization optimizations. It retains fine-grained control over data placement on PM and manages individual NVDIMMs separately with a fixed number of worker threads.

6.2 Testing crash consistency

This section discusses related work on crash-consistency testing for both traditional and PM storage systems.

6.2.1 Testing traditional storage systems

`dm-log-writes` is a Linux device mapper target that takes two devices and uses one of the devices to log all I/O issued to the other [175]. It is not a full testing tool on its own, but can be combined with a replay tool and tests to check potential crash states. Block Order Breaker [215] (BOB) is a tool that records disk I/O and replays it to generate possible post-crash states in which persistence guarantees do not hold. BOB is used alongside the Application-Level Intelligent Crash Explorer (ALICE) to explore application-level behavior in the event of a crash, but cannot also not be used on its own to comprehensively test low-level storage systems. Although neither of these tools constitutes a crash-consistency testing framework, both can be used to explore possible crash states and likely inspired subsequent testing tools.

Many full crash testing tools use a record-and-replay approach to generate and test potential crash states. Zheng et al. [275] use this method to test databases for ACID properties in the event of power loss. They record operations at the iSCSI layer, which allows one server to access another's storage via block-level operations over the network. CrashMonkey [201] records file system I/O at the kernel block layer

during systematically generated test cases in order to find crash-consistency bugs in those systems. Hydra [143] is a file-system fuzzer that focuses on crash consistency bugs and POSIX violations.

As discussed in Chapter 3, these tools cannot be used with PM storage systems because they rely on the block interface provided by traditional storage media or on software layers designed for this interface.

6.2.2 Testing PM file systems

Yat [161], PMTest [183], and Vinter [135] have all been used to test PM file systems for crash-consistency bugs. Yat was built for PMFS and records PM I/O using a custom hypervisor. Yat has limited optimizations to reduce the space of crash states. For example, the authors report that a workload of 1200 `creat`, `mkdir`, and `write` calls would take over five years to complete. PMTest was also only used on PMFS and found no crash consistency bugs.

Vinter [135] is a PM file-system testing tool that instruments instructions using PANDA and uses a heuristic based on locations that are read during recovery to reduce the crash state space. It was developed concurrently with CHIPMUNK and the authors of Vinter independently found several bugs also found by CHIPMUNK. Silhouette [127] is a similar tool, developed after Vinter and CHIPMUNK, that uses a set of crash-consistency mechanism-specific persistence invariants to further reduce the crash state space.

6.2.3 Testing PM applications

Recent research on PM crash consistency has focused on finding bugs caused by incorrect use of persistence primitives, such as missing cache line write backs or store fences. Some tools also try to detect performance bugs caused by redundant usage of these primitives. Tools in this category target PM programming mistakes and have limited support for identifying higher-level logic bugs. Many of these tools

also require manual source code annotation.

Pmemcheck [121] is a Valgrind-based tool designed to find PM programming errors in applications built with PMDK [118]. Using Pmemcheck without PMDK requires manual annotation of source code. PMTest [183] and XFDetector [182] also require developers to manually annotate regions of interest. PMFuzz [181] is a fuzzer built on AFL++ that uses XFDetector and Pmemcheck to detect bugs.

Agamotto [210] is a symbolic execution tool built on KLEE for user-space PM applications. Agamotto does not require source code annotation, but finding bugs other than low-level PM programming errors requires developer-provided oracles. Witcher [82] is designed to test key-value stores and targets both PM programming errors and "persistence atomicity violations" by statically inferring which sequences of writes are intended to be atomic. PmDebugger [63] is a tool for collecting and analyzing PM access traces without source code annotation.

6.3 Lightweight methods for crash consistency

Recent work has shown increasing interest in applying ideas from programming languages and formal methods literature to check storage systems in a way that is more lightweight than full verification. These techniques leverage language features or ideas like model checking to obtain confidence in the correctness of a system, but like the typestate approach presented in Chapter 4, cannot prove full system correctness.

Rust compiler-based approaches. Corundum [113] is a Rust crate (i.e., library) for building crash-consistent PM applications that uses the compiler to enforce PM-specific safety properties. For example, Corundum ensures that objects stored on PM are only updated inside of transactions, and that durable objects do not contain references to volatile state. It enforces these rules mainly using smart pointers, wrappers around PM-resident objects, and traits that represent properties about each type (e.g., whether it can be stored safely on PM or modified in a Corundum transaction).

Corundum was a major source of inspiration for SquirrelFS (Chapter 4), and the safety properties enforced by the `pmcopy` crate in CapybaraKV (§5.4.1.1) are similar to those enforced by Corundum's `PSafe` trait. The main difference between `PSafe` and our `PmSafe` trait is that `PmSafe` is opt-in: developers must manually annotate objects to store on PM to make them `PmSafe`. Attempting to annotate a type that cannot be stored safely on PM results in a compiler error. Corundum's `PSafe` is opt-out; it is automatically applied to any user-defined type that does not contain any non-`PSafe` fields. It explicitly specifies some types (e.g., references, raw pointers, function pointers) as non-`PSafe`, with some limitations. If a developer introduces a new type that is not safe to store on PM but does not contain any preexisting non-`PSafe` types, they must manually indicate this.

Several experimental operating system kernels have used Rust to enforce low-level safety properties. RedLeaf [209] is a microkernel that relies on Rust primarily to enforce isolation between different domains. Theseus [22] is an OS composed of many small components that uses Rust to check invariants about resource management and fault recovery. It has been extended to obtain additional correctness guarantees from the Rust type system (e.g., via the typestate pattern) and by verifying some properties in Prusti [10]. Both systems use a single address space and single privilege level and rely on Rust, rather than hardware primitives, for isolation.

There has been some work applying ideas from programming languages literature to check additional properties about Rust programs. This work has not been applied to crash-consistency properties, but could potentially be used this way. For instance, Flux [170] adds the idea of refinement types to Rust to perform lightweight type-level verification of Rust programs. The set of values a refinement type can have is constrained by a logical expression, which may specify a concrete value (e.g., in Flux, `i32[0]` means an `i32` with value 0) or a set of legal values (e.g., `{v. i32[v] | v > 0 }` specifies the set of positive `i32`s). Reasoning about invariants encoded in refinement types can be automatically handled using a constraint solver and requires less programmer effort than verification in a language like Verus. Refinement types

could allow developers to encode and check deeper properties than those allowed by the Rust compiler's typestate support. For example, Flux supports a vector type refined by the vector's size, which could be useful in a system like SquirrelFS to handle operations on collections of unbounded size. Unlike the techniques used in SquirrelFS, Flux requires a compiler plugin.

Model checking. Model checking has also been used in prior work to check correctness properties about storage systems. For example, Alloy has been used to model check the design of a flash file system [136]. This work also included a detailed model of the behavior of a flash device including aspects like the limited number of write-erase cycles a cell can support. It did not include an implementation of the file system described by the model.

FiSC [271] and eXplode [270] use *in situ* model checking to check properties of storage system implementations (rather than of a separate abstract model of the systems). While this eliminates the possibility of discrepancies between the model and implementation hiding bugs, implementation-level model checking introduces additional challenges. FiSC requires a modified Linux kernel to be run as a user-level process. eXplode is simpler but still requires kernel modifications and a custom kernel module to record operations.

Ferrite [25] is a tool for specifying and checking crash-consistency behaviors of file system interfaces that uses both exhaustive state enumeration and symbolic model checking. It focuses on clarifying the crash behavior of POSIX-based storage systems to aid automated reasoning and the development of more robust applications. The authors also build a synthesizer to insert synchronization primitives required to achieve application-level crash-consistency, and use Dafny to prove crash-consistency of several small example programs Ferrite is aimed at modeling and exploring crash behaviors, not finding implementation-level crash consistency bugs.

ShardStore [24] is a production storage system at Amazon Web Services that has been checked for concurrency issues using model checking. The developers of

ShardStore also used executable reference models and property-based testing to gain confidence in the correctness and crash consistency properties of the system. ShardStore's model checking, reference models, and property-based testing are all done in Rust. This reduces the strength of guarantees they can obtain compared to full verification or more exhaustive model checking techniques, but keeps the full validation framework more accessible to engineers.

6.4 Verifying storage systems

In this section, we discuss related work on verified storage systems. This section expands on the discussion of crash-consistency verification techniques in §5.1 and discusses additional work.

FSCQ. FSCQ [40] and its successsor DFSCQ [39] are file systems written and verified in the Rocq (formerly Coq) proof assistant [226]. FSCQ introduced Crash Hoare logic (CHL), which was built in Rocq as a domain-specific language (DSL), to facilitate reasoning about potential crash states. CHL also provided a notion of logical address spaces to make reasoning about predicates over different types of resources (such as disk blocks or inodes) simpler. DFSCQ added support for `fsync` and `fdatasync`, which are required in block-based POSIX systems to achieve good performance but add significant complexity and are difficult to formally specify. The developers of DFSCQ proposed a "metadata-prefix" specification that defines what updates are made durable by invocations of these system calls and when the system is allowed to flush data and metadata updates in the background.

VeriBetrKV. VeriBetrKV [104] is a key-value store written and verified in Dafny [171]. It is based on the intuition that storage systems are similar to distributed systems and can be verified using the same techniques. It follows the approach introduced in IronFleet [107] in which Hoare logic is combined with state machine refinement to reason about asynchronous behaviors between potentially faulty components. VeriBetrKV uses a B^{ε}-tree design with journaling for crash consistency. Its proof starts

with an abstract specification (e.g., a map for the B^{ε}-tree) and establishes refinement relationships between progressively more concrete and complex views of the system. Individual components are specified and implemented assuming that no crashes will occur, and then are proven to match a crashing version of the specification. Along with building VeriBetrKV, the authors also modified Dafny to support a linear type system [174], which subsequently led to the development of Verus [164].

Perennial and atomic invariants. Perennial [31, 33] is a tool for verifying concurrent, crash-consistent storage systems. It is based on Iris [129], a framework for verifying concurrent systems using separation logic written in Rocq [226]. It has been used to build and verify several systems – a crash-safe mail server [31], a journal [32], and several networked file systems [34]. Perennial combines the idea of crash conditions from Crash Hoare logic with a concept called *capabilities* from Iris, which provide a way to both describe the current state of a resource (e.g., a disk block) and give permission for a thread to access it.

Perennial also extends Iris' support for atomic invariants to crash consistency. An Iris invariant is a capability that must hold at every point during execution. Invariants can be "opened" to use the internal capability to perform an atomic operation, which must re-establish and "close" the invariant [31]. For example, each lock in an Iris program is associated with a lock invariant, which can be opened and closed by acquiring and releasing the lock. Perennial extends its specification of locks to also include crash invariants, which must be part of the CHL crash condition of the critical section [31]. Verus [163] also supports atomic invariants, but most languages do not.

Yggdrasil. Yggdrasil [240] is a tool for automated verification of file systems. It applies the idea of push-button verification [211], in which developers provide a specification and implementation but no proofs, to crash consistency. Systems built with Yggdrasil are structured as stacked layers of abstraction, each of which with its own specification and implementation. Although Yggdrasil removes the need to write

proofs, it imposes other restrictions. It does not support reasoning over infinite domains, so all aspects of the system must either be bounded or validated using other techniques. It currently only supports building systems in a subset of Python, which limits performance.

BilbyFS. BilbyFS [5] is a file system for raw flash devices implemented and verified in Cogent [213]. Cogent is a functional programming language that can be verified in Isabelle/HOL and compiled to C. BilbyFS has two verified operations, `sync` and `iget`. The `sync` function synchronizes data buffered in-memory with durable data, and `iget` looks up a durable inode given an inode number. It is an in-kernel system that is compatible with the Linux Virtual File System (VFS) layer. BilbyFS is only verified in terms of functional correctness; it does not have proven crash-consistency guarantees.

Chapter 7: Discussion

This chapter includes additional discussion and comparison of the techniques presented in Chapter 3, Chapter 4, and Chapter 5. We describe the assumptions and guarantees provided by each approach (§7.1) and their applicability to other storage media (§7.2). We also discuss our experience learning how to build verified systems (§7.3).

7.1 Comparison of guarantees and assumptions

In this section, we review and compare the assumptions Chipmunk, SquirrelFS, and CapybaraKV are based on and the guarantees they provide modulo those assumptions. We also briefly describe the consequences if these assumptions are incorrect.

Verified systems make concrete guarantees based on a set of assumptions, e.g., that the model describing the system's interaction with the external environment is accurate. Other techniques provide weaker assurances of correctness and thus do not clearly state specific guarantees or assumptions upon which they are based. Nevertheless, we observe that this is a useful framework for comparing these approaches, so this section will examine the three systems presented in this dissertation by considering their (implicit or explicit) guarantees and assumptions.

7.1.1 Comparison of guarantees

We first describe the guarantees provided by each of the three approaches to crash consistency.

Chipmunk makes the weakest promises out of the three. Its main guarantee is that it does not report false positives; all bug reports describe true crash-consistency bugs that could happen in practice. It does not explore every possible crash state in

a given workload, and it cannot test every possible workload, so it cannot prove the absence of bugs.

SquirrelFS guarantees that updates to durable objects with typestates will always occur in a safe, crash-consistent order. Unlike prior soft updates systems, it does not guarantee that every post-crash state will immediately be useable with no recovery, as its atomic rename procedure may require cleanup to prevent future inconsistencies. However, since it does use typestate-restricted operations during this cleanup, it does guarantee that recovery is idempotent with respect to crashes. Chipmunk and SquirrelFS base their guarantees on different characteristics of crash-consistency bugs, so SquirrelFS' guarantees are not strictly stronger than Chipmunk's. Typestate checking in SquirrelFS can only rule out ordering-related bugs, whereas Chipmunk's bug-finding abilities are based on the user-visible consequences of each bug and not its root cause. There are bugs that may be detected via typestate checking but not by Chipmunk, or vice versa.

