High Velocity Operating Systems Development

Samantha Miller

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

Doctor of Philosophy

University of Washington
2023

Reading Committee:
Thomas Anderson, Chair
Danyang Zhuo
Arvind Krishnamurthy

Program Authorized to Offer Degree:
Paul G. Allen School of Computer Science & Engineering

University of Washington

Abstract

High Velocity Operating Systems Development

Samantha Miller

Chair of the Supervisory Committee:
Thomas Anderson
Paul G. Allen School of Computer Science & Engineering

Development velocity is critical to modern software development, particularly in the cloud, because it allows developers to ship features to users quickly. Despite Linux being the most commonly used cloud operating system, Linux development velocity is lagging, limiting its ability to adapt to new workloads and new hardware. High development velocity in operating systems is difficult due to the complexity of the code, the ease of introducing bugs, the difficulty of finding and fixing these bugs, and the disruption caused by kernel upgrades. Improving kernel development velocity introduces challenges not fully addressed by past research: limiting bugs and improving the speed of deployment and debugging without sacrificing generality, compatibility, or performance. Past efforts to improve safety by eliminating or isolating bugs has often come at a significant cost in terms of those requirements.

We study bugs found in Linux to understand their causes and to suggest potential methods to reduce kernel bugs. We discover that around half of all security critical bugs are low level bugs that do not depend on the module logic. These could be prevented using language level compile-time safety checks, such as those performed by safe Rust.

In this thesis, we show that it is possible to improve development velocity in commercial operating systems. We use Rust's compile time safety checks to eliminate bugs without performance overhead or lack of generality. We also enable live upgrade and userspace debugging of kernel modules. We design, implement and evaluate Bento and Enoki, frameworks for improved development velocity in the Linux kernel. Bento is a framework for high velocity Linux kernel file systems that enables safe Rust, userspace debugging, and live upgrade. A file system implemented with Bento achieves performance comparable to Linux's default file system and can be upgraded with only 15ms of service downtime. Enoki enables high velocity Linux kernel schedulers written in safe Rust. It supports record and replay debugging, live upgrade, and bidirectional userspace communication. Enoki supports a variety of schedulers with performance on par with baseline Linux and research schedulers.

Table of Contents

Copyright ii

Abstract iii

List of Figures vii

List of Tables viii

Glossary x

Acronyms xii

1 Introduction 1

1.1 Thesis Statement and Contributions 5

1.2 Published Materials 7

1.3 Outline of Thesis 7

2 Background 9

2.1 Motivation 9

2.2 Development Velocity 11

2.3 Case Studies: File Systems and CPU Schedulers 14

2.4 Existing Mechanisms for Velocity in Linux 14

2.4.1 Linux Core Interfaces . . . 16
2.4.2 Userspace Trampolining . . . 17
2.4.3 Kernel Bypass . . . 19
2.4.4 Multicore Microkernels . . . 20
2.4.5 eBPF . . . 21
2.5 Relevant Background Knowledge . . . 22
2.5.1 Rust Primer . . . 22
2.5.2 Ext4 . . . 23
2.5.3 CFS . . . 24

3 Linux Kernel Bug Analysis 25
3.1 Summary of Prior Results . . . 27
3.2 Module Bug Analysis . . . 28
3.3 Linux CVE Analysis . . . 31
3.4 Summary . . . 33

4 Bento: High Velocity Linux Kernel File Systems 35
4.1 Approach . . . 37
4.2 Design . . . 39
4.3 Safe Interfaces . . . 39
4.3.1 Interacting with VFS . . . 39
4.3.2 Interacting with Kernel Services . . . 40
4.4 File System Upgrade . . . 43
4.5 Userspace Debugging . . . 44
4.6 Limitations . . . 45
4.7 Implementation . . . 46
4.8 Experiences with Bento . . . 47
4.9 Evaluation . . . 49

4.9.1 Experimental setup . . . 49
4.9.2 Microbenchmarks . . . 50
4.9.3 Application Workloads . . . 52
4.9.4 Live Upgrade . . . 55
4.10 Summary . . . 56

5 Enoki: High Velocity Linux Kernel Schedulers 57
5.1 Motivation and Approach . . . 59
5.2 Enoki . . . 62
5.2.1 Safe Interfaces . . . 62
5.2.2 Live Upgrade . . . 66
5.2.3 Custom Scheduler Hints . . . 68
5.2.4 Record and Replay . . . 68
5.3 Implementation . . . 70
5.3.1 Enoki . . . 70
5.3.2 Schedulers . . . 71
5.4 Evaluation . . . 74
5.4.1 Setting . . . 74
5.4.2 Microbenchmarks . . . 75
5.4.3 WFQ Scheduler Applications . . . 76
5.4.4 Locality Aware Scheduler . . . 81
5.4.5 Arachne Scheduler . . . 82
5.4.6 Live Upgrade . . . 83
5.4.7 Record and Replay . . . 84
5.5 Discussion . . . 84
5.6 Summary . . . 85

6 Related Work **87**

6.1 Using Safe Languages for Kernel Development 87

6.2 Software Fault Isolation 88

6.3 Moving Kernel Features to Userspace 89

6.4 OS Live Upgrade 90

6.5 Record and Replay 90

7 Conclusion and Future Work **91**

7.1 Future Work 92

7.1.1 Bento 92

7.1.2 Enoki 93

7.1.3 Extending to Other Subsystems 94

7.2 Conclusion 95

References **97**

List of Figures

3.1 Newly reported Linux CVEs per year. 31
3.2 Cumulative distribution function of reported CVEs in ext4 since release. . 32
3.3 Number of bugs fixed each year per line of code in three Linux file systems. 32

4.1 Design of Bento. 38
4.2 Bento performance on Filebench benchmarks. 53
4.3 Bento performance on application workloads. 54
4.4 Bento performance on Redis. 54
4.5 Bento create+delete performance during an upgrade. 55
4.6 Bento write performance during an upgrade. 56

5.1 Design of Enoki. 61
5.2 Enoki vs. ghOSt tail latency. 79
5.3 Enoki vs. ghOSt tail latency co-located with a batch application. 80
5.4 Enoki vs. ghOSt CPU share co-located with a batch application. 81
5.5 Performance of the Enoki Arachne scheduler. 83

List of Tables

2.1 Comparison of techniques for Linux kernel development velocity. 15

3.1 Low-level bugs in three Linux file systems. 29

4.1 The Bento file operations API. 41
4.2 The Bento kernel services API. 42
4.3 Bento performance on Filebench microbenchmarks. 51

5.1 The API of the EnokiScheduler Trait. 65
5.2 Lines of code for the Enoki components. 71
5.3 Performance of schedulers on the perf pipe benchmark. 74
5.4 Performance of schedulers on the schbench benchmark. 75
5.5 Performance of the Enoki WFQ scheduler on NAS Benchmarks. 77
5.6 Performance of the Enoki WFQ scheduler on Phoronix Multicore benchmarks. 78
5.7 Performance of the Enoki Locality-Aware scheduler. 82

Glossary

container a software object that provides an isolated view of an environment to a program and its dependencies.

continuous integration/continuous deployment a software development process where incremental changes are regularly and automatically compiled, tested, and deployed.

crash consistency a guarantee provided by storage systems that the system will be in a consistent state after a crash.

deployment velocity the rate at which new versions of a service can be made available.

development velocity the rate at which new code can be added to an existing product, including time programming, testing, and debugging.

extent based file system a file system that tracks file data blocks by recording the start and end of block ranges rather than recording every block separately.

immutable reference a memory reference that can only be used for reading.

journaling file system a file system that records in-process operations in a persistent log for crash consistency.

kernel bypass a technique where hardware virtualization is used to give userspace processes direct access to hardware.

lifetime a language feature where the compiler tracks the scope of variables to ensure that all references to the memory are valid when they are used.

linear types a type system where all objects are used exactly once.

microservice architecture a system architecture where jobs are composed from many small services.

mutable reference a memory reference that can be used for reading and writing.

oops a Linux kernel state that immediately precedes either a system crash or the loss of some system services.

ownership a language feature where all allocated memory is associated with an owner object and the memory is freed when the object goes out of scope.

quad-level cell solid state drive a high capacity solid state drive storing four bits per cell.

resource acquisition is initialization a language feature where resources for a variable are acquired and initialized when the variable is created and released when the varaible goes out of scope.

run-queue the queue of tasks to be run on a core.

serverless computing a computing architecture where resources are allocated on demand based on usage.

triple-level cell solid state drive a high capacity solid state drive storing three bits per cell.

zoned namespace solid state drive a type of solid state drive where the capacity is divided into zones that must be written sequentially, providing lower write amplification.

Acronyms

API Application Programming Interface.

CDF Cumulative Distribution Function.

CFS Completely Fair Scheduler.

CI/CD Continuous Integration/Continuous Deployment.

CPU Central Processing Unit.

CVE Common Vulnerabilities and Exposures.

DoS Denial of Service.

DPDK Data Plane Development Kit.

eBPF Extended Berkeley Packet Filter.

ext4 Fourth Extended File System.

FIFO First In First Out.

FPGA Field Programmable Gate Array.

FUSE File System in Userspace.

gdb GNU Debugger.

I/O Input/Output.

ID Identifier.

IP Internet Protocol.

IPC Inter-Process Communication.

NIC Network Interface Card.

NUMA Non-Uniform Memory Access.

QLC Quad-level Cell.

RAII Resource Acquisition is Initialization.

RDMA Remote Direct Memory Access.

RPC Remote Procedure Call.

SFI Software-Based Fault Isolation.

SLA Service Level Agreement.

SPDK Storage Performance Development Kit.

SSD Solid State Drive.

TCP Transmission Control Protocol.

TPU Tensor Processing Unit.

VFS Virtual File System.

WFQ Weighted Fair Queuing.

Acknowledgements

I would like to express my gratitude to everyone who helped me throughout my Ph.D. I really value the opportunities I have had to learn from and work with so many wonderful people here. I cannot imagine having a better Ph.D. experience, and I owe all of that to the amazing people I have had around me through the last five years.

I would first like to thank my advisors Dr. Thomas Anderson and Dr. Danyang Zhuo. I cannot express enough how happy I am to have worked with both of you. You have both taught me so much and helped me grow as a person and as a researcher over the last five years. Your guidance and support have been invaluable to me, and I feel so lucky to have been able to work with you. I hope I can make you proud as I continue my career, remembering the lessons you taught me.

I would also like to thank Dr. Simon Peter, who advised my undergraduate research thesis and has continued to be a source of wisdom throughout my Ph.D. With no exaggeration, I would not be here without your support and belief in me.

Thanks to my other collaborators over my Ph.D. Dr. Ang Chen has contributed so much knowledge to my projects. It has been a pleasure to work with you. Dr. Arvind Krishmurthy has been a constant source of support within the lab and served as a member of my reading committee. Dr. Jon Howell has been wonderful to work with and has helped guide me through the Ph.D process and beyond.

Thanks also to the undergraduate students I had the opportunity to work with, Frank Chen, Ryan Jennings, Anirudh Kumar, and Tanay Vakharia. I am glad I have been able to work with you all. I hope you have all been able to learn from me, as I have been able to learn from you.

I owe so much to the other graduate students I have shared my time with in the Syslab. You have all been there for me when I needed someone to talk to or laugh with or cry with. Leaving graduate school is bittersweet knowing that I will be leaving our cozy lab and the friendships we have made. Whether we have worked together on a project or not,

I would not have made it through my Ph.D. if it were not for you. There are too many people to name, but I want to specifically thank Niel Lebeck, Kaiyuan Zhang, Jialin Li, Ashlie Martinez, Henry Schuh, Katie Lim, Lequn Chen, Kevin Zhao, and Priyal Suneja.

Lastly, thanks to my parents, sister, and partner for their support over these five years. Things have not always been easy during this time, and I owe you all so much for keeping me alive and sane this whole time.

Chapter 1

Introduction

Cloud computing is changing the modern computing environment. With cloud computing, application developers can deploy their programs to run in datacenters owned by large companies with an abundance of capacity. This enables even small players to easily manage and scale up their applications in response to user demand without having to deploy and maintain their own on-premises server clusters. Application developers can leverage cloud services to simplify development, relying on services, scalability, and fault tolerance managed by the cloud provider. Since cloud providers manage all physical hardware and customers only pay for what they use, customers can reduce capital costs and move capital expenditure risk onto cloud providers. The cloud market has grown into an economic behemoth, with tens of billions of dollars spent on cloud infrastructure each quarter.

Cloud computing enables a new model of software development and deployment. Cloud products and services must be always available [15]. Cloud systems provide availability SLAs, often promising 99.9% or 99.99% uptime, leaving roughly 50-500 minutes of potential downtime in a year [60, 11, 12]. This uptime requirement includes updates. Perhaps surprisingly, these systems also see rapid development, with new features being deployed frequently—often at least once a week [49, 44, 22]. If services are shut down for every upgrade, it would be impossible to maintain the high availability promised by these systems and expected by its users. Upgrades must be deployed live to the running application.

To manage this rate of updates, cloud computing has ushered in age of continuous integration and continuous delivery/deployment (CI/CD). Traditionally, releases were managed manually. Engineers would select a set of commits to release. These commits would be tested against a test suite, then built and deployed, often after months of testing [44, 49]. CI/CD systems automatically manage the testing and release of new updates. Commits

are automatically batched by the system and pushed to the testing framework. If the tests pass, the CI/CD system automatically begins deploying the changes, monitoring for regressions along the way. If a regression is detected, the rollout is stopped and engineers are notified [44, 22]. CI/CD allows products and services to receive feature updates continuously and makes the design-code-debug-release faster and simpler.

Increased development velocity provides a number of advantages. 1) Rapid development and deployment enable providers to deliver new features to users earlier. These features can support specialized use cases for applications to better meet customer needs or improve performance. 2) More frequent releases means smaller updates between releases. The less code changed between updates, the more likely the update will not introduce bugs and the easier it is to debug any bugs that do appear. 3) Rapid releases enable prompt responses to security vulnerabilities. Security vulnerabilities often need to be patched as soon as they are announced to fix vulnerable code.

Rapid development velocity is also useful beyond the cloud. Any large scale deployment can take advantage of these techniques to make the development and deployment process more efficient. Because of these advantages, methods for increased development velocity have spread widely across datacenter and server deployments. This is currently the de facto way that large scale applications in datacenters are managed [44, 136].

However, this push for rapid development has not fully caught up to operating systems, despite this being a long-standing goal of OS research [137, 2, 55, 21, 90]. In Linux, the most widely used cloud server operating system, release cycles are still measured in months [93]. Larger systems, such as file systems, can take years to mature. Linux's BtrFS has been in development for more than 10 years, but still has unstable features. Zoned Namespace SSDs were introduced three years ago, but there has only recently been a proposal for a ZNS file system (SSDFS), and it is still highly unstable [146]. In networking and scheduling, there has been substantial work bypassing the kernel entirely for faster and easier development [77, 99, 67, 75, 114]. Kernel bypass requires rewriting the entire stack and sacrifices coordination with the rest of the kernel, introducing manageability problems [139].

Slow Linux development can be attributed to several factors. Linux has a large code base with relatively few mechanisms to prevent misuse, with complicated internal interfaces that are easily misused. Combined with the inherent difficulty of programming correct concurrent code in C, this means that new code is very likely to have bugs. Because there is no isolation between kernel modules, these errors often have non-intuitive effects and are difficult to track down. The lack of kernel-level debuggers and kernel testing frameworks makes this worse. We study in Chapter 3 the prevalence of bugs in Linux. The

restricted and different kernel programming environment also limits the number of trained developers. Finally, upgrading a kernel module requires either rebooting the machine or restarting the relevant module; neither is invisible to applications. In the cloud setting, this forces kernel upgrades to be batched to meet cloud-level availability goals, further slowing the design-code-debug-release cycle.

Linux and the wider development community have taken some steps that improve development velocity. Some types of subsystems, such as file systems and CPU schedulers, already support multiple implementations. New implementations can be written against the interfaces provided by these subsystems. However, these new implementations still suffer from the difficult programming environment in the kernel. Some subsystems can be implemented in userspace, using a user-to-kernel trampoline approach, such as FUSE (Filesystem in Userspace) [56] for file systems. Calls to the subsystem are forwarded from the kernel to a userspace module and back through the kernel to the user, isolating the module but potentially adding significant overhead [152]. eBPF (the Extended Berkeley Packet Filter) [53] provides a mechanism for running custom user code in the Linux kernel. To ensure safety of the kernel, eBPF programs are heavily restricted. For example, users must choose from a set of predefined locations, must use only provided data structures, and cannot use unbounded loops. In exchange, eBPF programs cannot interfere with correct functioning of the kernel. Due to these restrictions, it is difficult, or sometimes impossible, to write complicated kernel functionality using eBPF. Despite this, eBPF has been growing in popularity due to the benefits of safe kernel development. Proposals are regularly made to add functionality to eBPF or expand it to new areas [149, 40, 162].

There have also been research efforts to improve aspects of operating systems velocity. Some technologies reduce bugs or the scope of bugs. These do not specifically target development velocity but improve it regardless. Some projects have built operating systems in safe languages to prevent bugs [21, 43, 86, 68], but none of these apply to existing operating systems or directly address other aspects of development velocity. Software-based fault isolation (SFI) [153] and similar techniques [31] specifically focus on trust, isolating bugs to a module or region of memory to reduce the impact of bugs but not preventing them. Microkernel designs [2, 90] and moving modules to userspace [56, 67] isolate potentially buggy code from affecting the kernel. These enable use of userspace development technologies. However, this can introduce performance overhead [152]. Other technologies, such as those for live upgrade, improve velocity when handling errors, but are not intended for live upgrade of entire portions of the operating system. These tools provide fast, transparent update of commodity operating systems, but these do not address the difficulty of writing, debugging, and testing the code being deployed. They generally target small bug fixes and security updates to individual functions rather than whole modules or

feature updates [9, 113, 122, 118].

To support rapid development of operating systems code, we believe that we need to provide several properties to support rapid development velocity:

- **Bug Scope Reduction:** In a complex, monolithic C Linux kernel, bugs are easy to write and can have severe, non-local consequences. Developers must be very careful when writing new code to make sure they will not introduce new bugs and reduce the stability of the kernel. Limiting the scope and frequency of bugs will reduce the number of bugs introduced into a working system, allowing developers to write code more quickly with more confidence. Any bugs in new code should be isolated, as much as possible, to applications or containers that use the new code to limit the impact of the code change on the stability of the system. Unlike some prior work [68], we are interested in doing this in the context of existing operating systems.

- **Live upgrades:** Having to reboot the machine or kill or restart running processes to upgrade kernel code limits the rate of deployment of new updates. Live upgrade of kernel components would enable faster deployment cycles. Deployment of new code should require no more than a brief outage of the applications using that kernel service.

- **Userspace testing and debugging:** Many testing frameworks and debugging tools do not work in the kernel. This limits kernel developers' ability to find and fix issues, slowing development. Without access to a wide range of testing and debugging tools, developers are more cautions about deploying code. New code should be able to be debugged and tested with general purpose userspace tools so bugs can be found and fixed easily.

- **Generality:** New kernel code and kernel upgrades can require a wide range of functionality. Sometimes changes are simple bug fixes or security patches, but other times, developers want to introduce entirely new data structures or whole modules with complex functionality. The system should support a large variety of designs of new code.

- **Compatibility:** Requiring significant changes to existing applications or operating systems will greatly reduce the rate of adoption of new code. No or few changes should be required to existing systems and binaries to reduce the friction of adopting new code.

- **Performance:** Performance is critical for many kernel components. Any system that significantly impacts performance will not be useful for many use cases. Performance should be similar to that of the same functionality implemented in a less agile system.

Note that this does not address problems related to whether someone should trust the developer of the new code. Our trust model assumes that developers are well meaning but potentially clumsy. Our work tries to prevent accidental harm from new code but does not address malicious insider attacks from those writing the new code.

1.1 Thesis Statement and Contributions

Our hypothesis is that we can improve development velocity of commercial operating systems without sacrificing generality, compatibility and performance.

We believe that using Rust can provide a basis of a new approach to operating systems development velocity. Rust is a relatively new programming language for safe, high performance development of low level code. Rust provides a strong type system based on linear types to catch a wide class of bugs, including concurrency errors, at compile time without significant additional runtime performance overhead. In particular, Rust is type safe without a separate mark and sweep garbage collector. In the safe subset of Rust, bugs like `NULL` pointer dereferences, out of bounds accesses, and data races are prevented by the compiler. While safe Rust does eliminate some bug-free designs, most designs and data structures can be written in safe Rust. Using Rust, we can reduce bugs without significantly impacting generality.

The contributions of this thesis are threefold: 1) we categorize bugs leading to slow velocity, 2) we build a framework for higher velocity development of file systems within the Linux kernel, and 3) we extend and supplement this framework to support CPU schedulers. To this end, we explore several questions.

What kinds of bugs exist in deployed Linux releases today? We analyze bug fixes from several Linux kernel modules and categorize them by the type of bug. We find that around half of the bugs we analyze are low-level bugs, and that many could be addressed with compile-time memory safety checks. Specifically, the safety checks provided by safe Rust code would be able to prevent nearly half of the bugs we analyze. The remaining bugs are mostly high-level logic bugs that could not be prevented with generic mechanisms. Our results also show that for several widely used Linux file systems, the number of bugs

per line of code has stayed relatively stable over time, despite increased effort to identify and fix bugs [1, 41, 78, 79, 83, 94, 144].

Is high velocity development of kernel file systems possible? There are many new proposals for new file systems designs, reacting to new user workloads or new hardware (e.g., low-latency SSDs, persistent memory, and zoned namespace SSDs) [84, 154, 146]. These file systems are often not implemented in the kernel due to the difficulty of introducing new large, complex pieces of code into the kernel.

We design and implement Bento [102], a framework for high velocity Linux kernel file systems. Bento enables kernel file systems written entirely in safe Rust, the subset of Rust that guarantees memory safety, and ensures that safety is maintained across the boundary with the unsafe C code in the rest of Linux. Bento also enables userspace debugging and live upgrade of file systems. File systems implemented with Bento cannot cause the kernel to panic, though the file system can still deadlock or have logic bugs. Logic bugs do not disrupt execution of the rest of the kernel, only affecting the file system itself and applications using it. We show that Bento file systems can be written, debugged, and deployed more easily than traditional kernel file systems. We implement a file system using Bento, and found that it performs competitively against ext4, the default Linux file system. We show that the file system can be upgraded to include file provenance by changing only a few lines of code and only 15ms of downtime.

Is high velocity development of kernel schedulers possible? Scheduling algorithms can have a significant impact on the overall runtime of multi-threaded jobs, particularly in cloud environments where jobs often spawn many tasks and each task can be very short [16]. Heterogeneous hardware and energy aware operation are also adding complexity to scheduling decisions [14, 15, 85, 112]. Linux currently only supports three main schedulers, and these must all be compiled into the kernel source instead of implemented as dynamically replaceable modules.

Supporting scheduler updates introduces additional challenges not faced in Bento. File systems execute behind the page cache, so any added latency for file reads and writes imposed by system support for velocity is not directly visible to applications. This is not true for schedulers, which execute in-line with application code. Additionally, schedulers execute often and must be highly concurrent. To provide performance parity with the baseline kernel, the performance margins for a scheduler framework are much tighter than for file systems.

We develop Enoki, a framework for high velocity development of Linux kernel schedulers. As with Bento, Enoki supports writing safe Rust schedulers where bugs have less effect on the rest of the kernel. It also enables live upgrade and record and replay debugging of the schedulers. To quantify the performance impact, we use Enoki to implement several schedulers and evaluate these on a variety of workloads against other kernel and userspace schedulers. A scheduler implemented with Enoki intended to mimic the standard Linux CFS CPU scheduler has similar performance to it across a suite of standard benchmarks.

1.2 Published Materials

Parts of the thesis have been published in peer-reviewed settings:

- Samantha Miller, Kaiyuan Zhang, Danyang Zhuo, Shibin Xu, Arvind Krishnamurthy, Thomas Anderson. **Practical Safe Linux Kernel Extensibility.** *Proceedings of the 17th Workshop on Hot Topics in Operating Systems (HOTOS), 2019.*
- Samantha Miller, Kaiyuan Zhang, Mengqi Chen, Ryan Jennings, Ang Chen, Danyang Zhuo, Thomas Anderson. **High Velocity Kernel File Systems with Bento.** *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST), 2021.*
- Jialin Li, Samantha Miller, Danyang Zhuo, Jon Howell, Thomas Anderson. **An Incremental Path Towards a Safe OS Kernel.** *Proceedings of the 18th Workshop on Hot Topics in Operating Systems (HOTOS), 2021.*
- Samantha Miller, Anirudh Kumar, Tanay Vakharia, Ang Chen, Danyang Zhuo, Thomas Anderson. **Enoki: High Velocity Kernel Scheduler Development.** *Proceedings of the Nineteenth European Conference on Computer Systems (Eurosys), 2024. (to appear)*

1.3 Outline of Thesis

The rest of the thesis is organized as follows:

Chapter 2 expands on the motivation, defines development velocity, describes our focus on file systems and CPU schedulers, further discusses existing approaches, and provides

background on the Rust programming language and the default file system and CPU scheduler in Linux. In Chapter 3, we analyze types and causes of bugs and vulnerabilities in the Linux kernel. Chapter 4 presents our work on Bento, a framework that enables high velocity development of Linux kernel file systems. In Chapter 5, we discuss our work on Enoki, a framework to support high velocity development of Linux kernel schedulers. Chapter 6 describes related work. Chapter 7 concludes the thesis and discusses future work.

Chapter 2

Background

As background, we discuss several motivating factors behind our approach and provide context for some related efforts. First, we more thoroughly discuss the trends leading to more development pressure on the operating system. Next, we more precisely define development velocity and describe our approach to providing it. We then discuss why file systems and CPU schedulers are useful case studies. We also provide background on existing Linux frameworks for improving development velocity, and why they do not fully address the issue. We conclude with a discussion of topics that will be useful for later chapters, specifically Rust, ext4 (the default Linux file system), and CFS (the default Linux CPU scheduler).

2.1 Motivation

The cloud, and data centers more broadly, have been seeing accelerating changes in the computing landscape, both from new hardware designs and new application demands. These put increasing pressure on the operating system to react quickly to keep pace. This increased pace of development is itself causing a push towards more formal methods to catch bugs more quickly and automatically.

From below, new hardware is being developed with different interfaces and behavior than traditionally assumed by operating systems. In the world of storage, there are Zoned Namespace and Triple- and Quad-Level Cell SSDs (solid state drives) with better performance and different workload characteristics than older disks [103, 70]. Unlike earlier systems that tied the file system to a physical device, these hardware devices offer tradeoffs,

benefitting from file system management of the storage hierarchy. In the cloud, storage is often disaggregated [87]. Distributed file systems provide unified file systems across disaggregated storage [8].

CPUs are also becoming increasingly heterogeneous. Non-uniform memory access (NUMA) cores have different memory access latency for different cores, complicating compute and data placement within a machine [126, 112]. Process cache affinity and tiered memory further increase the variability of memory access times [129]. Energy awareness has led to the development of CPUs with low power cores for more optimal energy use [14]. "Turbo" frequencies allow some cores to run at a higher frequency, allowing higher performance for CPU intensive tasks [5, 6]. If developers want to ensure maximum performance, they must tune the CPU scheduler's task placement [85].

Other parts of server hardware are also experiencing change. In networking, network interface cards are getting faster, approaching 400 Gbps, and support new functionality, such as RDMA, protocol offload, and programmable hardware [45]. Compute hardware is growing to include customizable hardware like Field Programmable Gate Arrays (FPGAs) and use case specific accelerators, such as Tensor Processing Units (TPUs) [73, 33].

From above, new use cases are arising with different demands than traditional applications. For storage, large data intensive workloads of cloud applications and data pipelines operate at a massive scale. These generate many files across many machines within one or more data centers. These files can contain user data from cloud customers, so data integrity, security and provenance are essential.

CPU scheduling is also seeing new pressure from application trends. Microservices often spawn many small, microsecond scale tasks with wide fanout [69]. This puts a premium on low overhead scheduling of tasks on the critical path. Serverless computing pushes server resource management of small, event-driven workloads onto cloud providers, who try to compact these jobs onto a small number of servers as much as possible [32]. Energy is becoming increasingly important in the datacenter as energy costs rise, where optimizing energy usage is often different from optimizing for performance [15].

In order to effectively support these changes, the operating system must be changed as well. To make efficient use of new hardware designs, operating system primitives must be updated to understand new hardware features, capabilities, and performance. To support new application trends, the operating system must change hidden assumptions about how user applications behave. Without flexibility in the operating system, developers cannot react to these changes effectively.

However, reacting to this pressure is difficult without addressing operating systems development velocity. One major component is the prevalence of bugs. As we will show in

Chapter 3, over time the number of bugs in Linux has roughly kept pace with the number of lines of code. Users rely on the correctness of the operating system for reliability, availability, and security. If adding code implies adding bugs, the rate of new features in Linux will be limited.

2.2 Development Velocity

Development and deployment velocity refers to the rate at which programmers can develop and deploy new code. Fast development velocity leads to a rapid design-code-debug-release cycle, allowing developers to use their time efficiently as bugs are fixed as soon as possible after being introduced. Achieving fast develop velocity requires all parts of the process be efficient; designing and writing code should be simple, testing should be automated, debugging should be straightforward, and releasing code should be incremental and seamless.

Continuous integration and continuous delivery/deployment (CI/CD) are commonly used today to increase development velocity in many software applications and systems, other than operating systems [22, 44, 136]. In a CI/CD system, code changes are automatically batched and sent to a testing framework. The testing framework runs a barrage of tests on the changes. If the tests pass, the changes are often deployed on a canary test bed that mirrors real deployments. If no issues are discovered in the canary testing, the changes are then deployed to production, starting with a small percentage of machines and progressively rolled out. Built in monitoring checks for any issues during the deployment and, if any are discovered, pauses the rollout and alerts engineers. The whole process is done automatically and does not require input from developers unless issues are discovered.

