High Velocity Kernel File Systems with Bento

Samantha Miller, Kaiyuan Zhang, Mengqi Chen, and Ryan Jennings, *University of Washington;* Ang Chen, *Rice University;* Danyang Zhuo, *Duke University;* Thomas Anderson, *University of Washington*

https://www.usenix.org/conference/fast21/presentation/miller

This paper is included in the Proceedings of the 19th USENIX Conference on File and Storage Technologies.

February 23–25, 2021

978-1-939133-20-5

Open access to the Proceedings of the 19th USENIX Conference on File and Storage Technologies is sponsored by USENIX.

High Velocity Kernel File Systems with Bento

Samantha Miller Kaiyuan Zhang Mengqi Chen Ryan Jennings
Ang Chen[‡] Danyang Zhuo[†] Thomas Anderson

University of Washington [†]Duke University [‡]Rice University

Abstract

High development velocity is critical for modern systems. This is especially true for Linux file systems which are seeing increased pressure from new storage devices and new demands on storage systems. However, high velocity Linux kernel development is challenging due to the ease of introducing bugs, the difficulty of testing and debugging, and the lack of support for redeployment without service disruption. Existing approaches to high-velocity development of file systems for Linux have major downsides, such as the high performance penalty for FUSE file systems, slowing the deployment cycle for new file system functionality.

We propose Bento, a framework for high velocity development of Linux kernel file systems. It enables file systems written in safe Rust to be installed in the Linux kernel, with errors largely sandboxed to the file system. Bento file systems can be replaced with no disruption to running applications, allowing daily or weekly upgrades in a cloud server setting. Bento also supports userspace debugging. We implement a simple file system using Bento and show that it performs similarly to VFS-native ext4 on a variety of benchmarks and outperforms a FUSE version by 7x on 'git clone'. We also show that we can dynamically add file provenance tracking to a running kernel file system with only 15ms of service interruption.

1 INTRODUCTION

Development and deployment velocity is a critical aspect of modern cloud software development. High velocity delivers new features to customers more quickly, reduces integration and debugging costs, and reacts quickly to security vulnerabilities. However, this push for rapid development has not fully caught up to operating systems, despite this being a long-standing goal of OS research [1, 6, 16, 25, 44]. In Linux, the most widely used cloud operating system, release cycles are still measured in months and years. Elsewhere in the cloud, new features are deployed weekly or even daily.

Slow Linux development can be attributed to several factors. Linux has a large code base with relatively few guardrails, 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. The lack of isolation between kernel modules means that these errors often have non-intuitive effects and are difficult to track down. The difficulty of implementing kernel-level debuggers and kernel testing frameworks makes this worse. 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, either way rendering the machine unavailable during the upgrade. In the cloud setting, this forces kernel upgrades to be batched to meet cloud-level availability goals.

Slow development cycles are a particular problem for file systems. Recent changes in storage hardware (e.g., low latency SSDs and NVM, but also density-optimized QLC 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, 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 [2].

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 [17]. However, FUSE can incur a significant performance overhead, particularly for metadata-heavy workloads [48]. 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 FUSE and 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 are using 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 [34] 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 [26, 35] 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.

In this paper, we make the following contributions:

- We design and implement Bento, a framework that enables high-velocity development of safe, performant file systems in Linux.
- We develop an API that enables kernel file systems written in a type-safe language with both user and kernel execution and live upgrade.
- We demonstrate Bento's benefits by implementing and evaluating a file system developed atop Bento with ext4-like performance, and show that we can add provenance support without rebooting.

2 MOTIVATION

Development velocity is becoming increasingly important for the Linux kernel to adapt to emerging use cases and address security vulnerabilities. In this section, we describe several approaches for extending Linux file systems, and outline the properties of Bento.

2.1 High Velocity is Hard

Linux needs to adapt to support emerging workloads, address newfound vulnerabilities, and manage new hardware. On average 650,000 lines of Linux code are added and 350,000 removed every release cycle, resulting in a growth of roughly 1.5 million lines of code per year. Linux file systems are no exception in needing to adapt — with rapid change in both storage technologies and emerging application demands.