As a verified system, CapybaraKV guarantees the absence of all crash-consistency bugs. It also has verified functional correctness, which neither of the other systems provide; Chipmunk does not attempt to find non-crash-consistency bugs and SquirrelFS does not use typestate for volatile data structure management. CapybaraKV also has verified corruption detection and guarantees that the system will not erroneously report corruption after a crash. Chipmunk can (but is not guaranteed to) detect inconsistencies related to checksums after a crash, while SquirrelFS makes no attempt to detect corruption.

Chipmunk differs from the other two systems in the sense that it primarily tries to avoid reporting a crash-consistency issue when one does not exist. It may, however, fail to report real bugs. SquirrelFS and CapybaraKV, on the other hand, may reject a correct implementation, but guarantee that they will never admit a buggy one. We model checked SquirrelFS to ensure that its ordering rules did not allow inconsistent crash states, but we did not check that all consistent states were

possible. There are also some cases in SquirrelFS where a particular transition function depends on a single typestate when it could safely depend on several others because it was simpler to implement and did not impact performance. The potential rejection of valid implementations in CapybaraKV stems from the behavior of the underlying verifier. We discuss why attempting to verify a correct implementation may fail in §7.3.

7.1.2 Comparison of assumptions

All three techniques presented in this dissertation are based on a model of the semantics and behavior of the underlying storage device and assume that this model is correct. The models we use are based on documentation from Intel on their Optane DC Persistent Memory Module and resources from the Storage Networking Industry Association (SNIA) [9, 229, 258]. It is expected that PM over CXL will adhere to this same model [230], and we expect other future PM offerings to follow a similar (if not the same) model. Future hardware may support non-volatile CPU caches (as in the previously proposed eADR [122]). The current model is weaker than the expected model for a system with non-volatile caches. It admits some durable update orderings that may not happen with non-volatile caches, but there are no orderings that can happen with non-volatile but not volatile caches. Therefore, we believe this model is reasonable for both current and future PM systems.

Unlike the other systems we present, Chipmunk does not guarantee that it will comprehensively find or prevent any set of bugs. Thus, Chipmunk makes relatively few additional assumptions. Since it uses each file system's regular-case execution as an oracle during consistency checking, its primary assumption is that the system states reachable via regular system calls are in fact legal crash states.

SquirrelFS assumes that the order of durable operations encoded in its typestates is correct and cannot result in an inconsistent crash state. We improved our confidence in this by model checking the design in Alloy, but this does not con-

stitute a full proof of correctness. The correctness of the model checking depends on the correctness of Alloy itself and of the invariants used when checking. Discrepancies between the Rust implementation and the Alloy model could also introduce crash consistency bugs. We also assume that the implementation of each typestate transition function is correct and that the Rust compiler itself is correct.

CapybaraKV assumes that its high-level specification and its formal specification of PM are correct, including the specification of legal crash states. Like SquirrelFS, it also assumes that the underlying tools (in this case, Verus, Z3, and the Rust compiler) are correct. Several functions generated by the `PmCopy` macro rely on hard-coded primitive type size and alignment values, and we assume that the algorithm we use to calculate the layout of `#[repr(C)]` types is matches the one used by Rust. `PmCopy` generates static assertions to validate these assumptions, but this does not prove that they are correct. Due to current limitations in Verus, we cannot prove that the ghost and executable versions of several `PmCopy`-generated functions are equivalent, so the current implementation assumes this.

7.1.3 Impact of incorrect assumptions

The consequences of violated assumptions grow with stronger guarantees. Violating Chipmunk's assumptions could result in either false positives, where a report is generated for a bug that cannot happen, or false negatives in which bugs are not detected. False positives are more problematic, as Chipmunk does not guarantee the absence of false negatives even when its assumptions are met. Developers often prefer testing tools that do not generate false positives, as establishing whether each bug is real or not can take significant effort. However, even if they occurred due to violated assumptions, the overall impact of false positives from Chipmunk is relatively small. Unlike in SquirrelFS and CapybaraKV, such an issue does not have the ability to directly and immediately impact the correctness and crash-consistency properties of the target storage system.

Violating the assumptions of SquirrelFS could call into question the ordering guarantees it provides. For example, incorrect ordering rules that are too lax could allow an implementation that can produce inconsistent crash states. Note that this does not guarantee that the implementation will be buggy; it just means that if the implementation has a bug related to the incorrect rules, the bug will not be caught at compile time. If SquirrelFS' storage model is incorrect, its persistence typestates may not be accurate, but the operational typestates may still be able to prevent some higher-level ordering-related bugs.

Out of the three systems, CapybaraKV makes the strongest guarantees and has the most to lose if its assumptions are violated. As we discuss in §7.3, program verification aims to reduce the amount of code that a developer has to trust to believe the system is correct as a whole. In practice, for instance, this means that a developer who wishes to confirm the correctness of a system only needs to inspect its spec, not test or read the whole codebase. This provides very strong guarantees when assumptions are met, but those guarantees may be lost completely if the assumptions are violated. In the worst case, unsoundness introduced by a verification tool or user-provided specification may jeopardize the validity of the entire proof.

7.2 Applicability to other storage media

In this section, we discuss how the techniques presented in this dissertation may be applicable to block-addressable storage media.

Chipmunk. The techniques and contributions from Chipmunk are generally not applicable to block-based storage media. Chipmunk's function-based instrumentation is useful on PM systems, but is more complex than instrumentation approaches used in prior file system testing work. Current block-based media is too slow for testing tools to take advantage of other insights from Chipmunk, such as the observation that there are usually relatively few durable updates between each pair of ordering points. The most likely pattern to be an interesting line of research is that corruption-

detecting mechanisms can introduce new crash-consistency bugs; to our knowledge, this interaction has not been investigated in traditional systems.

SquirrelFS. Many fundamental ideas underlying SquirrelFS originally came from research on block-based storage systems and are not limited to byte-addressable domains. The order of durable updates is critical for crash consistency on any type of storage media, and soft updates was originally developed for use in block-based file systems. Synchronous Soft Updates would not be a good crash-consistency mechanism with current block devices because they are not fast enough to support synchronous operations. The typestate pattern could, however, be used to statically enforce durable update ordering rules in other crash-consistency mechanisms. Approaches like copy-on-write and journaling do not derive consistency entirely from ordering, so additional techniques would be needed to ensure their robustness. It would be more difficult to use typestate-checked soft updates in an asynchronous setting than a synchronous one, since the lifetime of a pending durable update would no longer neatly correspond to the scope of a system call, but we are not aware of any fundamental limitations that would prevent this with more engineering effort.

PoWER. The fundamental idea behind PoWER — that the set of crash states introduced by a write can be stated and reasoned about prior to that write — is not specific to byte-addressable storage. Our model of storage in CapybaraKV assumes that 8-byte aligned updates are atomic with respect to crashes; this could easily be modified to a different size (e.g., 512B or 4KiB) used by block-based storage devices. The structure of proofs in such a system would be different from CapybaraKV. In particular, both systems would support some degree of asynchrony, but whereas durable updates in CapybaraKV are issued to the storage device immediately, updates in a block-based system may first be reflected in a volatile cache and flushed and ordered separately. Other techniques used in CapybaraKV, such as the corruption-detecting Boolean (CDB), are designed for byte-addressable storage and would not provide as much benefit to a block-based system, but are not incompatible with other

storage types. Our general model of data corruption is compatible with any type of storage device and any data layout and could be useful in future verified systems for both byte- and block-addressable storage.

7.3 Experiences with systems verification

In this section, we discuss our experiences learning how to verify systems by building CapybaraKV. We focus on the observation that there are key differences between how systems experts approach development and best practices when building large verified systems, and describe what we learned about bridging this gap. These observations are based on my own experiences learning verification and on discussing verified code with other researchers and students from a systems background. This section draws primarily from experiences with Verus, but we believe that they may be more broadly applicable to other verification tools as well.

7.3.1 A mental model of verification performance and failures

We begin this section by building some intuition about several key aspects of verification tools like Verus. This section does not attempt to explain in detail how verification tools work. Rather, we focus on building intuition about two factors — performance and verification failures — that we argue are crucial when building a large verified system. First, the structure of a specification or proof has an impact on verification time, and slow proofs can significantly impact development speed. Understanding these factors and designing systems to avoid verification performance pitfalls is especially important for large systems with many proofs to check. Second, just because a proof is correct does not mean it will pass verification. It can be frustrating and time-consuming to debug such failures, but avoiding them requires some understanding of how the verification tool works.

The main aspect of a verification tool that we argue systems programmers need to consider when it comes to performance and proof failures is its *proof context*. The

proof context is the set of facts that are known at a given time during verification. We discuss the concept of a proof context abstractly here as a tool to understand better how verification works, but most verifiers do have some concrete notion of a proof context. Some tools, particularly proof assistants like Rocq, make the contents of this context explicit to aid the interactive theorem proving process. Verification-aware programming languages like Verus and Dafny do not show the developer the contents of the proof context. In order to have a higher degree of automation, these languages usually have a larger and more complex proof context that is not used directly by users to help write proofs. Since this section focuses primarily on experiences with Verus, we will focus on this latter category of verifiers.

Even though the exact contents of the proof context are not made explicit to the developer in many verification languages, it is still a useful lens for understanding how verification works. The process of verification can be thought of as a search for a proof of each property in the system. In SMT-based verifiers like Verus, the verifier builds logical formulas called *verification conditions*, and queries the SMT solver to determine if they are satisfiable. These queries are structured such that an `unsat` response implies that an assertion or pre/postcondition always holds, so the SMT solver is searching for a proof that no satisfying assignment exists. The proof context defines the search space, so the size and contents of the proof context directly determine what can be proven and how long it takes. A larger proof context results in a slower search, but a proof context that is missing key information can prevent a valid proof from being found. Verus (and most other tools mentioned in this dissertation) use modular verification, in which each function is evaluated separately with its own proof context. Verifying a program is thus a balancing act between ensuring that there is sufficient information to enable a successful search, without including extraneous facts that will slow the process down.

To make this mental model more concrete, we now discuss an example in which an obviously correct proof fails due to missing information in the proof context. Later in this section, we will discuss examples from CapybaraKV where the way a

```
mod M1 {
  pub struct Foo {
    val: u64
  }
  impl Foo {
    pub closed spec fn le(self, other: Foo) -> bool {
      self.val < other.val
    }
  }
  proof fn lemma_foo_le_transitive_success(a: Foo, b: Foo, c: Foo)
    requires
      a.le(b),
      b.le(c),
    ensures
      a.le(c)
  {}
}
mod M2 {
  proof fn lemma_foo_le_transitive_fail(a: Foo, b: Foo, c: Foo)
    requires
      a.le(b),
      b.le(c),
    ensures
      a.le(c)
      ^^^^^^^ failed this postcondition
  {}
}
```

Figure 7.1: **Closed `spec` function.** The listing shows how identical `proof` functions in different modules may fail or succeed to verify depending on the visibility of a `spec` function body.

specification was expressed had a significant impact on proof performance. Consider the Verus pseudocode in Figure 7.1. In Verus, one way a developer can improve proof times is to *close* `spec` functions, as on line 6 in the figure. The body of a closed function is hidden outside its own module (i.e., it is visible in `M1` but not `M2`). The function may still be called in other modules if it is public, but its meaning is not added to the proof context in those modules.

Figure 7.1 contains two `proof` functions with identical pre- and postconditions, one in `M1` and one in `M2`. Verus can automatically verify the function in `M1` but not its counterpart in `M2` because doing so requires knowledge that the `le` method invokes the less-than operator. This results in a potentially surprising verification failure that is caused by the structure of the code, not an incorrect proof or implementation. Keeping `spec fn` definitions open and globally visible can thus make proofs easier, but it can also slow verification down by adding information to the proof context of other functions.

In the rest of this section, we discuss observations and challenges we encountered both when learning how to build verified systems and when discussing the process of verification with other systems experts. We will continue to use the mental model described here to build intuition about challenges with verification performance or getting correct proofs to succeed. For more detailed information on how Verus works, see the official Verus guide's section on "Understanding the Prover" [256].

7.3.2 Restrictions simplify verification, but impact system design

Verification of large, complicated systems has been made tractable by addressing or eliminating sources of computational complexity for the underlying solver. Many techniques used in current systems verification stem from this goal. For example, modular verification, in which functions are checked individually with minimal proof contexts, reduces the size of solver queries to keep verification time low [190]. Since handling these queries generally involves solving NP-hard or undecidable prob-

lems, keeping them simple and small is crucial.

Another major source of complexity in verification is handling shared state. For instance, verifying concurrent programs is an active area of research that requires specialized techniques and frameworks [30, 103, 129]. Managing heap-resident values is also challenging, in part due to the restrictions imposed by modular verification. This is hard even in single-threaded programs because if there are multiple mutable aliases to a single value of the heap, it is difficult to keep track of how that value changes over time [23]. In fact, alias analysis is undecidable or uncomputable in general [160]. Many proof assistants use functional languages in which all values are immutable [203, 226, 245], which solves this issue but impacts runtime performance of the resulting system. Verus handles this by using Rust's strict type system, which prohibits multiple mutable references to individual values [164]. Dafny requires developers to explicitly specify what heap values a function accesses [55].

Restrictions on how systems may be built can thus simplify verification, but also may impact how systems are built and introduce confusing or surprising challenges for verification newcomers. This same phenomenon can also occur outside of verification; for example, consider the impact of Rust's type system on how programs are written. Rust is widely viewed as having a steep learning curve in large part due to its type system, which provides safety guarantees but can also increase the complexity of relatively simple tasks like implementing a doubly-linked list. This same effect is compounded in formal verification, in which developers can only implement what they can specify and prove and properties for which informal reasoning is straightforward may be surprisingly difficult to write proofs about.