CI/CD significantly increases development velocity in several ways. It enables constant releases of new features. Some companies using CI/CD report deploying new changes multiple times a day, every day [136, 44]. More frequent testing and deployment of changes also makes debugging easier. With fewer changes to inspect, it is easier for developers to find the root cause of bugs introduced between two releases. Any change was made recently, making it more likely the developer will remember their intent. The similarity between test deployments (especially canary) and production deployments also aids debugging. Bugs are more likely to show up before the production rollout and the test deployment can be instrumented for debugging. CI/CD also makes developers more efficient at other parts of their job by freeing time that would otherwise be spent on managing the release [44]. This time can instead be spent on developing new features to be deployed in the next cycle.

Development velocity is difficult to measure, and finding good ways to directly evaluate development velocity is its own area of research. It can be deceptively difficult to accurately

measure the rate at which new code can be written. Many factors can affect the rate at which code is written, such as the complexity of the code and the existence of reference implementations. Measuring the correctness of the code by the number of bugs in different implementations can give insight into different approaches. However it suffers from similar concerns as looking at the rate of writing code. Usability studies provide useful data about users experiences, but can be difficult to perform due to lack of resources. This is particularly true in regards to kernel code, where there are relatively few experienced developers, and the main commercial operating system is open source and free. Few companies have an incentive to improve Linux velocity.

Instead of trying to directly measure development velocity, we identify properties that we believe are important for development velocity in operating systems and aim to satisfy those properties.

Bug Scope Reduction: Bugs affect development velocity by reducing the time developers have to develop new features. This is particularly true in operating systems where bugs are can have severe consequences on parts of the kernel unrelated to the module where the change is made. Bugs in concurrent code can be highly dependent on context, only appearing under specific circumstances that can be difficult to reproduce. Operating systems developers must therefore slowly and carefully write, test, and deploy their code to reduce the rate of introducing new bugs into the system. Reducing bugs and the scope of potential consequences for those bugs can free developers to develop code more quickly and deploy code with more confidence. Eliminating some classes of bugs reduces the burden for testing and debugging. Limiting the scope of bugs' consequences reduces non-local effects, making bugs easier to find, and prevents bugs in new code from affecting applications that do not use the feature but are co-located by cloud software onto a server with an application that does use the feature.

Live upgrade: Continuous deployment is crucial to modern rapid development velocity. Live upgrade provides the ability to upgrade a service without shutting down the product or service or any of the applications using it. Live upgrade enables continuous deployment of new versions to users without requiring downtime during the upgrade. For operating systems, live upgrade means upgrading individual modules of the operating system without rebooting the machine or killing processes using those modules. This would enable installing new updated versions of operating system modules without needing to notify or modify applications.

Userspace Debugging: No matter how much effort is put into preventing bugs, some bugs will always remain that developers will need to find. Good debuggers help developers diagnose and fix bugs more quickly and easily. However, debugging in the Linux kernel

is difficult. The kernel runs in its own address space and is highly reentrant, so it can be difficult to apply standard debugging techniques to it [130]. Kernel bugs can cause the system to crash or hang, and it can be hard to extract information from these states. In order to use an interactive debugger in the kernel, the kernel must be compiled with debugging support and standard debuggers (such as the GNU Project debugger gbd) do not provide the full features of userspace interactive debuggers when operating on the kernel. To enable full debugging ability and access to breakpoints and single stepping, a developer must enable kdb, the kernel debugger, and run the target kernel in a virtual machine [72]. Wile using the debugger to inspect code, the entire kernel is paused, including interrupts.

Allowing kernel module code to be run at userspace would provide access to fully featured userspace debuggers. This can take different forms depending on the needs and use cases. One way to run modules in userspace is to use a trampoline approach. Applications call into the kernel like normal. Those calls are forwarded from the kernel to the module running in userspace. The module then responds to the kernel with the return value, and the kernel returns to the application. This is straightforward and keeps behavior the same whether the application is run in the kernel or in userspace, but does not work for all types of modules. Some modules, such as the CPU scheduler, need to complete and return to the user while holding shared locks and cannot afford the jump to the userspace module. In this case, we can use record and replay to record a run in the kernel, and then replay the run at userspace.

Generality: Some of our goals, such as safety, can be met at the cost of supporting a wide variety of designs. For example, eBPF (the extended Berkeley Packet Fileter) provides safety for user provided code running in the kernel, but does so by restricting the type of code that can be written. eBPF programsmust use predefined data structures, cannot have unbounded loops, cannot dynamically allocate memory, and must be under a certain size. While this helps eBPF ensure safety of untrusted code, it limits the scope of changes that eBPF can address. Further, eBPF adds to the difficulty of making additional changes to the kernel, by widening the API to include the behavior of the user installed filter. Our interest is in supporting velocity for general kernel code.

Compatibility: Retaining compatibility with existing operating systems and applications is important for ensuring adoption. Rewriting an application for a different operating system interface is time consuming and expensive. If our approach to improving development velocity requires developers to use an entirely new operating system with a new interface, our approach will likely see less use, at least initially. Additionally, changing the interface that is exposed to applications will also require application developers to rewrite their software. The easiest way to retain compatibility with Linux applications is to use Linux as a starting point and keep changes to code and interfaces minimal.

Performance: One of the motivations for developing new kernel code is to improve performance. New custom kernel code can be used to provide optimized support for new hardware and to specialize the kernel for specific applications, providing better performance for new hardware and known applications. If a system provides fast development velocity but imposes too much performance overhead, it will not make sense for these use cases.

2.3 Case Studies: File Systems and CPU Schedulers

In this thesis, we focus on improving development velocity for two types of Linux kernel modules: file systems and CPU schedulers. We focus on these for three main reasons: relevance to current trends, recent proposed modifications, and clean interfaces.

A number of the trends we discussed earlier are particularly relevant for file systems and CPU schedulers. New storage hardware and new use cases for file systems are placing new pressures on file systems to adapt. Similarly, new compute hardware, energy demands, and new use cases, such as microservices and serverless computing, push new demands onto CPU schedulers. By focusing on file systems and CPU schedulers, we can address some of these recent trends.

Second, there has been significant recent research on new file systems and new CPU schedulers, proving that there is demand for new approaches. For example, in file systems, recent work has developed new file systems for non-volatile memory and serverless workloads, among many other new use cases [8, 84, 82, 159, 88]. Similarly, CPU schedulers have seen recent development, often to cope with cloud demands or support new hardware [85, 75, 128, 47, 126].

Lastly, the internal Linux kernel interfaces for file systems and CPU schedulers are clean and usable. The Linux kernel already supports multiple implementations of both file systems and schedulers, and so provides clean interfaces for providing new implementations. Each implementation is neatly boxed into its own piece of code. Without this property, it is difficult to isolate the module from the rest of the Linux kernel, making it difficult to implement a framework for high development velocity.

2.4 Existing Mechanisms for Velocity in Linux

There are some existing ways of writing new modules for subsystems in Linux. We will describe these methods and their advantages and disadvantages in this section. A summary is shown in Table 2.1.

Approach	Bug Scope Reduction	Live Upgrade	Userspace Debugging	Generality	Compatibility	Performance
Linux Core	✗	✗	✗	✓	✓	✓
Userspace Trampoline	✓	✗	✓	✓	✓	✗
Kernel Bypass	✓	✗	✓	✓	✗	✓
Multicore Microkernel	✓	✓	✓	✓	✗	✗
eBPF	✓	✓	✗	✗	✓	✓
Bento/Enoki	✓	✓	✓	✓	✓	✓

Table 2.1: Comparison of techniques for Linux kernel development velocity. Implementing code in the core Linux kernel provides good generality, compatibility with Linux and Linux binaries, and performance, but does not provide support for eliminating bugs or reducing their effects, live upgrade of components, or debugging components in userspace. Trampolining calls to a userspace implementation of a component limits the scope of bugs to the userspace component which can be debugged with userspace tools, but comes at the cost of performance and historically does not include support for live upgrade. Kernel bypass, implementing the entire component stack in userspace, also restricts bugs to the component and enables use of userspace debugging tools, though debugging is often difficult due to the low level nature of the code. Kernel bypass breaks compatibility because the component stack can no longer coordinate with other parts of the kernel and applications often need to be rewritten to use the customized component stack. Multicore microkernels, where all components are implemented at userspace, breaks compatibility. On top of requiring applications to be ported to a new operating system, changes to the components can result in changes to the interfaces, requiring applications to be modified. The extended Berkeley Packet Filter (eBPF) eliminates some bugs, but cannot be debugged with userspace tools and significantly restricts code, limiting the set of support design.

2.4.1 Linux Core Interfaces

Some subsystems in Linux support new implementations of modules written against common interfaces, either as external kernel modules or compiled directly into the Linux source. File systems and schedulers both support this design and there are already multiple implementations of both in the kernel. Using this approach, new designs for file systems and new scheduler algorithms can implemented and run directly in Linux like the existing implementations. Because new code is written and executed just like other kernel code, this approach has the same advantages and disadvantages as other kernel code. That is, good generality, compatibility, and performance, without bug scope reduction, live upgrade, or userspace debugging.

Now we will describe the Linux interfaces for file systems (called VFS) and schedulers.

VFS. VFS is the Virtual File System framework for writing file systems in the Linux kernel. VFS implements a layer between the core kernel code and the individual file system. A new file system can be mounted in place of any subdirectory. Then, when the kernel receives a file system system call, the kernel uses name resolution to identify which file system and which function(s) should be called to execute the requested behavior. When the file system returns, VFS returns the relevant return value to the caller. VFS manages generic state about the file system, such as which file systems are registered and where different file systems are mounted. VFS also defines a set of functions that a file system should implement to be a VFS file system.

VFS file systems implement a subset of the functions specified by the VFS layer. The functions include things like opening, creating, reading, and writing files. Some functions, such as the `read_iter` and `write_iter` functions for asynchronous file reads and writes, are not required. The file system can optionally implement these functions for better performance. A VFS file system is responsible for implementing the functionality of the specific file system. It is responsible for managing the on disk layout and directly interacting with the storage device. To access a disk the file system calls the block device layer in the kernel. VFS file systems can be implemented as kernel modules and inserted at runtime. However, to upgrade or remove a file system, all applications with open file descriptors into the file system must first close their files so the kernel can recognize that the file system is not in use.

The VFS layer also manages the page, or buffer, cache and the inode cache. The page cache caches data from the file systems to speed up access. File reads can be served from the page cache if the relevant data is in the cache. Typically, writes to files first go through the page cache and are later written back to the file system, e.g. on a flush operation or

after a timeout. Provided the user applications allow buffered writes that can be lost on a system crash, this allows writes to proceed more quickly and allows VFS to batch writes to the file system. The inode cache caches directory entries to reduce the number of lookups required to determine if a file exists. Like the page cache, this reduces the number of calls to the file system.

Scheduler Core. Like the Linux filesystem, the Linux CPU scheduler is designed to allow multiple implementations to co-exist through a framework in the core scheduling code. The core scheduling code manages the implementation of scheduling actions. It handles context switching between tasks, moving tasks between cores, and creating timers, among other responsibilities. It defines a set of functions that a scheduler should implement and calls these functions when necessary. These functions inform the scheduler of task state changes, such as if tasks block or wake up, and delegate to the scheduler policy decisions about task placement and schedule order. When a core becomes idle, or when explicitly requested, the core scheduler code calls into the scheduler to pick which task to run next. The core scheduler code keeps a list of all registered schedulers and calls each one in a strict priority order when choosing a new task. That is, if the highest priority scheduler has a ready task to schedule, the core scheduler code will run that task. If not, the core scheduler code will check if the next highest priority scheduler has a ready task.

Schedulers are responsible for keeping track of which tasks are runnable and choosing which task to run next when asked by the core scheduler code. The scheduler also manages which task should run on which core. The scheduler can limit how long a task runs by calling into the core scheduler code to set a timer. When the timer fires, the scheduler can choose to force the core to reschedule. Schedulers cannot be implemented as kernel modules. New schedulers can be written against the scheduler core interface, but all schedulers must be compiled into the kernel before it is loaded onto the machine.

Because schedulers are called every time the running task changes, it is very important that schedulers can quickly choose which task to run next. Any overhead when picking the next task is directly observable by applications.

2.4.2 Userspace Trampolining

Another method to introduce new implementations into the kernel is to use a userspace trampolining approach. In this approach, calls from applications are first sent to the kernel like usual. A kernel component forwards those calls to a userspace module that implements the module behavior, akin to a remote procedure call (RPC) [26]. The userspace component sends the return value back to the kernel component, which then returns it to the user. This

approach enables writing, testing and debugging the module in userspace, but can impose significant performance overhead due to the trampolining. While systems implementing this approach could support live upgrade, currently they typically do not.

This is implemented for file systems by FUSE and recently proposed for schedulers in ghOSt.

FUSE. File System in Userspace (FUSE) [56] is a framework for writing Linux kernel file systems in userspace. FUSE consists of two components: a kernel component to intercept and forward calls intended for the FUSE file system and a userspace component that receives the forwarded calls, directing them to the user provided file system and returning the result back to the kernel component. The kernel component of FUSE is implemented as a VFS file system. It translates calls from VFS to the low level FUSE interface and forwards the calls to the userspace component using a special file (`/dev/fuse`). The kernel component waits for the userspace file system to respond in the same file and returns the relevant value to VFS. The kernel component manages internal kernel state about the file system, such as locking and unlocking locks and tracking reference counts.

FUSE file systems are implemented in userspace by interacting with the user space component of FUSE. Like VFS, FUSE file systems are implemented by providing implementations for a set of functions defined by the framework, though the exact set of functions differ. The FUSE interface differs from the kernel VFS interface because FUSE needs to communicate with its user space component over a message passing API through a file. Buffers for reading and writing must be copied through the special file rather than passing pointers. Some VFS calls, like `llseek` are handled entirely by the kernel component. The userspace file system has to manage interacting with the storage device. This is usually done by opening the device file using direct I/O and reading and writing to the device through the file.

FUSE's design adds performance overhead in a number of cases [152]. The trampoline approach where calls are forwarded from the kernel to userspace and back introduces overhead, and interacting with block devices through direct I/O is much slower than using the in kernel block device layer. The kernel buffer cache is able to serve many reads and writes without calling into FUSE, allowing these to often have very good performance. However file creates and deletes need to be synchronously executed by the userspace file system, so they show noticeable performance overhead. Additionally buffered writes must be written out to the file system eventually, so performance overhead is occasionally noticeable on writes.

GhOSt. GhOSt [67] is a recently developed framework for userspace Linux kernel schedulers. It was published after the file system work described in this thesis but before the

CPU scheduling work. GhOSt consists of two components: a kernel scheduler that hooks into the core scheduler code, and a userspace scheduler agent. Like the core scheduler code, ghOSt defines a set of functions that a scheduler should implement. Calls to the kernel scheduler are asynchronously forwarded to the userspace agent, which responds with decisions that are then applied in the kernel. The kernel does not wait for decisions from the userspace scheduler to schedule a task. Instead decisions are communicated back to the kernel, and applied the next time a call is made into the kernel scheduler. The user and kernel components communicate using a shared memory queue and ioctls on a special file.

A GhOSt scheduler is implemented as a userspace process. It polls the queue, waiting for messages from the kernel. When the scheduler receives a message, it executes the relevant function. The userspace scheduler decides which task should be run on which core, sending those decisions to the kernel component via ioctl calls. The interface implemented by the ghOSt scheduler is slightly different from the one used by the kernel scheduler. In the kernel, moving tasks between cores and choosing which task on a core to run are separate calls. However, allowing this interface from userspace is dangerous. If a task is chosen to run on a specific core, but it is not currently located on that core, the kernel will crash. To ensure that a userspace program cannot crash the kernel, these must be combined into one operation in ghOSt.

Because ghOSt needs to receive decisions from userspace, it can add significantly to the scheduling latency. Additionally, the asynchronous model adds additional latency between when the scheduler makes a decision and when the decision is enacted, meaning the scheduling decisions can be based on out of date information. This also makes it difficult to precisely mirror what a native kernel scheduler would do. Scheduling latency is directly visible to applications, so this can noticeably affect application performance. Additionally, the ghOSt scheduler runs as a userspace process and must itself be scheduled to run on the CPU. To reduce the latency burden, the scheduler is often pinned to a core where it spins polling for updates. This core cannot be used to run other user code, effectively reducing the total number of usable cores, and increasing the overhead of the trampoline approach.

To optimize performance, ghOSt has also implemented eBPF hooks in the scheduler code so that parts if the scheduler can be run in the kernel.

2.4.3 Kernel Bypass

Some I/O devices support virtualization. This can be used to bypass the kernel and give userspace applications direct access to the I/O device. Typically, the device is managed

by a userspace library that implements abstractions for applications. For example, in kernel bypass networking, packets are sent directly from the NIC to the userspace network stack, often implementing TCP (Transmission Control Protocol)/IP (Internet Protocol). Applications call into the userspace network stack instead of the kernel to send and receive packets. Kernel bypass is often used to increase performance by removing the system call overhead and enabling new custom subsystem implementations that are not bound to prior kernel design decisions [77, 119, 47].

Using kernel bypass requires modifying the software stack so the application and device use userspace libraries. Tools like the Data Plane Development Kit (DPDK) [51] for networking and the Storage Performance Development Kit (SPDK) [145] for storage provide libraries and drivers for exposing devices to userspace. Developers unbind the device from the standard kernel driver, and bind it to the DPDK or SPDK driver. The developer can then use provided libraries to interface with the driver from userspace. Developers must modify applications so they call into the userspace library instead of the kernel. This is often done by directly modifying the application or standard library code or using binary rewriting on the application binary [84, 156].

Kernel bypass is commonly used in research work. Several research file systems [84, 30, 157], schedulers [47, 114, 125], and network stacks [77, 71] have been implemented using kernel bypass.

Kernel bypass is a useful mechanism for implementing new designs for kernel subsystems and achieving maximum performance. However, it comes with a number of costs. Because the kernel is completely bypassed, kernel bypass solutions must reimplement the entire subsystem and cannot easily share resources with applications that do not use the kernel bypass library [139]. While kernel bypass approaches support userspace debugging, it is often still difficult to debug the low level parts of the kernel bypass software stacks.

2.4.4 Multicore Microkernels

A microkernel is an operating system where the kernel consists of a small amount of privileged code, and most of the operating system, including device drivers and file systems, is implemented at userspace as independent processes [90, 81]. Applications send requests to the operating system processes using inter-process communication (IPC). Microkernel designs can provide good bug scope reduction and generality of approaches. Bugs in one process cannot affect the other processes, so bug scope is limited to the affected process. Because most of the operating system is unprivileged and operates at userspace, developers that wish to customize behavior of the operating system for a specific application

can implement custom parts of the operating system stack. These user-provided custom components operate exactly the same as the native operating system components.

Microkernels are popular in the research community due to their strong isolation properties, but have seen little widespread adoption. The overhead from IPC on all communication between different components has plagued microkernels [66]. On top of that, adoption of any new operating system is difficult because applications must be ported to the new operating system's interface. Additionally in microkernels, changes to an operating system component often result in changes to the IPC interface exposed by the component, meaning all applications using the component must be updated. Google uses a microkernel based approach for their new networking stack Snap [99].

2.4.5 eBPF

eBPF (the extended Berkely Packet Filter) [53] is a mechanism for running untrusted user code in the kernel to customize its behavior. It was first created for customizing and implementing network packet tracing, but has since been expanded to support several different use cases within the kernel [104]. An example is system call filtering, for implemeting a restricted execution environment for untrusted code. While Linux does not currently support eBPF programs for file systems and schedulers, there have been proposals to use eBPF to accelerate storage device accesses [160] and enable scheduler implementations [67].

eBPF allows application and kernel developers to run custom code at certain selected points in the kernel. The developer writes their eBPF program in a restricted language. The developer loads the compiled eBPF program into the kernel, targeting a specific location. The kernel runs the eBPF verifier on the program to ensure that it does not violate safety properties. If the verifier passes, the program is run whenever the kernel encounters the target location or condition.

Since the trust model is that eBPF programs are per-application, eBPF programs must be restricted to ensure that the rest of the kernel will be safe in case the eBPF developer is malicious. For example, the code of the program cannot contain direct memory accesses, dynamic memory allocation, or unbounded loops. These prevent the program from reading and manipulating kernel memory, exhausting kernel memory, or looping infinitely. To allocate memory and store data, the kernel provides a set of pre-compiled data structures for the eBPF code to use. Also, eBPF programs can only be inserted at prespecified locations in the kernel. The locations specify when the program is called and what type of data is passed to the program. Further, eBPF programs must complete within a certain number of instructions to ensure that the timeliness of the rest of the kernel is not impacted.

eBPF is very useful for use cases such as small custom optimizations, monitoring, and logging. However, for larger and more complex programs or programs that need to carefully manage their data, the restrictions and trust model make it difficult to use. File systems are too complicated to be implemented within eBPF, and schedulers often rely on custom data structures that are not provided by eBPF. In either case, it can be difficult to verify that malicious code, powerful enough to implement a scheduler or file system, can be constrained as to avoid corrupting or disrupting the kernel.

2.5 Relevant Background Knowledge

In this section, we provide some background that will be useful for the later chapters. We describe Rust, the programming language we use to reduce kernel bugs, as well as the default file system and scheduler in Linux that we use to show our approach does not carry a significant performance overhead. Understanding Rust's guarantees will be useful to understanding how we prevent bugs and which bugs are prevented. The descriptions of ext4, the default Linux file system, and CFS, the default Linux CPU scheduler, will be relevant to understanding our implementations and evaluations.

2.5.1 Rust Primer

Rust is a strongly-typed, memory safe, data race free, non-garbage collected language. With these properties, Rust is able to provide strong safety guarantees without high performance overhead or the performance unpredictability caused by garbage collection.

Rust relies on its type system to enforce memory safety. The type system restricts how objects can be created and cast, so if an object exists and is of a certain type, this guarantees that the memory backing the object is valid and correctly represents that type. Since raw pointers can be `NULL` and can be cast to nonequivalent types, dereferencing pointers and creating strongly typed objects from pointers is unsafe and must be tagged as `unsafe` to compile. Calling unsafe functions is additionally unsafe.

Rust prevents most memory leaks without mark and sweep garbage collection by tracking the lifetime of objects. All objects must be owned by one variable at a time. When the variable owning an object goes out of scope, the lifetime of the object is over and the memory backing the object can be safely reclaimed. References allow other variables to refer to data without claiming ownership of the memory. References are either immutable or mutable, enabling read-only or read-write accesses, respectively; references cannot outlive

the owner. Developers can provide custom functionality to be performed when an object goes out of scope by implementing the `drop` method. Memory can be leaked in safe Rust, causing the memory to go out of scope without calling `drop`, so it is possible that the `drop` function will not be called even if there is no unsafe code. However, in order to leak memory, a developer must explicitly call a `leak` function, so accidental memory leaks are unlikely.

Data races are avoided by enforcing that all objects, except those that can be safely modified concurrently, must only have one mutable reference at a time. For non-thread safe objects that must be shared between threads, some synchronization mechanism such as locking must be used to safely obtain references. Acquiring the lock gives the caller access to the underlying data. Lock acquisition methods generally return a guard that automatically unlocks the lock in `drop`, preventing the caller from forgetting to unlock. However, deadlocks, such as circular waiting for locks, are possible in safe Rust code as preventing them is beyond the power of the Rust type system.

2.5.2 Ext4

Ext4 (extended file system, version 4) is the default file system in Linux. It is an extent based [147] journaling file system [124] that provides good performance, even for large files.

In traditional file systems using fixed blocks, the inode keeps track of the disk locations of the data blocks used to store the file data. If more blocks are required than fit in the inode, indirect blocks are used. The inode points to the indirect block, and the indirect block is filled with the disk locations of the data blocks. If the file is even larger, a double indirect block may be required. Although indirect and doubly indirect blocks may be cached, given limited main memory, reads and writes to large files may require multiple reads from the storage device. With extents, the file system inode stores the disk locations of the start and end of a contiguous region. These regions can be arbitrarily large. This allows the inode to point to a larger number of data blocks without indirection, as long as the file data is (mostly) contiguous.

Ext4 uses a journal to ensure crash consistency [121]. Depending on the journaling mode, metadata or both data and metadata are written to the journal first. After state is completely written to the journal along with a commit record, ext4 copies the data/metadata to their final location on disk, and garbage collects the log. If the file system crashes while writing data to the log, ext4 can discard the operation. If it crashes after the log commit, ext4 can use the journal to ensure that all writes in the log are fully written. Different levels of safety are guaranteed depending on the journaling mode.

Ext4 also provides a number of other useful file system features, such as pre-allocation of disk space for a file, delayed allocation of data blocks, journal and metadata checksumming, and automatic defragmentation.

2.5.3 CFS

CFS is the default CPU scheduler in Linux. CFS stands for the Completely Fair Scheduler. CFS uses per-core run-queues, meaning it first assigns tasks to cores, and then chooses the next task for the core from among the assigned tasks. CFS arranges that most scheduling operations are done on core-local data structures. On each core, CFS implements a version of weighted fair queuing, dividing CPU time proportionally between groups of tasks, and then within each group, while respecting priority. It uses a calculation called *vruntime* to track which group/task to choose next, choosing the group/task with the lowest weighted accumulated runtime. The *vruntime* is calculated based on the task's runtime, modified by its priority; tasks with higher priority accrue *vruntime* slower. To prevent sleeping tasks from accruing a large *vruntime* debt and therefore running for too long after they wake, newly woken tasks receive the maximum of their old *vruntime* and the *vruntime* of the task with the lowest *vruntime* in the run-queue minus a several millisecond threshold. If a newly woken task has a smaller *vruntime* than the current task, it preempts the current task at the next system timer tick. Otherwise, task switches occur only after a task has run for its allocated time slice. In order to prevent starvation, CFS attempts to run every task at least once per time period, where the period depends on the number of tasks, with a minimum of 6ms. When tasks are created or woken, the length of the period adjusts to include the new task. New tasks are run at the end of the period, but recently woken tasks can be run earlier.

CFS places running tasks onto cores and moves tasks between cores to achieve better performance. By default, CFS will attempt to even out the amount of work per core, based on information such as the amount of time the core is idle or overloaded, the priority of the tasks on the core, the capacity of the cores on the machine, and the preferences of the tasks. In certain cases, CFS will co-locate tasks on specific cores, such as if it is required to by the user. CFS attempts to place tasks so that newly woken tasks can be scheduled promptly. When a task is woken or a core becomes idle, CFS will move tasks so the newly idle cores are used. CFS rebalances task placement every 1-10ms, depending on the configuration, or when cores become newly idle. CFS first tries to move tasks to cores within the same NUMA node, and will not balance tasks across NUMA nodes unless the difference in load across the NUMA nodes is more than a defined threshold.

Chapter 3

Linux Kernel Bug Analysis

This chapter discusses bugs in the Linux kernel. We first discuss why bugs in the kernel are important. We then describe prior work done to categorize bugs in Linux and present two bug studies we performed. We also include a discussion of what techniques could be used to prevent these bugs.

Operating systems are the first line of defense for security isolation of untrusted application code, and bugs can provide a means for attackers to compromise data and computation integrity. One of the primary jobs of the operating system is to isolate and protect the applications running on the machine from each other. Bugs in the kernel can hinder its ability to provide this isolation and security. Many bugs in the Linux kernel result in crashing the kernel, causing the machine to cease functioning. Even if a particular bug cannot be used in a kernel compromise, if it can be triggered by users, it can cause Denial of Service (DoS). Other bugs can allow privilege execution, bypassing of security checks, arbitrary code execution, or information leaks.

Bugs also impact reliability and availability. If the kernel crashes, all processes on the machine will be killed, hurting the uptime of those applications. Even bugs that do not cause kernel panics can affect reliability, resulting in either improper kernel functionality leading to application crashes or unexpected behavior.

In order to maintain the security and reliability of the Linux kernel, we must ensure that rapid development velocity does not come at the cost of increasing bugs. Any framework we build to improve Linux kernel development velocity must address bugs somehow, either by eliminating bugs or making it easier for developers to find and fix bugs. To do this, we first need to understand the types of bugs that occur in the Linux kernel because different types of bugs will require different approaches. `NULL` pointer dereferences are found and

fixed differently than race conditions, which are different again from logic bugs or security flaws. Another important difference is whether a kernel bug has a local effect within its own module, e.g. affecting only the applications using a particular file system, or a global effect on unrelated applications.

In this chapter, we analyze how bugs manifest in Linux. We first discuss prior bug studies in Linux and then present the results of two bug studies we perform. The first study we perform looks at three kernel modules in depth, to understand the types of bugs that occur in the kernel, how serious those bugs are, and what could be done to prevent the bugs. We analyze all bug fix git commits in three Linux kernel modules (OverlayFS, AppArmor, and Open vSwitch Datapath) from 2014-2018 and categorize these bugs by the cause and possible effect of the bug. This study only analyzes bugs fixed in git commits, so it cannot draw conclusions about bugs that still remain in the code, bugs that were caught before or after the study period, and bugs that were caught during the development and testing process and so were not committed to the git repository. Additionally, we only analyze bugs that were found in the three kernel modules, so the results may not be representative of the kernel more broadly.

We find that around half of the bugs across these are high-level logic bugs that depend on the semantics of the module and the other half are low-level bugs that could be addressed with generic mechanisms. Many of these low-level bugs are memory errors or occur along error handling pathways. The vast majority of the low-level bugs could be prevented by compile-time type checks, such as those performed by safe Rust.