As a concrete example, consider what is needed to add a feature like data provenance to a Linux file system. Increasingly, enterprise customers want to track the source data files used in producing each data analysis output file to perform security forensics. While this might be implemented with existing tools for system call tracking, that would be incomplete — the file system has more comprehensive information (e.g., whether two file paths are hard links to the same inode); a distributed file system can further enable cross-network forensics. To implement this as a new feature in the file system, developers have to modify the file system, test it, and push this modification to production clusters.

The most widely used approach is to directly modify the kernel source code. Linux has standard kernel interfaces for extending its key subsystems — e.g., virtual file systems (VFS) for file systems, netfilter for networking, and Linux Security Module (LSM) for security features. Sometimes, it is also possible to add new features using loadable kernel modules, which can be integrated at runtime without kernel recompilation or reboot. Several VFS filesystems, including ext4, overlayfs, and btrfs, are implemented in the kernel source and can be inserted as loadable kernel modules.

However, high velocity kernel development (including kernel file system development) is hard to come by. To start with, kernel modifications are notoriously difficult to get right. Kernel code paths are complex and easy to accidentally misuse.

Bug	Number	Effect on Kernel
Use Before Allocate	6	Likely `oops`
Double Free	4	Undefined
`NULL` Dereference	5	`oops`
Use After Free	3	Likely `oops`
Over Allocation	1	Overutilization
Out of Bounds	4	Likely `oops`
Dangling Pointer	1	Likely `oops`
Missing Free	18	Memory Leak
Reference Count Leak	7	Memory Leak
Other Memory	1	Variable
Deadlock	5	Deadlock
Race Condition	5	Variable
Other Concurrency	1	Variable
Unchecked Error Value	5	Variable
Other Type Error	8	Variable

Table 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.

Worse, debugging kernel source code is much harder than user-level debugging. This is because a kernel debugger operates below the kernel, typically remotely, and it cannot leverage Posix APIs such as ptrace. Upgrading kernel modules is also an intrusive operation. In the case of file systems, this requires shutting down applications, unmounting the old file system and remounting the new, and restarting the application. In a multi-tenant cloud setting, most cloud services are upgraded live on a daily or weekly basis. To meet four or five nine application uptime service-level objectives [33] within a reboot model, however, kernel changes need to be batched and applied en masse every few months. Getting needed functionality upstreamed into Linux, so that it is compatible with the 1.5M lines of new code being added each year, takes even longer.

To provide intuition into the difficulty of developing and deploying new kernel features, Table 1 shows an analysis we conducted of 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. 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 accounted for 50% of the total bugs fixed in these modules.

The second set concern low-level bugs that are apply to any C language module, but when found in the kernel can potentially undermine the correctness or operation of the rest of the kernel. We categorized 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). Of the 50% of fixed bugs that were low-level bugs, we found that 68% are memory bugs. Of these, half are a type of memory leak. Many of the bugs occur in error-handling code, e.g., incorrect checking of return values, missing cleanup procedures. 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% caused a kernel `oops` which either kills the offending process or panics the kernel. An additional 34% of the analyzed bugs result in a memory leak, potentially causing out-of-memory problems or even DoS attack vectors. Many of these low-level bugs, particularly memory and type errors, result from inherent challenges of C code and could be prevented if the programming language had more safety checks.

2.2 Existing Alternatives

Besides directly modifying the Linux kernel, there are two other approaches to adding functionality to Linux, with their respective pros and cons.

Upcall (FUSE [17]): One common technique, particularly for file systems and I/O devices, is to implement new functionality as a userspace server. A stub is left in the kernel that converts system calls to upcalls into the server. Filesystem in Userspace (FUSE) does this for file systems. As opposed to implementing new file system functionality directly in the kernel, this isolates low-level memory errors such as use-after-free to the userspace process. (Low-level bugs can still affect file system functionality, of course.) Development speed is faster because engineers can use familiar debugging tools like gdb. All this comes at a performance cost for metadata-operations [48]. Our evaluation (§5.2) confirms this finding, revealing even worse performance overheads than previously reported, particularly for write-heavy workloads. Additionally, FUSE file systems can't reuse many existing kernel features, such as disk accesses through the buffer cache. Userspace file systems can mitigate the performance overhead by sharing mapped memory with the kernel, but this neither fully removes the performance overhead due to the extra kernel crossing nor allows the file system to access existing kernel functionality.