The fact that verifiers trade completeness for soundness can also impact and restrict system designs in unexpected ways. A true statement may fail to verify, for a variety of reasons. For instance, it may require solving an undecidable problem, or the necessary information is not present in the proof context, or the query is too large to solve in a reasonable amount of time. We will further discuss differences between

how systems and verification experts view soundness and correctness in §7.3.3. The fact that the verifier cannot prove all true statements can also impose restrictions on developers that require them to change how they write the system or proofs.

During the development of CapybaraKV, there were several times when we were unable to prove a particular design or abstraction, even though we believed it to be correct. We now describe a few examples of this.

PM management abstractions. CapybaraKV is made up of several durable components that each require a (logically) contiguous region of PM but need not all be stored together. One natural approach would be to use a PM abstraction that stores each component in a logically-separate region. For example, each component could each be stored in a separate PM-resident file. Note that such an abstraction does not preclude informal global reasoning about the entire PM device. In particular, a potential performance optimization is to use a single `sfence` to order writes to *all* the files. If we have independent updates to multiple separate components that must all become durable before a subsequent operation (e.g., updating CapybaraKV's key and item tables before updating the journal), we can safely use just one `sfence` to enforce this ordering.

We initially tried to build CapybaraKV using this approach and optimizations to share `sfence` invocations between components. However, we quickly found that formalizing this global reasoning was very challenging. There was no straightforward way to invoke an `sfence` in one component and use the fact that it had been invoked in proofs elsewhere. A method of one component that uses `sfence` can indicate in its postcondition that it has no outstanding writes, but it cannot use this to update the state of other components. This is due to the modular approach used by Verus; to keep proofs fast, no function has a full view of global system state. However, developers can and do use this information in informal reasoning about the system's design and implementation. This challenge is similar to the aliasing problem. Here, there are no explicit mutable aliases to a single value, but a key part of system state

(the durability of recent writes) can be changed by one component in a way that affects others.

We tried writing proofs that could be invoked after an `sfence` to establish that prior updates could now be considered durable. This was difficult to do soundly because we needed to ensure that an `sfence` made at time t_n could not be used to prove that data written at t_{n+1} was durable. We could have used techniques from distributed systems (e.g., logical clocks) to establish an order of writes and `sfence` instructions, but this would make proofs more much more complicated and only bring a minor performance improvement. We decided instead to use a single memory-mapped file on PM to store all components. CapybaraKV keeps track of the location of each component in the file, and each component has the ability to update the durability-related ghost state of the entire file.

Corruption-detecting Boolean. The corruption-detecting Boolean primitive we propose in Chapter 5 was influenced by challenges we encountered when introducing corruption detection to our verified log, which is now used in CapybaraKV. We initially attempted a design similar to the Tick-Tock algorithm used in NOVA-Fortis [267] for crash-consistent CRC management on PM. However, we found it challenging to verify on several fronts. First, we struggled to prove it correct with the corruption-related axioms we were using. Our axioms required that the developer know where the correct CRC for a given chunk of data be stored in order to prove that the data was uncorrupted. This is a reasonable restriction; if a developer is not required to use the stored CRC for a particular region of data, they could make up a CRC for use in the proof, which would not be correct. However, after a crash, we do not know which of the two Tick-Tock-maintained CRCs associated with a piece of data is expected to be correct.

Second, even if we had managed to prove the algorithm correct (perhaps by using different axioms), we found that Tick-Tock was not sufficient under our model of corruption. The current model used in CapybaraKV relies on the fact some

number c of bit flips will not cause CRC collisions [152] and assumes that no more than c bits will be corrupted. Tick-Tock uses CRCs both to detect corruption and also to determine which version of a data structure is valid; the two versions may be arbitrarily different, which violates our assumption.

Tick-Tock is a reasonable algorithm under the standard assumptions that corruption will not cause CRC collisions and that the two versions of a given data structure will not have the same CRC. It also has built-in redundancy to recover from corruption in many cases, which the CDB-based approach does not. However, when formalizing our model of corruption and attempting to prove our implementation correct, the restrictions imposed by verification forced us to look for a different approach. This is not necessarily a bad thing; we found the CDB very useful when developing CapybaraKV, and we likely would not have come up with it if we had not encountered these obstacles. The additional restrictions and challenges imposed by verification can lead to the development of new techniques and ideas that may not have been considered otherwise but could have broader utility.

7.3.3 Soundness is crucial and relies on a small but critical set of assumptions

Verification tools strive to be *sound* and ensure that a valid proof will never imply that false is equivalent to true. Since anything follows from a contradiction, if a developer can accidentally (or maliciously) prove false, they can make the verifier admit dangerous or incorrect code and endanger the utility of verification as a whole. The axioms that are part of, e.g., Verus's standard library are carefully considered to ensure they are sound, and unsoundness bugs are taken very seriously by developers.

It is also possible for developers of verified code to accidentally introduce soundness issues. Verified programs include trusted specifications about how the system interacts with the external, unverified environment it executes in. In CapybaraKV, this includes assumptions about, e.g., the behavior of persistent memory in the event of a crash and the conditions under which we can prove the absence of cor-

```
impl Drop for X {
  fn drop(&mut self)
    requires false,
  {
    // do something dangerous...
  }
}
```

Figure 7.2: `Drop` **trait unsoundness bug.** The listing shows an implementation of Rust's `Drop` trait in Verus with a precondition that introduces unsoundness.

ruption. If these assumptions are not always correct, then proofs that rely upon them are at risk of unsoundness. There are also no verifier-imposed restrictions on writing new axioms or unverified functions, so developers are responsible for the correctness of these additions.

Thus, in verified systems, soundness is not optional. Maintainers of verification tools often prioritize potential soundness issues, even if exploiting them requires a convoluted set of steps or rare external conditions. We observe that this differs from how many systems developers think about correctness. In many systems domains, how problematic but rare events are handled is one dimension in a much larger trade-off space, and safety guarantees are not always the top priority. For example, most production file systems accept the possibility that some data may be lost or corrupted in the event of a crash in order to improve regular-case performance. As another example, most low-level systems are developed in non-memory-safe languages like C, which increases the likelihood of issues like security vulnerabilities and serious bugs in order to gain better performance. These trade-offs are a standard part of systems development, and developers are used to weighing correctness considerations against their impact on the performance and complexity of the system.

To compare these two lines of thinking, consider the code snippet in Figure 7.2, which is unsound; a developer could write anything in the body of `drop`, and verification would still succeed. This code is referenced in a (now-fixed) Verus GitHub

issue [102] which noted that it incorrectly passed verification. The root cause of this issue is that calls to `drop` are inserted automatically by the Rust compiler whenever an object goes out of scope, and it is rarely called directly by developers. Verus would not allow explicit calls to this `drop` implementation because its precondition cannot be met, but automatically-added `drop` call sites are not checked during verification. This issue was fixed within several days by preventing developers from including preconditions on most `drop` implementations. On one hand, this is a nontrivial problem; it gives developers an easy way to introduce unsoundness that may then be exploited. On the other hand, this code is obviously wrong, probably malicious, and would most likely be caught immediately in a code review.

In our experience, verification experts tend to prioritize fixing these bugs and try to strengthen their systems by seeking out ways that an adversary could exploit newly-added assumptions. Verification newcomers with systems experience tend to be confused about why this is necessary at all. As a concrete example, when building CapybaraKV, the axiom we used for CRC reasoning went through multiple revisions because it was difficult to ensure soundness. Our initial model of corruption simply assumed that a CRC collision would not cause stored data to appear uncorrupted. We were willing to accept that our proofs would be compromised if that did happen (although we ultimately strengthened the model and our assumption to make them more realistic). However, we spent several weeks working on this axiom trying to ensure that an adversarial developer could not prove false by manufacturing a collision. Some potential "attacks" we considered were relatively straightforward; for example, the axiom should not allow a developer to prove the absence of corruption using a CRC they did not read from storage. We also considered more complicated cases, such as where the attacker could wait for a specific corruption event that did not violate our main assumption, use it to construct a CRC collision, and prove false. Systems developers (including, initially, the author of this dissertation) involved with these conversations were generally confused about why preventing this type of thing was worth so much effort.

Verification experts and tool maintainers prioritize this type of issue because a key goal of verification is to reduce the amount of code that developers must manually reason about to trust the correctness of the system. If assuring the correctness of a verified system requires manually checking all of its code to ensure it does not attempt (accidentally or intentionally) to exploit unsoundness, the fully utility of verification has not been realized.

These observations have several interesting implications. First, in our experience, it is not uncommon for those unfamiliar with verification to assume that a verified system is guaranteed to be completely correct. However, as we have just discussed, these guarantees are based on the assumption of soundness of the underlying verification tool and developer-provided specifications. The latter is particularly important, because it means that the developers of a verified system have a responsibility to carefully ensure not just that they are proving the correct properties, but also that they have not introduced potentially-exploitable issues. Verification newcomers are likely aware that the correctness of their system depends on its specification, but in our experience it seems less widely known that there are other avenues via which correctness can be jeopardized.

Second, we believe that it would be interesting and useful to further investigate static analysis techniques that provide partial correctness guarantees, such as those used in SquirrelFS. The reaction of systems experts to this aspect of verification indicates that at least some subset of the community may be more interested with simply improving confidence in their systems by some amount than in fully verifying them. There has been recent interest from industry practitioners in more lightweight techniques [24, 158, 212] as well, since these techniques require less specialized knowledge and time investment than full verification. Defining the exact guarantees and limitations of different techniques and how these characteristics concretely impact system design and development would be a valuable contribution.

Alternatively, a potentially more controversial approach could be to apply

verification tools in a different, less-complete way to achieve different goals. It may be more useful in some cases to use verification not to eliminate the need to trust most of a codebase, but rather to help a developer develop a more formal specification of their system and ensure that their implementation meets some less-complete correctness criteria. Again, establishing how to express what such an approach guarantees would be crucial so that developers can use the right mix of dynamic and static methods to check the correctness of their system.

7.3.4 Understanding quantifiers and triggers is key to writing good verified code

Quantifiers and triggers are one of the more challenging aspects of Verus for newcomers to understand, but they are arguably also one of the most important. How they are used both impacts whether code verifies successfully (even if it is correct) and how long verification takes. The key challenge stems from the fact that solving logical formulas including universal and/or existential quantifiers is undecidable. However, quantified expressions are extremely common, so understanding how they impact verification results and performance can help developers write better proofs.

We first explain how verifiers handle quantified expressions. Solvers like Z3 handle quantified expressions using pattern matching [198]. In Verus and Dafny, the patterns to match on are specified using annotations called triggers. When the pattern is used, an instance of the quantified expression is instantiated in the proof context. This prevents the verifier from needing to somehow handle an infinite set of facts established by the quantified expression, and also ensures that only facts likely to be needed for the proof are included in the proof context.

For example, consider Figure 7.3, which contains a code listing borrowed from the Verus documentation [257]. There are two proof functions (lines 4 and 11), both of which have preconditions requiring that all elements of a sequence `s` are even. In both, `is_even(s[i])` is chosen as a trigger. In `test_use_forall`, the quantifier is instantiated with the use of `is_even(s[3])` on line 9. In our mental model, this in-

```
1  spec fn is_even(i: int) -> bool {
2    i % 2 == 0
3  }
4  proof fn test_use_forall(s: Seq<int>)
5    requires
6      5 <= s.len(),
7      forall|i: int| 0 <= i < s.len() ==> #[trigger] is_even(s[i]),
8  {
9    assert(is_even(s[3]));
10 }
11 proof fn test_use_forall_fail(s: Seq<int>)
12   requires
13     5 <= s.len(),
14     forall|i: int| 0 <= i < s.len() ==> #[trigger] is_even(s[i]),
15 {
16   assert(s[3] % 2 == 0);
17            ^^^^^^^^^^^^^^ assertion failed
18 }
```

Figure 7.3: **Trigger example.** The listing shows how the choice of trigger in a `forall` statement can impact whether a correct assertion passes verification.

stantiation adds the fact `0 <= 3 < s.len() ==> is_even(s[3]) == 3` to the proof context. This fact is known to be true from the precondition, and it can be used to prove that the assertion on line 9 holds.

In `test_use_forall_fail`, the assertion on line 16 is logically equivalent to the one on line 9, but it does not use the trigger pattern. Thus, `0 <= 3 < s.len() ==> is_even(s[3]) == 3` is not added to the proof context (even though it is true), so Verus does not have a way to prove that the assertion holds.

In this example, `is_even(s[i])` is not the only valid trigger; `s[i]` could also be selected, and would make both assertions pass since they both use the pattern `s[i]`. In this way, the choice of trigger can impact the result of verification. However, more general triggers will result in more instantiations, which can have a significant impact on performance in large systems. Since each instantiation increases the size of the proof context, many instantiations will make verification slow. The verifier cannot

```
spec fn spec_no_duplicates(list: Seq<u64>) -> bool {
  forall |i: int, j: int| {
    &&& 0 <= i < list.len()
    &&& 0 <= j < list.len()
    &&& i != j
  } ==> list[i] != list[j]
}
```

Figure 7.4: **No duplicates `spec` function.** The listing shows a `spec` function that returns whether a sequence contains any duplicate elements.

tell *a priori* which instantiated facts will be necessary for the current proof, so it must instantiate each quantifier every time the trigger pattern is matched, even if the resulting information is not helpful.

Quantifier instantiations are a common source of verification performance problems, and Verus includes a built-in profiler to detect problematic triggers. However, as systems grow larger, it becomes more and more difficult to choose the right triggers and keep track of the patterns to use in new proofs. More general triggers will cause performance problems, but more restrictive triggers can lead to frustrating proof failures that are hard to debug.