In our second study, we focus on the rate of change of bugs over time. We analyze new Linux CVEs (Common Vulnerabilities and Exposures) and bug fix commits in three Linux file systems (OverlayFS, ext4, and btrfs) from their release to 2020 to track the rate of new bugs in relation to the age and stability of the code. We collect Linux vulnerabilities by analyzing all Linux vulnerabilities listed in the CVE records. This does not catch bugs that cannot be triggered by user code or bugs that do not have security implications. Like our other bug study, the analysis of both CVEs and bug fix commits cannot catch bugs that remain in the code or were caught before they were committed. The CVE analysis also cannot catch bugs that were caught and fixed before being compiled into a kernel release version. The study of bugs over time in file systems is not necessarily representative of the overall prevalence of bugs in Linux.

We find that the rate of CVEs in Linux has increased and the number of bugs per line of code stays relatively stable throughout the lifetime of the code, despite increased efforts to prevent, find, and fix bugs. We also analyze the cause of the CVEs and again found that a significant portion (42%) could be addressed with compile-time safety checks.

Together, these bug studies imply that there are growing issues caused by bugs in the Linux kernel, and that a significant portion of these bugs could be addressed by language level mechanisms.

3.1 Summary of Prior Results

There have been several past studies on bugs and CVEs within the Linux kernel and its file systems. Here, we summarize the past results from these studies.

There have been two major papers studying the types of faults found broadly within the Linux kernel: a 2001 paper that used static analysis the study faults in Linux [39] and a 2011 paper that redid the analysis on the newer version of the kernel [116]. As far as we know, there has not been a subsequent redoing of the analysis, so we will focus on the 2011 paper. Both of these papers identified and categorized bugs using static analysis searching for dangerous code patterns. The types of bugs identified include deadlocks caused by holding a spinlock when calling a blocking function, dereferencing potentially `NULL` pointers, allocating large stack variables (potentially overrunning the fixed size kernel stack), out of bounds accesses, double acquiring locks, use after free, using floating point values, and allocating the incorrect amount of memory for a variable. The 2011 study found that many of the bugs were in driver code. The most common types of bugs were 1) dereferencing pointers that were `NULL`, despite first checking for `NULL`, 2) not checking a pointer that could be `NULL`, 3) calling a blocking function when holding a spinlock, and 4) using a value after it had been freed. This study also analyzed bugs over time. It shows a relatively consistent number of bugs as new bugs get added and old ones are fixed. The total bugs per line of code slowly went down between 2004 and 2010, from 0.2 to around 0.1 bugs per 1000 lines of code. For comparison, there were 13 million lines of code in the Linux kernel, implying that in 2010, Linux had 1300 bugs. Because this study used static analysis, it did not catch many higher level logic bugs.

A different study analyzed eight years (December 2003 to May 2011) of patches in several Linux file systems (ext3, ext4, XFS, ReiserFS, and JFS) [96]. Ext3 and ext4 are general purpose journaling file systems and have been the default file systems in many distributions of Linux since ext3's introduction in 2001. Ext4 is an extension of ext3, adding extents for supporting large files and delayed allocation. Ext3 was itself an extension of ext2. XFS is a journaling file system that is optimized for large files and parallel file I/O. It was first proposed in 1993 and implemented in Linux in 2001. ReiserFS was a general purpose journaling file system introduced in 2001 that competed with ext2 and ext3. It has since been marked as obsolete. JFS is a journaling file system that is optimized for

IBM's AIX operating system. It was released for AIX in 1990 and ported to Linux in 2001. Since 2008, it has seen relatively little support, but is still maintained by IBM. Because this study only analyzes bugs that have been found and fixed, it is unable of understanding the distribution of latent errors in the file systems.

This study found that most of the patches during this time period were for maintenance; 40% were bug fixes. Many of the bug fix patches were in more complicated parts of the code, specifically the file, inode, and superblock structures. Of the bug fixes, over 50% were semantic bugs. The rest were concurrency (around 20%), memory (around 15%) and error handling (around 10%). Some of the semantic bugs (10-20%) were not based on the specific behavior of the module, of which some were type errors. Most of the concurrency bugs were either atomicity violations, where locks were not grabbed correctly, or deadlocks. Memory bugs were primarily memory leaks or `NULL` pointer dereferences. Roughly one third of bugs were found along error handling pathways, and many of these were memory bugs.

3.2 Module Bug Analysis

We perform a study to provide intuition into the difficulty of developing and deploying new kernel features and understand whether or not the safety properties provided by Rust can address or prevent these bugs. We analyze all bug fix git commits from 2014-2018 for three modules that modify core Linux functionality used by Docker containers: OverlayFS, AppArmor, and Open vSwitch Datapath.

OverlayFS provides support for layered overlay file systems, where one file system is layered on top of another and the combined view is exposed to applications. Overlay file systems are useful for exposing a read-only directory to an application without preventing the application from writing to its view of the directory. Docker containers use OverlayFS to share directories with large files among containers. Only the containers' modifications to the directories need to be stored. AppArmor is a security utility that is used to restrict the capabilities of an application. Docker uses it to protect the operating system and other applications from misbehaving containers. Open vSwitch is a virtual switch used to multiplex network connections between multiple IP addresses on one machine. The Open vSwitch Datapath processes packets in the kernel and directs them to the correct application.

We divide bugs in these systems into two types. One set are semantic bugs in the high-level correctness properties of each module. These can range from mission critical to configuration errors, but generally impair just the functionality of the module. These account for 50% of the total bugs fixed in these modules.

Memory Bugs	Number	Fixed by Rust?	Effect on Kernel
Use Before Allocate	6	Y	Likely `oops`
`NULL` Dereference	5	Y	`oops`
Over Allocation	1	Y	Overutilization
Use After Free	3	Y	Likely `oops`
Double Free	4	Y	Undefined
Dangling Pointer	1	Y	Likely `oops`
Missing Free	18	Y	Memory Leak
Reference Count Leak	7	Y	Memory Leak
Out of Bounds	4	Y	Likely `oops`
Other Memory	1	N	Variable
Concurrency Bugs			
Deadlock	5	N	Deadlock
Race Condition	5	Y	Variable
Type Errors			
Unchecked Error Value	5	Y	Variable
Other Type Error	3	Y	Variable
Total	68	62	

Table 3.1: Low-level bugs in released versions of OverlayFS, AppArmor, and Open vSwitch Datapath between 2014-2018, categorized as memory bugs, concurrency bugs, or type errors, and the likely effect of each bug on kernel operation. `Oops` is a Linux kernel condition that immediately causes the kernel to shutdown the current process or the entire kernel.

The second set concern low-level bugs of the type found in any C language code, but when found in the kernel can potentially undermine the correctness or operation of the rest of the kernel. Table 3.1 categorizes these low level bugs in more detail. We categorize these as (1) memory bugs, such as `NULL` pointer dereferences, out-of-bounds errors, and memory leaks; (2) concurrency bugs, such as deadlocks and race conditions; and (3) type errors, such as incorrect usage of kernel types (e.g., interpreting error values as valid data).

We also estimate by hand the likely possible consequence of the bugs. Of the 50% of fixed bugs that are low-level bugs, we find that 68% are memory bugs. Of these, half are a type of memory leak, potentially causing out-of-memory problems or even DoS attack vectors. Such bugs are hard to uncover by testing but can lead to serious impacts on the integrity of the kernel. Of all identified low-level bugs, 26% cause a kernel `oops`, after which the kernel either kills the offending process or panics, crashing the machine.

We observe that bugs regularly occur along error handling pathways. Often, developers forget to check return values for `NULL` or negative error values or forget to perform necessary cleanup, like freeing memory or releasing locks, along the error pathway. These bugs are particularly difficult to catch because triggering error cases in test cases can be difficult, particularly when those error cases are due to rare hardware failures.

Many of these low-level bugs, particularly memory and type errors, result from inherent challenges of C code and could be prevented with a type safe programming language. Specifically, safe Rust is able to catch almost all of these low-level bugs. Rust uses a technique called Resource Acquisition Is Initialization (RAII) where the resources used by an object are owned by the object and tied to the lifetime of the object. When a variable is created, resources are acquired and initialized for the object. When the object goes out of scope, its lifetime ends, and the resources are reclaimed. Because creation of a variable is tied to the memory backing the variable, no memory can be accessed before it is initialized. This prevents Use Before Allocation and `NULL` Dereference bugs. Additionally, because memory is acquired to back a variable, Over Allocation bugs are also generally eliminated. In RAII, the memory backing a variable is reclaimed when the variable goes out of scope, and the variable and its memory address can no longer be accessed. This prevents Use After Free, Double Free, and Dangling Pointer bugs. Because the memory is freed automatically, this also generally prevents Missing Free and Reference Count Leak bugs, although memory and reference counts can be intentionally leaked in safe Rust. Rust also includes bounds checking on memory accesses, preventing Out of Bounds errors. The Other memory bug that we encountered was due to not cleaning a reusable buffer when it was placed on a free list. This cannot be caught by Rust because it is caused by the code managing its own pool of buffers.

Safe Rust can also catch many of the concurrency bugs we discovered. Safe Rust keeps track of whether objects are read-only or read-write. If the developer needs to modify the value of a variable, the variable must be marked as mutable. Attempting to modify the value of an immutable variable will result in a compiler error. Similarly, references can be mutable or immutable. Mutable references allow the reference holder to modify the value of the object the reference points to. Immutable references only allow reading the value of the object. Safe Rust requires that any objects or references that are shared across threads are immutable so all threads can safely concurrently read the value. To allow multiple threads to share a mutable object, the object must be wrapped with a lock. The lock itself can be shared safely. A thread can obtain a mutable reference to the shared object by acquiring the lock. This prevents data race conditions, but cannot prevent deadlocks.

Safe Rust is able to catch all of the type errors we encountered in our bug study. Most of these type errors were due to incorrect handling of error values. In C, a function returns

an error by returning an invalid value. The callee must check the return value to ensure that it is valid before using it. In the Linux kernel, functions that return pointers in the normal case often return errors using either a `NULL` pointer or a small negative number cast to a pointer, called `ERR_PTR`. The callee must then check for either `NULL` or `ERR_PTR` before using the returned pointer. Often, developers will forget to check for one or both of these, and execute the wrong behavior or access the invalid pointer, causing a kernel `oops`. Rust includes a special type for returning error values called the `Result` type. A `Result` can be either a valid value of the function's standard return type or an error code. The callee must check the returned `Result` for errors before getting access to the valid return value. This prevents all of the Unchecked Error Value bugs we encountered. Other type errors were often due to incorrect functions being called or forgetting to mark arguments as `const` so they would be immutable.

In total, using safe Rust catches 91% of the low-level bugs, only leaving deadlocks and miscellaneous memory errors unchecked.

3.3 Linux CVE Analysis

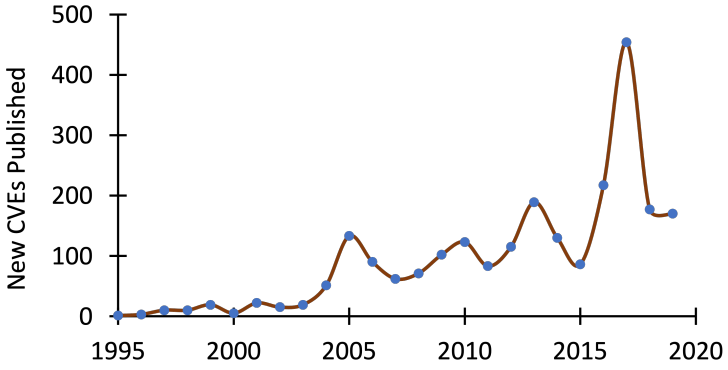

Figure 3.1: Newly reported Linux CVEs per year.

As with other commercial systems, Linux tries to strike a balance between growth and reliability. A typical release cycle consists of a two-week merge window and eight or more weeks of bug fixes and stabilization. During testing periods, developers are encouraged to test changes through static analysis tools for C [144, 41], automated testing frameworks [78, 83, 1], and continuous integration tests [94, 79]. However, even after years of rigorous

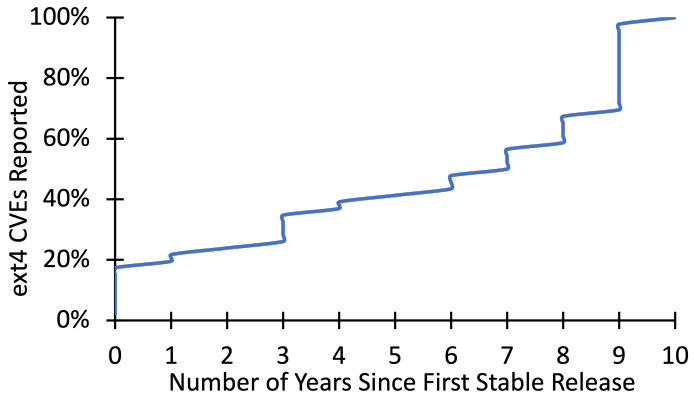

Figure 3.2: Cumulative distribution function of reported CVEs in ext4 since release.

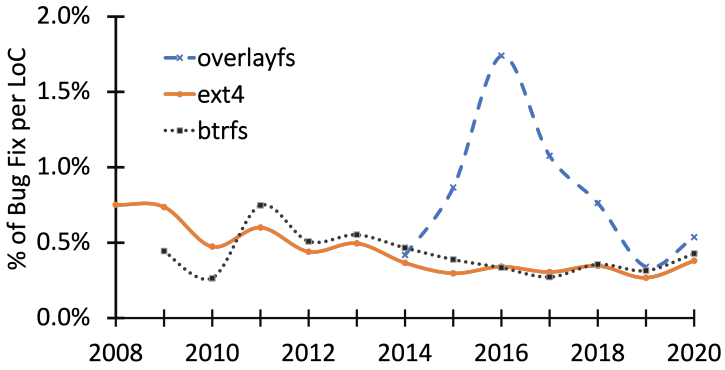

Figure 3.3: Number of bugs fixed each year per line of code in three Linux file systems.

testing and development, hundreds of common vulnerabilities and exposures (CVEs) are still found in Linux each year. Figure 3.1 is computed from the number of CVEs reported in the Linux kernel each year from 1995 to 2020.

One may hope that vulnerabilities come from newer components that become reliable as they stabilize. However, when we look at ext4, the default Linux file system, this does not appear to be the case. Figure 3.2 shows a CDF of the CVEs reported in ext4 since its release in 2008 through 2020. In these 12 years, 50% of the CVEs were found after 7 years or more of use. Despite seeing wide use for 12 years since its first stable release, ext4 has only seen an increase in the number of vulnerabilities over time.

Other Linux file systems share a similar trend. In Figure 3.3, we show the number of new bug patches per lines of code in overlayfs, ext4, and btrfs since their initial releases to 2020. Btrfs is a copy-on-write file system that uses B-trees for efficient on disk storage. While the file systems generally become more stable over time, they all maintain a significant number of bug fixes per line of code. Even after 10 years, there are still new bugs. All three file systems have approximately one new bug found each year for every 200 lines of code.

3.4 Summary

Despite the best efforts of developers and maintainers, bugs continue to exist in Linux. In the three file systems we analyzed, the number of bugs has grown proportionally with the number of lines of code that are added. As the rate of development on Linux continues to increase, we can expect the rate of bugs and CVEs to do the same, unless changes are made. If kernel development could take advantage of language level compile-time safety checks, a significant portion of these bugs could be prevented before the code is ever run.

By developing a framework that allows developers to write Linux kernel code in safe Rust, we can eliminate large classes of bugs that are hard to debug, making kernel development much higher velocity. With this type of framework, kernel developers can be sure that their code will not crash or have data races and can focus their efforts on the logic and security of their module.

A framework that relies on language level safety checks would still leave the 50-60% of logic and security bugs we found. To make these easier to find and fix, our framework should also support enhanced testing and debugging. We leave for future work other methods of preventing kernel bugs, such as lightweight or heavyweight verification [142, 41, 36, 64].

Chapter 4

Bento: High Velocity Linux Kernel File Systems

Slow development cycles are a particular problem for file systems. Recent changes in storage hardware (e.g., low latency SSDs (solid state drives), but also density-optimized QLC (quad-level cell) SSD and shingle disks) have made it increasingly important to have an agile storage stack. Likewise, application workload diversity and system management requirements (e.g., the need for container-level SLAs (service level agreements), or provenance tracking for security forensics) make feature velocity essential. Indeed, the failure of file systems to keep pace has led to perennial calls to replace file systems with blob stores that would likely face many of the same challenges despite having a simplified interface [3].

Existing alternatives for higher velocity file systems sacrifice either performance or generality. FUSE is a widely-used system for user-space file system development and deployment [56]. However, FUSE can incur a significant performance overhead, particularly for metadata-heavy workloads [152]. We show that the same file system runs a factor of 7x slower on 'git clone' via FUSE than as a native kernel file system. Another option is Linux's extensibility architecture eBPF. eBPF is designed for small extensions, such as to implement a new performance counter, where every operation can be statically verified to complete in bounded time. Thus, it is a poor fit for implementing kernel modules like file systems with complex concurrency and data structure requirements.

Our research hypothesis is that we can enable high-velocity development of kernel file systems without sacrificing performance or generality, for existing widely used kernels like Linux. Our trust model is that of a slightly harried kernel developer, rather than an untrusted application developer as with eBPF. This means supporting a user-friendly

development environment, safety both within the file system and across external interfaces, effective testing mechanisms, fast debugging, incremental live upgrade, high performance, and generality of file system designs.

To this end, we built Bento, a framework for high-velocity development of Linux kernel file systems. Bento hooks into Linux as a VFS file system, but allows file systems to be dynamically loaded and replaced without unmounting or affecting running applications except for a short performance lag. As Bento runs in the kernel, it enables file systems to reuse well-developed Linux features, such as VFS caching, buffer management, and logging, as well as network communication. File systems are written in Rust, a type-safe, performant, non-garbage collected language. Bento interposes thin layers around the Rust file system to provide safe interfaces for both calling into the file system and calling out to other kernel functions. Leveraging the existing Linux FUSE interface, a Bento file system can be compiled to run in userspace by changing a build flag. Thus, most testing and debugging can take place at user-level, with type safety limiting the frequency and scope of bugs when code is moved into the kernel. Because of this interface, porting to a new Linux version requires only changes to Bento and not the file system itself. Bento additionally supports networked file systems using the kernel TCP stack. The code for Bento is available at https://gitlab.cs.washington.edu/sm237/bento.

We use Bento for our own file system development, specifically to develop a basic, flexible file system in Rust that we call Bento-fs. Initially, we attempted to develop an equivalent file system in C for VFS to allow a direct measurement of Bento overhead. However, the debugging time for the VFS C version was prohibitive. Instead, we quantitatively compare Bento-fs with VFS-native ext4 with data journaling, to determine if Bento adds overhead or restricts certain performance optimizations. We found no instances where Bento introduced overhead – Bento-fs performed similarly to ext4 on most benchmarks we tested and never performs significantly worse while outperforming a FUSE version of Bento-fs by up to 90x on Filebench workloads. Bento-fs achieves this performance without sacrificing safety. We use CrashMonkey [105] to check the correctness and crash consistency of Bento-fs; it passes all depth two generated tests. With Bento, our file system can be upgraded dynamically with only around 15ms of delay for running applications, as well as run at user-level for convenient debugging and testing. To demonstrate rapid feature development within Bento, we add file provenance tracking [91, 107] to Bento-fs and deploy it to a running system.

Bento's design imposes some limitations. While Rust's compile-time analysis catches many common types of bugs, it does not prevent deadlocks and or semantic guarantees such as correct journal usage—those errors must be debugged at runtime. While correctness testing is possible at user-level, performance testing generally must be done in the kernel.

Also, like other live upgrade solutions, Bento upgrades also require backward-compatibility of the new code with the previous data layout on disk—though the file system itself can perform disk layout changes. The current implementation of Bento imposes some usability limitations similar to FUSE, such as only supporting one mounted file system per inserted file system module. And while we compare Bento-fs performance to ext4, we should note that Bento-fs is a prototype and lacks some of ext4's more advanced features.

4.1 Approach

Bento addresses the following challenges for high velocity development of Linux file systems:

- **Buggy code:** Bugs are difficult to avoid in the monolithic C Linux kernel, but bugs in the kernel can have severe consequences for the entire machine. Any bugs in a newly installed file system should be limited, as much as possible, to applications or containers that use that file system.
- **Disruptive upgrade:** Upgrades that require rebooting the machine or shutting down any processes using the file system cause availability downtime, restricting the rate at which upgrades can be deployed. The framework should support dynamic upgrades to running file system code, transparently to applications, except for a small delay
- **Slow debugging:** Many common debugging tools do not work in the kernel, limiting developers' ability to find and fix bugs. File system code should be easily migrated between userspace and the kernel to enable user-level debugging and correctness testing.
- **Generality:** There are a large variety of file system designs that developers might want to implement. Bento should address the above challenges without limiting the types of file systems that can be developed.
- **Compatibility:** Changes to kernel primitives or operating system structure can require significant changes to applications. File systems added to Linux via our framework should work with existing application binaries without recompiling or relinking. Further, Bento should not require substantial changes to Linux's internal architecture, to make Bento easier to adopt.

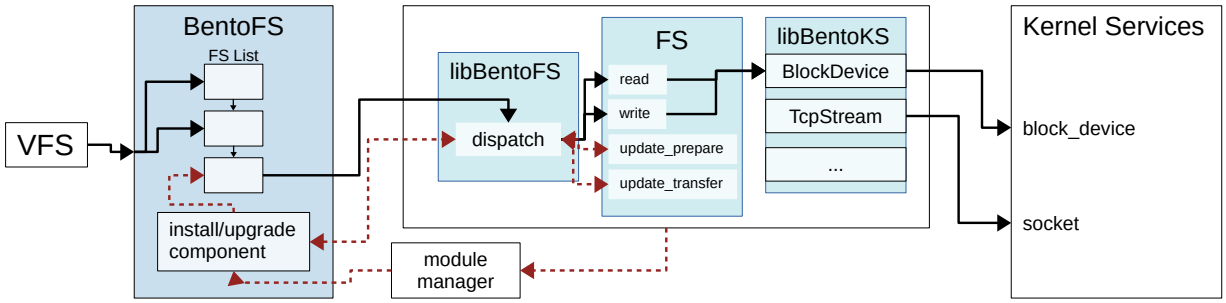

Figure 4.1: Design of Bento. Shaded components are parts of Bento. BentoFS is in C. The other shaded components are in Rust. Solid black lines represent the common-case operation pathway, detailed in §4.3.1 and §4.3.2. Dashed red lines represent the install/upgrade pathway and are described in §4.4

- **Performance:** New file systems are often developed to realize performance gains on new hardware or for new applications. If Bento imposes significant performance overhead, that will hinder its ability to support these use cases. Performance should be similar to that of the same functionality implemented using VFS.

At a high level, Bento addresses *buggy code* without sacrificing *generality* or *performance* by enabling developers to write file systems in Rust. Of course, safely using Rust within Linux is a challenge of its own. The other three challenges are addressed via careful architectural design. To limit *buggy code* without sacrificing *compatibility*, Bento avoids directly using the Linux VFS interface, because it requires data structures to be directly passed back and forth between the file system and the kernel, making it difficult to provide verifiable data structure ownership safety. Instead, Bento introduces a message-passing based API for file systems that enforces ownership safety. Second, Bento introduces a different API to enable safe access to C-language kernel services, by translating unsafe kernel interfaces into ones that can be safely used by Rust. To address *disruptive upgrade*, Bento supports live upgrades. It includes a component that quiesces the running file system and then transfers file system-defined state to the new instance, passing ownership of long-lived, in-memory data structures between the file systems so they can be shared across the upgrade. For *slow debugging*, Bento enables kernel file system code to be run at userspace by simply changing a build flag. Bento is designed with the same set of API calls whether it runs in the kernel or in the userspace to facilitate this.

4.2 Design

Figure 4.1 shows the Bento architecture; the shaded portions are the Bento framework. Bento is a thin layer that, to the rest of Linux, operates like a normal VFS file system. The Linux kernel is unmodified other than the introduction of Bento. In turn, like VFS, Bento defines a set of function calls that Bento file systems implement and provides a mechanism for file systems to register themselves with the framework by exposing the necessary function pointers. Unlike VFS, Bento is designed to support file systems written in safe Rust.

Bento consists of three components. First, BentoFS interposes between VFS and the file system module and acts as a controller that manages registering and running file systems. BentoFS is written in C and inserted as a separate kernel module. The other two components are Rust libraries that are compiled into the file system module. LibBentoFS translates unsafe calls from BentoFS into the safe file operations API that is implemented by the file system. LibBentoKS provides a safe API for file systems to access kernel services, such as to perform I/O. The file system itself is written in safe Rust and is compiled as a Rust static library that includes libBentoFS and libBentoKS. When a file system module is loaded, it registers itself with BentoFS which adds it to the list of active file systems.

4.3 Safe Interfaces

4.3.1 Interacting with VFS

The VFS layer poses a fundamental challenge to memory safety. For example, VFS file systems allocate a single inode data structure to hold both VFS and file system-specific data. When the kernel needs a new inode, it requests one from the file system which allocates it from its own memory pool. Both sides access their half of the data structure, and when done, the kernel releases the inode to the file system so the memory can be reclaimed. Independent of whether this is a good design pattern for minimizing kernel memory errors, it is inconsistent with Rust compile time analysis and therefore would compromise our ability to prevent memory safety errors within the file system code itself.

Instead, we define a new interface for safe kernel file systems. This API is shown in Table 4.1. The BentoFS module receives all calls from the VFS layer, determines which mounted file system is the target, and handles any necessary operations on kernel data structures. BentoFS then sends requests to the libBentoFS dispatch function using

a similar API to that of the file system, but with unsafe pointers instead of Rust data structures. LibBentoFS parses the request, converts pointers to safe data structures, and calls the correct function in the file system. The key idea is that the file system's compiler can statically verify its own data accesses, including its inode. To create an inode, BentoFS calls into the file system (via libBentoFS) and gets back an opaque reference (the inode number). In turn, BentoFS allocates and returns to VFS a separate kernel inode data structure. BentoFS never touches the contents of the file system inode.

BentoFS and libBentoFS are responsible for ensuring that Rust's safety properties are maintained as memory is passed across the file operations API so the assumptions made by the Rust compiler will be true. When passing references to kernel memory to the file system, such as data for read and write calls, BentoFS guarantees that the memory will remain valid until the call completes and, if a mutable reference is passed, must ensure that no other thread is modifying the memory. When passing references to structured data, BentoFS and libBentoFS also ensure that the memory is correctly structured and never cast to an incompatible type. Passing ownership across the file operations API requires careful handling of the memory in libBentoFS and is only done during live upgrade (§4.4).

4.3.2 Interacting with Kernel Services

Bento file systems need access to kernel functionality such as block I/O for access to underlying storage devices. These kernel interfaces, like those in the VFS layer, are not designed with type safety in mind and so cannot be directly used by a Bento file system. Instead, libBentoKS implements safe versions of kernel data structures and functions needed by file systems.

As an example, we will focus on kernel block I/O. File systems in Linux access block devices via the buffer cache. To read from (or write to) a block device, a Linux file system calls `__bread_gfp`, passing in a pointer to the `block_device` data structure, a block number, the block size, and a page allocation flag. This function returns a `buffer_head` data structure representing the requested block. The block's data is represented as a pointer and size in the `buffer_head`. The file system can then read and/or write to this memory region. When the file system is done using the `buffer_head`, it must call `brelse` or buffers can be leaked.

Like many kernel interfaces, kernel block I/O relies heavily on pointers. However, raw pointers cannot be deferenced in safe Rust, and directly exposing these pointers to the file system results in safety errors. If the block I/O functions exposed to the file system accept

API Function	Description
bento_init(&mut self, req, devname, fc_info)	Initialize the file system.
bento_destroy(&mut self, req)	Destroy the file system.
bento_lookup(&self, req, parent, name, reply)	Lookup a file
bento_forget(&self, req, ino, nlookup)	Forget lookups of a file
bento_getattr(&self, req, ino, reply)	Get attributes
bento_setattr(&self, req, args..., reply)	Set attributes
bento_readlink(&self, req, ino, reply)	Read a symbolic link
bento_mknod(&self, req, parent, name, mode, rdev, reply)	Create a file node
bento_mkdir(&self, req, parent, name, mode, reply)	Create a directory
bento_unlink(&self, req, parent, name, reply)	Unlink a file
bento_rmdir(&self, req, parent, name, reply)	Remove a directory
bento_symlink(&self, req, parent, name, link, reply)	Create a symbolic link
bento_rename(&self, req, parent, name, newparent, newname, flags)	Rename a file
bento_link(&self, req, ino, newparent, newname, reply)	Create a hard link
bento_open(&self, req, ino, flags, reply)	Open a file
bento_read(&self, req, ino, fh, offset, size, reply)	Read data from a file
bento_write(&self, req, ino, fh, offset, data, flags, reply)	Write data to a file
bento_flush(&self, req, ino, fh, lk_own, reply)	Called on each close of a file
bento_release(&self, req, ino, fh, flags, lk_own, flush, reply)	Called on the last close of an open file
bento_fsync(&self, req, ino, fh, datasync, reply)	Sync a file
bento_opendir(&self, req, ino, flags, reply)	Open a directory
bento_readdir(&self, req, ino, fh, offset, reply)	Read a directory
bento_releasedir(&self, req, ino, fh, flags, reply)	Called on the last close of a directory
bento_fsyncdir(&self, req, ino, fh, datasync, reply)	Sync a directory
bento_statfs(&self, req, ino, reply)	Get file system statistics
bento_setxattr(&self, req, ino, name, value, flags, position, reply)	Set extended attributes of a file
bento_getxattr(&self, req, ino, name, size, reply)	Get extended attributes of a file
bento_listxattr(&self, req, ino, size, reply)	List extended attributes of a file
bento_removexattr(&self, req, ino, name, reply)	Remove an extended attribute of a file
bento_access(&self, req, ino, mask, reply)	Check file permissions
bento_create(&self, req, parent, name, mode, flags, reply)	Create and open a file
bento_getlk(&self, req, ino, fh, lk_own, start, end, typ, pid, reply)	Test for a file lock
bento_setlk(&self, req, ino, fh, lk_own, start, end, typ, pid, sleep, reply)	Acquire a file lock
bento_bmap(&self, req, ino, blocksize, idx, reply)	Map a block index within a file
bento_update_prepare(&mut self) -> Option<TransferOut>	Prepare to be removed during a live upgrade
bento_update_transfer(&mut, Option<TransferIn>)	Initialize during a live upgrade

Table 4.1: The file operations API, based on the FUSE lowlevel API with *bento_update_prepare* and *bento_update_transfer* added for live upgrade. File systems implement a subset of the provided functions. The *req* includes the requesting application's user id, group id, and process id. The *reply* data structures are used to return data or error values.

a pointer, the block I/O functions cannot be marked safe and the file system as a whole cannot be safe.