In-Kernel Interpreter: Using an interpreter inside the kernel for a dynamically loaded program in a safe language is another approach to ensure safety of kernel extensions. Linux supports eBPF (extended Berkeley Packed Filter) [32], an in-kernel virtual machine that allows code to be dynamically loaded and executed in the kernel at predefined points defined by the kernel. eBPF is used heavily for packet filtering, system call filtering, and kernel tracing. The idea is to allow kernel customization in a safe manner. The Linux eBPF virtual machine validates memory safety and execution termination before it JIT compiles the virtual machine instructions into native machine code. As such, eBPF can sandbox untrusted extensions, but the restrictions placed on eBPF make it very difficult to implement larger or more complex pieces of functionality. We argue that untrusted eBPF extensions are not the right model for kernel file system extensibility, as it is particularly difficult to imagine implementing mutable file system operations using eBPF and still enforcing crash consistency.

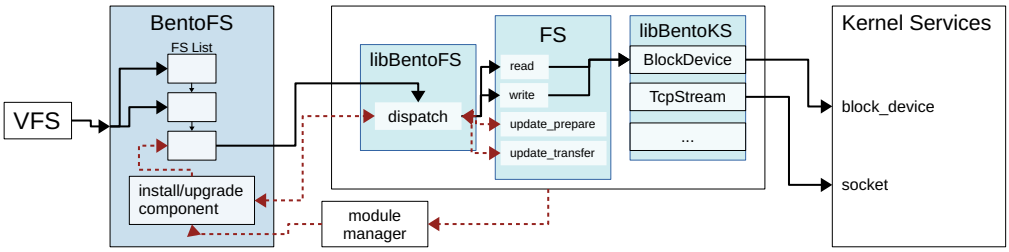


Figure 1: Design of Bento. Shaded components are parts of Bento. BentoFS is in C. The other shaded components are in Rust. Solid black lines represent the common-case operation pathway, detailed in §3.2 and §3.3. Dashed red lines represent the install/upgrade pathway and are described in §3.4

2.3 Our Approach: Bento

We have designed Bento for high velocity development of Linux file systems with the following properties:

- **Safety:** Any bugs in a newly installed file system should be limited, as much as possible, to applications or containers that use that file system.
- **Performance:** Performance should be similar to that of the same functionality implemented using VFS.
- **Generality:** There are a large variety of file system designs that developers might want to implement. Bento should not limit the types of file systems that can be developed.
- **Compatibility:** 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 upstream.
- **Live upgrades:** The framework should support dynamic upgrades to running file system code, transparently to applications, except for a small delay.
- **User-level debugging:** File system code should be easily migrated between userspace and the kernel to enable user-level debugging and correctness testing.

At a high level, Bento achieves the first three goals by enabling developers to write file systems in Rust, a type-safe, non-garbage collected, general-purpose language that is receiving increasing attention for kernel implementations. Of course, safely using Rust within Linux is a challenge of its own. The other three goals are achieved via careful architectural design. To provide *compatibility* without sacrificing *safety*, 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. For *live upgrades*, Bento 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 *user-level debugging*, Bento is designed with the same set of API calls whether it runs in the kernel or in the userspace. A simple build flag change is sufficient to choose a different mode.

2.4 Rust Primer

As background, 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 collectors. These provide useful building blocks for Bento.

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 type objects from pointers is unsafe and must be tagged as `unsafe` to compile. Calling unsafe functions is additionally unsafe. Although some systems allow unsafe Rust, Bento requires that its file systems contain no unsafe code.

Rust prevents most memory leaks 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. Leaking memory is not a safety violation in Rust, so the `drop` function is not guaranteed to be called, but memory leaks must be explicit instead of accidental.

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, synchronization mechanism such as locking must be used to safely obtain references. Acquiring the lock gives the caller access to the underlying data. Lock acquisitions 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.