As a concrete example in CapybaraKV, several system components maintain volatile free lists as Rust `Vec`s to help allocate persistent memory. One important free list invariant is that the list does not contain any duplicates. We initially specified this property as shown in Figure 7.4. The quantifier must have a trigger for both `i` and `j`, and the only patterns that can function as valid triggers are `list[i]` and `list[j]` (in Verus and Dafny, arithmetic and (in)equality expressions cannot be triggers). However, as this is part of an invariant that is included as a pre/postcondition in many functions, the quantifier will be instantiated each time any index in `list` is accessed. In CapybaraKV, the impact of all of these instantiations on performance quickly became apparent. We discuss how we overcame this particular performance issue in §7.3.5.

We encountered many expensive quantifier instantiations during the development of CapybaraKV, and there is no one-size-fits-all solution to fix them. These issues arose in large part because the author of this dissertation did not fully understand the impact that quantifier usage and trigger choice would have on the rest of the system until those impacts were realized. We thus believe that obtaining a good understanding of quantifiers and triggers is important for those learning verification, and that designing systems and abstractions with this in mind will result in cleaner code that verifies faster and is easier to write proofs for.

7.3.5 Proofs and specifications are not executable code

Verification of imperative programs blends ideas familiar to systems programmers with concepts from functional programming and automated logical reasoning. We observe that one sticking point for some verification newcomers is understanding how these newer aspects differ from the code they are used to writing. One potential cause of this confusion is that many parts of a program written in, e.g., Verus, are syntactically very similar to executable code but are expressing purely mathematical properties. Understanding this is crucial for nearly all aspects of writing a verified system. We now describe several examples from the development of CapybaraKV where this idea came into play.

Proofs can take advantage of theoretical properties that programs can't. In our experience, a common pattern among verification newcomers is that they approach writing proofs and specifications as if they were imperative code. This is often fine, but we have also seen that approaching ghost code as if it will be executed can leave simplifications and verification performance optimizations on the table.

Consider the `spec_no_duplicates` function shown in Figure 7.4. It looks similar to an executable function a developer might write to check whether an arbitrary list contains duplicate values. The main difference is the `forall` statement, but we could replace this with loops over the elements of `list` to make an equivalent impera-

```
spec fn is_reverse_mapping(s: Seq<u64>, f: spec_fn(u64) -> int)
    -> bool
{
  forall |i: int| 0 <= i < s.len() ==> i == f(s[i])
}
```

Figure 7.5: **Reverse mapping definition.** The listing shows a function defining a reverse mapping from a sequence's elements to their indices. The type `spec_fn(u64) -> int` represents a function from `u64` to `int`.

tive function. Such an implementation is a standard way to check a list for duplicates if no other information is known about it (e.g., whether it is sorted). However, as discussed in §7.3.4, this function is not ideal as a specification because it can result in frequent trigger instantiation and impact verification performance.

An alternative approach, proposed to us by Chris Hawblitzel, is to specify what it means for a list to be free of duplicates using the mathematical definition of a function. This is a different definition of *function* than what programmers generally use. We normally think of functions as callable units of code, but Verus can also express functions in the mathematical sense of a mapping that assigns each member of a set X to exactly one member of a set Y. In this case, we can take advantage of this to write a no-duplicates specification that has a much lower verification performance impact.

We can write an improved specification as follows. Consider the function shown in Figure 7.5, which defines whether a function `f` is a *reverse mapping* function for a sequence `s` of `u64`s. Essentially, `f` is a reverse mapping if it maps each element of `s` to its index in `s`. Because `f` is a function in the mathematical sense, a particular sequence `s` only has a reverse mapping if each element maps to exactly one index. Therefore, `s` contains no duplicate elements iff a reverse mapping function exists for `s`. To prove that a sequence contains no duplicates, we can keep a corresponding reverse mapping function in ghost state, maintaining as an invariant that it is a valid reverse mapping, as shown in Figure 7.6. It's not strictly necessary to maintain the

```
struct FreeList {
  contents: Vec<u64>,
  reverse_mapping: Ghost<spec_fn(u64) -> int>,
}
impl FreeList {
  spec fn invariant(self) -> bool {
    is_reverse_mapping(self.contents@, self.reverse_mapping@) && ...
  }
}
```

Figure 7.6: **Specification with reverse mapping.** The listing shows a structure that stores a free list in a vector. It also maintains, as a field, a ghost reverse mapping function to help prove that the free list has no duplicates. A `spec` function defining an invariant for the free list specifies that the reverse mapping must match the concrete free list. The `@` symbols take the view of the `contents` vector and unwrap the `Ghost` type, respectively.

ghost reverse mapping function (it is possible to establish the existence of a reverse mapping, which is sufficient, without it), but it simplifies proofs and introduces no runtime overhead. If we still want to use the definition of `spec_no_duplicates` in some places, it is trivial to prove that the existence of the reverse mapping implies there are no duplicates, and relatively easy to prove the converse. This was not a total panacea; we still found it easier to write proofs using `spec_no_duplicates` in some places, and we made some additional optimizations to speed up proofs about this property. Still, this example illustrates one way in which engaging with the practice of writing ghost code as writing mathematical expressions, rather than translating imperative code into something the verifier can reason about, can be a beneficial way to approach verification.

Reasoning about sequences. Verus' standard library includes several types for use in specifications, including sequences (`Seq`), maps (`Map`), and mathematical sets (`Set`). We use sequences of bytes to model the contents of persistent memory in CapybaraKV because they are a natural abstraction for array-structured objects. Sequences are widely used in Verus, so it has good support for specifications and proofs

```
struct MySeq {
  s: Seq<int>
}
proof fn test_proof() {
  let s1 = MySeq { s: Seq::new(4, |i: int| i * 2) };
  let s2 = MySeq {
    s: Seq::empty().push(0).push(2).push(4).push(6)
  };
  assert(s1 == s2);
           ^^^^^^^^ assertion failed
}
```

Figure 7.7: **Sequence equality.** The listing shows a `proof` function that fails to verify that two identical sequences are equivalent.

about them. Despite this, the way that Verus handles reasoning about sequences is often unintuitive and does not match how systems developers may informally reason about them.

For example, consider the Verus code in Figure 7.7. We define a type, `MySeq`, that is a wrapper around the Verus standard library type `Seq` and stores a sequence of integers. We then create two instances of `MySeq` in two different ways: `s1` uses the `Seq` constructor to create a list of four elements in which the value at index `i` is `i * 2`, and `s2` is created by pushing values onto an empty `Seq`. It is easy to see that `s1 == s2`, but Verus fails to prove this. Verus uses *extensional equality* in determining if two objects are equal; for `Seq`s, this means it considers two sequences equal if they have the same elements. However, Verus will not always automatically check that two objects are extensionally equal, because a simple basic equality check (which here would be checking if the two `Seq`s were constructed in the same way) is faster. This is why the code in the listing does not verify: the two `MySeq` instances are not checked for extensional equality and are not obviously equivalent by definition. At the time of writing, Verus does automatically check extensional equality for `Seq` types, but not for user-defined types containing `Seq` such as `MySeq`.

```
spec fn spec_padding_needed(offset: nat, align: nat) -> nat
{
  let misalignment = offset % align;
  if misalignment > 0 {
    (align - misalignment) as nat
  } else {
    0
  }
}
```

Figure 7.8: **Memory layout padding function.** The listing shows a `spec` function that specifies how to calculate the padding needed for a field in a `#[repr(C)]` structure.

Nonlinear arithmetic. Linear arithmetic encompasses basic mathematical expressions including operations involving constants; for example, $x * 3 \leq 15$. Integer linear arithmetic is NP-hard and supported by SMT solvers like Z3. Nonlinear arithmetic covers expressions in which variables are multiplied or divided (e.g., $x * y \geq 15$), and is undecidable [197].

Unfortunately, nonlinear arithmetic is common in systems programming. For example, CapybaraKV's durable circular log uses modular arithmetic (which is nonlinear if the modulus is not constant) to calculate addresses when log contents wrap around. We encountered challenges trying to efficiently verify parts of CapybaraKV that involved modular arithmetic. Verus supports several optional solvers for handling nonlinear arithmetic, one that handles a decidable subset and Z3's built-in solver that can often (but not always) solve general nonlinear expressions. We primarily used the latter, as support for the former is less mature and comes with some restrictions (e.g., no support for inequalities or division). However, because this approach attempts to solve an undecidable problem, code dealing with modular arithmetic was often more difficult to prove and took longer to verify.

One particular example of this had to do with how our `PmCopy` macro is used to calculate the memory layout of each durable object. For each type that derives

`PmCopy`, the macro generates code that uses the `spec_padding_needed` function shown in Figure 7.8 to determine how much padding (if any) is needed between each field. This function is based on the algorithm used by Rust to determine the layout of `#[repr(C)]` types [231]. Because the value of `align` is not constant, line 4 involves nonlinear arithmetic. Many proofs rely on the result of this calculation, so we initially kept the body of this function open and visible. For example, crash-consistency proofs in CapybaraKV often require proving that updating a particular durable data structure does not overwrite part of an adjacent structure, which involves reasoning about the size of the update. However, we found that making this specification globally visible caused significant verification slowdown. To demonstrate this, measured how long CapybaraKV takes to verify with the current version of Verus at the time of writing on a ThinkPad laptop. The version of CapybaraKV described in Chapter 5, which hides the definition of `spec_padding_needed`, verifies in about 60 seconds with 1 thread. After making the definition of `spec_padding_needed` open, verification of many functions initially times out. After increasing timeouts to allow for verification to complete, it takes about 120 seconds with 1 thread.

7.3.6 Summary

This section provides an overview of some of our experiences learning Verus. We initially build intuition using a mental model of verification based on a set of known facts forming a proof context, and we use this to describe challenges and lessons that we learned about verification when building CapybaraKV. A common thread in many of these lessons is that verification impacts how a system is built beyond just requiring the developer to write additional code for proofs and specifications; it also requires developers to think about trade-offs and what it means to be correct in new ways. These lessons may be unsurprising to verification experts, but we believe that they may be useful to engineers or researchers with a background in systems who are interested in learning verification.

7.4 Summary

In this chapter, we first describe the guarantees and assumptions made by Chipmunk, SquirrelFS, and CapybaraKV. These are standard aspects of verified systems to discuss explicitly, and although not all of these systems are verified, we find this to be a useful framework for comparing them. We also discuss the impact of violating these assumptions, noting how the severity of these consequences increases alongside the strength of the provided guarantees.

Next, we discuss how the techniques discussed in this dissertation could be applied to other types of storage hardware. Although the techniques we describe here were originally developed for PM, some of them are more widely applicable and may be useful to gain confidence in the correctness of systems for block-based storage devices as well.

Finally, we describe our experiences learning systems verification when building CapybaraKV and distill them into a set of observations. We provide examples from CapybaraKV to back up these observations. The key goal of this section is to explain the intuition we built over the development of CapybaraKV about how to design verified systems and write and debug proofs.

Chapter 8: Future work

In this chapter, we discuss possible directions for future work.

8.1 Rust for lightweight static checking

As shown in this dissertation with SquirrelFS and in prior work [113], Rust's powerful type system can be used to check certain crash safety properties in persistent memory file systems. We believe an interesting line of work would be to investigate additional ways to use the Rust type system to statically enforce other properties, both in storage systems and more broadly.

As discussed in §6.3, there has been some work on using ideas from the programming languages and formal methods communities to statically check other properties using the Rust compiler. These techniques could be useful for enforcing additional properties in storage systems (or low-level systems more broadly) without high proof burden. Further investigation is required to determine if and how these techniques could be applied to check high-level correctness and safety properties like crash consistency.

Members of the Rust development community have also recently proposed the addition of language support for *contracts*, predicates about program state and correctness that could be used for both static and dynamic analysis techniques [149,150]. Another proposal suggested the addition of ghost code that would be checked by the compiler but erased in the resulting executable [162] to the main language. Contracts and ghost code are widely used in program verification and are handled differently by each Rust verification tool. These proposals aimed to standardize support for these features and make them easier to use. Neither proposal has been adopted, but such constructs could be used to express properties to check in a lightweight way, although this may require additional tooling beyond the compiler.

8.2 Typestate for asynchronous file systems

As discussed in §7.2, the typestate pattern could be applied to check ordering-related properties in asynchronous file systems built for slower media. We believe another interesting line of work would be to investigate the use of typestate in such systems. As with traditional soft updates systems and other crash-consistency mechanism currently in wider use, such a system would resolve most updates asynchronously from a page cache, rather than in the critical path of a system call handler. Typestates could potentially be used in this part of the system's implementation to enforce ordering properties.

A challenge we encountered in SquirrelFS that may become more prominent in an asynchronous system is dealing with collections of unbounded size. For example, it would be difficult to statically check that a set of pending journal entries have become durable before committing them. One solution may be to wrap such collections in fixed-size abstractions for typestate checking purposes, as we did in SquirrelFS. It is also possible that new system designs that manage the page cache or a journal differently from existing systems would be easier to check using typestate, similar to how we designed a non-standard durable layout for SquirrelFS. Extensions to Rust, such as those proposed by Flux [170], could also potentially be useful.

8.3 Studying corruption in byte-addressable storage

The problem of data corruption on persistent memory has not been well studied. We present a model of corruption and techniques to detect it in Chapter 5, as does the NOVA-Fortis file system [267], but there have been no large-scale studies as there have been for other storage media [14, 15, 98, 99, 237, 252, 276]. Further research is required both to understand how best to model and detect corruption on byte-addressable storage devices as well as to understand how to deploy them in large-scale systems in a reliable way.

Prior studies on other types of storage hardware have either studied data collected over months or years on millions of drives [14, 15, 237] or used custom-built hardware to stress-test storage devices by repeatedly cutting their power supply [98, 252, 276]. To our knowledge, no study of either type has been conducted on any kind of persistent memory hardware. The closest related work examined the nature of PM-related patches submitted to the Linux kernel [85], some of which were related to corruption detection and handling.