Exposing kernel services safely. Bento provides wrapping abstractions for kernel services so they can be used safely by the file system. These abstractions can be used like any other Rust data structures and functions. Several of the provided abstractions are detailed

in Table 4.2.

Object Type	Method	Kernel Equivalent	Description
BlockDevice	*bread(&self, ...)->Result<BufferHead>*	*__bread_gfp(...)*	Read a block from disk
	getblk(&self, ...)->Result<BufferHead>	*__getblk_gfp(...)*	Get access to a block
	sync_all(&self)->Result<i32>	*blkdev_issue_flush(...)*	Flush the block device
BufferHead	*data(&self)->&[u8]*	*buffer_head->b_data*	Get read access to data
	data_mut(&mut self)->&mut [u8]	*buffer_head->b_data*	Get write access to data
	drop(&mut self)	*brelse(...)*	Release the buffer
	sync_dirty_buffer(&mut self)->Result<c_int>	*sync_dirty_buffer(...)*	Sync a block
GlobalAllocator	*alloc(&self, ...)->*mut u8*	*kmalloc(...)/vmalloc(...)*	Allocate memory
	dealloc(&self, ...)	*kfree(...)/vfree(...)*	Free allocated memory
RwLock<T>	*new(data:T)->RwLock<T>*	*init_rwsem(...)*	Create a RwLock of type T
	read(&self)->LockResult<ReadGuard<'_,T>>	*down_read(...)*	Acquire the read lock
	write(&self)->LockResult<WriteGuard<'_,T>>	*down_write(...)*	Acquire the write lock
TcpStream	*connect(addr: SockAddr) -> Result<TcpStream>*	*sock_create_kern(...)* *kernel_connect(...)*	Create and connect
	read(&mut self, ...)->Result<usize>	*kernel_recvmsg(...)*	Read a message
	write(&mut self, ...)->Result<usize>	*kernel_sendmsg(...)*	Send a message
	drop(&mut self)	*sock_release(...)*	Cleanup the TcpStream
TcpListener	*bind(addr: SockAddr)->Result<TcpListener>*	*sock_create_kern(...)* *kernel_bind(...)* *kernel_listen(...)*	Create, bind, and listen
	accept(&self)->Result<(TcpStream, SockAddr)>	*kernel_accept(...)*	Accept a connection

Table 4.2: The Bento kernel services API. These are some of the data structures and methods provided to the file system. Methods that take *&mut self* can modify the object. Methods that take *&self* can access but not modify the object.

To be concrete, we address the example discussed above. We provide a safe `BlockDevice` abstraction to represent a kernel block device. A `BlockDevice` takes the name of the block device file and the block size; it contains a pointer to the kernel block device and the block size as fields. It provides several methods, including a safe `bread` method that takes a block number as an argument, performs safety checks, and calls `__bread_gfp` using the correct page allocation flag. The `bread` method returns a `BufferHead` that wraps the kernel `buffer_head`. A `BufferHead` method converts the pointer and size fields into a sized memory region that can be used safely. That method must use unsafe code to make the sized memory region out of the unsized pointer and size fields, but the file system can call the method safely. To prevent accidental memory leaks, we call the `brelse` function in the `drop` method of the `BufferHead` wrapper. With this, buffer management has the same properties as memory management in Rust: memory leaks are possible but difficult.

LibBentoKS provides synchronization primitives including `RwLock<T>`, a wrapper around the kernel read-write semaphore. It has the same interface as the Rust standard library `RwLock<T>`, a read-write lock that protects data of type `T`. To obtain an immutable ref-

erence to the protected data, the user must acquire the read lock; to obtain a mutable reference, the user must acquire the write lock. `ReadGuard` calls `up_read` in `drop` and `WriteGuard` calls `up_write` in `drop`, preventing the user from forgetting to unlock.

In addition libBentoKS provides an implementation of the Rust global allocator that uses `kmalloc` and `kfree` for small regions (less than 8 pages) and uses `vmalloc` and `vfree` for larger regions. In this way, file system developers can use dynamically allocated types such as a growable array (Rust's `alloc::vec::Vec`) and collection types (from Rust's `alloc::collections`). LibBentoKS provides `TcpStream` and `TcpListener` to support networked file systems.

These abstractions can, in some cases, add a small amount of performance overhead. If a kernel function has requirements on its arguments, the wrapping method likely will need to perform a runtime check to ensure that the requirements hold.

4.4 File System Upgrade

To enable online upgrades that are transparent to applications using the file system, we must first identify when it is safe to upgrade the file system and how to handle long-lived file system state. If an upgrade occurs while file system operations are still pending, there may be race conditions where some operations are executed on the old file system and others on the new, leading to correctness problems. In addition, any state that affects the semantic behavior of the file system, such as in-progress disk requests, file system journals, and TCP connections for networked file systems, must be correctly preserved across the upgrade. State that affects performance but not semantics, such as clean data in caches, can be optionally preserved.

Bento addresses these challenges by ensuring that the old file system is in a quiescent state and that semantic state is transferred to the new file system. Bento quiesces the file system by pausing new calls into the file system module during the upgrade and waiting for in progress operations to complete. To achieve this, Bento uses a read-write lock on the file system connection. All calls into libBentoFS acquire the read lock, while upgrades acquire the write lock. Therefore, file system operations can be executed concurrently in normal mode but will be blocked during an upgrade; the upgrade will be blocked until previous operations complete.

Second, a constraint on the old file system is that it must be able to transfer its semantic state to the new file system. Of course, the specific content of this state will vary from file system to file system. Each file system defines two data structures: one that is returned

when the file system is removed and one that is expected when the file system is replacing a previous live file system. This design pattern, of needing to write code to support both past and future versions, is common in cloud settings. During upgrade, ownership of the data structure is passed from the old file system to the new one. BentoFS handles passing the data structure from the old file system to the new file system.

The detailed mechanisms involved for live upgrades are shown in Figure 4.1 and described below:

1. A new file system upgrade instance is loaded into the kernel. At module load, it calls into BentoFS to register itself and indicate that it is an upgrade.
2. BentoFS identifies the file system that needs to be unloaded and acquires the lock to pause new operations and wait for existing operations to complete.
3. BentoFS sends a `bento_update_prepare` request to the old file system through libBentoFS.
4. The old file system instance handles the `bento_update_prepare` request, performing any necessary cleanup and creating and returning its defined output state transfer struct to BentoFS through libBentoFS.
5. BentoFS sends a `bento_update_transfer` request to the new file system through libBentoFS, passing the state transfer data structure to the new file system.
6. The new file system instance initializes itself using the provided state and returns.
7. BentoFS modifies the connection state by replacing the old file system reference with the new file system reference and releases the write lock, allowing calls to proceed to the new instance.

4.5 Userspace Debugging

Bento also introduces a feature that enables a new file system to be seamlessly hoisted to userspace for debugging. This enables developers to leverage gdb and other familiar utilities for higher velocity development. Debugged code can then be dropped back into the kernel without any modification. Bento supports this feature by exposing identical interfaces to both the kernel version and the userspace version of a developed file system. Whether the file system runs in the kernel or at userspace is determined by a compilation

configuration flag which specifies which libraries will be linked and how the file system should register itself during initialization.

Our solution leverages Linux kernel FUSE support to forward file operations to userspace. By itself, this is not sufficient — a FUSE file system is not runnable in the kernel. At a high level, we design our kernel interfaces to mirror existing userspace interfaces when possible, and implement userspace libraries to expose additional abstractions otherwise.

Many kernel interfaces can be designed to expose the same interfaces as userspace abstractions. For example, kernel read-write semaphores are used the same way as Rust's `std::sync::RwLock<T>` and the kernel TCP stack provides similar interfaces to Rust's `std::net::TcpStream` and `std::net::TcpListener`. In these cases, our kernel services API provides interfaces that are identical to the analogous userspace interface.

However, some kernel interfaces do not have obvious userspace analogues. The file operations API (Table 4.1), for example, adds functions to implement state transfer and passes immutable references to ensure correct concurrency behavior. Additionally, operations on the backing storage device are performed differently from the kernel and userspace. FUSE file systems typically use file I/O to access the storage device while kernel file systems directly interface with the kernel buffer cache. Using a file I/O interface in the kernel would significantly hinder performance and functionality, adding extra data copies and preventing certain optimizations. However, there is no standard userspace abstraction that closely mirrors the kernel buffer cache.

To address this, we provide two additional libraries The userspace version of libBentoFS translates calls from FUSE into the file operations API. The userspace version of libBentoKS implements a basic buffer cache that uses file I/O under the hood, providing the `BlockDevice` and `BufferHead` abstractions to Bento file systems when running at user level.

4.6 Limitations

Bento has a number of limitations. Bento cannot catch all types of bugs that occur in Linux kernel modules. Because Bento relies on language-level safety checks, it cannot catch high level logic bugs. Bento also cannot catch deadlocks.

Bento's userspace debugging is has poor performance because it writes to the disk through userspace files, incurring significant overhead per file write. Additionaly, for the userspace debugging to work, any kernel functions called by the file system must have a

userspace library. This is simple for functions that mirror those provided by userspace libraries, but for other functions, Bento must provide a userspace implementation of the behavior.

Live upgrade quiesces the system, resulting in a short blackout period for the file system, where a more complicated mechanism for live upgrade could reduce or eliminate the blackout period. State transition data structures must also be carefully constructed. For the state transition to function correctly across the upgrade, the data structure exported by the old file system must match the data structure expected by the new file system. Bento does not verify that the data structures are correct, so incorrect data structure specifications could result in bad behavior.

4.7 Implementation

We have developed Bento as a Linux kernel module for BentoFS and a Rust library containing both libBentoKS and libBentoFS in 5240 lines of C and 5072 lines of Rust. The userspace versions of libBentoKS and libBentoFS are another 986 lines of Rust. The current implementation targets Linux kernel version 4.15. The file system is compiled as a Rust a static library, which can be linked with any required C code to generate the .ko kernel module. Kernel code in Rust cannot use standard libraries, but we do enable use of the Rust alloc crate.

BentoFS We built BentoFS by modifying the existing Linux FUSE kernel module. In place of upcalls, BentoFS communicates with libBentoFS using function calls. A file system module registers itself with BentoFS by providing a pointer to the `dispatch` function when it is mounted. Like the VFS layer, BentoFS maintains a list of active file systems, locking the list and adding and removing entries when file systems are registered or unregistered. This list is additionally locked during a live upgrade.

Upgrade State Transfer. Ownership of state transfer data structures must be moved between the Rust file system modules during an upgrade to allow the new file system instance to take ownership of state owned by the old file system instance. We implement this ownership transfer in libBentoFS using the Rust *Box* type. When the old file system instance returns its state to libBentoFS, we create a *Box* to take ownership of the data and pass the box as a raw pointer to BentoFS. The new libBentoFS converts the pointer back to a *Box*, claiming ownership of the data before passing it to the file system. Rust deletes

the old file system data structure when it goes out of scope at the end of the transfer; the old file system is uninstalled in the background.

4.8 Experiences with Bento

We began this project developing both a Bento version of a file system and its VFS equivalent in C, as a way to quantify the performance cost of Bento. However, we eventually stopped development on the VFS version because implementing and debugging new features were significantly more time consuming and difficult than for the Bento version. In VFS, we were much more likely to accidentally write memory errors, such as `NULL` pointer dereferences and memory leaks. These bugs took much longer to diagnose and fix than bugs in the Bento version because they would crash the kernel, forcing us to reboot between tests, and they were difficult to isolate.

We further illustrate our experience developing with Gento on three axes: functionality, performance, and correctness.

Functionality. Using Bento, we implemented Bento-fs, a file system designed to have ext4-like performance, in 3038 lines of safe Rust code. Bento-fs is structurally similar to the xv6 file system, a simple file system included in MIT's teaching operating system xv6 [42]. This simplicity made the xv6 file system an attractive starting point for our prototype. Bento-fs includes several modifications for improved functionality and performance. For example, xv6 does not fully support the functionality necessary to run our benchmarks. Likewise, we added double indirect blocks to support files up to 4GB, instead of 4MB in xv6.

We also added a provenance feature to Bento-fs. The architecture of provenance tracking is borrowed from existing work [91, 107]. It consists of two pieces: a) a file system component that tracks file creations, deletions, and opens; and b) a syscall-level component that tracks the process hierarchy and operations on open file descriptors, such as dup and sendmsg.

The file system-level component is implemented by logging information to a special file. To track existing files, 'create', 'rename', 'symlink', and 'unlink' operations log the user process ID of the request, the names and inode numbers of relevant files, any request flags, and, for 'unlink', whether or not the file was deleted. The current implementation does not track hard links, but adding such support could follow a similar strategy. Since Bento-fs is not called for every read or write operation due to kernel caching, we track file accesses

by logging 'open' and 'close' calls, recording the read/write mode of the open call along with the process ID of the request and the inode number of the file. If a file is opened as writable while another file is opened as readable, provenance tracking assumes that the writable file's contents depends on the readable file's contents.

The syscall-level component tracks process creation through 'fork'/'exec' and operations on open file descriptors so the provenance system can correctly handle instances where a process gains access to a file without using the open syscall. This component is implemented as a collection of eBPF programs that log the relevant system calls, namely 'clone', 'exec', 'pipe', 'dup', 'dup2', and 'sendmsg'. 'Open' calls are also logged so the file descriptors used in the system calls can be matched to the file system tracking on file names.

Overall, these features were added to Bento-fs in 145 lines of code in two weeks of development. In our development process, we never caused a crash of the operating system and were able to test and debug code within minutes of making changes. In fact, many of our changes worked correctly once they compiled, something that has not been true of our C development.

Performance. To be able to bound the overhead imposed by Bento by comparing it to ext4, we added various optimizations to Bento-fs to match ext4 behavior. We particularly noticed overhead on multi-threaded and metadata intensive benchmarks. The xv6 free inode and free block implementations, for example, are needlessly inefficient. The journal used by xv6 is small by default and assumes that each operation will use the maximum number of blocks, limiting it to only three concurrent operations at once. It also commits operations to the device synchronously when transactions are completed. We increased the size of the log and leveraged the Linux journal module JBD2 (also used by ext4). In JBD2, transactions request the necessary number of blocks and commit in the background. [1]

Similarly, xv6 uses an inefficient list structure for directories. We added tree-structured directories that use the hash of the file name to locate directory entries.

Most of the code changes for the journal modifications were in libBentoKS and mkfs. Tree structured directories were implemented within Bento-fs in around 800 lines of code, split across utility functions for the hash tree and directory lookup, linking, and reading. Having access to dynamically allocated data structures from Rust's `alloc` crate simplified this implementation. The tree structure uses the B-tree implementation provided by the

[1] Although we implemented a log manager for the userspace version, it is likely less optimized than the kernel version, and there may be additional ways to improve userspace write performance that we have not yet discovered.

crate and the directory lookup, linking, and reading code use Rust's dynamically allocated array `Vec`.

Correctness. We tested the correctness of our file system using CrashMonkey [105]. It generates workloads based on operations supported by the file system, and exhaustively tests all combinations up to a defined sequence length. We ran the `seq-2` benchmarks [105], which test sequences of two operations, using the operations supported by Bento-fs. This resulted in 47314 benchmarks in total. CrashMonkey did not find any crash consistency bugs in Bento=fs. It found a known bug from the FUSE kernel module in the C code used in BentoFS where opening a directory then calling `rmdir` followed by `mkdir` on the directory name before closing it resulted in an unusable directory due to inode reuse. We fixed this by always allocating a new inode during directory creation.

The provenance extension to Bento-fs was also used by two groups of students to create two applications in the context of a class. One of these applications automatically recreated derived files when input files changed, specifically recompiling an executable based on the input C files, inspired by past work on transparent make [151]. The other application performed automatic directory synchronization, syncing files in a local directory to remote storage. In these student projects, we found that Bento was robust enough to support a smooth development experience.

4.9 Evaluation

Our evaluation of Bento aims to answer two questions: a) How well does Bento-fs perform on different workloads? and b) How expensive are live upgrades?

4.9.1 Experimental setup

Baselines. We compare: a) ext4-o: ext4, the default file system on most Linux versions, using the default `data=ordered` option with metadata journaling, b) ext4-j: ext4 with data journaling (`data=journal` mode) c) Bento-fs, and d) Bento-fs running in userspace. We focus our evaluation on ext4 with journaling because Bento-fs also implements data journaling. Note that Bento-fs has implemented only a subset of ext4's optimizations. The userspace version of Bento interacts with the storage device by opening it with the `O_DIRECT` flag.

Environment. All experiments were run on a machine with Intel Xeon Gold 6138CPU (2 sockets, each with 20 cores, 40 hyperthreads), 96 GB DDR4 RAM, and a 480 GB Intel Optane SSD 900P Series with 2.5 GB/s sequential read speed and 2 GB/s sequential write speed. All benchmarks were run using the SSD as the backing device using the cores and memory on the socket connected to the SSD.

4.9.2 Microbenchmarks

We ran microbenchmarks from the Filebench benchmarking suite. The workloads included sequential read, random read, sequential write, random write, and create and delete benchmarks. All workloads except for sequential write are run with both 1 thread and 40 threads. Read and write benchmarks were executed on a 4GB file using four different operation sizes: 4, 32, 128, and 1024KB. The create workloads create 800,000 16KB files in the same directory, allocating half before the start of the benchmark. The delete workloads delete 300,000 16KB files across many directories, with an average of 100 files per directory. All benchmarks were run 10 times, and averages and standard deviation were calculated. Table 4.3 shows the results on ext4 with both the default metadata journaling and data journaling, Bento-fs, and Bento-user, the userspace version of Bento-fs. Results are colored based on the performance compared to ext4.

Reads. Reads on all three file systems have similar performance for all sizes and both single-threaded and 40-threaded, and large reads achieve greater bandwidth than provided by the device. This is because data is cached quickly after the first read, and all subsequent reads hit in the page cache. The userspace version uses the kernel cache in the FUSE kernel module before forwarding requests to userspace, so it performs similarly to direct kernel implementations.

Writes. For small write benchmarks, Bento-fs and ext4-j have fairly similar write performance. Bento-fs has higher performance than ext4-j and similar performance to ext4-o on large write benchmarks due to slight implementation differences. Whereas ext4-j logs blocks to the journal on the write syscall path, Bento-fs logs asynchronously in the writeback cache when data is flushed. This performance difference is more prominent for single-threaded benchmarks with large writes because these are more likely to stress the journal in ext4-j without stressing the writeback cache. For all cases, the user-level implementation is much slower because it incurs additional kernel crossings and issues block I/O from userspace. Each operation must first pass from the kernel back to the userspace, which will then be translated into several read/write operations on the storage device. Each system call to the device file must in turn pass through the VFS layer to

Benchmark	ext4-o	ext4-j	Bento-fs	Bento:ext4	Bento-user	user:ext4
seq. read, 1-t, 4k	286 (±2)	287 (±2)	289 (±4)	1.01	290 (±2)	1.01
seq. read, 1-t, 32k	1811 (±20)	1796 (±21)	1817 (±18)	1.01	1807 (±18)	1.00
seq. read, 1-t, 128k	4170 (±55)	4071 (±75)	4119 (±82)	1.01	4112 (±50)	1.01
seq. read, 1-t, 1024k	6434 (±129)	6580 (±197)	6730 (±197)	1.02	6510 (±160)	0.99
seq. read, 40-t, 4k	429 (±7)	433 (±9)	436 (±7)	1.00	429 (±9)	0.99
seq. read, 40-t, 32k	3372 (±65)	3561 (±332)	3488 (±184)	0.98	3417 (±56)	0.96
seq. read, 40-t, 128k	17668 (±143)	17878 (±162)	17784 (±132)	0.99	17833 (±168)	1.00
seq. read, 40-t, 1024k	21407(±1774)	22024 (±101)	22082 (±339)	1.00	22136 (±101)	1.00
rand. read, 1-t, 4k	150 (±1)	149 (±2)	149 (±2)	1.01	149 (±3)	1.00
rand. read, 1-t, 32k	1037 (±6)	1044 (±6)	1049 (±8)	1.00	1041 (±6)	0.99
rand. read, 1-t, 128k	2901 (±20)	2955 (±36)	2957 (±33)	1.00	2908 (±31)	0.98
rand. read, 1-t, 1024k	5836 (±68)	5961 (±152)	5967 (±116)	1.00	5890 (±131)	0.99
rand. read, 40-t, 4k	223 (±24)	211 (±2)	217 (±5)	1.02	218 (±5)	1.02
rand. read, 40-t, 32k	1717 (±34)	1712 (±34)	1737 (±37)	1.01	1738 (±31)	1.02
rand. read, 40-t, 128k	9265 (±104)	9232 (±70)	9206 (±132)	1.00	9224 (±55)	1.00
rand. read, 40-t, 1024k	21635 (±46)	21650 (±49)	21637 (±50)	1.00	21569 (±54)	1.00
seq. write, 1-t, 4k	234 (±7)	172 (±3)	252 (±6)	1.46	3.7 (±0.0)	0.02
seq. write, 1-t, 32k	860 (±86)	409 (±1)	1003 (±65)	2.45	4.0 (±0.1)	0.01
seq. write, 1-t, 128k	1058 (±109)	430 (±44)	1774 (±352)	4.12	4.0 (±0.1)	0.01
seq. write, 1-t, 1024k	1365 (±0)	469 (±62)	1843 (±329)	3.93	4.0 (±0.0)	0.01
rand. write, 1-t, 4k	142 (±3)	120 (±1)	139 (±2)	1.16	8.5(±0.14)	0.07
rand. write, 1-t, 32k	875 (±7)	395 (±22)	898 (±9)	2.27	10.1 (±0.0)	0.03
rand. write, 1-t, 128k	1952 (±16)	330 (±18)	2167 (±62)	6.55	10.3 (±0.1)	0.03
rand. write, 1-t, 1024k	3051 (±35)	309 (±8)	3789 (±56)	12.24	10.1 (±0.3)	0.03
rand. write, 40-t, 4k	230 (±3)	208 (±4)	241 (±14)	1.15	9.2 (±0.1)	0.04
rand. write, 40-t, 32k	1237 (±46)	357 (±61)	1500 (±34)	4.20	10.0 (±0.2)	0.03
rand. write, 40-t, 128k	1414 (±43)	303 (±10)	1894 (±39)	6.24	10.4 (±0.1)	0.03
rand. write, 40-t, 1024k	1391 (±49)	296 (±13)	1924 (±78)	6.50	11.0 (±0.0)	0.04
create, 1-t, ops/s	12510 (±418)	8564 (±186)	12087 (±390)	1.41	194 (±5)	0.02
create, 40-t, ops/s	34377(±2157)	17858 (±0)	18819 (±663)	1.05	216 (±2)	0.01
delete, 1-t, ops/s	23331 (±878)	22913 (±0.3)	24997 (±0)	1.09	827 (±11)	0.03
delete, 40-t, ops/s	60493(±7088)	63253(±7101)	57253(±6258)	0.91	808 (±27)	0.01

Table 4.3: Bento performance results for ext4 in `data=ordered` mode (ext4-o), and `data=journal` mode (ext4-j), Bento-fs, and a userspace version of Bento-fs (Bento-user) on Filebench microbenchmarks using varying operation sizes and 1 and 40 threads. Reads and writes are measured in MBps. Reads and writes are cached in the kernel and so can outperform the 2.5 GBps and 2.0 GBps device read and write speed. Results are averaged over 10 runs and standard deviations are included in parentheses. Color indicates performance relative to ext4-j. Bento-fs performs similarly to ext4-j for most benchmarks. Both significantly outperform Bento-user.

reach the kernel block cache; this is much slower than direct accesses to the kernel block cache by a kernel file system. Additionally, Bento-user does not have access to the JBD2 module, so it uses a simpler journal that is less efficient on large write workloads. This journal is also affected by slow userspace block I/O.

Creates+Deletes. On the create and delete benchmarks, ext4-j and Bento-fs have

similar performance. Bento-fs outperforms ext4-j on single-threaded creates, likely due to the write speedup. Ext4-o outperforms Bento-fs on multi-threaded creates. Both ext4 modes and Bento-fs outperform the user-level file system for the same reason as the write benchmarks.

4.9.3 Application Workloads

Next, we run three application-style workloads from Filebench, four applications, and two workloads each on two different key-value stores. All workloads were run 10 times and averages and standard deviation were calculated. From Filebench, we ran 'varmail', 'fileserver', and 'webserver'. (1) The 'varmail' mail-serving workload uses 16 threads to create and delete 1000 files in one directory and performs reads and writes followed by fsyncs to these files. (2) The 'fileserver' file-serving workload uses 50 threads to create and delete 10,000 files across 500 directories and executes reads and appends to these files. (3) The 'webserver' web-serving workload uses 100 threads to read from 1000 small (16KB average size) files across around 50 directories and append to an operation log. All benchmarks execute for one minute. For application workloads, we used 'tar', 'untar', and 'grep' on the Linux kernel source code and 'git clone' on the xv6 source repository.

We also evaluate read and write workloads on the Redis [132] and RocksDB [134] key-value stores. Redis is an in memory key-value store used in distributed environments. By default, it periodically dumps the database to a file but can be configured to also log all operations to an append-only-file (AOF) for persistence. In our evaluation, we use the AOF and configure it to sync every second. We run the 'set' and 'get' workloads from redis-benchmark, the provided benchmarking utility, for 1,000,000 operations using 100B values. RocksDB is a persistent key-value store developed by Facebook based on Google's LevelDB [46]. Using db_bench, the included benchmarking utility, we evaluate the 'fillrandom' and 'readrandom' workloads each for 1,000,000 operations using 100B values.

Filebench: Figure 4.2 presents the application-style Filebench results for the three file systems described earlier, plus Bento-fs with file provenance (Bento-prov). Across all benchmarks, Bento-fs (with or without provenance) outperforms Bento-user by 10-400x due to the reasons discussed earlier. For varmail and webserver, ext4-j and Bento-fs exhibit similar performance, but for fileserver, Bento-fs significantly outperforms ext4-j due to an unintentional quirk in the benchmark. Filebench 'fileserver' executes many sequences of create-write-delete operations, but it does not sync the file before the file is deleted. With

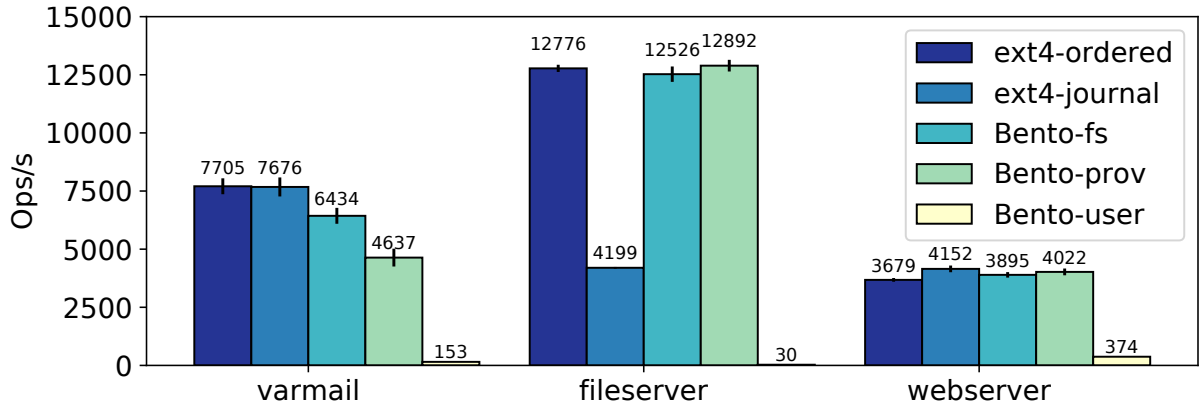

Figure 4.2: Bento performance results for ext4 in `data=ordered` mode and `data=journal` mode, Bento-fs, Bento-fs with provenance, and a userspace version of Bento-fs on Filebench application-style workloads in ops/s. Bento-user performs much worse on all benchmarks. Bento-fs and Bento-prov outperform ext4-journal on 'fileserver' due to different handling of un-synced writes to deleted files.

writeback caching, Bento recognizes that the pages belong to files that no longer exist, and drops the writes.

In ext4-j, on the other hand, writes are associated with the appropriate location on the storage device during the write syscall path by mapping the written page to the appropriate buffer head. This writeback code path therefore has no need to identify the written file and executes the block I/O regardless of whether the file exists or not. Like Bento-fs, ext4-o is able to drop the writes to the deleted files so both file systems show similar performance.

Applications: Figure 4.3 shows the results for application workloads. Here, Bento-fs outperforms Bento-user by 4-36x. The difference is particularly noticeable for 'untar' which involves many creates. Creates are particularly impacted by slow block I/O from userspace due to the large number of separate disk operations needed to modify the directory, allocate an inode, and fill the allocated inode. Relative to ext4-j, Bento-fs performs similarly on 'untar', 'tar', and 'git clone' and 19% worse on 'grep'. The slowdown is due to optimized page caching in ext4 that is not implemented in Bento-fs. Relative to ext4-o, Bento-fs performs 13% worse on 'untar' due to data journaling and the lack of delayed allocation. On other benchmarks, ext4-o shows similar results to ext4-j.

For most tested workloads, Bento-prov has similar performance to Bento-fs. Bento-fs outperforms Bento-prov on 'varmail' by 39%, 'untar' by 13%, 'grep' by 68% because

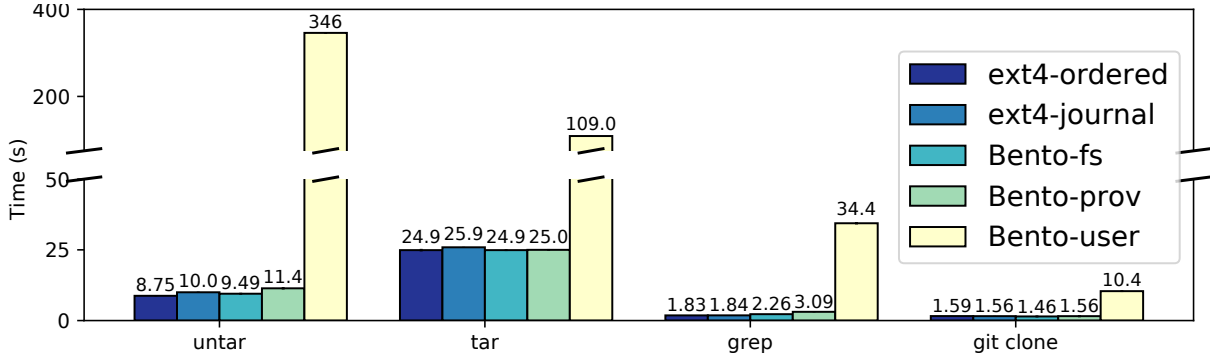

Figure 4.3: Bento performance results for ext4 in `data=ordered` mode and `data=journal` mode, Bento-fs, Bento-fs with provenance, and a userspace version of Bento-fs on application workloads 'tar', 'untar', and 'grep' on Linux source code and 'git clone' on xv6. Bento-user performs much worse than the other file systems. Ext4-journal performs somewhat better than Bento-fs and Bento-prov on 'grep'.

Bento-prov logs information on creates, deletes, opens, and closes. Similarly, Bento-prov is 25% slower on the multithreaded create microbenchmark.

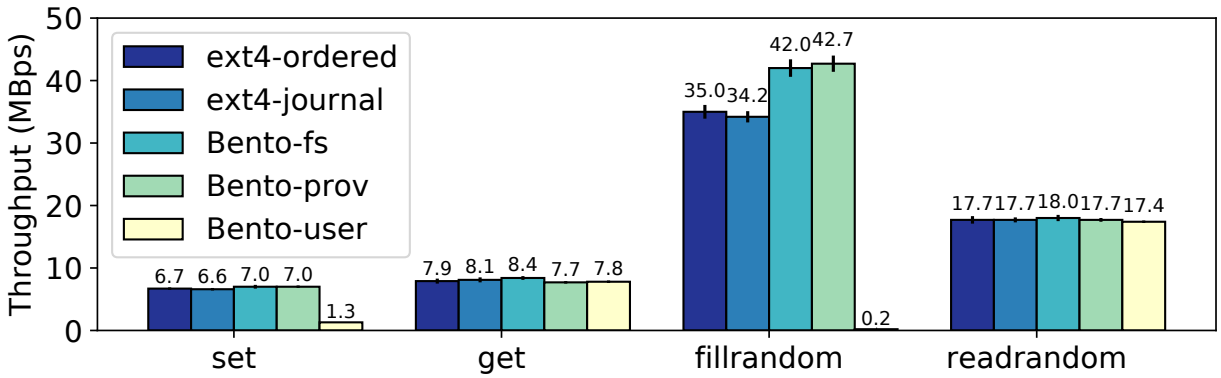

Figure 4.4: Bento performance results for ext4 in `data=ordered` mode and `data=journal` mode, Bento-fs, Bento-fs with provenance , and a userspace version of Bento-fs on Redis 'set' and 'get' and RocksDB 'fillrandom' and 'readrandom'. Bento-user performs much worse on write benchmarks.

Key-Value Stores: Figure 4.4 shows the results for Redis ('set' and 'get') and RocksDB ('fillrandom' and 'readrandom') workloads on the four file systems. Due to caching, Bento-

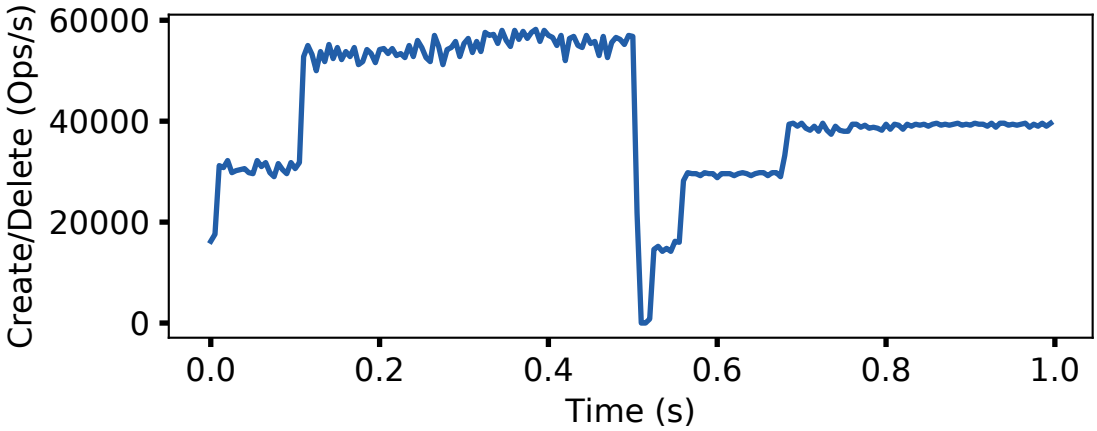

Figure 4.5: Bento create+delete performance during an upgrade from Bento-fs to Bento-prov, a provenance-tracking version of Bento-fs. At 0.5 seconds, Bento-fs is upgraded to Bento-prov. The system experiences around 15ms of downtime.

user performs similarly to the others on read-intensive workloads, but it performs much worse on writes. Bento-fs and Bento-prov show similar performance to ext4-j and ext4-o on reads but slightly outperform them on writes.

4.9.4 Live Upgrade

In this section, we measure the effect of a live upgrade on application file system performance during an upgrade of the file system from Bento-fs to Bento-prov. We do not use Filebench for these benchmarks so we can collect latency of individual operations. We ran two tests, both using a directory that initially contained 400,000 files. In the first, we executed a single thread that repeatedly created and deleted files. In the second, we executed 10 threads that repeatedly wrote and synced 64Kb writes to random files; we used 10 threads because with too many threads any service interruption caused by the upgrade was hidden by the latency variability of individual operations. In both tests, we upgraded to the version with provenance tracking after 0.5 seconds and completed the test after another 0.5 seconds. We converted the latency measurements into throughput by calculating the number of operations that occur each 5ms interval to smooth the data slightly. The results are shown in Figure 4.5 and Figure 4.6.

These graphs show a performance drop where the upgrade occurred at 0.5 seconds. In both tests, the upgrade took around 15ms, during which time the file system was

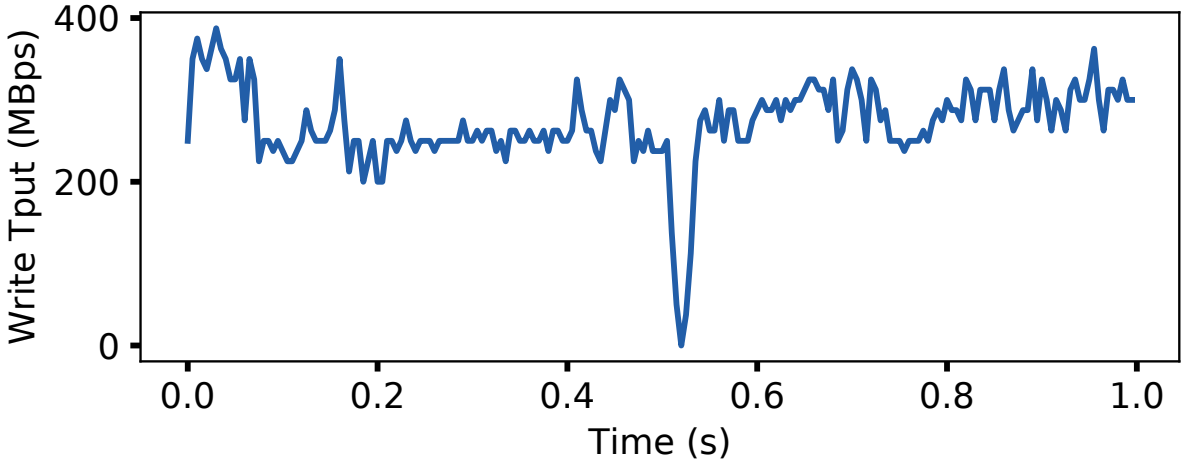

Figure 4.6: Bento synced writes with 10 threads performance during an upgrade from Bento-fs to Bento-prov, a provenance-tracking version of Bento-fs. At 0.5 seconds, Bento-fs is upgraded to Bento-prov. The system experiences around 15ms of downtime.

unavailable and a single operation per thread was blocked in the kernel. The performance recovered after the upgrade completed but create and delete performance was lower because the provenance-tracking file system performs extra work on these operations.

4.10 Summary

Bento is a framework for high velocity development of Linux kernel file systems that enables several goals: safety, performance, generality, compatibility with existing operating systems, ability to do live upgrade, and support for easy debugging. Bento provides these properties for file systems written in Rust, by translating Linux interfaces into safe interfaces with restricted memory sharing, supporting live upgrade with state transfer, and exposing identical interfaces to kernel and userspace file systems for userspace debugging. We implement Bento-fs, a simple file system using Bento and show that it has similar performance to ext4 and significantly outperforms the version of Bento-fs compiled to run in userspace. We develop a provenance tracking version of Bento-fs, and show that we can transparently upgrade Bento-fs to it with only 15 ms of service interruption to running applications.

Chapter 5

Enoki: High Velocity Linux Kernel Schedulers

Kernel scheduler behavior is central to application performance, adaptability to new hardware, and complex user requirements. Many applications have short, latency sensitive tasks and bursty workloads, where suboptimal kernel decisions can lead to high tail latency and poor overall job performance [75, 114, 47]. Heterogeneous hardware, such as non-uniform and tiered memory or accelerators, increases the complexity of scheduling decisions [14, 112]. Energy use is also increasingly important; neither underloading nor fully loading a server provides peak server energy efficiency [15, 85]. Additionally, applications may have information that can help improve scheduling decisions, such as workload characteristics or hardware preferences, but in most kernels the scheduler is oblivious to user preferences beyond simple priorities and hand coded placement [110, 123].

Efficiently supporting these changes will require new scheduler designs or new features in existing schedulers. Although kernel schedulers could theoretically be adapted to handle these new demands, developing and testing new kernel schedulers can be difficult and time consuming. Kernel code is difficult to write correctly and debug and slow to deploy. There are only three mainline schedulers implemented in the Linux kernel, the most widely used cloud operating system.

To address this, some researchers have used kernel bypass to implement new schedulers [75, 114, 128, 47]. This approach increases development velocity by removing the need to recompile the kernel and providing access to userspace debugging tools. However, it interferes with resource sharing between the scheduler and the rest of the system and complicates deployment and maintenance, limiting the potential reach of the research [139].

GhOSt [67] aims to provide a general purpose deployable solution for userspace schedulers in Linux. GhOSt uses an upcall approach where scheduling policy decisions are made by a userspace scheduler while the mechanism remains in the kernel. GhOSt schedulers can be implemented in small amounts of userspace code and redeployed easily, but each scheduling decision requires scheduling the userspace scheduler adding significant overhead to scheduling decisions. To mitigate some of the latency overhead, ghOSt uses an asynchronous model where the kernel can continue to take interrupts and make scheduling decisions while the userspace scheduler runs. This means the scheduling decisions may be out of date.

Other kernel subsystems, particularly networking, can use eBPF [53] for high velocity kernel development, and ghOSt implemented support for scheduler eBPF hooks. Using eBPF, users can load programs to customize kernel scheduler code, provided that the structure of the scheduler does not change. It is difficult to implement large or complicated code, such as an entire scheduler, using eBPF. For example, the default Linux scheduler, the Completely Fair Scheduler (CFS), is over 6,000 lines of code. Additionally, the eBPF trust model is not a good match for scheduling. eBPF considers loaded programs potentially malicious and verifies that they will not disrupt kernel execution. It is not clear how to implement this for scheduling since bad scheduling policy decisions, particularly choosing to run a task on a CPU where it is not queued, can cause the kernel to crash, violating eBPF's safety requirements.

In the previous chapter, we described Bento [102]. Bento [102] shows that we can have high development velocity and high performance for Linux kernel file systems by writing file systems in safe Rust and supporting live upgrade and userspace debugging. However, file systems and schedulers have different environments and performance tolerances, and the techniques from Bento cannot be applied directly to schedulers. Bento sits behind the file page cache, reducing the cost of interposition. It is not clear if Bento's design is low enough latency for scheduling. Schedulers cannot be run in userspace using the same type of synchronous trampoline as Bento because the scheduler code cannot block waiting for a response from userspace. While a 15 ms pause during redeployment is short for file systems, it would be too long to pause scheduling. It is also not obvious if Bento supports code that is general enough to meet the demands of scheduler designs. Schedulers are more performance critical than file systems, and may need to rely on specific data structure or algorithm designs to ensure performance Additionally, Bento does not address logical correctness errors. In the file system, non-memory safety correctness errors will generally affect only the applications using the file system, while correctness errors in the scheduler code can crash the kernel.

Inspired by our past insights in Bento, we build a framework called Enoki. The goal of

Enoki is to enable high velocity development and deployment of high performance schedulers in the Linux kernel. We envision that Enoki schedulers will be used for both research prototyping and production deployments. Enoki schedulers are fast to write and debug and easy to test with seamless resource sharing with the rest of the kernel. Enoki supports schedulers running in the Linux kernel for wide deployability, but our approach is not restricted to the Linux kernel. A new Enoki scheduler is written in safe Rust against a clean interface, making it less likely to introduce bugs. Enoki enables dynamic update of scheduling code in a live kernel without rebooting, with a pause of only 10μs. Using a record and replay system, scheduling policies can be debugged using userspace tools. Since Enoki schedulers are implemented in the kernel, they can coordinate easily with other kernel schedulers, such as passing cores between schedulers or applications.

We used Enoki to implement several different scheduling algorithms to demonstrate its flexibility in supporting a variety of scheduler designs and to understand the overhead and performance of Enoki compared to native implementations of the same schedulers. We implemented a weighted fair queuing scheduler and evaluated it against Linux's default scheduler CFS. Despite our scheduler being much simpler than CFS, and including the Enoki framework overhead, it achieves an average of only 0.74% slowdown across 36 application benchmarks, with a maximum slowdown of 8.57%. We also implemented the Shinjuku [75] scheduler, a locality aware scheduler that co-locates tasks of the same user-defined class, and the core arbiter from the Arachne multilevel thread scheduler. The Enoki Shinjuku and Arachne schedulers performed competitively with native implementations, and the locality aware scheduler showed the potential to provide significant performance benefit. Two of the schedulers were implemented by undergraduates with no prior Linux kernel programming experience. The code is available at https://github.com/smiller123/enoki.

5.1 Motivation and Approach

Linux includes three schedulers: a real time scheduler, an earliest deadline first scheduler, and the Completely Fair Scheduler (CFS). CFS is the default and implements a version of weighted fair queuing. These schedulers are all quite large and complex; CFS is over 6000 lines of code, and even the simpler real time and deadline schedulers are over 1500 lines of code. This complexity has led to a number of bugs, particularly performance bugs due to a complicated load balancing mechanism [95].

Although CFS's weighted fair queuing algorithm works well for many of the tasks run on desktop or server machines. Other schedulers can have advantages in some cases. With more application knowledge, more optimal decisions are possible. For example, for

workloads composed of many very small tasks, shorter time slices can lower average job completion time [75]. Multilevel scheduling can give better performance isolation and flexibility by allowing applications to define their own policies based on their workloads [128]. Nest [85] improves energy efficiency for jobs with fewer tasks than cores by reusing warm cores rather than spreading tasks across many cold cores. Because these schedulers do not need to work well in all circumstances, they can potentially be much smaller and simpler than CFS. Nevertheless, uptake into Linux has been slow.

We propose Enoki, a framework for high velocity development of Linux kernel schedulers. Enoki's trust model assumes that scheduler developers are trusted but clumsy; they are not trying to break the kernel but may accidentally introduce bugs. Our overall goal is to allow non-expert programmers to be able to successfully design, implement, debug, and deploy new schedulers.

There are several challenges to achieving high development velocity for Linux kernel schedulers:

- **Buggy code:** The Linux kernel is a monolith of complicated C code, causing bugs to be common. The lack of modularity in the Linux kernel and the potentially large consequences of kernel bugs force developers to code slowly and carefully. In Enoki, we rely on safe language features. By enabling schedulers implemented entirely in safe Rust and by introducing a novel application of the type system to check for scheduling correctness errors, we eliminate whole classes of bugs, such as memory errors and race conditions, without introducing significant performance overhead or overly limiting scheduler designs.

- **Disruptive upgrade:** Current Linux schedulers are compiled into the kernel source, so deploying a new scheduler requires recompiling the kernel and rebooting the machine, decreasing the availability of running applications. Enoki schedulers can be upgraded quickly, without rebooting the machine and with only a short period of service downtime.

- **Slow debugging:** The kernel does not have access to debugging tools, such as those commonly used in userspace. We support record and replay debugging in Enoki. To diagnose bugs not caught by the Enoki framework, developers can record the calls between the kernel and the scheduler and replay later entirely at userspace.

- **Limited interaction:** Some scheduler designs, such as two level schedulers that require coordination with user threads, are difficult to implement in Linux because they require interaction with the userspace tasks that Linux does not support. By

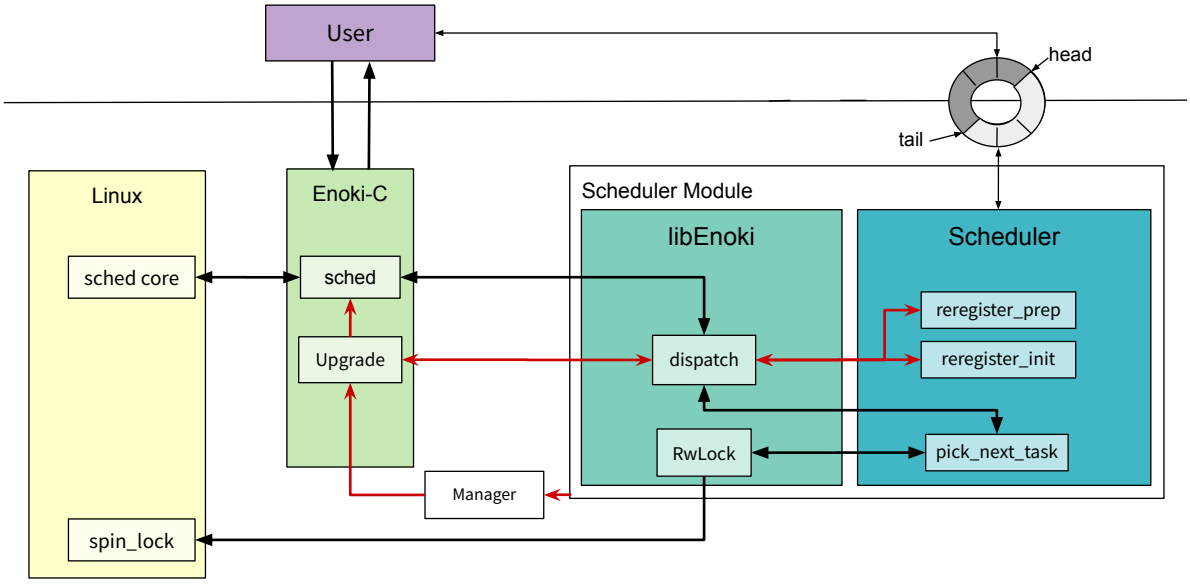

Figure 5.1: A high level diagram of Enoki. Enoki-C is written in C and compiled into the kernel. It interacts with libEnoki, a Rust library that is compiled with the scheduler code into the dynamically loaded scheduler module. The black lines represent code pathways during normal execution. Red lines represent module insertion and upgrade.

allowing generic, scheduler-defined interactions between the scheduler and userspace tasks, Enoki can support scheduler designs that cannot be implemented in Linux today, such as those that require application specific hints or two level schedulers.

- **Resource sharing:** Schedulers should be able to share resources with the rest of the system. In Linux, different applications can use different schedulers, sharing cores and cycles between the schedulers. Enoki schedulers are implemented in the kernel to enable fine grained core sharing across applications and schedulers.

- **Performance:** Schedulers execute often and concurrently on many cores and scheduler latency directly impacts application performance. Enoki is careful to impose low performance overhead and enable highly concurrent designs.

5.2 Enoki

The high level overview of Enoki is shown in Figure 5.1. Enoki is composed of two major components, Enoki-C and libEnoki.

Part of Enoki, Enoki-C is implemented in C and compiled into the Linux kernel. It interfaces directly with the core scheduling code and the kernel scheduling data structures. Enoki-C handles registering, deregistering, and upgrading schedulers and sets up and manages infrastructure for communication channels between userspace and the kernel scheduler and the record and replay system. It handles the unsafe work that is required for scheduling on behalf of Enoki schedulers, such as performing state updates to kernel `task_struct` data structure, managing interactions with the kernel run-queues when adding or moving tasks, and manipulating raw pointers to read and pass data to the scheduler module. Enoki-C also translates the calls from the core scheduling code into calls that the scheduler can implement safely by ensuring that all data passed to the scheduler can be safely accessed. Enoki schedulers do not directly manipulate kernel state or run-queues.

The other component, libEnoki, is a Rust library that is compiled with the scheduler code into a module that is dynamically loaded into the kernel. This library provides safe interfaces so the scheduler code can access the kernel and implements functions for loading and managing the scheduler. It contains some unsafe Rust because it must handle interactions with the C code in Enoki-C, which is inherently unsafe. Each scheduler is written entirely in safe Rust and only needs to provide the logic for the scheduling algorithm. The scheduler module is not sandboxed further; once it is loaded into the kernel, it runs like any other kernel code.

When the scheduler module is loaded, libEnoki calls Enoki-C to register the newly available scheduler. This registers the ID of the scheduler being loaded and a processing function in libEnoki for parsing calls from Enoki. User tasks can switch to using the new scheduler using its defined ID value. During regular operation, Enoki-C processes calls from the core scheduler code for these tasks, forwarding the calls to the processing function in libEnoki and managing updates to kernel data structures, such as the CPU's run queue. When the module is unloaded, libEnoki similarly unregisters the scheduler with Enoki-C, and no new tasks can be attached to the scheduler.

5.2.1 Safe Interfaces

Enoki provides schedulers with safe interfaces, both the interface that schedulers are required to implement and the interfaces for schedulers to access kernel functionality, such as

locks and timers. With safe interfaces, Enoki schedulers can be implemented entirely in safe Rust, preventing whole classes of bugs and reducing time spent debugging new schedulers. Enoki-C and libEnoki work together to provide safe interfaces for the scheduler code.

Enoki's threat model assumes that the developer is well meaning and knowledgeable but makes occasional mistakes. Enoki aims to help this developer prevent these mistakes by catching low-level errors that do not depend on the scheduler behavior, such as NULL pointer dereferences and data races, at compile time. Enoki does not aim to prevent all bugs, and bugs that depend on the scheduler's semantic behavior can remain uncaught. For example, schedulers implemented with Enoki can deadlock, lose tasks, and violate work conservation. We attempt to catch as many of these bugs as we can at runtime, but cannot guarantee that all instances are caught. We will discuss our mechanism for catching some of the semantic bugs later when discussing the API.

Enoki-C takes the interface defined by the core scheduler code and translates it into an interface based on message passing. Like the core scheduler code, this is a synchronous interface; it consists of function calls where the caller waits for the the callee to return before progressing. It is not necessary for safety that the interface be based on message passing, but it helps enforce certain safety properties, such as no shared pointers, preventing memory bugs across the interface. Enoki-C handles direct interactions with kernel data structures, such as pulling information like the runtime of the task or the current CPU of the task. This information is placed into per-function type "message" data structures that are passed to the registered processing function in libEnoki.

The processing function in libEnoki parses each "message" to determine which scheduler function is being invoked and handles the unsafety of interacting directly with C code through the Rust FFI (Foreign Function Interface) layer. The processing function pulls the fields from the "message" and passes them to the scheduler function being called. If the function returns a value, libEnoki writes that value back into the "message" data structure to return the value to Enoki-C.

To enforce that the scheduler code implements the required behavior, libEnoki provides a Rust trait for scheduler types. This trait defines the functions that a scheduler module must implement to be loadable as an Enoki scheduler. A trait defines a set of functionality that a type must have to implement the trait and can provide default implementations for shared behavior on those types. Traits can be used as bounds on function arguments, and any type that implements a trait can be used where the trait is called for.

The EnokiScheduler trait (shown in Table 5.1) specifies the functions that a scheduler should provide. Most of these functions are very similar to the functions defined by the core scheduling code to implement a Linux scheduler, with some notable differences.

The core function for a scheduler is `pick_next_task` which tells the core kernel scheduler code which task should be run next. Other important functions are `task_new`, `task_wakeup` and similar functions for tracking task state, `migrate_task_rq` for moving tasks between cores, `balance` for telling the core scheduler to move tasks to rebalance load, and error functions to return error values from the kernel. While these functions are modeled after the Linux scheduler interface they could be adapted to a different operating system. An Enoki scheduler is only expected to manage its own state in response to these calls; the kernel's core scheduling code decides when to call each function and Enoki-C manages kernel state.

For example, consider a simple scheduler that keeps a queue of tasks assigned to each core and schedules these tasks first come, first serve on each core. When a task is created on this scheduler, the kernel calls `select_task_rq`. The scheduler returns the core the task should be assigned to. The kernel attaches the task to the core and calls `task_new`. The scheduler places this task at the back of the queue for the specified core, storing the `Schedulable` (discussed more later). When a core becomes idle, the kernel calls `pick_next_task` for that core. The scheduler pops the first task off the core's queue and returns the `Schedulable` for the task. The kernel switches to the task and run it. If there is an error, the kernel calls `pnt_err`. If the task blocks, the kernel calls `task_blocked`. When the task wakes up, the kernel calls `task_wakeup`, and the scheduler adds the task to the back of the wakeup core's queue. Periodically, the kernel calls the scheduler's `balance` function on each core so the scheduler can rebalance tasks across cores. The scheduler returns the ID of any tasks it wants moved to the balancing core. The kernel then tries to move the task to the specified core. If it succeeds, the kernel calls `migrate_task_rq`, and the scheduler moves the task to the new core's queue. If it fails, the kernel calls `balance_err`.

While these functions are modeled after the Linux scheduler interface they could be adapted to a different operating system. An Enoki scheduler is only expected to manage its own state in response to these calls; the kernel's core scheduling code decides when to call each function and Enoki-C manages kernel state.

In Linux, schedulers are expected to track their tasks' runtimes using kernel timing functions. In our system, Enoki-C tracks the runtime of tasks on behalf of the scheduler and passes this to the scheduler when the task state changes, such as blocking, waking, and yielding, and to `pick_next_task`. Enoki records this information for deterministic replay.

API Function	Description
`get_policy(&self) → i32`	Get the policy number.
`pick_next_task(&self, ..., sched: Option⟨Schedulable⟩) → Option⟨Schedulable⟩`	Pick the next task for the CPU.
`pnt_err(&self, ..., sched: Schedulable)`	Could not schedule chosen task.
`task_dead(&self, pid)`	A task died.
`task_blocked(&self, ...)`	A task blocked.
`task_wakeup(&self, ..., sched: Schedulable)`	A task woke up.
`task_new(&self, ..., sched: Schedulable)`	There is a new task.
`task_preempt(&self, ..., sched: Schedulable)`	A task was preempted.
`task_yield(&self, ..., sched: Schedulable)`	A task yielded.
`task_departed(&self, ...) → Schedulable`	A task left the scheduler.
`task_affinity_changed(&self,..)`	Change task's allowed CPUs.
`task_prio_changed(&self,..)`	Change task's priority.
`task_tick(&self, ...)`	A timer has triggered.
`select_task_rq(&self, ...) → i32`	Choose the CPU for a task.
`migrate_task_rq(&self, ..., sched: Schedulable) → Schedulable`	A task is moving CPUs.
`balance(&self, ...) → Option ⟨u64⟩`	Rebalance tasks onto CPU.
`balance_err(&self, ...)`	Could not move the chosen task.
`reregister_prepare(&mut self) → Option⟨TransferOut⟩`	Prepare for an upgrade.
`reregister_init(&mut self, Option⟨TransferIn⟩)`	Initialize during an upgrade.
`register_queue(&self, RingBuffer⟨UserMessage⟩) -> i32`	Register a user to kernel queue.
`register_reverse_queue(&self, RingBuffer⟨RevMessage⟩) -> i32`	Register a kernel to user queue.
`enter_queue(&self, id: i32, ...)`	Check the user hint queue.
`unregister_queue(&self, id: i32) → RingBuffer⟨UserMessage⟩)`	Unregister the user to kernel queue.
`unregister_rev_queue(&self, id: i32) → RingBuffer⟨RevMessage⟩)`	Unregister the kernel to user queue.
`parse_hint(&self, hint: UserMessage)`	Synchronously parse hint.