Detailed measurements from individual devices as well as large-scale studies of corruption in production systems would both provide valuable information about these devices and how they behave in real deployments in the presence of crashes or other corrupting events. Such a study would be particularly valuable *before* PM is widely deployed in cloud systems and/or personal computers in order to understand and mitigate corruption issues before they occur.

8.4 Environmental impact of persistent memory

The sustainability and environmental impact of data center technologies is becoming increasingly important. An interesting line of inquiry would be to study the environmental impact of different types of storage-class memory and develop techniques to reduce it.

Recent research on data center sustainability has shown that embodied carbon, or carbon emissions from manufacturing components, is a large problem [19, 184]. Reusing or continuing to use components beyond their standard lifetime, potentially in a degraded form, has been proposed as a way to reduce embodied carbon [185, 219]. This solution, while effective, requires more investigation to apply it successfully to storage devices, which can experience higher failure rates over time [184] or have limited write endurance [186]. Understanding the lifetime, failure rates, and write endurance of different PM technologies well enough to consider sustainability approaches like this will require more research.

There are other non-carbon-related environmental concerns, such as water consumption and e-waste, that should also be considered. In particular, batteries can produce particularly hazardous waste [255], so using them for large amounts of battery-backed DRAM could have additional impacts on the amount of e-waste produced by data centers. The materials used in the production of future PM hardware could also be environmentally costly to obtain or produce. To our knowledge, these factors have not been investigated with regard to future PM technologies.

8.5 Other PM use cases

This dissertation focuses on systems that use PM for its durability. However, PM differs from existing DRAM technology in a number of ways besides persistence, and recent research has also begun to investigate how these additional characteristics may be useful for different types of systems. For example, systems and techniques for tiered memory like Nimble [268], KLOCs [139], and HeMem [220] have explored how to use slower types of memory (e.g., Optane PM) to expand main memory capacity. These systems generally treat PM as additional, slower layer in the memory hierarchy to store colder data. They have used Optane PM in both Memory Mode (which configures the hardware to treat DRAM as an L4 cache and Optane as volatile main memory [139]) and App Direct mode, but do not rely on PM's durability in either case. Memory Mode, which presents Optane PM as volatile, in fact encrypts and stores data durably and throws out the encryption key on a power cycle to prevent future access [116].

Note, however, that Optane PM still provides persistence regardless of whether it is used. By not utilizing PM's durability at all, these systems leave some features on the table. For example, if a server one of these systems loses power, it has no way to recover the information stored on its PM even though the data is still present. Although losing this data does not impact the functionality of the system after a crash, it could impact performance by requiring the system to read the data from

block storage or other servers. However, the designers of these systems would likely not want to give up any performance in order to obtain crash consistency on PM. The techniques described in this dissertation prioritize consistency over performance, so they would not work well in this use case. An interesting line of work would be to look at how these systems could leverage the inherent persistence of PM devices they already use with minimal performance impact. Since flush/write-back and ordering primitives increase performance overheads, one potential approach could be to apply ideas from NoFS [44], which achieves crash consistency without ordering durable updates.

Recent work has also proposed developing storage-class memory technologies that trade shorter retention times for better write endurance and capacity [169]. This class of device, *managed-retention memory* (MRM), could be of particular benefit for AI workloads that currently depend on high-bandwidth memory (HBM), which is necessary to hit bandwidth requirements but is expensive and is expected to face yield and scaling challenges in the future [169]. Systems using PM that only promises to retain data for hours or days may have different crash-consistency and data integrity considerations than the storage systems examined in this dissertation. We believe that investigating the reliability requirements of these systems would also be an interesting future direction for PM software research.

Chapter 9: Lessons learned and conclusion

9.1 Lessons learned

The three projects discussed in this dissertation use techniques that are generally studied by separate research communities. In this section, we discuss lessons learned from studying all of these techniques to address storage system robustness.

Lesson 1: Most prior work has focused on individual robustness techniques, but the best way to build truly robust systems is to use multiple techniques together.

This dissertation presents three approaches to building robust storage systems that are generally considered independent and completely distinct. Based on experiences with all three techniques, I argue that the best way to build truly robust storage systems is to use these approaches *together*. Although lightweight static checking and verification provide stronger guarantees than testing, neither can prove the absolute correctness of a system because they rely on assumptions about the external environment and what it means for the system to be correct. Using multiple techniques together can help catch additional bugs and detect flawed assumptions before they cause real-world issues.

For instance, I argue that verified systems should be thoroughly tested, just as unverified systems are. While such testing will almost certainly find fewer bugs than in unverified systems, it can help find bugs in the specification of correctness or the model of the external environment the proofs are based on. It is not hard to write an incorrect specification or make potentially dangerous assumptions that are not easy to notice when manually checking the code. Aspects of a system's specification that can be time-consuming to handle but seem somewhat less critical to correctness may be particularly at risk. For example, during the development of CapybaraKV, we prioritized specifying regular-case and crash-related behaviors, sometimes at the expense of error-case specifications. At one point, when adding lists (a feature not

discussed in this dissertation) an incorrectly-specified error case allowed an operation to *always* return an out-of-space error regardless of how much space was actually left.[1] The root cause of the bug was the use of an incorrect Boolean operator, and code with this bug verified because the implementation was valid for the incorrect specification we had provided. This bug was only found when testing the system using pre-existing tests written for an unverified system.

We also combined standard testing techniques, including CHIPMUNK, with lightweight static checking in SQUIRRELFS to find regular-execution and non-ordering-related crash-consistency bugs. An interesting avenue for future work could be combining verification with typestate checking; for example, verifying the functionality of typestate transition functions and using typestate to check higher-level operations.

The techniques we describe in this dissertation occupy various points in the trade-off space between complexity and confidence, but they are not mutually exclusive. We can further strengthen our confidence in systems by combining these techniques to use the strengths of each approach while compensating for their weaknesses. Combining these techniques, as well as techniques from related work, in different ways can help developers find the right spot in the trade-off space and obtain the guarantees they need for their particular use case.

Lesson 2: All robustness techniques, not just verification, make assumptions that are crucial to their ability to detect or prevent problems.

Developers of verified systems are generally explicit about the assumptions they make and the unverified components they depend on. In contrast, developers of testing tools do not usually state the assumptions that the correctness of their tools rely on. In lightweight static checking, the authors of Corundum [113] list some assumptions made by the library. We did not include such a list in the original SQUIRRELFS paper, but a discussion of the assumptions made in that project is

[1] This bug was found by Jay Lorch.

included here in §7.1.

We observe that *all* of these approaches make key assumptions upon which their utility and ability to detect or prevent bugs depends. As discussed in §7.1, all of these techniques assume that their model of the underlying PM's crash behavior is correct. Especially when trying to ensure properties like crash consistency, where systematic testing requires modeling hardware behaviors we have little direct insight into, the assumption that this model is correct is the crucial. Although the consequences of getting these assumptions wrong in a testing tool are less problematic than in a verified system, they can still result in false positives that waste developer time or unnecessary false negatives that cause data loss later on. For example, an early version of Chipmunk did not model cache lines correctly in all cases, resulting in several false positive bugs. We did not realize they were false positives until after finding and fixing the issue with cache line modeling, when they could no longer be replicated.

I argue that the practices of explicitly stating and carefully scrutinizing assumptions, already present in the verification community, should be adopted by developers of other techniques as well. I recommend the following practices that are currently in use in the verification community. First, papers and/or documentation about verified systems generally explicitly state which components are trusted for correctness. Along with the specification of correctness and models of the external environment, developers of these tools list the dependencies upon which the verified code depends for correctness. The authors are generally explicit about what system components should be manually audited by a human to confirm that they are correct. Second, it is common for verified codebases to be organized such that each file contains only trusted or untrusted code, and for this to be reflected in the name of each file. For example, many projects (including CapybaraKV and CapybaraNS) use a `_t` suffix on file names containing only trusted code and `_v` for files with verified code. This practice makes it easier to determine which parts of the codebase should be audited. Unverified systems would also benefit from clear documentation about

assumptions and organizational practices that make it easier to understand how these assumptions are made and used in different parts of the codebase.

It is more difficult to draw a clear distinction in unverified systems between code that makes critical assumptions and code that relies on those assumptions for correctness. More work is likely required to determine the best principles for understanding and stating the assumptions made in unverified code. One possible model for this is unsafe code in Rust [234]. Unsafe Rust is somewhat analogous to unverified code; it is subject to weaker compiler-enforced restrictions than regular Rust, and generally makes some unchecked assumptions about how underlying resources are being used. Unsafe Rust is the only way to perform certain operations (e.g., dereferencing a raw pointer or interacting with other languages), but it must be used carefully to avoid introducing memory safety issues. It is standard practice for Rust developers to document each unsafe function with a description of the assumptions the function makes and how to use it safely. It is also common to include comments with each invocation of an unsafe function to explain why that particular call is safe. Unsafe Rust only targets specific, potentially-memory-unsafe operations, but it demonstrates how critical assumptions can be handled more safely in unverified programs via both language support and standard documentation practices.

Lesson 3: Most storage system robustness techniques are developed specifically for asynchronous contexts, but starting from a synchronous storage model can help keep these techniques simple and does not preclude generalizing to asynchrony later.

Thanks to our focus on PM, the systems discussed in this dissertation were originally developed for mostly synchronous use cases. In particular, SquirrelFS provides synchronous system calls, our original version of CapybaraKV's storage model assumed completely synchronous writes to storage and used synchronous operations on keys and values. As discussed in Chapter 3 and Chapter 4, synchrony allows systems to provide stronger, better-defined guarantees than can be achieved in asynchronous settings, and often simplifies reasoning about properties like crash

consistency. It was only after establishing the key ideas behind SquirrelFS and CapybaraKV in a synchronous setting that we realized that some aspects in both projects could be generalized to asynchronous settings as well.

I believe that starting in a synchronous context played a significant role in the development of many of the techniques described in this dissertation, and is a potentially useful tool for developing future robustness techniques as well. Specifically, we started with a synchronous model of PM in SquirrelFS and PoWER/CapybaraKV that we later extended to encompass asynchrony. The intuition behind this approach is that it is easier to reason about crash behaviors in a synchronous setting than an asynchronous one. In our experience, it is also easier to assume that each storage I/O operation is synchronous, and determine where this assumption could be relaxed later, than to have to consider all possible reorderings from the start.

This approach was particularly valuable when building CapybaraKV. The complexity eliminated by starting in a synchronous setting helped us land on the key insight behind PoWER – that crash consistency can be enforced via Hoare logic preconditions. We have been asked why prior work did not develop and use PoWER, since it is simpler and more flexible than existing techniques. I believe the fact that prior work had to reason about crashes and asynchrony together played a major role in why these approaches are more complicated. SquirrelFS is more closely tied to synchrony, but the core idea underlying the use of the typestate pattern to enforce crash consistency is compatible with asynchrony. Furthermore, when determining the ordering rules to enforce using typestate, our initial informal reasoning was based on a synchronous model of PM, which we extended to an asynchronous model when we built the system.

It is likely more difficult in general to generalize a synchronous system design to an asynchronous setting; the approach proposed here is more useful when reasoning about the underlying storage model. For example, much of SquirrelFS's design

(e.g., Synchronous Soft Updates) relies on synchrony, and the complex nature of file system operations complicates the transition from synchrony to asynchrony. However, we did successfully transition CapybaraKV, which has a much simpler KV store interface, from a fully-synchronous design to an asynchronous, transaction-based design, so this approach is not limited to the storage model.

Crash consistency is already difficult to reason about, so focusing on it and keeping other properties as simple as possible helped facilitate the development of the techniques presented in this dissertation. PM provides an avenue for building synchronous storage systems, and so it helped us develop novel and straightforward storage-system robustness techniques that are applicable to traditional storage as well.

9.2 Closing words

In this dissertation, we have explored a set of techniques for ensuring the robustness of persistent memory storage systems. We first presented Chipmunk, a testing tool for POSIX PM file systems that is useful with both new and existing systems and effective at finding serious crash-consistency bugs. Next, we presented SquirrelFS, a new PM file system demonstrating the use of the typestate pattern for crash consistency together with Synchronous Soft Updates, a new crash-consistency mechanism. Finally, we presented PoWER, a new technique for verifying crash consistency, and CapybaraKV, a verified PM key-value store that uses it. Each approach falls at a different point in the trade-off space between effort and confidence. They contribute to the set of tools available for storage system developers, enabling them to build more robust systems with a variety of resources and requirements. We have also discussed the guarantees and assumptions made by each technique and described our experiences learning how to build verified systems in Verus, focusing on unexpected challenges and gaps that had to be bridged coming from a background in systems with limited verification experience. We have described lessons learned by

working on testing, lightweight static methods, and verification together and made recommendations based on our experiences with these techniques.

We hope that the techniques presented in this dissertation inform the development of future storage systems for PM as well as traditional and potential future storage devices. We have focused on developing techniques that are useful for practitioners and provide avenues for developers of real-world systems to adopt more rigorous static approaches to robustness. In working at the intersection of previously-separate research communities, we have also learned and shared valuable lessons both about developing new robustness approaches and building robust systems themselves. We hope that these lessons aid the development of both future robustness techniques and of new storage systems, and that they inspire continued work on approaches that utilize the knowledge from many different research communities to provide high confidence and well-defined guarantees.

Appendix A: Open-source code

All software presented in this dissertation is open source. The source code for Chipmunk can be found at https://github.com/utsaslab/chipmunk. The source code for SquirrelFS can be found at https://github.com/utsaslab/squirrelfs. The source code for CapybaraKV can be found at https://github.com/microsoft/verified-storage.

References

[1] Aeneas Developers. Aeneas. https://github.com/AeneasVerif/aeneas.

[2] Paul Alcorn. 3D XPoint: A Guide To The Future Of Storage Class Memory. https://www.tomshardware.com/reviews/3d-xpoint-guide,4747.html, November 2016.

[3] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In *Proceedings of the ACM Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, OOPSLA '09, pages 1015–1022, New York, NY, USA, 2009. Association for Computing Machinery.

[4] Alloy Developers. Alloy 6. https://alloytools.org/.

[5] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. Cogent: Verifying high-assurance file system implementations. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 175–188, April 2016.

[6] Amazon Web Services. Amazon S3 Storage Classes. https://aws.amazon.com/s3/storage-classes/.

[7] Amazon Web Services. AWS GovCloud (US). https://aws.amazon.com/govcloud-us/.

[8] Thomas E. Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N. Schuh, and Emmett Witchel. Assise: Performance and availability via client-local NVM in a distributed file system. In *Proceedings of the USENIX Symposium on Operating Systems Design*

and Implementation (OSDI), pages 1011–1027. USENIX Association, November 2020.

[9] Storage Networking Industry Association. NVM programming model v1.2. https://www.snia.org/tech_activities/standards/curr_standards/npm, June 2017.

[10] Vytautas Astrauskas, Aurel Bílý, Jonáš Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J. Summers. The Prusti project: Formal verification for Rust. In *NASA Formal Methods: 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24–27, 2022, Proceedings*, page 88–108, Berlin, Heidelberg, 2022. Springer-Verlag.

[11] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. How do programmers use unsafe rust? *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.

[12] Valerie Aurora. Soft updates, hard problems. https://lwn.net/Articles/339337/, July 2009.

[13] Jeremy Avigad, Leonardo de Moura, Soonho Kong, and Sebastian Ullrich. Theorem proving in lean 4. https://leanprover.github.io/theorem_proving_in_lean4/.

[14] 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 *6th USENIX Conference on File and Storage Technologies (FAST 08)*, San Jose, CA, February 2008. USENIX Association.

[15] 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 International Conference on Measurement and*

Modeling of Computer Systems, SIGMETRICS '07, page 289–300, New York, NY, USA, 2007. Association for Computing Machinery.

[16] Piotr Balcer. Exploring the Software Ecosystem for Compute Express Link (CXL) Memory. https://pmem.io/blog/2023/05/exploring-the-software-ecosystem-for-compute-express-link-cxl-memory/, May 2023.

[17] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.

[18] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. Viper: an efficient hybrid PMem-DRAM key-value store. *Proc. VLDB Endow.*, 14(9):1544–1556, May 2021.

[19] Daniel S. Berger, David Brooks, Fiodar Kazhamiaka, Mark D. Hill, Ricardo Bianchini, Carole-Jean Wu, Karin Strauss, Kali Frost, Jaylen Wang, Kevin Martins, Sharon Gillett, Esha Choukse, Dan Ernst, Rodrigo Fonseca, Kari Lio, Bhargavi Narayanasetty, Pratyush Patel, Celine Irvene, Akshitha Sriraman, George Porter, Alex Jones, Udit Gupta, Bilge Acun-Uyan, Kim Hazelwood, and Doug Carmean. Reducing embodied carbon is important. https://www.sigarch.org/reducing-embodied-carbon-is-important/.

[20] BetrFS Developers. Checksumming. https://btrfs.readthedocs.io/en/latest/Checksumming.html, 2024.

[21] Nicolas Boichat. Issue 502898: ext4: Filesystem corruption on panic, June 2015. https://code.google.com/p/chromium/issues/detail?id=502898.

[22] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. Theseus: an experiment in operating system structure and state management. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1–19. USENIX Association, November 2020.

[23] A. Borgida, J. Mylopoulos, and R. Reiter. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, 21(10):785–798, 1995.

[24] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. Using lightweight formal methods to validate a key-value storage node in Amazon S3. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 836–850, October 2021.

[25] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. Specifying and checking file system crash-consistency models. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 83–98, Atlanta, GA, USA, April 2016.

[26] Box. Box for federal government. https://www.box.com/industries/government-federal.

[27] Neil Brown. Block layer introduction part 2: the request layer. https://lwn.net/Articles/738449/, 2017.

[28] BTRFS Developers. BTRFS documentation. https://btrfs.readthedocs.io/en/latest/.

[29] Can Cebeci, Yonghao Zou, Diyu Zhou, George Candea, and Clément Pit-Claudel. Practical verification of system-software components written in standard C. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 455–472, November 2024.

[30] Tej Chajed. *Verifying a concurrent, crash-safe file system with sequential reasoning*. PhD thesis, Massachusetts Institute of Technology, 2022.

[31] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 243–258, 2019.

[32] Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai Zeldovich. GoJournal: A verified, concurrent, crash-safe journaling system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 423–439, July 2021.

[33] Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai Zeldovich. GoJournal: A verified, concurrent, crash-safe journaling system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 423–439, July 2021.

[34] Tej Chajed, Joseph Tassarotti, Mark Theng, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying the DaisyNFS concurrent and crash-safe file system with sequential reasoning. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 447–463, Carlsbad, CA, July 2022. USENIX Association.

[35] Aleks Chakarov, Jaco Geldenhuys, Frank Michel, Matthew Heck, Mike Hicks, Sam Huang, Georges Axel Jaloyan, Anjali Joshi, Rustan Leino, Mikael Mayer,

Sean McLaughlin, Akhilesh Mritunjai, Clément Pit Claudel, Sorawee Porncharoenwase, Florian Rabe, Marianna Rapoport, Giles Reger, Cody Roux, Neha Rungta, Robin Salkeld, Matthias Schlaipfer, Daniel Schoepe, Johanna Schwartzentruber, Serdar Tasiran, Aaron Tomb, Emina Torlak, John Tristan, Lucas Wagner, Mike Whalen, Remy Willems, Jenny Xiang, Tae Joon Byun, Joshua Cohen, Ruijie Wang, Junyoung Jang, Jakob Rath, Hira Taqdees Syeda, Dominik Wagner, and Yongwei Yuan. Formally verified cloud-scale authorization. 2025.

[36] Aleksandar Chakarov, Aleksandr Fedchin, Zvonimir Rakamaric, and Neha Rungta. Better counterexamples for Dafny. 2022.

[37] Donald D. Chamberlin, Morton M. Astrahan, Michael W. Blasgen, James N. Gray, W. Frank King, Bruce G. Lindsay, Raymond Lorie, James W. Mehl, Thomas G. Price, Franco Putzolu, Patricia Griffiths Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost. A history and evaluation of System R. *Commun. ACM*, 24(10):632–646, October 1981.

[38] Chia Chao, Robert English, David Jacobson, Alexander Stepanov, and John Wilkes. Mime: a high performance parallel storage device with strong recovery guarantees. Technical Report HPL–CSP–92–9rev1, HP Laboratories, Palo Alto, CA, USA, November 1992.

[39] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay İleri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 270–286, October 2017.

[40] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using Crash Hoare logic for certifying the FSCQ

file system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 18–37, October 2015.

[41] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. Flatstore: An efficient log-structured key-value storage engine for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1077–1091, New York, NY, USA, 2020. Association for Computing Machinery.

[42] Vijay Chidambaram. *Orderless and Eventually Durable File Systems*. PhD thesis, University of Wisconsin, Madison, Aug 2015.

[43] 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, page 228–243, New York, NY, USA, 2013. Association for Computing Machinery.

[44] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Consistency Without Ordering. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 101–116, San Jose, California, February 2012.

[45] Brian Chin, Shane Markstrum, and Todd Millstein. Semantic type qualifiers. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 85–95, New York, NY, USA, 2005. Association for Computing Machinery.

[46] Jeongdong Choe. Intel's 2nd Generation XPoint Memory - Will it be worth the long wait ahead? https://www.techinsights.com/blog/memory/intels-2nd-generation-xpoint-memory, 2021.

[47] Compute Express Link Consortium. Compute Express Link (CXL) specification. https://www.computeexpresslink.org/download-the-specification.

[48] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 133–146, New York, NY, USA, 2009. Association for Computing Machinery.

[49] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, pages 143–154, June 2010.

[50] Jonathan Corbet. ext4 and data loss, March 2009. http://lwn.net/Articles/322823/.

[51] Intel Corporation. Intel Optane Persistent Memory. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/overview.html.

[52] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, page 262–275, New York, NY, USA, 1999. Association for Computing Machinery.

[53] Iris Crawford. How much CO2 is emitted by manufacturing batteries? https://climate.mit.edu/ask-mit/how-much-co2-emitted-manufacturing-batteries, July 2022.

[54] crc64fast Developers. crc64fast. https://crates.io/crates/crc64fast, 2024.

[55] Dafny Developers. Dafny reference manual. https://dafny.org/dafny/DafnyRef/DafnyRef.html.

[56] Debendra Das Sharma, Robert Blankenship, and Daniel Berger. An Introduction to the Compute Express Link (CXL) Interconnect. *ACM Comput. Surv.*, 56(11), July 2024.

[57] Data Center Knowledge. Equinix Outages Through the Years: Key Incidents and Lessons Learned. https://www.datacenterknowledge.com/outages/equinix-outages-through-the-years-key-incidents-and-lessons-learned, 2024.

[58] DDRdrive. DDRdrive X1: A Solid-State Storage System. https://ddrdrive.com/menu1.html.

[59] Leonardo de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *2008 Tools and Algorithms for Construction and Analysis of Systems*, pages 337–340. Springer, Berlin, Heidelberg, March 2008.

[60] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, page 59–69, New York, NY, USA, 2001. Association for Computing Machinery.

[61] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. Creusot: A foundry for the deductive verification of Rust programs. In *Proceedings of the International Conference on Formal Engineering Methods (ICFEM)*, pages 90–105, October 2022.

[62] Linux Kernel Developers. Journal (jbd2). https://www.kernel.org/doc/html/latest/filesystems/ext4/journal.html.

[63] Bang Di, Jiawen Liu, Hao Chen, and Dong Li. Fast, flexible, and comprehensive bug detection for persistent memory programs. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 503–516, April 2021.

[64] Larry Dignan. Ready for ReRAM? HP and Hynix think so. https://www.zdnet.com/article/ready-for-reram-hp-and-hynix-think-so/.

[65] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering with PANDA. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, PPREW-5, New York, NY, USA, 2015. Association for Computing Machinery.

[66] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. Performance and protection in the ZoFS user-space NVM file system. *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019.

[67] Mingkai Dong and Haibo Chen. Soft updates made simple and fast on non-volatile memory. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 719–731, Santa Clara, CA, July 2017. USENIX Association.

[68] Dropbox. Dropbox. https://www.dropbox.com/.

[69] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

[70] Edge Delta. Discover the most crucial data storage statistics in 2024. Gain insights on trends and key figures shaping the data storage landscape. https://edgedelta.com/company/blog/data-storage-statistics, 2024.

[71] Robert Elder. What causes bit flips in computer memory? https://blog.robertelder.org/causes-of-bit-flips-in-computer-memory/.

[72] Everspin Technologies, Inc. Spin-transfer Torque MRAM Technology. https://www.everspin.com/spin-transfer-torque-mram-technology.

[73] Gregory Ewing. Reverse-Engineering a CRC Algorithm. https://www.csse.canterbury.ac.nz/greg.ewing/essays/CRC-Reverse-Engineering.html, March 2010.

[74] ext4 Developers. ext4 Data Structures and Algorithms. https://docs.kernel.org/filesystems/ext4/, 2024.

[75] John Field, Deepak Goyal, G. Ramalingam, and Eran Yahav. Typestate verification: Abstraction techniques and complexity results. Springer Berlin Heidelberg, October 2005.

[76] FileCloud. Cloud Storage for Banks. https://www.filecloud.com/cloud-storage-for-banks/.

[77] Robert W. Floyd. Assigning meanings to programs. In *Proceedings of the Symposium on Applied Mathematics*, pages 19–32, 1967.

[78] Pedro Fonseca, Rodrigo Rodrigues, and Björn B. Brandenburg. SKI: Exposing kernel concurrency bugs through systematic schedule exploration. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 415–431, Broomfield, CO, October 2014. USENIX Association.

[79] Free Software Foundation, Inc. Program instrumentation options. https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html.

[80] FreeBSD Project. Chapter 22. The Z File System (ZFS). https://docs.freebsd.org/en/books/handbook/zfs/.

[81] Christopher Frost, Mike Mammarella, Eddie Kohler, Andrew de los Reyes, Shant Hovsepian, Andrew Matsuoka, and Lei Zhang. Generalized file system dependencies. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, SOSP '07, pages 307–320, New York, NY, USA, 2007. Association for Computing Machinery.

[82] Xinwei Fu, Wook-Hee Kim, Ajay Paddayuru Shreepathi, Mohannad Ismail, Sunny Wadkar, Dongyoon Lee, and Changwoo Min. Witcher: Systematic crash consistency testing for non-volatile memory key-value stores. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 100–115, October 2021.

[83] 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 *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 149–166, Santa Clara, CA, February 2017. USENIX Association.

[84] Gregory R. Ganger and Yale N. Patt. Metadata update performance in file systems. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 49–60, Monterey, CA, November 1994. USENIX Association.

[85] Om Rameshwar Gatla, Duo Zhang, Wei Xu, and Mai Zheng. Understanding persistent-memory-related issues in the Linux kernel. *ACM Trans. Storage*, 19(4), October 2023.

[86] Robin Geuens. How much data is generated every day? https://soax.com/research/data-generated-per-day, February 2025.

[87] Sanjay Ghemawat and Jeff Dean. LevelDB. https://github.com/google/leveldb.

[88] Jon Gjengset. *Rust for Rustaceans*. No Starch Press, 2022.

[89] Georges Gonthier. A computer-checked proof of the Four Color Theorem. Technical report, Inria, March 2023.

[90] Google. Data availability and durability. https://cloud.google.com/storage/docs/availability-durability.

[91] Google. Google Cloud for financial services. https://cloud.google.com/solutions/financial-services?hl=en.

[92] Google. Google Drive. https://workspace.google.com/products/drive/.