Table 5.1: The API of the EnokiScheduler Trait. This is the API that a scheduler module must implement to be loadable as an Enoki scheduler. Most of the functions are used for managing the state of tasks in the scheduler. The `reregister` functions handle live upgrade. The queue functions and `parse_hint` are for user-to-kernel communication, and the reverse queue functions are for kernel-to-user communication.

The `pick_next_task` function in Linux expects the scheduler to choose a task on the CPU's run-queue, and if this expectation is violated, the kernel can crash. While this is a semantic bug, we attempt to catch it to limit kernel crashes due to buggy schedulers. To prevent this bug, we introduce a new type called `Schedulable` that represents a task and what core it can safely be scheduled on. When tasks are created, blocked or unblocked, or moved between run-queues, libEnoki creates a `Schedulable` data structure to indicate which core's run-queue the task is on and passes ownership of it to the scheduler at the corresponding call. The scheduler returns the data structure back to Enoki as the return value for `pick_next_task` as proof that the task can be safely scheduled on the core. In libEnoki, the core in the `Schedulable` is checked against the core being assigned, and if the check fails, the data structure's ownership is returned to the scheduler using the `pnt_err` call. A `Schedulable` also cannot be copied or cloned, so the scheduler cannot hold onto an old `Schedulable` to act as validation after it has returned it to Enoki when moving or running the task. The scheduler must instead receive a new `Schedulable`, such as through the `task_wakeup` call or as the current task in `pick_next_task`.

Enoki cannot always prevent the scheduler from holding onto an invalid `Schedulable`. The `migrate_task_rq` function, which moves tasks between cores, passes in a new `Schedulable` for the new core and requires the scheduler to return the old one so the scheduler will only have validation to run the task on one core. We cannot check at compile time that the scheduler returns the correct `Schedulable` for the old core, so it is possible for a scheduler to keep the wrong data structure. Additionally, we cannot require that the scheduler return a `Schedulable` in `task_blocked` or `task_dead` because these functions are sometimes called when the task is not schedulable and the scheduler has nothing to return.

In libEnoki, we also provide a safe interface for receiving hints from userspace (§5.2.3).

5.2.2 Live Upgrade

We support live upgrade of schedulers in Enoki. Live upgrade allows an upgraded version of a scheduler to replace an older version of the same scheduler without rebooting the machine or killing any of the tasks using the scheduler. Because the scheduling code is running throughout the upgrade, we must ensure that any state maintained by the scheduler, such as task runtimes and which task is runnable on which core, remains consistent across the upgrade and is available to the new scheduler after the upgrade.

We ensure consistency of the state by quiescing the scheduler module during the upgrade. Since the scheduler module is only invoked from Enoki-C, we know that the state of the scheduler will remain consistent as long as all calls from Enoki-C are completed

before the upgrade begins and any new calls wait until the upgrade is completed. We implement this using a simple per-scheduler read-write lock. Non-upgrade calls into the scheduler module acquire the lock in read mode, allowing multiple concurrent calls into the scheduler module. When an upgrade begins, the lock is acquired in write mode, preventing any of the non-upgrade calls from entering the scheduler module. When the upgrade is finished, the lock is released, and the non-upgrade calls can proceed, now directed to the new version of the scheduler.

After the scheduler module is quiesced, Enoki-C calls into `reregister_prep`, informing the scheduler that an upgrade is occurring. The module then performs any necessary maintenance and returns a data structure with any state it wishes to pass to the new version of the scheduler. Enoki-C then calls `reregister_init` in the new version of the scheduler, passing the data structure returned from the old scheduler. The new scheduler can then initialize itself based on the provided state, claiming ownership of some or all of the state if it wishes. Because Enoki-C acquired the read-write lock described above in write mode, we know that no other calls can enter either scheduler module during the upgrade, and it is safe for the scheduler to manipulate its internal state.

After ensuring that state remains consistent across the scheduler during an upgrade, the rest of the upgrade is fairly straightforward. Enoki-C swaps pointers so that its internal data structure now points to the new module, and the upgrade completes. Calls to the scheduler will use the new pointer, and so will call into the new scheduler module. The old scheduler module can then be safely unloaded.

Limitations. Because we quiesce the scheduler during an upgrade, there is a period when the scheduler is blocked and cannot accept non-upgrade calls, leading to a short service blackout. Enoki trusts the scheduler to have short, well-defined code paths so the read locks will be given up quickly and the upgrade can progress. We also trust the scheduler to upgrade and release the write lock quickly. Another limitation is that the state passing data structure that the new scheduler expects must be the same as the state passing data structure exported by the old scheduler because the memory is passed directly. This data structure is otherwise completely custom, and the new scheduler can export a different data structure to the next scheduler upgrade. This does not mean the upgraded scheduler must use the same state layout as the old scheduler. Rather the new scheduler needs to initialize itself using the state passing data structure exported from the old scheduler, but can define a new state passing data structure for the next version.

5.2.3 Custom Scheduler Hints

Correct scheduling decisions often depend on the behavior of the workload being scheduled. Tasks that communicate with each other or operate on the same data benefit from being scheduled on the same NUMA node, or with more recent split last level cache architectures, on cores with the same last level cache [110]. On a heterogeneous CPU, some tasks might prefer to be scheduled on certain cores or devices or co-located with other tasks [112].

In addition to sending scheduling hints from userspace to the kernel, it can also be useful for information to flow from the kernel to userspace. For example, in scheduler activations [7], the scheduler provides information about scheduling events to the user-level thread scheduler, such as when cores become available or when user tasks block in the kernel. Another example could be to utilize machine learning to improve scheduling decisions [59].

Enoki supports custom scheduler-defined hints, both from userspace to the kernel and vice versa. Each scheduler that supports hints defines data structures indicating the type of hints that it expects to receive and the type of hints it will send to the user. We enforce that these types can be read-shared across the user/kernel boundary without violating memory safety, but otherwise put no restrictions on them. For example, for our locality aware scheduler, we pass locality hints in the form of the task ID and the hint value. For our two-level scheduler, we pass core requests in the form of the process ID and the number of cores per priority level, and core reclamation requests as a single boolean value. Other applications can define the hint data structure as needed based on the use case.

Queues can be shared across a live upgrade as long as both versions of the scheduler use the same hint data structures. The scheduler passes the queues as part of the shared state during the upgrade. If the next scheduler version will use different hints, old queues must be closed before the upgrade, and new ones can be created after.

5.2.4 Record and Replay

Kernel debugging is notoriously difficult. Many debugging tools are difficult or even impossible to use on the kernel. While Enoki prevents many bugs that would crash the kernel, logic bugs can still exist in the schedulers. We want to provide a mechanism to simplify debugging by allowing the scheduler code to be run and debugged at userspace.

To this end, Enoki implements a record and replay system for scheduling events. When running in record mode, libEnoki records each call and hint sent to the scheduler. The replay system implements a replacement version of libEnoki to replay these records to the

scheduler, now running in userspace. The exact same scheduler code is run during both record and replay. The replay is primarily aimed at understanding the behavior of the recorded scheduler. The trace could be used as input to a modified scheduler, but the policy changes may affect the order of scheduler actions and the scheduler behavior. Consulting a userspace scheduler synchronously on every operation, like Bento's approach to file system debugging, is infeasible because the scheduler subsystem cannot block waiting for a response from a userspace program that itself needs to be scheduled to run. Likewise, consulting a userspace scheduler asynchronously, as in ghOSt, can result in different behavior between the kernel and user versions.

Record. LibEnoki must record the sequence of information provided to the scheduler, both procedure calls into the scheduler and hints from the queue, so they can be replayed exactly as they were originally played in the kernel. We also record responses returned by the scheduler so we can alert the user if the scheduler returns a different result during replay. We do not record or attempt to replay the exact timing of the messages. Where timing is relevant to scheduler decisions, such as the runtime of a task, that information is provided in the message from the kernel and so will be recorded. We assume that the scheduler does not attempt to validate this information or track its own timing and that the scheduler does not contain other sources of nondeterminism.

Enoki's record and replay system must also handle concurrency. The Linux kernel is multi-threaded, and multiple kernel threads can call into an Enoki scheduler at once, e.g. on different cores. Due to Rust's properties and Enoki's structure, we know that potential race conditions are protected by locks, and we can identify and record the order of lock acquisitions. As long as locks are acquired in the same order during record and replay and the behavior of the scheduler is deterministic, the results should be the same [62]. In libEnoki, we include recording functionality in the shim wrappers around the kernel lock functions to record lock creation, acquisition and release, along with the address of the lock and the ID of the accessing kernel thread. All other message records are also tagged by the thread ID of the kernel thread calling into the scheduler.

Recording messages in the scheduler stack is non-trivial. In many cases, Enoki is called while the kernel has interrupts disabled. Writing to a file has the potential to sleep, so we cannot write the messages to a log file while in the scheduler context. Even printing to the kernel log must be delayed until out of the scheduler context.

Similarly to userspace hints, we use a ring buffer to solve this. We run a separate userspace task that creates a ring buffer queue shared with Enoki-C. When libEnoki wants to record a message, it sends it to Enoki-C, which adds the message to the queue. The userspace task consumes messages on the queue and writes them to a file. If the buffer

overruns, events may be dropped.

In order to record messages, the userspace record task must be running and the scheduler must have been compiled in record mode. By default, libEnoki does not record messages.

Replay. Replay consumes the file created during recording. The replay utility sends the recorded messages directly to the scheduler code in the same order they were called and validates the responses against the recorded ones.

To handle concurrent replay, we ensure that all locks are acquired in the same order in replay as they were during record. First, the replay system analyzes the log and parses out the lists of operations on each lock, using the lock's memory address to differentiate the locks. The locks are then created, passing in the list of acquisitions for each lock. We assume that locks are created in the same order during replay as they were during record and are not deallocated.

To allow for concurrent operations, the replay system starts a thread per recorded message in the record log as it replays the log. When each replay thread is created, the replay system names it with the ID of the associated kernel thread. When the replay thread attempts to acquire a lock, the lock checks whether it is the next to acquire the lock. If not, the thread is blocked until its turn.

Enoki does not support upgrading the scheduler during the record and replay process.

5.3 Implementation

5.3.1 Enoki

The lines of code for Enoki are shown in Table 5.2. Enoki is implemented in Linux 5.11 and is based on the kernel component from ghOSt [67]. The scheduler libEnoki includes the EnokiScheduler trait, hint queues, and support for record and live upgrade. Other libEnoki provides safe access to kernel data structures and functions and general support for Rust kernel programming. It builds upon the libBentoKS library from Bento. Enoki could be ported to newer versions of the kernel with relative ease, particularly because the scheduler interface appears to be quite stable. Note that the ideas behind Enoki are not unique to Linux; however their application to other operating systems is left for future work.

Component	Lang.	LOC	Unsafe LOC
Enoki-C	C	2411	N/A
Scheduler libEnoki	Rust	962	94
Other libEnoki	Rust	5870	2858
Userspace Record	Rust	95	10
Replay	Rust	646	0

Table 5.2: Lines of code for the Enoki components. The scheduler libEnoki includes the EnokiScheduler trait, hint queues, and support for record and live upgrade. Other libEnoki provides safe access to kernel data structures and functions. Part of Other libEnoki is borrowed from the analogous library from Bento (libBentoKS).

5.3.2 Schedulers

To demonstrate Enoki's flexibility, we implemented a selection of schedulers: a weighted fair queuing scheduler based on Linux CFS, a specialized research scheduler based on Shinjuku, and schedulers that utilize the userspace hinting and bidirectional communication features of Enoki for locality aware scheduling and two level scheduling. As baselines, we use the default Linux CFS and the ghOSt [67]

CFS and Weighted Fair Queuing.

CFS uses per-core run-queues, meaning it first assigns tasks to cores, and then chooses the next task for the core from among the assigned tasks. On each core, CFS implements a version of weighted fair queuing, dividing CPU time proportionally between groups of tasks, and then within each group, while respecting priority. It uses a calculation called *vruntime* to track which group/task to choose next, choosing the group/task with the lowest weighted accumulated runtime. The *vruntime* is calculated based on the task's runtime, modified by its priority; tasks with higher priority accrue *vruntime* slower. To prevent sleeping tasks from accruing a large *vruntime* debt and therefore running for too long after they wake, newly woken tasks receive the maximum of their old *vruntime* and the *vruntime* of the task with the lowest *vruntime* in the run-queue minus a several millisecond threshold. If a newly woken task has a smaller *vruntime* than the current task, it preempts the current task when a system timer ticks. Otherwise, task switches occur only after a task has run for its allocated time slice. In order to prevent starvation, CFS attempts to run every task at least once per time period, where the period depends on the number of tasks, with a minimum of 6ms. When tasks are created or woken, the length of the period

adjusts to include the new task. New tasks are run at the end of the period, but recently woken tasks can be run earlier.

CFS places running tasks onto cores and moves tasks between cores to achieve better performance. By default, CFS will attempt to even out the amount of work per core, based on information such as the amount of time the core is idle or overloaded, the priority of the tasks on the core, the capacity of the cores on the machine, and the preferences of the tasks. In certain cases, CFS will co-locate tasks on specific cores, such as if it is required to by the user. CFS attempts to place tasks so that newly woken tasks can be scheduled promptly. When a task is woken or a core becomes idle, CFS will move tasks so the newly idle cores are used. CFS rebalances task placement every 1-10ms, depending on the configuration, or when cores become newly idle. CFS first tries to move tasks to cores within the same NUMA node, and will not balance tasks across NUMA nodes unless there are more than a threshold more tasks running on the busier NUMA node.

Enoki Weighted Fair Queuing. To evaluate the overhead of Enoki, we implemented our own scheduler based on CFS. Our version does not provide the full complexity of the algorithm; instead, we are interested in showing the overhead of our approach on benchmarks relative to CFS. We compute *vruntime* for per-core time slices but use a much simpler method for determining task placement. If a core is about to become idle and another core had a waiting task, our scheduler steals waiting work from the core with the longest queue of tasks. Otherwise, our scheduler does not rebalance tasks. We found this compromise allowed our scheduler to achieve good performance on a wide array of benchmarks with much less complexity - 646 lines of code versus 6247 for CFS.

GhOSt and the Shinjuku Scheduler.

GhOSt [67] is a research framework for userspace schedulers. GhOSt implements a trampoline approach; calls from the core scheduler code are forwarded to the userspace scheduler, which sends its scheduling decisions to the kernel. GhOSt uses an asynchronous message passing model, i.e. the core scheduler code does not wait for the userspace scheduler to respond before choosing what to run. We evaluate microbenchmarks against two ghOSt schedulers, the per-CPU FIFO scheduler, which runs per-core schedulers that manage tasks assigned to that core, and the SOL scheduler, a latency-optimized FIFO scheduler that manages all cores, running the scheduler on a separate core. We also implement a version of the Shinjuku [75] scheduler and evaluate it against ghOSt's version of the same scheduler.

The Shinjuku scheduler [75] achieves low latency for workloads with short, high priority tasks and longer, low priority tasks with an efficient version of shortest task first. Shinjuku uses centralized first-come-first-serve scheduling. After each task has run for 5 to 15μs Shinjuku preempts it, placing it at the back of the queue. However, Shinjuku does not run in Linux, instead depending on the Dune [19] operating system. Unlike Linux, Dune applies a single scheduling algorithm for all tasks on the machine.

To evaluate Enoki's ability to support a specialized scheduler for Shinjuku-style workloads, we implemented a scheduler based on the Shinjuku scheduler using Enoki. Our scheduler implements an approximation of a first-come-first-serve queue of tasks with fast preemption across the multiple kernel run-queues. Our preemption slice is 10μs instead of 5μs to prevent overloading the scheduler. This scheduler was implemented in 285 lines of code.

Locality Aware Scheduler.

We also implemented a locality aware scheduler using Enoki that co-locates tasks that communicate heavily with each other or benefit from cache sharing. This scheduler uses Enoki's userspace hinting mechanism to inform the scheduler about which tasks to co-locate. The application sends the ID of each newly created thread and a locality value to indicate which tasks should be co-located. Unlike Linux's *taskset* cgroup, these hints do not need to specify the core for each task, only its colocation, which the scheduler can ignore if non-optimal, such as when there are too many tasks on a given core. This scheduler was implemented in 203 lines.

Two-level Scheduler.

The Arachne user-level scheduler provides two-level thread management: applications request cores and manage user-level threads on the assigned cores [128, 7].

In Arachne, both the core arbiter and the runtime are implemented in userspace. The core arbiter relies on Linux's `cpuset` mechanism to manage core assignments. The runtime sends messages to the core arbiter over a socket, and the core arbiter either responds on the socket or uses a shared memory page. This socket allows the runtime to manually block if a core is not available.

We reimplemented the Arachne core arbiter as a kernel scheduler using Enoki. This scheduler uses Enoki's bidirectional userspace hints. We use the user-to-kernel queue to send core requests to the Enoki core arbiter; we use the kernel-to-userspace queue for core

reclamation requests. The Enoki core arbiter executes the same decisions as the Arachne core arbiter, but uses standard kernel scheduling mechanisms for assigning, moving, and blocking user scheduler activations rather than relying on `cpuset` and sockets. The Enoki version of the core arbiter is implemented in 579 lines of code. We compare this scheduler against both CFS and unmodified Arachne.

5.4 Evaluation

In our evaluation, we test Enoki's ability to meet our goal of high velocity development for a wide variety of high performance schedulers with minimal runtime overhead. In §5.5, we discuss the development experience of Enoki. In this section, we evaluate the performance of the baseline and Enoki schedulers. We evaluate whether these schedulers can achieve equivalent performance to native implementations on microbenchmarks and application workloads. For research schedulers, we evaluate the scheduler against using benchmarks from the original paper.

Latency (μs)	**One Core**	**Two Cores**
CFS	3.0	3.6
GhOSt SOL	6.0	5.8
GhOSt FIFO	9.1	7.0
WFQ	3.6	4.0
Shinjuku	4.0	4.4
Locality	3.5	3.9
Arachne	0.1	0.2

Table 5.3: Scheduler latency for the `perf bench sched pipe` benchmark in μs per wakeup. Enoki adds around 0.6μs of latency over CFS in the worst case and outperforms the ghOSt schedulers due to the synchronous nature of the benchmark.

5.4.1 Setting

Benchmarks were performed on either an 8 core, one-socket machine with an Intel i7-9700 CPU running at 3.00GHz or (for scalability tests) an 80 core, two-socket machine with Intel Xeon Gold 6138 CPUs running at 2.00GHz.

5.4.2 Microbenchmarks

We used microbenchmarks to evaluate the latency and scalability imposed by the Enoki framework. We ran the same microbenchmarks on all the schedulers we implemented.

Threads	2 (μs)		40 (μs)	
	50th	**90th**	**50th**	**90th**
CFS	74	101	139	320
GhOSt SOL	66	132	192	1354
GhOSt FIFO	101	170	152	1806
WFQ	78	104	170	323
Shinjuku	79	109	168	307
Locality	80	105	175	324
Arachne	1	1	1	1

Table 5.4: Schbench benchmark with two message threads and 2 and 40 worker threads per message thread. Thread wakeup latencies are measured in μs.

Latency. Table 5.3 shows the results of the `perf bench sched pipe` benchmark, averaged over three runs of the benchmark. This benchmark starts two tasks that send 1 million messages back and forth using the `pipe` system call. After each message, the sending task sleeps until the other task responds. By default, all schedulers put the two tasks on different cores. We also ran the benchmarks forcing both tasks to be on the same core.

The CFS and Enoki WFQ schedulers implement similar behavior. The ghOSt SOL and FIFO schedulers implement different algorithms. The ghOSt SOL algorithm attempts to schedule tasks as quickly as possible, and so should have minimum scheduling latency. The Arachne scheduler uses userspace threads while the others use processes because the Arachne userspace scheduler is built to manage userspace threads.

Compared to CFS, our Enoki WFQ scheduler adds 0.4μs (0.6μs) of latency per message, for the two (one) core case. This represents a 12% to 20% overhead on this benchmark. We found that this overhead was due to 100-150 ns of overhead per invocation of the Enoki scheduler. The scheduler is invoked four times per schedule operation: once due to the current task blocking, once due to the next task waking, once to allow the scheduler to rebalance tasks, and once for the scheduler to choose the next task. Together this results in 400-600 ns, or 0.4-0.6 μs, of overhead per schedule operation. This overhead could possibly be reduced with further optimization. Our version of the Shinjuku scheduler has slightly

higher overhead because it starts a reschedule timer on every operation, while CFS and our WFQ scheduler only start a reschedule timer when multiple tasks are present. The locality aware scheduler is slightly faster because it is simpler. The Enoki version of Arachne is much faster than the others because it uses userspace threads instead of processes for blocking and waking threads.

The GhOSt schedulers perform significantly worse than both CFS and Enoki. The per-CPU FIFO scheduler performs worse when both tasks are placed on the same core because they are sharing the core with the ghOSt userspace scheduler. On every schedule operation, the scheduler first must be scheduled and run on the core.

The more often an application triggers scheduling actions, the more scheduling latency will impact the application. Workloads with many small tasks or many sleeps and wakes, and therefore many invocations of the scheduler, will be most affected by this overhead.

Scalability. To measure the scalability of Enoki, we evaluate the tail latency of task schedules when there are a large number of tasks using the schbench benchmark. This benchmark starts a number of message threads and worker threads. Each message thread and its worker threads send messages back and forth. Schbench reports the median and 99% tail latency of task schedules throughout the benchmark. The tested configurations use 2 message threads and 2 or 40 worker threads, resulting in a maximum of 80 worker threads, the same as the number of cores on the machine. When the number of worker threads is larger than the number of cores, the scheduling latency is influenced by waiting time more than scheduler performance. We use a 5s warmup time for each run.

The results are shown in Table 5.4. CFS and the Enoki WFQ scheduler showed similar results except for the median with 40 threads. The Enoki version of Arachne has lower latency than the other schedulers because of its userspace mechanisms for blocking and waking threads.

5.4.3 WFQ Scheduler Applications

To evaluate whether our WFQ scheduler performs competitively with CFS, we ran multi-threaded application benchmarks using benchmark suites that covered a wide range of use case patterns. We ran benchmarks from the NAS Parallel Benchmark suite [13] and the Phoronix Multi-core Test Suite [120]. The NAS Parallel Benchmark suite from NASA is a set of benchmarks to evaluate parallel performance on supercomputers. We ran nine of the ten NAS Benchmarks, excluding the DC benchmark that targets computational grids. We used the "C" benchmark size, the largest standard benchmark size. The Phoronix Multi-Core Test Suite contains a very large collection of multithreaded application benchmarks.

NAS Benchmark	CFS	WFQ	Slowdown
BT (total Mops/s)	26 669.1	26 682.3	−0.05 %
CG (total Mops/s)	4535.8	4475.7	1.32 %
EP (total Mops/s)	487.9	491.9	−0.83 %
FT (total Mops/s)	14 886.8	14 716.5	1.14 %
IS (total Mops/s)	1297.4	1284.9	0.96 %
LU (total Mops/s)	30 469.4	29 811.4	2.16 %
MG (total Mops/s)	8601.4	8535.9	0.76 %
SP (total Mops/s)	11 797.0	11 705.6	0.78 %
UA (total Mops/s)	73.8	73.1	0.87 %

Table 5.5: Performance comparison of CFS and the Enoki WFQ scheduler on the NAS Parallel Benchmarks. The maximum slowdown was 2.16%. The geometric mean of the slowdowns over all benchmarks was 0.74%.

There are over 90 applications, and many have multiple workloads. We report the same collection of 27 benchmarks as reported in Nest [85]. Phoronix runs each benchmark three times unless the standard deviation is greater than 5%, in which case it will rerun the benchmark until the standard deviation is low enough, up to a maximum of 15 times. The results for the NAS benchmarks are in Table 5.5, and the Phoronix Multicore benchmark results are in Table 5.6.

The NAS benchmarks show very little performance difference between the CFS and Enoki WFQ schedulers, with a maximum of 2.16% difference on the LU benchmark. The NAS benchmarks all start one task per core, so this is expected behavior. Performance on the Phoronix benchmarks is also quite similar for both schedulers across the benchmarks. The largest slowdowns were 8.22% and 8.57% on the Cassandra Writes benchmark, which issued writes to an Apache Cassandra database, and the Zstandard compression benchmark using compression level 3, long mode, respectively. This was likely due to the WFQ scheduler's simpler mechanism for rebalacing tasks making less optimal decisions for these benchmarks. We found that the balancing mechanism most affected the Arrayfire, Cassandra, and Zstandard compression benchmarks. Interestingly, we also saw speedup for Enoki on some benchmarks. The largest speedup was on the Zstandard compression benchmark using level 8, long mode. This was likely due to the simplified balancing mechanism. Overall, the geometric mean of the performance differences between the schedulers across the benchmarks was 0.74%.

We also ran these benchmarks on ghOSt. The SOL minimal FIFO scheduler described in the ghOSt paper is much slower and does not run some of the benchmarks. A new

Phoronix Multicore Benchmark	CFS	WFQ	Slowdown
Arrayfire, BLAS CPU (GFLOPS)	812.98	820.29	−0.90 %
Arrayfire, Conjugate Gradient CPU (ms)	26.72	26.71	−0.04 %
Cassandra, Writes (Op/s)	55 100	50 573	8.22 %
ASKAP, Hogbom Clean OpenMP (Iterations/s)	161.46	161.12	0.22 %
Cpuminer, Blake-2 S (kH/s)	258 100	261 667	−1.38 %
Cpuminer, Myriad-Groestl (kH/s)	9499.87	9494.90	0.05 %
Cpuminer, Quad SHA-256 Pyrite (kH/s)	35 667	35 390	0.78 %
Cpuminer, Skeincoin (kH/s)	29 400	29 323	0.26 %
Cpuminer, Triple SH-256 Onecoin (kH/s)	51 363	51 390	−0.05 %
Ffmpeg, libx264, Live (s)	23.98	24.73	3.13 %
Graphics-Magick, Resizing (Iterations/m)	781	779	0.26 %
OIDN, RT.hdr_alb_nrm.3840x2160 (Images/s)	0.31	0.30	3.23 %
OIDN, RT.ldr_alb_nrm.3840x2160 (Images/s)	0.31	0.30	3.23 %
OIDN, RTLightmap.hdr.4096x4096 (Images/s)	0.15	0.15	0 %
Rodina, OpenMP Leukocyte (s)	159.32	160.00	0.43 %
Zstd, 3 Long Mode Compression (MB/s)	856.1	782.7	8.57 %
Zstd, 8 Long Mode Compression (MB/s)	153.1	165.4	−8.03 %
AVIFEnc, 6 Lossless (s)	14.94	15.33	2.62 %
Libgav1, Summer Nature 1080p (FPS)	262.95	261.21	0.66 %
Libgav1, Summer Nature 4k (FPS)	67.28	66.58	1.04 %
Libgav1, Chimera 1080p (FPS)	222.70	216.51	2.78 %
Libgav1, Chimera 1080p 10-bit (FPS)	64.10	63.54	0.87 %
OneDNN, IP Shapes 1D, f32 (ms)	4.26	4.18	−1.85 %
OneDNN, IP Shapes 3D, f32 (ms)	9.71	9.10	−6.31 %
OneDNN, RNN Training, bf16bf16bf16 (ms)	4164.25	4163.34	−0.02 %
OneDNN, RNN Training, f32 (ms)	4166.31	4164.74	−0.04 %
OneDNN, RNN Training, u8s8f32 (ms)	4166.40	4161.15	−0.13 %

Table 5.6: Performance comparison of CFS and the Enoki WFQ scheduler on a selection of the Phoronix Multicore application benchmarks. The maximum slowdown was 8.57%. The geometric mean of the slowdowns over all benchmarks was 0.74%.

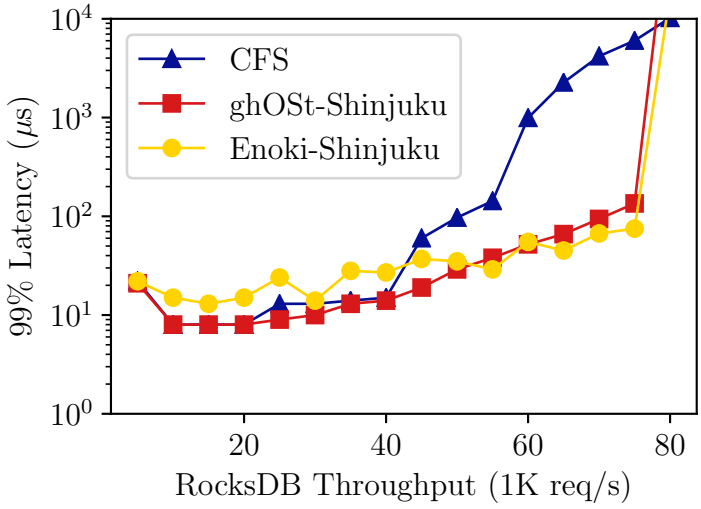

Figure 5.2: Tail latency comparison of the Enoki Shinjuku (μs scale preemption) scheduler compared to CFS and the ghOSt Shinjuku scheduler of a dispersive load on RocksDB. Graph is in log scale.

ghOSt-based weighted fair queueing scheduler is in progress, and runs correctly on all but one benchmark. It is slower on most benchmarks than Enoki. However, since it is not finished, a quantitative comparison would not be fair to ghOSt at this time.