[93] Google. Google for government. https://cloud.google.com/gov?hl=en.

[94] Google. OSS-Fuzz. https://github.com/google/oss-fuzz.

[95] Google. Syzkaller. https://github.com/google/syzkaller/, 2021.

[96] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. Jaaru: Efficiently model checking persistent memory programs. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 415–428, April 2021.

[97] Bogdan Gribincea. Ext4 data loss. https://bugs.launchpad.net/ubuntu/+source/linux/+bug/317781, January 2009.

[98] Laura M. Grupp, Adrian M. Caulfield, Joel Coburn, Steven Swanson, Eitan Yaakobi, Paul H. Siegel, and Jack K. Wolf. Characterizing flash memory: Anomalies, observations, and applications. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 24–33, 2009.

[99] Laura M. Grupp, John D. Davis, and Steven Swanson. The bleak future of NAND flash memory. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, page 2, USA, 2012. USENIX Association.

[100] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 653–669, Savannah, GA, November 2016. USENIX Association.

[101] Robert B. Hagmann. Reimplementing the Cedar file system using logging and group commit. In Les Belady, editor, *Proceedings of the Eleventh ACM Symposium on Operating System Principles, SOSP 1987, Stouffer Austin Hotel, Austin, Texas, USA, November 8-11, 1987*, pages 155–162. ACM, 1987.

[102] Travis Hance. impl Drop should be disallowed. https://github.com/verus-lang/verus/issues/723, 2023.

[103] Travis Hance. *Verifying Concurrent Systems Code*. PhD thesis, Carnegie Mellon University, August 2024.

[104] Travis Hance, Andrea Lattuada, Chris Hawblitzel, Jon Howell, Rob Johnson, and Bryan Parno. Storage systems are distributed systems (so verify them that way!). In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 99–115, November 2020.

[105] Jim Handy. Intel's Optane DIMM Price Model. https://thememoryguy.com/intels-optane-dimm-price-model/, 2019.

[106] John Harrison, Josef Urban, and Freek Wiedijk. History of interactive theorem proving. In Jörg H. Siekmann, editor, *Computational Logic*, volume 9 of *Handbook of the History of Logic*, pages 135–214. North-Holland, 2014.

[107] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving practical distributed systems correct. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–17, October 2015.

[108] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad Apps: End-to-end security via automated full-system verification. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 165–181, October 2014.

[109] Hewlett Packard Enterprise. HPE Persistent Memory. https://www.hpe.com/us/en/servers/persistent-memory.html.

[110] Son Ho and Jonathan Protzenko. Aeneas: Rust verification by functional translation. *Proc. ACM Program. Lang.*, 6(ICFP), August 2022.

[111] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[112] Gerard Holzmann. *The Spin model checker: primer and reference manual*. Addison-Wesley Professional, first edition, 2003.

[113] Morteza Hoseinzadeh and Steven Swanson. Corundum: Statically-enforced persistent memory safety. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 429–442, April 2021.

[114] HP. Smart Array technology: advantages of battery-backed cache. http://h10032.www1.hp.com/ctg/Manual/c00257513.pdf.

[115] Hyper OS Systems. Why DDR SSDs are the next step. http://www.hyperossystems.co.uk/07042003/products.htm.

[116] Intel Corporation. Brief: Intel Optane Persistent Memory. https://www.intel.la/content/www/xl/es/products/docs/memory-storage/optane-persistent-memory/optane-dc-persistent-memory-brief.html.

[117] Intel Corporation. Learn more part 2 – persistent memory architecture. https://www.intel.com/content/www/us/en/developer/articles/training/pmem-learn-more-series-part-2.html.

[118] Intel Corporation. Persistent memory development kit. https://pmem.io/pmdk/.

[119] Intel Corporation. Pmem-redis. https://github.com/pmem/pmem-redis.

[120] Intel Corporation. Pmem-rocksdb. https://github.com/pmem/pmem-rocksdb.

[121] Intel Corporation. Discover Persistent Memory Programming Errors with Pmemcheck. https://www.intel.com/content/www/us/en/developer/articles/technical/discover-persistent-memory-programming-errors-with-pmemcheck.html, 2018.

[122] Intel Corporation. eadr: New opportunities for persistent memory applications. https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html, January 2021.

[123] Isabelle Developers. Isabelle. https://isabelle.in.tum.de/.

[124] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amir Saman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the Intel Corporation Optane DC persistent memory module. *CoRR*, abs/1903.05714, 2019.

[125] Daniel Jackson. *Software Abstractions*. The MIT Press, 2016.

[126] Daniel Jackson and Craig A. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Trans. Softw. Eng.*, 22(7):484–495, July 1996.

[127] Bing Jiao, Ashvin Goel, and An-I Andy Wang. Silhouette: Leveraging consistency mechanisms to detect bugs in persistent Memory-Based file systems. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 407–423, Santa Clara, CA, February 2025. USENIX Association.

[128] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: securing the foundations of the Rust programming language. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 1–34, January 2018.

[129] Ralf Jung, R. Krebbers, Jacques-Henri Jourdan, A. Bizjak, L. Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28(20):1–73, 2018.

[130] Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. The future is ours: prophecy variables in separation logic. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 45:1–32, January 2020.

[131] Rohan Kadekodi. *Transparently achieving high performance for applications on Persistent Memory*. PhD thesis, University of Texas at Austin, Aug 2023.

[132] Rohan Kadekodi, Saurabh Kadekodi, Soujanya Ponnapalli, Harshad Shirwadkar, Gregory R. Ganger, Aasheesh Kolli, and Vijay Chidambaram. WineFS: A hugepage-aware file system for persistent memory that ages gracefully. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, SOSP '21, pages 804–818, New York, NY, USA, 2021. Association for Computing Machinery.

[133] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. SplitFS: Reducing software overhead in file

systems for persistent memory. In *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP '19, pages 494–508, New York, NY, USA, 2019. Association for Computing Machinery.

[134] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young ri Choi. SLM-DB: Single-Level Key-Value store with persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 191–205, Boston, MA, February 2019. USENIX Association.

[135] Samuel Kalbfleisch, Lukas Werling, and Frank Bellosa. Vinter: Automatic non-volatile memory crash consistency testing for full systems. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 933–950, July 2022.

[136] Eunsuk Kang and Daniel Jackson. Formal modeling and analysis of a flash filesystem in Alloy. In Egon Börger, Michael Butler, Jonathan P. Bowen, and Paul Boca, editors, *Abstract State Machines, B and Z*, pages 294–308, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[137] Kani Developers. The Kani Rust Verifier. https://model-checking.github.io/kani/, 2024.

[138] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning LSMs for nonvolatile memory with NoveLSM. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 993–1005, Boston, MA, July 2018. USENIX Association.

[139] Sudarsun Kannan, Yujie Ren, and Abhishek Bhattacharjee. KLOCs: kernel-level object contexts for heterogeneous memory systems. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 65–78, New York, NY, USA, 2021. Association for Computing Machinery.

[140] Rajat Kateja, Anirudh Badam, Sriram Govindan, Bikash Sharma, and Greg Ganger. Viyojit: Decoupling battery and DRAM capacities for battery-backed DRAM. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 613–626, New York, NY, USA, 2017. Association for Computing Machinery.

[141] Michael Kazar, Bruce Leverett, Owen Anderson, Vasilis Apostolides, Beth Bottos, Sailesh Chutani, Craig Everhart, William Mason, Shu-Tsui Tu, and Edward Zayas. Decorum file system architectural overview. pages 151–164, 01 1990.

[142] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. Lessons learned from the Chameleon Testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020.

[143] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 147–161, October 2019.

[144] Wonbae Kim, Chanyeol Park, Dongui Kim, Hyeongjun Park, Young ri Choi, Alan Sussman, and Beomseok Nam. ListDB: Union of Write-Ahead logs and persistent SkipLists for incremental checkpointing on persistent memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 161–177, Carlsbad, CA, July 2022. USENIX Association.

[145] KIOXIA America, Inc. XL-Flash. https://americas.kioxia.com/en-us/business/memory/xlflash.html, 2024.

[146] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, USA, 2018.

[147] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computing Systems*, 32(1):1–70, February 2014.

[148] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery.

[149] Felix Klock. Contracts and invariants. https://rust-lang.github.io/rust-project-goals/2024h2/Contracts-and-invariants.html, 2024.

[150] Felix Klock. Contracts for Rust. Rust Verification Workshop 2024, 2024.

[151] Philip Koopman. CRC Polynomial Zoo. https://users.ece.cmu.edu/~koopman/crc/crc64.html, November 2019.

[152] Philip Koopman. *Understanding Checksums and Cyclic Redundancy Checks*. 2024.

[153] Greg Kramer. Direct Drive - Azure's Next-generation Block Storage Architecture. https://www.sniadeveloper.org/events/agenda/session/347, 2022. SNIA Developer Conference.

[154] Rahul Kumar. How Open Source Projects are Using Kani to Write Better Software in Rust. https://aws.amazon.com/blogs/opensource/how-open-source-projects-are-using-kani-to-write-better-software-in-rust/, November 2023.

[155] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 460–477, New York, NY, USA, 2017. Association for Computing Machinery.

[156] Emre Kültürsay, Mahmut Kandemir, Anand Sivasubramaniam, and Onur Mutlu. Evaluating STT-RAM as an energy-efficient main memory alternative. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 256–267, 2013.

[157] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., USA, 2002.

[158] Leslie Lamport. Industrial Use of TLA+. https://lamport.azurewebsites.net/tla/industrial-use.html, 2019.

[159] Leslie Lamport and Stephan Merz. Prophecy made simple. *ACM Trans. Program. Lang. Syst.*, 44(2), April 2022.

[160] William Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, dec 1992.

[161] Philip Lantz, Subramanya Dulloor, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. Yat: A validation framework for persistent memory software. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 433–438, Philadelphia, PA, June 2014. USENIX Association.

[162] Andrea Lattuada. Initiative: Ghost types and blocks. https://github.com/rust-lang/lang-team/issues/161, 2022.

[163] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Oded Padon, and Bryan Parno. Verus:

A practical foundation for systems verification. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 438–454, 2024.

[164] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying Rust programs using linear ghost types. In *Proceedings of the ACM Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 286–315, October 2023.

[165] Hayley LeBlanc, Jacob R. Lorch, Chris Hawblitzel, Cheng Huang, Yiheng Tao, Nickolai Zeldovich, and Vijay Chidambaram. PoWER never corrupts: Tool-agnostic verification of crash consistency and corruption detection. OSDI (to appear), 2025.

[166] Hayley LeBlanc, Shankara Pailoor, Om Saran K R E, Isil Dillig, James Bornholt, and Vijay Chidambaram. Chipmunk: Investigating crash-consistency in persistent-memory file systems. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, pages 718–733, May 2023.

[167] Hayley LeBlanc, Nathan Taylor, James Bornholt, and Vijay Chidambaram. SquirrelFS: using the Rust compiler to check file-system crash consistency. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 387–404, July 2024.

[168] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. *SIGARCH Comput. Archit. News*, 37(3):2–13, June 2009.

[169] Sergey Legtchenko, Ioan Stefanovici, Richard Black, Antony Rowstron, Junyi Liu, Paolo Costa, Burcu Canakci, Dushyanth Narayanan, and Xingbo Wu. Managed-retention memory: A new class of memory for the AI era, 2025.

[170] Nico Lehmann, Adam T. Geller, Niki Vazou, and Ranjit Jhala. Flux: Liquid types for Rust. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023.

[171] Rustan Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, pages 348–370, April 2010.

[172] Lenovo. memcached-pmem. https://github.com/lenovo/memcached-pmem.

[173] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009.

[174] Jialin Li, Andrea Lattuada, Yi Zhou, Jonathan Cameron, Jon Howell, Bryan Parno, and Chris Hawblitzel. Linear types for large-scale systems verification. pages 1–28, December 2022.

[175] Linux Developers. dm-log-writes. https://docs.kernel.org/admin-guide/device-mapper/log-writes.html.

[176] Bug 15910 - zero-length files and performance degradation, 2010. https://bugzilla.kernel.org/show_bug.cgi?id=15910.

[177] Linux Kernel Developers. Direct Access for files. https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[178] Linux Kernel Developers. Kernel Probes (Kprobes). https://www.kernel.org/doc/Documentation/kprobes.txt.

[179] Linux Kernel Developers. Uprobe-tracer: Uprobe-based event tracing. https://www.kernel.org/doc/html/latest/trace/uprobetracer.html.

[180] R. J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, December 1975.

[181] Sihang Liu, Suyash Mahar, Baishakhi Ray, and Samira Khan. PMFuzz: Test case generation for persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, pages 487–502, New York, NY, USA, 2021. Association for Computing Machinery.

[182] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. Cross-failure bug detection in persistent memory programs. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 1187–1202, March 2020.

[183] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. PMTest: A fast and flexible testing framework for persistent memory programs. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 411–425, April 2019.

[184] Jialun Lyu, Jaylen Wang, Kali Frost, Chaojie Zhang, Celine Irvene, Esha Choukse, Rodrigo Fonseca, Ricardo Bianchini, Fiodar Kazhamiaka, and Daniel S. Berger. Myths and misconceptions around reducing carbon embedded in cloud platforms. In *Proceedings of the 2nd Workshop on Sustainable Computer Systems*, HotCarbon '23, New York, NY, USA, 2023. Association for Computing Machinery.

[185] Jialun Lyu, Marisa You, Celine Irvene, Mark Jung, Tyler Narmore, Jacob Shapiro, Luke Marshall, Savyasachi Samal, Ioannis Manousakis, Lisa Hsu, Preetha Subbarayalu, Ashish Raniwala, Brijesh Warrier, Ricardo Bianchini, Bianca Schroeder, and Daniel S. Berger. Hyrax: Fail-in-Place server operation in cloud platforms. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 287–304, Boston, MA, July 2023. USENIX

Association.