To evaluate our version of the Shinjuku scheduler, we ran the RocksDB benchmarks used in the Shinjuku and ghOSt papers. These benchmarks send queries to an in-memory RocksDB database, with 99.5% GET requests and 0.5% range queries. Replicating how this benchmark was run in ghOSt, each GET is assigned to take 4μs and each range query to take 10ms. If RocksDB responds too quickly, the benchmark spins until the assigned time has elapsed. Three cores were reserved, one for background tasks, one for the load generator, and one for the scheduler if required. The load generator passes tasks to a total of 50 workers running on the other five cores. We evaluate against the ghOSt version of the Shinjuku scheduler. We could not compare against the original Shinjuku scheduler because it requires a specific NIC that we do not have. Both the ghOSt Shinjuku and Enoki Shinjuku schedulers use a preemption timer of 10μs. The results are shown in Figure 5.2, Figure 5.3, and Figure 5.4.

In the first benchmark, shown in Figure 5.2, only the RocksDB application is run. Note that the y-axis is in log scale. Both he Enoki and ghOSt Shinjuku schedulers achieve low

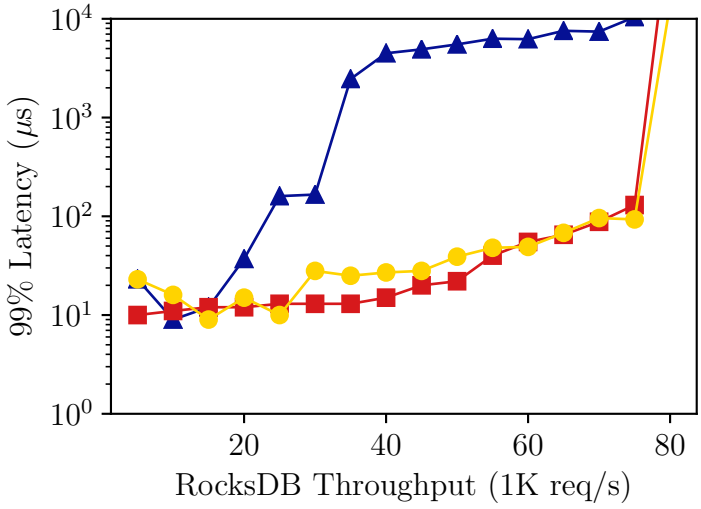

Figure 5.3: Tail latency comparison of the Enoki Shinjuku (μs scale preemption) scheduler compared to CFS and the ghOSt Shinjuku scheduler of a dispersive load on RocksDB co-located with a batch application. Graph is in log scale.

tail latency at high loads due to the short preemption timer; long running range query tasks are preempted quickly, allowing the short GET queries to run. On CFS, tasks run for much longer before being preempted, by default 750μs, so the GET queries spent more time waiting to be run.

In the second benchmark, a batch application was co-located with RocksDB. For the CFS and Enoki tests, the batch application was run using CFS. RocksDB is given a priority of -20 while the batch application is 19 (lower is higher priority). The batch application is run on ghOSt using a lower priority than RocksDB. Figure 5.3 shows the tail latency of RocksDB, and Figure 5.4 shows the CPU share of the batch application.

Both the Enoki and ghOSt Shinjuku schedulers achieve similar tail latency as with no background task because they give priority to the RocksDB workers, with a small time slice. The tail latency on CFS worsens with addition of the batch task. In the first experiment with no batch application, the Enoki scheduler achieved 30% lower latency than the ghOSt scheduler at high load (above 65K req/s). High load results in the most stress on the scheduler because it must repeatedly preempt long running tasks to schedule the short tasks.

The batch application receives a similar share of the CPU on CFS and the Enoki

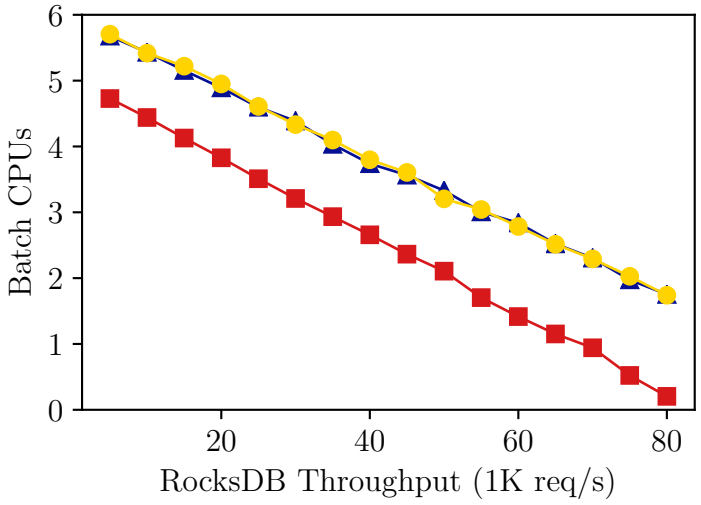

Figure 5.4: CPU share comparison of the Enoki Shinjuku (μs scale preemption) scheduler compared to CFS and the ghOSt Shinjuku scheduler of a dispersive load on RocksDB co-located with a batch application. Graph is in log scale.

Shinjuku scheduler. CFS itself shares cycles between the RocksDB workers and the batch application according to their niceness values. When there are no RocksDB requests the Enoki scheduler seamlessly cedes cycles to CFS to run the batch application. The ghOSt Shinjuku scheduler provides substantially less CPU time to the batch application because it must pay the overhead of the userspace scheduler. In Enoki and CFS, the scheduler is run on the same core as the application as part of regular kernel calls. All of the schedulers give the batch task a much higher CPU share than the original Shinjuku scheduler would [67].

5.4.4 Locality Aware Scheduler

To evaluate the effectiveness of using hints in our locality aware scheduler, we used a modified version of the schbench benchmark. This benchmark starts a specified number of message threads and worker threads. Each message thread and its worker threads send messages back and forth. The benchmark records the wakeup latency of the worker threads to evaluate scheduler overhead. When a message thread and its worker threads are on the same core, wakeup latency can be very low. However, the benchmark uses a futex to wait which does not set the WF_SYNC flag when waking the workers, so Linux can not detect

Latency	CFS	CFS One Core	Random	Hints
50th (μs)	33	17	46	2
99th (μs)	50	32032	49	4

Table 5.7: Wakeup latency for the schbench benchmark with two message threads and two worker threads per message thread. Thread wakeup latencies were in microseconds. All runs used a 5s warmup time and ran the benchmark for 30s.

this pattern [110]. In our modified version of the benchmark, we send hints to an Enoki locality aware scheduler to co-locate the message thread with its workers, but place each set of message and worker threads on a different core. We compare this approach to CFS and to the locality aware scheduler with random placement (no hints) as baselines. We also compare to CFS using cgroups to test if the flexibility provided by the hints is necessary for a performance benefit. Cgroups enable specifying a set of cores that a process should run on, but do not support different sets of cores for different threads within the same process. We use cgroups to place all the threads on one core.

The results are shown in Table 5.7. We use two message threads and two worker threads per message thread. CFS and the locality aware scheduler with random placement perform similarly because both spread tasks across cores. The locality aware scheduler with hints achieves significantly lower 99% latency. Using cgroups to put all threads on one core improves median latency at a cost of much worse tail latency due to the added competition between threads.

5.4.5 Arachne Scheduler

We evaluate the Enoki version of the Arachne scheduling using a version of the Realistic memcached workload from the Arachne paper [128]. We use the Mutilate benchmark utility [108] to generate load for the memcached server, using the key size and distribution, value size and distribution, and inter-arrival distribution of the Facebook ETC workload [10], 1 million records, and 3% updates. Four clients are used to generate load, enough to make the benchmark server-bound in our tests. Each client creates 16 threads and four connections per thread. An extra client is used to evaluate latency. We compare the Enoki version of Arachne against the original Arachne code and a baseline version of memcached running on CFS. Both of the Arachne versions automatically scale between two and seven cores, reserving one for background tasks. The baseline version uses all eight cores on the machine.

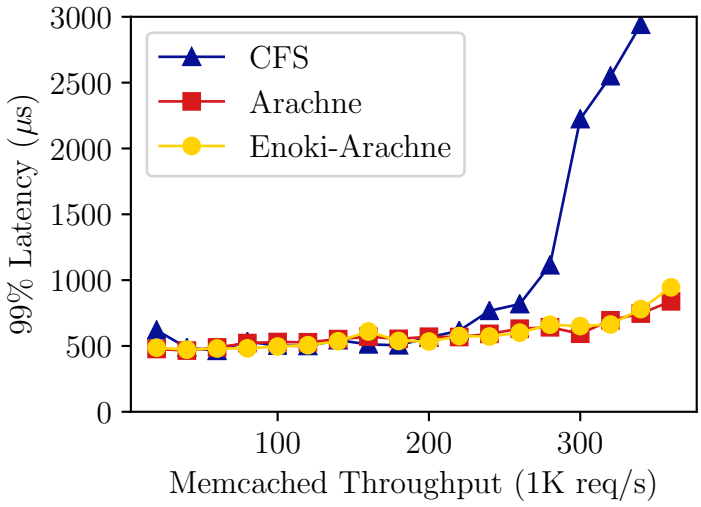

Figure 5.5: Tail latency of requests to a Memcached server using Mutilate comparing baseline Memcached using CFS to a version using Arachne and a version using Arachne modified to use an Enoki core arbiter.

The Enoki version of Arachne achieves similar performance to the original Arachne scheduler, better than CFS at high load.

5.4.6 Live Upgrade

We evaluate the performance impact of live upgrade using the schbench benchmark and timing instrumentation in the kernel. We track the tail latency of schedule operations before, during, and after the upgrade. The upgrade interruption was too short to affect the tail latency of the schbench operations, so we repeated the evaluation with timing calls inserted in the kernel. We evaluate this benchmark on both the one socket machine using 2 message threads and 2 workers per message thread and the two socket machine with 2 message threads and both 2 and 40 workers per message thread. On the one socket machine, the upgrade takes 1.5μs. On the two socket machine, the upgrade takes 9.9μs and 10.1μs for 2 and 40 workers, respectively. All were averaged over three runs.

5.4.7 Record and Replay

We evaluate the performance of record and replay using the `perf bench sched pipe` benchmark on the WFQ scheduler. This benchmark completes in around 4 seconds during regular operation. During record, the benchmark completes in around 30 seconds, and the replay takes around 3 minutes. Record is slower than regular operation because all relevant operations must be sent to the record program, which writes a file asynchronously on a different core.

During replay, the first 30 seconds are spent reading the file and parsing lock operations. The rest is due to the mechanism for ensuring concurrent operations occur in the same order as the live execution. This benchmark has many calls to the scheduler. In the kernel, these operations are very fast. During replay, if threads arrives in a different order than what was recorded, we need to block the thread and wake it up later to try again. These constant sleeps and wakes add latency and account for most of the execution time during replay.

5.5 Discussion

In this section, we report on our experience using Enoki to develop kernel schedulers. Enoki's design made scheduler development much more straightforward than it would be in Linux. In fact, two of the schedulers (the Shinjuku and Locality Aware schedulers) were written by undergraduates with no prior experience in Linux kernel development. They were able to start coding with relatively little ramp up and were able to build, test, and debug their code on their own. They primarily benefited from modularity, because they were using Enoki when record and replay and live upgrade had not been implemented.

Being able to use the `Schedulable` data structure as proof that a task is schedulable on a core allowed us to ensure that we were picking runnable tasks. This helped us identify and address bugs faster because we would see a compile time error from `pick_next_task` returning an incorrect `Schedulable` thread rather than putting the kernel into a bad state by trying to run a task on the wrong core.

We ran into relatively few runtime bugs, and those bugs were fairly easy to address. Most have been deadlocks, which are painful to encounter because they force a reboot, but proved surprisingly easy to debug, in line with prior OS development experience [25]. Often, relatively few lock operations were touched in between two runs of the code, so finding the relevant changes to lock operations was easy. With more complicated deadlocks, it was

easy to look through the scheduler code and check the order of lock acquires because the largest scheduler (WFQ) was only around 600 lines of code. Of the non-deadlock bugs, most were conceptual, such as a benchmark having tasks yield when we had not yet implemented `task_yield` or miscalculating the fair time slice to assign tasks in WFQ. While fixing these bugs sometimes required rebooting the machine, we never had to recompile the kernel to fix them, so our iteration times were quite short.

The general design of an interposable, message-passing based interface that uses memory sharing under the hood had a number of benefits. We gained many of the benefits of modularity for development velocity because of clean interfaces and isolation from the rest of the kernel, allowing the developer to focus their energy on the algorithm being implemented. In its implementation, Enoki uses function calls and memory sharing, limiting the overhead that can come with modular designs. Because Enoki schedulers run in the kernel, the core scheduling code can quickly and easily make synchronous calls to the scheduler, allowing it to quickly respond to changes in state.

Enoki's design also made it easy to implement additional features. Because all the functionality was contained in the module and Enoki-C contacts the scheduler through a single function pointer, live upgrade is as simple as quiescing the state and replacing the function pointer. Due to Rust's support for generic data types and traits, custom hint data structures could be defined as type parameters on the scheduler and any requirements on the data types can be expressed as trait bounds. Enoki's design supported record and replay debugging very smoothly. The interposable, message passing based interface and made it simple to record relevant state that was passed into the scheduler or returned from calls into the kernel. Recording nondeterministic behavior is one of the main challenges for record and replay on parallel systems, but using safe Rust made this much simpler. Due to Rust's safety guarantees, we knew that the scheduler could not contain race conditions or any other undefined behavior, so the only sources of nondeterminism were timing and the order of lock acquisitions. All timing state was handled by the kernel and passed into the scheduler, and so was automatically handled by recording the messages. To correctly handle concurrency, we only needed to record and replay the order in which locks were acquired.

5.6 Summary

Enoki is a framework for rapid development of high performance Linux kernel schedulers. Enoki enables safe, high performance kernel schedulers with seamless live upgrade, bidirectional user communication channels, and record and replay debugging. A scheduler

implemented with Enoki is able to achieve similar performance to CFS, the default Linux scheduler, on a wide range of benchmarks. Other Enoki schedulers mimic recent research schedulers, but integrated with Linux. Enoki's schedulers can be upgraded with only $10.1us$ of service interruption, and the record and replay debugging allows for slow but functional userspace debugging of the kernel scheduler code.

Chapter 6

Related Work

Our work is related to prior research along several dimensions. Several projects have used safe languages to reduce bugs in operating systems, though none of these target other aspects of high development velocity. Other work has improved safety or for modules in operating systems. SFI (software-based fault isolation) uses binary rewriting techniques to isolate bugs to modules, enabling untrusted code to be run safely in kernel mode. Running modules in userspace provides a safe and user friendly userspace development environment, but can introduce performance or compatibility issues. Live upgrade and record and replay debugging have been used in non-kernel contexts.

6.1 Using Safe Languages for Kernel Development

Several systems, including Pilot [131], SPIN [21], Singularity [68], Biscuit [43], Redox [133], Tock [86], Theseus [28], and RedLeaf [109], write the entire operating system, including the kernel, in a high-level language, with the goal of reducing bugs, simplifying the implementations, or improving performance. None directly target high development velocity in commercial operating systems. SPIN, Singularity, and Biscuit use garbage collected languages, while Tock, Theseus, and RedLeaf use Rust to provide safety without kernel garbage collection. SPIN leverages type safety to allow application-specific customization of kernel behavior. These approaches show that benefits can be gained by using safe languages, but all involve rewriting the entire operating system.

We are not the only project to integrate Rust into the Linux kernel [89, 92]. Most similar to our work is the Rust-for-Linux project [138] which is working to make it possible

to upstream Rust code into the Linux kernel. The file system portion of our work predates the published output of the Rust-for-Linux effort. Like our work, Rust-for-Linux introduces interfaces for Rust code to interface with the Linux kernel. It initially targeted device drivers. It attempts to provide safe interfaces, but does not make using them a requirement and does not currently include interfaces for safe file systems or schedulers. Unlike our work, Rust-for-Linux does not introduce other mechanisms for high development velocity such as debugging support or live upgrade. As far as we know, we are the first project to incorporate modules that are entirely in safe Rust into a commodity operating system and to combine Rust modules with other tools for high development velocity.

The extended Berkeley Packet Filter (eBPF) [101] is a type safe language for safe kernel extensibility with untrusted user specific code. It can be seen as an adaptation of the ideas in the Spin research operating system [21]. Users can insert eBPF programs at predefined kernel locations, and the kernel verifies the safety of the inserted programs before running them. For example, eBPF specifically checks to make sure all loops are bounded, all instructions are reachable, and there are no calls to disallowed functions. ExtFUSE [24] has enabled writing parts of a stackable file system using eBPF. Similarly, GhOSt [67] extends eBPF to reduce some of the runtime cost of running ghOSt. To ensure safety of user code running in the kernel, eBPF is quite restrictive. Programs are limited to only running at pre-defined points in the kernel, only calling whitelisted functions, and not having unbounded loops. This is a useful tool for inserting untrusted user code into the kernel at runtime and has been used for a number of projects [160, 161, 27]. However, it is not a good fit for larger modules written by trusted kernel developers because eBPF limits the scope of the inserted code to what can be statically verified. Additionally, eBPF limits the types of data structures extensions can use, further restricting designs. eBPF programs cannot be updated while they are in use by an application. Compared to eBPF, this thesis shows that it is possible to develop large feature-rich modules in a safe language to allow continuous integration of new features into a commodity operating system.

6.2 Software Fault Isolation

Software fault isolation (SFI) [153, 31, 98] is a technique for sandboxing the impact of faults in a module to the module itself. SFI is most widely used for isolating third party device drivers. Checks are performed at compile-time or runtime to constrain memory references to the memory used by the untrusted code—without affecting the rest of the system. SFI reduces the effects of bugs on the system as a whole, but it does not prevent bugs and can incur noticeable overhead when there are frequent calls across trust boundaries, particularly

where state is shared across the boundary. We see both in our setting. CPU scheduling code can be invoked very frequently, and we use references to enable safe memory sharing when we need to share data structures between the kernel and the module code. In SFI, for example, the entire inode would need to be copied in and out and revalidated on each call into the file system. We chose to use Rust instead as it has lower runtime overhead and provides the additional benefit of bug prevention within a module in addition to isolating errors across modules.

6.3 Moving Kernel Features to Userspace

Microkernel design, where kernel services run in userspace, is another way to speed operating system development [2, 90] especially when safety and/or development velocity are more important than raw performance. Most similar to our project are FUSE [56] and ghOSt [67]. Filesystem in Userspace (FUSE) allows for developers to write userspace file systems for Linux and ghOSt similarly supports userspace schedulers in Linux. FUSE shows significant overhead for create and delete heavy workloads since these cannot be serviced by the kernel page cache, and GhOSt introduces significant scheduling latency and CPU usage overhead.

A related approach is to run the userspace OS service on dedicated processor cores, where applications communicate with the service via asynchronous message queues in shared memory [20, 17, 77, 99]. This approach has been widely used for implementing new network stacks and scheduler algorithms [75, 128, 99, 77]. Performance can often be competitive with an equivalent kernel implementation, except when processors need to busy wait, block, or when the system needs to remap pages for efficient zero copy I/O.

Rump kernels (or anykernels) enable running unmodified kernel code as userspace libraries by hijacking system calls and providing userspace implementations of necessary kernel internals. They are used for untrusted execution of kernel code, e.g., when mounting an untrusted file system, or userspace debugging. Implementations exist for NetBSD as a rump kernel [76] and Linux in the libOS [148] and Linux Kernel Library [127] projects. Similarly, User Mode Linux [50] enables running a Linux kernel as a userspace process.

In kernel bypass solutions, hardware virtualization is used to expose I/O devices directly to userspace. Userspace applications can directly access the device through userspace libraries to avoid crossing into the kernel at all on I/O operations. This enables I/O stacks that are customized for an application's use cases, but introduces challenges for resource sharing and security. Moving device management out of the kernel limits the ability to

share the device with applications that still use the kernel I/O stack and prevents global optimization decisions that could be made by the kernel [139]. Allowing user applications to directly manage the hardware can allow users to launch attacks on or using the hardware [143].

6.4 OS Live Upgrade

Three commercially available tools for live upgrade of Linux systems are ksplice [9, 113], kpatch [122], and kGraft [118]. All three perform live upgrade of Linux kernel diffs and focus on security patches that do not modify data structure layout. The internals of each approach differ, but all three reroute calls from modified functions to new functions. Some research systems provide support for upgrade of more complex components. K42 [18] is a research operating system that enables upgrade of modular components by quiescing the component then transferring state to the new instance and updating references. PROTEOS [58], another research operating system, also supports live upgrade of modular components. LUCOS [37] and DynAMOS [97] enable live upgrade of complex components in Linux without the need for state quiescence by using shadow data structures and virtualization, respectively, to maintain state. Our work enables live upgrade of large modules in a commodity operating system with state transfer by introducing a framework layer that handles quiescing state.

6.5 Record and Replay

Record and replay systems enable debugging code by first recording a trace of operations and then later replaying those operations against the target code, enabling debugging of the recorded trace. These systems require instrumentation around the target code in order to record the necessary data. There are a variety of approaches to record and replay, from recording the whole system [52, 155] to using kernel instrumentation to record user programs [135, 48, 100] to running both the target code and the instrumentation in userspace [140, 57, 117, 23] to instrumenting language and application runtimes [4, 29]. One project [35] specifically records and replays kernel modules using mechanisms for whole binary analysis and instrumentation of kernel module interfaces. As far as we know, we are the first project to selectively record a kernel module, as opposed to the whole kernel, and to replay the behavior at userspace on the same code as ran in the kernel.

Chapter 7

Conclusion and Future Work

Development velocity is critical for modern software development, particularly in the cloud where new code is often deployed weekly. Cloud providers have embraced the concept of CI/CD (continuous integration/continuous deployment) for quickly building, testing, and releasing changes to users. New demands by user applications and new hardware designs are increasingly creating a need for higher development velocity for operating system kernel code. Despite this, the techniques that have enabled faster development for userspace applications are not applied to operating systems code, and development velocity in the Linux kernel, the most widely used server operating system, is still slow. Linux's development velocity is hindered by the the complexity of the code base, the ease of writing bugs, the difficulty of systematically testing for bugs, the limited debugging environment, and the disruption caused by redeploying new code.

In this thesis, we explore how to apply modern software development principles to Linux to improve development velocity. We start by analyzing the types of bugs that occur in Linux kernel modules and the root cause and effects of those bugs. We then turn to file systems due to the long history of new design proposals in the area. We design, build, and evaluate Bento, a framework for high velocity development of Linux kernel file systems. We also design, build, and evaluate Enoki for high velocity Linux kernel schedulers to support the growing number of proposals for new scheduler algorithms.

Our bug studies revealed that around half of the bugs from Linux are low level bugs that could be prevented using compile-time language level mechanisms. Based on that, we designed Bento and Enoki to allow developers to write Linux kernel file systems and schedulers entirely in safe Rust, eliminating almost all of these low level bugs. Bento also enables userspace debugging and online upgrades for file system modules to easy debugging

and redeployment. A file system written with Bento was able to perform competitively with ext4, the baseline Linux file system and could be upgraded with only 15 ms of service interruption.

Enoki follows the same design philosophy as Bento, but extends the work to support CPU scheduler modules. Enoki allows schedulers to communicate with userspace processes using custom data structures over bidirection pathways. Enoki schedulers can be debugged using a record and replay mechanism that replays entirely at userspace. Schedulers written with Enoki can perform competitively with the default Linux scheduler CFS and with research schedulers that are optimized for cloud use cases.

Since we started this work, the use of Rust in the Linux kernel has been gaining steam. The Rust-for-Linux project is working to have Rust code in the mainline Linux kernel, and as of kernel version 6.0, they have been successful. There are ongoing efforts to grow the amount of Rust code in Linux by writing more subsystems in Rust. Currently, developers are working to add safe Rust bindings for file systems in Linux, including an effort to port Bento to kernel version 6.0 so it can be upstreamed.

7.1 Future Work

We believe that Bento and Enoki open up a number of new directions for operating systems kernel research. One of the long term goals of the project is to make it easier to conduct research on key kernel modules as well as to reduce the friction of deploying new ideas into the kernel. Here we focus on research into improving the generality of our mechanisms.

7.1.1 Bento

Bento currently has a number of limitations that we could address in future work. We use the Rust compiler guarantees to catch classes of bugs that are most likely to crash the kernel, but this leaves logic bugs uncaught. Logic bugs in file systems can be severe, potentially causing data loss or corruption. Additionally, Bento only enables replacements to Linux kernel file system modules. It does not support replacing other parts of the file system stack, such as the file cache or the write ahead log. We also did not explore areas where Bento's design could enable new designs or optimizations.

To expand the types of bugs caught by Bento, we could add support for formal verification of Bento file systems. There have been several past works on formal verification

of file systems [36, 142], operating systems components [111, 81], and Rust modules [65]. We could extend these works to enable verification of file systems to Bento. This could be used to verify individual properties of the file system, such as crash consistency, or could be used to ensure that the file system completely matches a provided specification.

In our work on file systems, we reused the kernel buffer cache and write ahead log implementations. However, some developers may wish to use different designs for these modules. We could extend Bento to allow developers to implement new buffer caches and write ahead logs and choose which implementation to use with different file systems.

We could also explore potential optimizations enabled by Bento's architecture. For example, we could extend Bento to provide additional support for stacked file systems, where one or more stacked file systems are layered on top of a base file system to modify the file system's behavior. Stacked file systems can be used for automatically encrypting files (eCryptfs [54]), automatically compressing files (Gzipfs [63] or similar), or making files from one file system visible to another file system (UnionFS [150] and OverlayFS [115]). Stacked file systems can be implemented in Linux today by having the upper layer file system call into the lower layer, through the VFS layer. This adds some overhead because the VFS code is run per layer, redoing unnecessary work. If the stacked file system can assume that the lower layer is implemented in Bento, we could allow stacked file systems to call into Bento instead of into the VFS layer, bypassing the overhead of the extra VFS layer calls without compromising on generality.

Bento could be used for research on new file systems. New hardware, such as Triple- and Quad-Level Cell SSDs and Zoned Namespace SSDs, are introducing new storage structures that would benefit from new file system designs. Additionally, new workloads, such as microservices and serverless computing in the cloud, are placing new demands on file systems. We could use Bento to research file systems designs that could better meet new and upcoming environments.

7.1.2 Enoki

Our current version of Enoki has a number of limitations. Like Bento, Enoki uses the Rust compiler to prevent memory bugs, but is unable to prevent logic bugs. Logic bugs in the scheduler can have very serious consequences, such as never scheduling a ready task or trying to run a task on a CPU it is not located on, causing a crash in Linux. Additionally, Enoki imposes few changes on the Linux kernel. If we changed more of the Linux scheduler code, we could support different use cases and optimizations.

We could use formal verification to prevent more types of bugs in Enoki schedulers. Formal verification is a powerful tool for ensuring correctness of code. Verification has recently seen growing applications to operating systems components, particularly file systems [36, 163, 34, 142, 141]. While Enoki already prevents many bugs that cause the scheduler to crash the kernel, using verification to check correctness properties would give developers more confidence in their schedulers. For example, developers could use verification to check for work conservation, starvation freedom, or other high level correctness properties. The Verus project [65] supports verifying Rust code, and could be integrated into Enoki for scheduler verification.

Some features of Enoki could be improved by making more substantial changes to the Linux core scheduler code. For example, Linux uses a strict hierarchy of schedulers. If a high priority scheduler has a ready task, it will always have the opportunity to run that task before a lower priority scheduler's tasks. Therefore, if a high priority scheduler is very busy, it will starve the tasks on the lower priority scheduler. While this may be the desired behavior sometimes, it is not necessarily always the optimal choice. We could modify the Linux scheduler code to introduce a more complicated form of hierarchical scheduling so cycles could be shared more fairly between different schedulers.

We could use Enoki to aid research on CPU schedulers. Microsecond scale tasks, heterogeneous CPUs, and rising energy concerns are motivating development of new CPU scheduling algorithms [47, 114, 75, 158, 85]. Enoki could be used to accelerate development of research schedulers to address these new demands. We could also use the userspace hinting mechanism to implement a scheduler that uses machine learning. Neural networks make heavy use of floating point arithmetic, and so cannot be implemented in the kernel. We could implement the machine learning portion of the scheduler in userspace and send its output as hints to the kernel scheduler.

7.1.3 Extending to Other Subsystems

File systems and schedulers are both good targets for this work because there is considerable research interest as well as new design proposals and the Linux kernel is already designed to support modularity in these areas.

Other areas within Linux, such as networking and memory management are also seeing these kinds of pressure. We could improve development velocity of new network protocols, memory managers, and beyond by extending our work on Bento and Enoki to these new domains.

Networking Due to rapidly increasing networking speeds and the overheads of the Linux kernel TCP stack, there have been proposals for new networking protocols [106], networking stacks [99], and TCP stacks [77, 71]. Most of these are implemented at userspace to avoid the difficulty of modifying the kernel and the overheads of the Linux kernel TCP stack. To provide low latency despite being implemented in userspace, these systems often dedicate cores or shared memory to the network stack [77, 38, 99]. Extendeding Bento and Enoki to support new designs for networking and TCP stacks would allow these to be implemented in the kernel, providing low latency without dedicated resources.

Memory Management If we extended the ideas from this thesis to network stacks, we could also support network file systems that are co-designed with the network protocol they use. Networked file systems have long been in use (e.g. NFS). More recently, they have been proposed for tiered storage [8, 80, 74] where network latencies are very low so the performance of the network stack can matter greatly. Enabling higher velocity development of Linux kernel network stacks could make it possible to write kernel network stacks that are optimized to give good performance for a specific file system.