[186] Sara McAllister, Yucong "Sherry" Wang, Benjamin Berg, Daniel S. Berger, George Amvrosiadis, Nathan Beckmann, and Gregory R. Ganger. FairyWREN: A sustainable cache for emerging Write-Read-Erase flash interfaces. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 745–764, Santa Clara, CA, July 2024. USENIX Association.

[187] Marshall Kirk McKusick and Gregory R. Ganger. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *1999 USENIX Annual Technical Conference (USENIX ATC 99)*, Monterey, CA, June 1999. USENIX Association.

[188] Marshall Kirk McKusick and T. J. Kowalski. Fsck - The UNIX** File System Check Program. https://docs-archive.freebsd.org/44doc/smm/03.fsck/paper.html, October 1996.

[189] Marshall Kirk McKusick, George Neville-Neil, and Robert N.M. Watson. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley Professional, 2nd edition, 2014.

[190] Sean McLaughlin, Georges-Axel Jaloyan, Tongtong Xiang, and Florian Rabe. Enhancing proof stability. Dafny Workshop 2024, 2024.

[191] memcached Developers. memcached. https://memcached.org/.

[192] Micron Technology, Inc. Micron Technology, Inc., and AgigA Tech Collaborate to Develop Nonvolatile DIMM Technology. https://investors.micron.com/news-releases/news-release-details/micron-technology-inc-and-agiga-tech-collaborate-develop, November 2012.

[193] Microsoft Corporation. Azure for US Government. https://azure.microsoft.com/en-us/explore/global-infrastructure/government.

[194] Microsoft Corporation. Azure Storage redundancy. https://learn.microsoft.com/en-us/azure/storage/common/storage-redundancy.

[195] Microsoft Corporation. Microsoft for financial services. https://www.microsoft.com/en-us/industry/financial-services/microsoft-cloud-for-financial-services.

[196] Microsoft Corporation. Microsoft OneDrive. https://www.microsoft.com/en-us/microsoft-365/onedrive/online-cloud-storage.

[197] Microsoft Corporation. Z3 Guide: Arithmetic. https://microsoft.github.io/z3guide/docs/theories/Arithmetic, 2024.

[198] Microsoft Corporation. Z3 Guide: Quantifiers. https://microsoft.github.io/z3guide/docs/logic/Quantifiers, 2024.

[199] E.L. Miller, S.A. Brandt, and D.D.E. Long. HeRMES: high-performance reliable MRAM-enabled storage. In *Proceedings Eighth Workshop on Hot Topics in Operating Systems*, pages 95–99, 2001.

[200] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, March 1992.

[201] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. CrashMonkey and ACE: Systematically testing file-system crash consistency. *ACM Transactions on Storage*, 15(2):1–34, April 2019.

[202] Steve Morgan. The World Will Store 200 Zettabytes Of Data By 2025. https://cybersecurityventures.com/the-world-will-store-200-zettabytes-of-data-by-2025/, 2024.

[203] Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *Proceedings of the International Conference on Automated Deduction (CADE)*, pages 625–635, July 2021.

[204] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 9583*, VMCAI 2016, page 41–62, Berlin, Heidelberg, 2016. Springer-Verlag.

[205] David Mulnix. Third Generation Intel Xeon Processor Scalable Family Technical Overview. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-xeon-processor-scalable-family-overview.html, June 2020.

[206] David Mulnix. Technical Overview Of The 4th Gen Intel Xeon Scalable processor family. https://www.intel.com/content/www/us/en/developer/articles/technical/fourth-generation-xeon-scalable-family-overview.html, July 2022.

[207] Madan Musuvathi, Shaz Qadeer, and Thomas Ball. CHESS: A systematic testing tool for concurrent software. Technical Report MSR-TR-2007-149, November 2007.

[208] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. An analysis of persistent memory use with WHISPER. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 135–148, New York, NY, USA, 2017. Association for Computing Machinery.

[209] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. RedLeaf: Isolation and communication in

a safe operating system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 21–39. USENIX Association, November 2020.

[210] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. AGAMOTTO: How persistent is your persistent memory application? In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 1047–1064, November 2020.

[211] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. Hyperkernel: Push-button verification of an OS kernel. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 252–269, October 2017.

[212] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services uses formal methods. *Communications of the ACM*, 2015.

[213] Liam O'Connor, Zilin Chen, Christine Rizkallah, Vincent Jackson, Sidney Amani, Gerwin Klein, Toby Murray, Thomas Sewell, and Gabriele Keller. Cogent: uniqueness types and certifying compilation. *Journal of Functional Programming*, 31:e25, 2021.

[214] ParaTools, Inc. Freja manual chapter 5.3: Non-temporal data. http://www.nic.uoregon.edu/~khuck/ts/acumem-report/manual_html/ch05s03.html, November 2016.

[215] 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 USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 433–448, October 2014.

[216] Soujanya Ponnapalli. *Minimizing I/O Bottlenecks to Achieve Scalable and High-Throughput Systems*. PhD thesis, University of Texas at Austin, Dec 2023.

[217] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and evolution of journaling file systems. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)*, Anaheim, CA, April 2005. USENIX Association.

[218] Matthew Prince, John Graham-Cunning, and Jeremy Hartman. Major data center power failure (again): Cloudflare code orange tested. https://blog.cloudflare.com/major-data-center-power-failure-again-cloudflare-code-orange-tested/, 2024.

[219] Jehan-François Pâris, Darrell D.E. Long, and S.J. Thomas Schwarz. Zero-maintenance disk arrays. In *2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing*, pages 140–141, 2013.

[220] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. HeMem: Scalable tiered memory management for big data applications and real NVM. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 392–407, New York, NY, USA, 2021. Association for Computing Machinery.

[221] Redis Ltd. Redis. https://redis.io/.

[222] Yujie Ren, Changwoo Min, and Sudarsun Kannan. CrossFS: A cross-layered Direct-Access file system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 137–154. USENIX Association, November 2020.

[223] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the IEEE Symposium on Logic in Computer Science (LICS)*, pages 55–74, July 2002.

[224] RocksDB Developers. RocksDB. https://rocksdb.org/.

[225] Rocq Development Team. Coq Reference Manual. https://coq.inria.fr/doc/V8.20.0/refman/.

[226] Rocq Development Team. Rocq. https://rocq-prover.org/, January 2025.

[227] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, feb 1992.

[228] Andy Rudoff. Deprecating the PCOMMIT instruction. https://www.intel.com/content/www/us/en/developer/articles/technical/deprecate-pcommit-instruction.html.

[229] Andy Rudoff. Persistent memory programming. *;login:*, 42(2):34–40, 2017.

[230] Andy Rudoff. Persistent memory on CXL. PM+CS Summit 2021, April 2021.

[231] Rust Developers. Type Layout. https://doc.rust-lang.org/reference/type-layout.html, 2024.

[232] Rust for Linux Developers. Rust for linux. https://rust-for-linux.com/.

[233] Rust Resources Team. Typestate programming. https://docs.rust-embedded.org/book/static-guarantees/typestate-programming.html.

[234] Rust Team. Unsafe rust. https://doc.rust-lang.org/book/ch20-01-unsafe-rust.html.

[235] Salesforce. Data Cloud for Financial Services. https://www.salesforce.com/financial-services/financial-services-data-platform/.

[236] Samsung. Samsung electronics unveils far-reaching, next-generation memory solutions at flash memory summit 2022. https://semiconductor.samsung.com/news-events/news/samsung-electronics-unveils-far-reaching-next-generation-memory-solutions-at-flash-memory-summit-2022/, August 2022.

[237] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash reliability in production: The expected and the unexpected. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 67–80, Santa Clara, CA, February 2016. USENIX Association.

[238] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM errors in the wild: a large-scale field study. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '09, page 193–204, New York, NY, USA, 2009. Association for Computing Machinery.

[239] Anand Lal Shimpi. Gigabyte's i-RAM: Affordable Solid State Storage. https://www.anandtech.com/show/1742, July 2005.

[240] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button verification of file systems via crash refinement. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 1–16, November 2016.

[241] Gopalan Sivathanu, Charles P. Wright, and Erez Zadok. Ensuring data integrity in storage: techniques and applications. In *Proceedings of the ACM Workshop on Storage Security and Survivability*, pages 26–36, November 2005.

[242] Snowflake, Inc. Snowflake's financial services data cloud. https://www.snowflake.com/en/resources/solution-brief/snowflakes-financial-services-data-cloud-2/.

[243] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, SE-12(1):157–171, 1986.

[244] Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, and Éric Tanter. First-class state change in Plaid. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, page 713–732, New York, NY, USA, 2011. Association for Computing Machinery.

[245] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 256–270, January 2016.

[246] Dan Swinhoe. Electrical distribution system failure causes outage at Microsoft data center. https://www.datacenterdynamics.com/en/news/electrical-distribution-system-failure-causes-outage-at-microsoft-data-center/, 2023.

[247] Symas Corporation. LMDB. http://www.lmdb.tech.

[248] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *;login:*, 41(1):6–12, 2016.

[249] Joseph Tassaroti, Ralf Jung, Nickolai Zeldovich, Upamanyu Sharma, Pierre Roux, Pierre-Marie Pédrot, and Tej Chajed. Perennial asynchronous disk prophecy model. https://github.com/mit-pdos/perennial/blob/master/src/goose_lang/ffi/async_disk_proph.v, 2024.

[250] The Open Group. The Open Group Base Specifications Issue 7. https://pubs.opengroup.org/onlinepubs/9699919799/, 2018.

[251] Tom's Hardware. Samsung's Memory-Semantic CXL SSD Brings a 20X Performance Uplift. https://www.tomshardware.com/news/samsung-memory-semantic-cxl-ssd-brings-20x-performance-uplift, August 2022.

[252] Hung-Wei Tseng, Laura Grupp, and Steven Swanson. Understanding the impact of power loss on flash memory. In *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 35–40, 2011.

[253] Aubrey Turner. How Big is the Cloud? https://www.pingidentity.com/en/resources/blog/post/how-big-is-the-cloud.html, 2022.

[254] Ubuntu Bugs LaunchPad. Bug #317781: Ext4 Data Loss. https://bugs.launchpad.net/ubuntu/+source/linux/+bug/317781?comments=all.

[255] United States Environmental Protection Agency. Used lithium-ion batteries. https://www.epa.gov/recycle/used-lithium-ion-batteries, March 2024.

[256] Verus Developers. Verus Tutorial and Reference. https://verus-lang.github.io/verus/guide/, 2024.

[257] Verus Developers. Verus Tutorial and Reference: forall and triggers. https://verus-lang.github.io/verus/guide/forall.html, 2024.

[258] Doug Voigt. Persistent memory atomics and transactions. https://www.snia.org/educational-library/persistent-memory-atomics-and-transactions-2017, October 2017.

[259] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*,

ASPLOS XVI, page 91–104, New York, NY, USA, 2011. Association for Computing Machinery.

[260] An-I Wang, Peter Reiher, Gerald Popek, and Geoffrey H. Kuenning. Conquest: Better performance through a Disk/Persistent-RAM hybrid file system. In *2002 USENIX Annual Technical Conference (USENIX ATC 02)*, Monterey, CA, June 2002. USENIX Association.

[261] Michael Wu and Willy Zwaenepoel. envy: a non-volatile, main memory storage system. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, page 86–97, New York, NY, USA, 1994. Association for Computing Machinery.

[262] XFS Developers. XFS Self Describing Metadata. https://www.kernel.org/doc/Documentation/filesystems/xfs-self-describing-metadata.txt, 2024.

[263] XFS Developers. XFS Filesystem Documentation. https://docs.kernel.org/filesystems/xfs/index.html, 2025.

[264] xfstests Developers. xfstests. https://github.com/kdave/xfstests.

[265] Jian Xu, Juno Kim, Amir Saman Memaripour, and Steven Swanson. Finding and fixing performance pathologies in persistent memory software stacks. In Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck, editors, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 427–439. ACM, 2019.

[266] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 323–338, February 2016.

[267] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. NOVA-Fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 478–496, October 2017.

[268] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble page management for tiered memory systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 331–345, New York, NY, USA, 2019. Association for Computing Machinery.

[269] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 169–182, February 2020.

[270] Junfeng Yang, Can Sar, and Dawson Engler. EXPLODE: A lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, page 131–146, USA, 2006. USENIX Association.

[271] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 273–287, December 2004.

[272] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. MatrixKV: Reducing write stalls and write amplification in LSM-tree based KV stores with matrix container in NVM. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 17–31. USENIX Association, July 2020.

[273] Pamela Zave. Using lightweight modeling to understand Chord. *SIGCOMM Comput. Commun. Rev.*, 42(2):49–57, March 2012.

[274] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. ChameleonDB: a key-value store for Optane persistent memory. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 194–209, New York, NY, USA, 2021. Association for Computing Machinery.

[275] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S. Yang, Bill W Zhao, and Shashank Singh. Torturing databases for fun and profit. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 449–464, Broomfield, CO, October 2014. USENIX Association.

[276] Mai Zheng, Joseph Tucek, Feng Qin, and Mark Lillibridge. Understanding the robustness of SSDs under power fault. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 271–284, San Jose, CA, February 2013. USENIX Association.

[277] Diyu Zhou, Vojtech Aschenbrenner, Tao Lyu, Jian Zhang, Sudarsun Kannan, and Sanidhya Kashyap. Enabling high-performance and secure userspace NVM file systems with the Trio architecture. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 150–165, 2023.

[278] Yuan Zhou. Accelerating Redis with Intel Optane DC Persistent Memory. https://ci.spdk.io/download/2019-summit-prc/02_Presentation_13_Accelerating_Redis_with_Intel_Optane_DC_Persistent_Memory_Dennis.pdf, 2019. SPDK, PMDK and Intel VTune Amplifier PRC Summit.