Similarly, enormous server memories, superpages, non-volatile memory and tiered and disaggregated memory are changing the memory hierarchy of servers, placing new demands on memory managers and motivating new memory manager designs [61, 129]. We could extend the work in this thesis to support new designs of memory managers safely. For both networking and memory management, the Linux kernel subsystems are intertwined with surrounding code, so any of these extensions would require rearchitecting the software stack to increase agility.

7.2 Conclusion

In summary, we believe that it is possible to significantly improve the development and deployment velocity of key elements of the operating system kernel. Researchers and developers do not need to avoid modifying Linux. The Spin project [21] argued for applications to customize operating system behavior using a new operating system designed for extensibility. Today, eBPF is used for extensibility in Linux, allowing untrusted user programs written for a restricted environment to be run at prespecified points in the kernel. We propose a different approach. We combine the modular flexibility of Spin with the Linux compatibility of eBPF. Researchers and developers do not need to choose between putting up with existing Linux implementations, limiting kernel modifications to highly restricted

programs, or replacing Linux entirely. We can instead push Linux to become more extensible. With efficient development and testing and minimal downtime during deployment in Linux, we no longer need to build application specific operating systems. Linux itself can be modified to become application specific.

References

[1] Autotest - Fully Automated Testing under Linux. https://autotest.github.io/.

[2] Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., and Young, M. Mach: A New Kernel Foundation For UNIX Development. In *Summer USENIX* (1986).

[3] Aghayev, A., Weil, S., Kuchnik, M., Nelson, M., Ganger, G. R., and Amvrosiadis, G. File Systems Unfit as Distributed Storage Backends: Lessons from 10 Years of Ceph Evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (2019), SOSP '19.

[4] Alpern, B., Choi, J.-D., Ngo, T., Sridharan, M., and Vlissides, J. M. A Perturbation-Free Replay Platform for Cross-Optimized Multithreaded Applications. In *Proceedings of the 15th International Parallel &; Distributed Processing Symposium* (2001), IPDPS '01.

[5] AMD Turbo Core Technology, 2023. https://www.amd.com/en/technologies/turbo-core.

[6] Intel Turbo Boost Technology 2.0: Higher Performance When You Need It Most, 2023. https://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html.

[7] Anderson, T. E., Bershad, B. N., Lazowska, E. D., and Levy, H. M. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles* (1991), SOSP '91.

[8] Anderson, T. E., Canini, M., Kim, J., Kostić, D., Kwon, Y., Peter, S., Reda, W., Schuh, H. N., and Witchel, E. Assise: Performance and Availability

via Client-local NVM in a Distributed File System. In *14th USENIX Symposium on Operating Systems Design and Implementation* (2020), OSDI '20.

[9] Arnold, J., and Kaashoek, M. F. Ksplice: Automatic Rebootless Kernel Updates. In *Proceedings of the Fourth European Conference on Computer Systems* (2009), EuroSys '09.

[10] Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., and Paleczny, M. Workload Analysis of a Large-Scale Key-Value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems* (2012), SIGMETRICS '12.

[11] AWS SLAs, 2023. https://aws.amazon.com/compute/sla/.

[12] Azure SLAs, 2023. https://www.microsoft.com/licensing/docs/view/Service-Level-Agreements-SLA-for-Online-Services?lang=1.

[13] Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Schreiber, R. S., Simon, H. D., Venkatakrishnan, V., and Weeratunga, S. K. The NAS Parallel Benchmarks—Summary and Preliminary Results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing* (1991), Supercomputing '91.

[14] Barbalace, A., Sadini, M., Ansary, S., Jelesnianski, C., Ravichandran, A., Kendir, C., Murray, A., and Ravindran, B. Popcorn: Bridging the Programmability Gap in Heterogeneous-ISA Platforms. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), EuroSys '15.

[15] Barroso, L., Hölzle, U., and Ranganathan, P. *The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition*, vol. 13. 2018.

[16] Barroso, L., Marty, M., Patterson, D., and Ranganathan, P. Attack of the Killer Microseconds. *Commun. ACM 60* (2017).

[17] Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., and Singhania, A. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles* (2009), SOSP '09.

[18] Baumann, A., Heiser, G., Appavoo, J., Da Silva, D., Krieger, O., Wisniewski, R. W., and Kerr, J. Providing Dynamic Update in an Operating System. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (2005), ATEC '05.

[19] Belay, A., Bittau, A., Mashtizadeh, A., Terei, D., Mazières, D., and Kozyrakis, C. Dune: Safe User-Level Access to Privileged CPU Features. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (2012), OSDI '12.

[20] Bershad, B., Anderson, T. E., Lazowska, E. D., and Levy, H. M. User-level Interprocess Communication for Shared Memory Multiprocessors. *ACM Transactions on Computer Systems* (1991).

[21] Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M. E., Becker, D., Chambers, C., and Eggers, S. Extensibility Safety and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (1995), SOSP '95.

[22] Betsy Beyer, Chris Jones, J. P., and Murphy, N. R. *Site Reliability Engineering*. O'Reilly Media, Inc., 2016.

[23] Bhansali, S., Chen, W.-K., de Jong, S., Edwards, A., Murray, R., Drinić, M., Mihočka, D., and Chau, J. Framework for Instruction-Level Tracing and Analysis of Program Executions. In *Proceedings of the 2nd International Conference on Virtual Execution Environments* (2006), VEE '06.

[24] Bijlani, A., and Ramachandran, U. Extension Framework for File Systems in User Space. In *2019 USENIX Annual Technical Conference* (2019), ATC '19.

[25] Birrell, A. An Introduction to Programming with Threads. Tech. Rep. 35, Digital Systems Research Center, January 1989.

[26] Birrell, A. D., and Nelson, B. J. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems 2* (1984).

[27] Bonola, M., Belocchi, G., Tulumello, A., Brunella, M. S., Siracusano, G., Bianchi, G., and Bifulco, R. Faster Software Packet Processing on FPGA NICs with eBPF Program Warping. In *USENIX Annual Technical Conference* (2022), ATC '22.

[28] Boos, K., Liyanage, N., Ijaz, R., and Zhong, L. Theseus: an Experiment in Operating System Structure and State Management. In *14th USENIX Symposium on Operating Systems Design and Implementation* (2020), OSDI '20.

[29] Burg, B., Bailey, R., Ko, A. J., and Ernst, M. D. Interactive Record/Replay for Web Application Debugging. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology* (2013), UIST '13.

[30] Cao, W., Liu, Z., Wang, P., Chen, S., Zhu, C., Zheng, S., Wang, Y., and Ma, G. PolarFS: An Ultra-Low Latency and Failure Resilient Distributed File System for Shared Storage Cloud Database. *Proceedings of the VLDB Endowment 11* (2018).

[31] Castro, M., Costa, M., Martin, J.-P., Peinado, M., Akritidis, P., Donnelly, A., Barham, P., and Black, R. Fast Byte-granularity Software Fault Isolation. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (2009), SOSP '09.

[32] Castro, P., Ishakian, V., Muthusamy, V., and Slominski, A. The Rise of Serverless Computing. *Communications of the ACM 62* (2019).

[33] Caulfield, A. M., Chung, E. S., Putnam, A., Angepat, H., Fowers, J., Haselman, M., Heil, S., Humphrey, M., Kaur, P., Kim, J.-Y., Lo, D., Massengill, T., Ovtcharov, K., Papamichael, M., Woods, L., Lanka, S., Chiou, D., and Burger, D. A Cloud-Scale Acceleration Architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture* (2016), MICRO '16'.

[34] Chajed, T., Tassarotti, J., Theng, M., Kaashoek, M. F., and Zeldovich, N. Verifying the DaisyNFS Concurrent and Crash-Safe File System with Sequential Reasoning. In *16th USENIX Symposium on Operating Systems Design and Implementation* (2022), OSDI '22.

[35] Chen, B., Yang, Z., Lei, L., Cong, K., and Xie, F. Automated Bug Detection and Replay for COTS Linux Kernel Modules with Concolic Execution. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering* (2020), SANER '20.

[36] Chen, H., Chajed, T., Konradi, A., Wang, S., undefinedleri, A., Chlipala, A., Kaashoek, M. F., and Zeldovich, N. Verifying a High-Performance

Crash-Safe File System Using a Tree Specification. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), SOSP '17.

[37] Chen, H., Chen, R., Zhang, F., Zang, B., and Yew, P.-C. Live Updating Operating Systems Using Virtualization. In *Proceedings of the 2nd International Conference on Virtual Execution Environments* (2006), VEE '06.

[38] Chen, J., Wu, Y., Lin, S., Xu, Y., Kong, X., Anderson, T., Lentz, M., Yang, X., and Zhuo, D. Remote Procedure Call as a Managed System Service. In *20th USENIX Symposium on Networked Systems Design and Implementation* (Boston, MA, 2023), NSDI '23.

[39] Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D. An Empirical Study of Operating Systems Errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles* (2001), SOSP '01.

[40] [RFC PATCH v2 0/5] mm: Select victim using bpf_oom_evaluate_task, 2023. https://lore.kernel.org/lkml/20230810081319.65668-1-zhouchuyi@bytedance.com/.

[41] Coccinelle: A Program Matching and Transformation Tool for Systems Code. https://coccinelle.gitlabpages.inria.fr/website/.

[42] Coxx, R., Kaashoek, F., and Morris, R. Xv6, A Simple Unix-like Teaching Operating System, 2020.

[43] Cutler, C., Kaashoek, M. F., and Morris, R. T. The Benefits and Costs of Writing a POSIX Kernel in a High-Level Language. OSDI '18.

[44] Daniel V. Klein, D. M. B., and Monroe, M. G. Making "Push on Green" a Reality. *login Usenix Mag. 39* (2014).

[45] Dastidar, J., Riddoch, D., Moore, J., Pope, S., and Wesselkamper, J. The AMD 400-G Adaptive SmartNIC System on Chip: A Technology Preview. *IEEE Micro 43* (2023).

[46] Dean, J., and Ghemawat, S. LevelDB: A Fast Persistent Key-Value Store, 2011.

[47] Demoulin, M., Fried, J., Pedisich, I., Kogias, M., Loo, B. T., Phan, L. T. X., and Zhang, I. When Idling is Ideal: Optimizing Tail-Latency for Highly-Dispersed Datacenter Workloads with Perséphone. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (2021), SOSP '21.

[48] Devecsery, D., Chow, M., Dou, X., Flinn, J., and Chen, P. M. Eidetic Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation* (2014), OSDI '14.

[49] Dickson, C. How Embracing Continuous Release Reduced Change Complexity. USENIX Association.

[50] Dike, J. A User-Mode Port of the Linux Kernel. In *Annual Linux Showcase & Conference* (2000).

[51] Intel Data Plane Development Kit (Intel DPDK) Programmer's Guide. https://doc.dpdk.org/guides/prog_guide/, 2017.

[52] Dunlap, G. W., King, S. T., Cinar, S., Basrai, M. A., and Chen, P. M. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. *SIGOPS Operating Systems Review 36* (2003).

[53] eBPF.io. eBPF, 2023. https://ebpf.io/.

[54] eCrypt Filesystem. https://www.ecryptfs.org/.

[55] Engler, D. R., Kaashoek, M. F., and O'Toole, Jr., J. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (1995), SOSP '95.

[56] Filesystem in Userspace. https://github.com/libfuse/libfuse.

[57] Geels, D., Altekar, G., Shenker, S., and Stoica, I. Replay Debugging for Distributed Applications. In *2006 USENIX Annual Technical Conference* (2006), ATC '06.

[58] Giuffrida, C., Kuijsten, A., and Tanenbaum, A. S. Safe and Automatic Live Update for Operating Systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2013), ASPLOS '13.

[59] Glaubius, R., Tidwell, T., Gill, C., and Smart, W. D. Real-Time Scheduling via Reinforcement Learning. In *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence* (2010), UAI '10.

[60] Google Cloud SLAs, 2023. https://cloud.google.com/terms/sla.

[61] Gouk, D., Lee, S., Kwon, M., and Jung, M. Direct Access, High-Performance Memory Disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference* (Carlsbad, CA, 2022), ATC '22.

[62] Guo, Z., Wang, X., Tang, J., Liu, X., Xu, Z., Wu, M., Kaashoek, M. F., and Zhang, Z. R2: An Application-Level Kernel for Record and Replay. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (USA, 2008), OSDI '08.

[63] Gzip Filesystem. https://www.filesystems.org/docs/zadok-thesis-proposal/node94.html.

[64] Hance, T., Lattuada, A., Hawblitzel, C., Howell, J., Johnson, R., and Parno, B. Storage Systems are Distributed Systems (So Verify Them That Way!). In *14th USENIX Symposium on Operating Systems Design and Implementation* (2020), OSDI '20.

[65] Hawblitzel, C., and Lattuada, A. Verus, 2022. https://github.com/verus-lang/verus.

[66] Heiser, G., and Elphinstone, K. L4 Microkernels: The Lessons from 20 Years of Research and Deployment. *ACM Transactions on Computer Systems 34* (2016).

[67] Humphries, J. T., Natu, N., Chaugule, A., Weisse, O., Rhoden, B., Don, J., Rizzo, L., Rombakh, O., Turner, P., and Kozyrakis, C. GhOSt: Fast & Flexible User-Space Delegation of Linux Scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (2021), SOSP '21.

[68] Hunt, G. C., and Larus, J. R. Singularity: Rethinking the Software Stack. *SIGOPS Operating Systems Review* (2007).

[69] Huye, D., Shkuro, Y., and Sambasivan, R. R. Lifting the Veil on Meta's Microservice Architecture: Analyses of Topology and Request Workflows. In *2023 USENIX Annual Technical Conference* (2023), ATC '23.

[70] Jaffer, S., Mahdaviani, K., and Schroeder, B. Improving the Reliability of Next Generation SSDs using WOM-v Codes. In *20th USENIX Conference on File and Storage Technologies* (2022), FAST '22.

[71] Jeong, E. Y., Woo, S., Jamshed, M., Jeong, H., Ihm, S., Han, D., and Park, K. mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems.

In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (2014), NSDI'14.

[72] Jonathan Corbet, Alessandro Rubini, G. K.-H. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005.

[73] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P.-l., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T. V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C. R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Killebrew, D., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Ross, M., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., and Yoon, D. H. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (2017), ISCA '17.

[74] Kadekodi, R., Lee, S. K., Kashyap, S., Kim, T., Kolli, A., and Chidambaram, V. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (2019), SOSP '19.

[75] Kaffes, K., Chong, T., Humphries, J. T., Belay, A., Mazières, D., and Kozyrakis, C. Shinjuku: Preemptive Scheduling for μsecond-Scale Tail Latency. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation* (2019), NSDI '19.

[76] Kantee, A. Rump File Systems: Kernel Code Reborn. In *USENIX Annual Technical Conference* (2009), ATC '09.

[77] Kaufmann, A., Stamler, T., Peter, S., Sharma, N. K., Krishnamurthy, A., and Anderson, T. E. TAS: TCP Acceleration as an OS Service. In *Proceedings of the Fourteenth EuroSys Conference 2019* (2019), EuroSys '19.

[78] Kernel self-test. https://kselftest.wiki.kernel.org/.

[79] KernelCI. https://kernelci.org/.

[80] Kim, J., Jang, I., Reda, W., Im, J., Canini, M., Kostić, D., Kwon, Y., Peter, S., and Witchel, E. LineFS: Efficient SmartNIC Offload of a Distributed File System with Pipeline Parallelism. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (2021), SOSP '21.

[81] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (2009), SOSP '09.

[82] Klimovic, A., Wang, Y., Stuedi, P., Trivedi, A., Pfefferle, J., and Kozyrakis, C. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation* (2018), OSDI '18.

[83] ktest. https://elinux.org/Ktest.

[84] Kwon, Y., Fingler, H., Hunt, T., Peter, S., Witchel, E., and Anderson, T. E. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), SOSP '17.

[85] Lawall, J., Chhaya-Shailesh, H., Lozi, J.-P., Lepers, B., Zwaenepoel, W., and Muller, G. OS Scheduling with Nest: Keeping Tasks Close Together on Warm Cores. In *Proceedings of the Seventeenth European Conference on Computer Systems* (2022), EuroSys '22.

[86] Levy, A., Campbell, B., Ghena, B., Giffin, D. B., Pannuto, P., Dutta, P., and Levis, P. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), SOSP '17.

[87] Li, N., Kalaba, A., Freedman, M. J., Lloyd, W., and Levy, A. Speculative Recovery: Cheap, Highly Available Fault Tolerance with Disaggregated Storage. In *2022 USENIX Annual Technical Conference* (2022), ATC '22.

[88] Li, Q., Chen, L., Wang, X., Huang, S., Xiang, Q., Dong, Y., Yao, W., Huang, M., Yang, P., Liu, S., Zhu, Z., Wang, H., Qiu, H., Liu, D., Liu, S., Zhou, Y., Wu, Y., Wu, Z., Gao, S., Han, C., Luo, Z., Shao, Y., Tian, G.,

Wu, Z., Cao, Z., Wu, J., Shu, J., Wu, J., and Wu, J. Fisc: A Large-scale Cloud-native-oriented File System. In *21st USENIX Conference on File and Storage Technologies* (2023), FAST '23.

[89] Li, Z., Wang, J., Sun, M., and Lui, J. C. Securing the Device Drivers of Your Embedded Systems: Framework and Prototype. In *Proceedings of the 14th International Conference on Availability, Reliability and Security* (2019), ARES '19.

[90] Liedtke, J. On Microkernel Construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (1995), SOSP '95.

[91] Lineage File System. https://crypto.stanford.edu/~cao/lineage.

[92] Linux-kernel-module-rust. https://github.com/fishinabarrel/linux-kernel-module-rust.

[93] Linux Kernel Releases, 2023. https://www.kernel.org/category/releases.html.

[94] LKFT - Linaro's Linux Kernel Functional Test framework. https://lkft.linaro.org/.

[95] Lozi, J.-P., Lepers, B., Funston, J., Gaud, F., Quéma, V., and Fedorova, A. The Linux Scheduler: A Decade of Wasted Cores. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), EuroSys '16.

[96] Lu, L., Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., and Lu, S. A Study of Linux File System Evolution. In *11th USENIX Conference on File and Storage Technologies* (2013), FAST '13.

[97] Makris, K., and Ryu, K. D. Dynamic and Adaptive Updates of Non-Quiescent Subsystems in Commodity Operating System Kernels. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (2007), EuroSys '07.

[98] Mao, Y., Chen, H., Zhou, D., Wang, X., Zeldovich, N., and Kaashoek, M. F. Software Fault Isolation with API Integrity and Multi-principal Modules. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), SOSP '11.

[99] Marty, M., de Kruijf, M., Adriaens, J., Alfeld, C., Bauer, S., Contavalli, C., Dalton, M., Dukkipati, N., Evans, W. C., Gribble, S., and

ET AL. Snap: A Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (2019), SOSP '19.

[100] MASHTIZADEH, A. J., GARFINKEL, T., TEREI, D., MAZIERES, D., AND ROSENBLUM, M. Towards Practical Default-On Multi-Core Record/Replay. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (2017), ASPLOS '17.

[101] MCCANNE, S., AND VAN, J. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Winter USENIX* (1993).

[102] MILLER, S., ZHANG, K., CHEN, M., JENNINGS, R., CHEN, A., ZHUO, D., AND ANDERSON, T. High Velocity Kernel File Systems with Bento. In *19th USENIX Conference on File and Storage Technologies* (2021), FAST '21.

[103] MIN, J., ZHAO, C., LIU, M., AND KRISHNAMURTHY, A. eZNS: An Elastic Zoned Namespace for Commodity ZNS SSDs. In *17th USENIX Symposium on Operating Systems Design and Implementation* (2023), OSDI '23.

[104] MOGUL, J., RASHID, R., AND ACCETTA, M. The Packet Filter: An Efficient Mechanism for User-Level Network Code. *SIGOPS Operating Systems Review 21* (1987).

[105] MOHAN, J., MARTINEZ, A., PONNAPALLI, S., RAJU, P., AND CHIDAMBARAM, V. Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (2018), OSDI '18.

[106] MONTAZERI, B., LI, Y., ALIZADEH, M., AND OUSTERHOUT, J. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (2018), SIGCOMM '18.

[107] MUNISWAMY-REDDY, K.-K., HOLLAND, D. A., BRAUN, U., AND SELTZER, M. Provenance-Aware Storage Systems. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference* (2006), ATEC '06.

[108] Mutilate, 2015. https://github.com/leverich/mutilate.

[109] NARAYANAN, V., HUANG, T., DETWEILER, D., APPEL, D., LI, Z., ZELLWEGER, G., AND BURTSEV, A. RedLeaf: Isolation and Communication in a Safe Operating

System. In *14th USENIX Symposium on Operating Systems Design and Implementation* (2020), OSDI '20.

[110] Nayak, K. P. sched: Userspace Hinting for Task Placement, 2022.

[111] Nelson, L., Sigurbjarnarson, H., Zhang, K., Johnson, D., Bornholt, J., Torlak, E., and Wang, X. Hyperkernel: Push-Button Verification of an OS Kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), SOSP '17.

[112] Nightingale, E. B., Hodson, O., McIlroy, R., Hawblitzel, C., and Hunt, G. Helios: Heterogeneous Multiprocessing with Satellite Kernels. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (2009), SOSP '09.

[113] Oracle Ksplice. https://ksplice.oracle.com/.

[114] Ousterhout, A., Fried, J., Behrens, J., Belay, A., and Balakrishnan, H. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation* (2019), NSDI '19.

[115] Overlay Filesystem. https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt.

[116] Palix, N., Thomas, G., Saha, S., Calvès, C., Lawall, J., and Muller, G. Faults in Linux: Ten Years Later. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2011), ASPLOS XVI.

[117] Patil, H., Pereira, C., Stallcup, M., Lueck, G., and Cownie, J. PinPlay: A Framework for Deterministic Replay and Reproducible Analysis of Parallel Programs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (01 2010), CGO '10, pp. 2–11.

[118] Pavlik, V. kGraft: Live Kernel Patching. https://www.suse.com/c/kgraft-live-kernel-patching/.

[119] Peter, S., Li, J., Zhang, I., Ports, D. R. K., Woos, D., Krishnamurthy, A., Anderson, T., and Roscoe, T. Arrakis: The Operating System is the Control Plane. In *11th USENIX Symposium on Operating Systems Design and Implementation* (2014), OSDI '14.

[120] Phoronix Multicore, Jan. 2023. https://openbenchmarking.org/suite/pts/multicore.

[121] Pillai, T. S., Chidambaram, V., Alagappan, R., Al-Kiswany, S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Crash Consistency. *Communications of the ACM 58* (2015).

[122] Poimboeuf, J. Introducing kpatch: Dynamic Kernel Patching. https://www.redhat.com/en/blog/introducing-kpatch-dynamic-kernel-patching.

[123] Pourhabibi, A., Sutherland, M., Daglis, A., and Falsafi, B. Cerebros: Evading the RPC Tax in Datacenters. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (2021), MICRO '21.

[124] Prabhakaran, V., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Analysis and Evolution of Journaling File Systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (2005), ATEC '05.

[125] Prekas, G., Kogias, M., and Bugnion, E. ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), SOSP '17.

[126] Psaroudakis, I., Scheuer, T., May, N., Sellami, A., and Ailamaki, A. Scaling up Concurrent Main-Memory Column-Store Scans: Towards Adaptive NUMA-Aware Data and Task Placement. *Proceedings of the VLDB Endowowment 8* (2015).

[127] Purdila, O., Grijincu, L. A., and Tapus, N. LKL: The Linux Kernel Library. In *9th RoEduNet IEEE International Conference* (2010).

[128] Qin, H., Li, Q., Speiser, J., Kraft, P., and Ousterhout, J. Arachne: Core-Aware Thread Management. In *13th USENIX Symposium on Operating Systems Design and Implementation* (2018), OSDI '18.

[129] Raybuck, A., Stamler, T., Zhang, W., Erez, M., and Peter, S. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (2021), SOSP '21.

[130] Redell, D. D. Experience with Topaz Telebugging. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging* (1988), PADD '88.

[131] Redell, D. D., Dalal, Y. K., Horsley, T. R., Lauer, H. C., Lynch, W. C., McJones, P. R., Murray, H. G., and Purcell, S. C. Pilot: An Operating System for a Personal Computer. *Commununications of the ACM 23* (1980).

[132] Redis. https://redis.io.

[133] Redox. https://www.redox-os.org/.

[134] RocksDB. https://rocksdb.org/.

[135] Ronsse, M., and De Bosschere, K. RecPlay: A Fully Integrated Practical Record/Replay System. *ACM Transactions on Computer Systems 17* (1999).

[136] Rossi, C. Rapid release at massive scale. https://engineering.fb.com/web/rapid-release-at-massive-scale.

[137] Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrmann, F., Kaiser, C., Langlois, S., Leonard, P., and Neuhauser, W. CHORUS Distributed Operating Systems. *Computing Systems* (1988).

[138] Rust for Linux. https://github.com/Rust-for-Linux.

[139] Sadok, H., Zhao, Z., Choung, V., Atre, N., Berger, D. S., Hoe, J. C., Panda, A., and Sherry, J. We Need Kernel Interposition over the Network Dataplane. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (2021), HotOS '21.

[140] Saito, Y. Jockey: A User-Space Library for Record-Replay Debugging. In *Proceedings of the Sixth International Symposium on Automated Analysis-Driven Debugging* (2005), AADEBUG'05.

[141] Schellhorn, G., Ernst, G., Pfähler, J., Haneberg, D., and Reif, W. Development of a Verified Flash File System. In *Proceedings of the ABZ Conference* (2014), vol. 8477.

[142] Sigurbjarnarson, H., Bornholt, J., Torlak, E., and Wang, X. Push-button Verification of File Systems via Crash Refinement. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (2016), OSDI '16.

[143] Simpson, A. K., Szekeres, A., Nelson, J., and Zhang, I. Securing RDMA for High-Performance Datacenter Storage Systems. In *12th USENIX Workshop on Hot Topics in Cloud Computing* (2020), HotCloud '20.

[144] Smatch: pluggable static analysis for C. https://lwn.net/Articles/691882/.

[145] Build Ultra High-Performance Storage Applications with the Storage Performance Development Kit. https://spdk.io/, 2023.

[146] SSDFS, 2023. https://lore.kernel.org/linux-fsdevel/20230225010927.813929-1-slava@dubeyko.com/.

[147] Sweeney, A. Scalability in the XFS File System. In *USENIX 1996 Annual Technical Conference* (1996), ATC '96.

[148] Tazaki, H., Nakamura, R., and Sekiya, Y. Operating System with Mainline Linux Network Stack. In *netdev 0.1* (2015).

[149] [PATCHSET RFC] sched: Implement BPF extensible scheduler class, 2022. https://lore.kernel.org/lkml/20221130082313.3241517-1-tj@kernel.org/.

[150] Union Filesystem. https://www.filesystems.org/project-unionfs.html.

[151] Vahdat, A., and Anderson, T. E. Transparent Result Caching. In *USENIX Annual Technical Conference* (1998), ATC '98.

[152] Vangoor, B. K. R., Tarasov, V., and Zadok, E. To FUSE or Not to FUSE: Performance of User-Space File Systems. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies* (2017), FAST '17.

[153] Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. Efficient Software-based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles* (1993), SOSP '93.

[154] Xu, J., and Swanson, S. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies* (2016), FAST '16.

[155] Xu, M., Malyugin, V., Sheldon, J., Venkitachalam, G., Weissman, B., and Inc, V. Retrace: Collecting execution trace with virtual machine deterministic replay. In *In Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation* (2007), MoBS '07.

[156] Yasukata, K., Tazaki, H., Aublin, P.-L., and Ishiguro, K. Zpoline: A System Call Hook Mechanism Based on Binary Rewriting. In *2023 USENIX Annual Technical Conference* (2023), ATC '23.

[157] Yoshimura, T., Chiba, T., and Horii, H. EvFS: User-level, Event-Driven File System for Non-Volatile Memory. In *11th USENIX Workshop on Hot Topics in Storage and File Systems* (2019), HotStorage '19.

[158] Yuan, W., and Nahrstedt, K. Energy-Efficient CPU Scheduling for Multimedia Applications. *ACM Transactions on Computer Systems 24* (2006).

[159] Zhang, J., Kwon, M., Gouk, D., Koh, S., Lee, C., Alian, M., Chun, M., Kandemir, M. T., Kim, N. S., Kim, J., and Jung, M. FlashShare: Punching Through Server Storage Stack from Kernel to Firmware for Ultra-Low Latency SSDs. In *13th USENIX Symposium on Operating Systems Design and Implementation* (2018), OSDI '18.

[160] Zhong, Y., Li, H., Wu, Y. J., Zarkadas, I., Tao, J., Mesterhazy, E., Makris, M., Yang, J., Tai, A., Stutsman, R., and Cidon, A. XRP: In-Kernel Storage Functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation* (2022), OSDI '22.

[161] Zhou, Y., Wang, Z., Dharanipragada, S., and Yu, M. Electrode: Accelerating Distributed Protocols with eBPF. In *20th USENIX Symposium on Networked Systems Design and Implementation* (2023), NSDI '23.

[162] [PATCH RFC] HotBPF: Prevent Kernel Heap-based Exploitation, 2023. https://lore.kernel.org/lkml/20230719155032.4972-1-wzc@smail.nju.edu.cn/.

[163] Zou, M., Ding, H., Du, D., Fu, M., Gu, R., and Chen, H. Using Concurrent Relational Logic with Helpers for Verifying the AtomFS File System. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (2019), SOSP '19.