These represent about 7% of the low-level bugs found in our analysis of popular kernel modules.

3 THE BENTO SYSTEM

In this section, we describe the architecture of Bento, explain how it interfaces with VFS and the rest of the kernel, and detail how it enables live upgrades and user-level debugging.

3.1 The System Architecture

Figure 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.

3.2 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. A selection of this API is in Table 2; the rest in the appendix. 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

Bento File Operations API (partial)
bento_init(&mut self, req, devname, fc_info)
bento_destroy(&mut self, req)
bento_read(&self, req, ino, fh, offset, size, reply)
bento_write(&self, req, ino, fh, offset, data, flags, reply)
bento_update_prepare(&mut self) -> Option<TransferOut>
bento_update_transfer(&mut, Option<TransferIn>)

Table 2: A subset of the Bento File Operations API. *req* includes the user application's uid, gid, and pid. *reply* includes data or error values. The full API is included in supplementary material.

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 (§3.4).

3.3 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, as described in §2.4, 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 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 3.

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 reference 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.

3.4 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 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.

3.5 Userspace Debugging Support

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

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: SocketAddr) -> 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: SocketAddr) -> Result<TcpListener>*	*sock_create_kern(...)* *kernel_bind(...)* *kernel_listen(...)*	Create, bind, and listen
	accept(&self) -> Result<(TcpStream, SocketAddr)>	*kernel_accept(...)*	Accept a connection

Table 3: 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.

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 2), 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 IMPLEMENTATION & EXPERIENCES

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.

4.1 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.

4.2 Experiences Using 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 Bento 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 [12]. 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 [26, 35]. 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 required 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 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 [34]. 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 [34], which test sequences of two operations, using the operations supported by Bento-fs. This resulted in 47314 benchmarks in total. CrashMonkey did

[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.

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 [47]. 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.

5 EVALUATION

Our evaluation of Bento aims to answer several questions: a) How well does Bento-fs perform on different workloads? b) How robust is the file system under crash consistency testing? and c) How expensive are live upgrades?

5.1 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.

5.2 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 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 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.

5.3 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

Benchmark	ext4-o	ext4-j	Bento-fs	Bento:ext4-j	Bento-user	user:ext4-j
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: Performance results for ext4 in `data=ordered` mode (ext4-o), and `data=journal` mode (ext4-j), Bentofs, 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.

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 [41] and RocksDB [43] 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 [14]. 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 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 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 3 shows the results for application

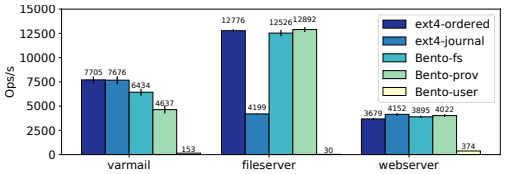


Figure 2: Performance results for ext4 in `data=ordered` mode and `data=journal` mode, Bento-fs, Bento-fs with provenance, and a userspace version of Bento-fs on Filebench application-style workloads in ops/s. Bento-user performs much worse on all benchmarks. Bento-fs and Bento-prov outperform ext4-journal on 'fileserver' due to different handling of un-synced writes to deleted files.

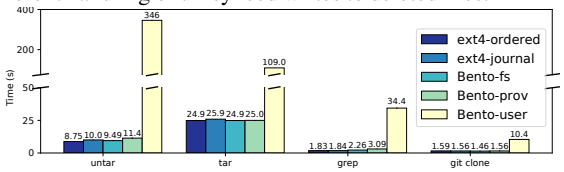


Figure 3: Performance results for ext4 in `data=ordered` mode and `data=journal` mode, Bento-fs, Bento-fs with provenance, and a userspace version of Bento-fs on application workloads 'tar', 'untar', and 'grep' on Linux source code and 'git clone' on xv6. Bento-user performs much worse than the other file systems. Ext4-journal performs somewhat better than Bento-fs and Bento-prov on 'grep'.

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

Key-Value Stores: Figure 4 shows the results for Redis ('set' and 'get') and RocksDB ('fillrandom' and 'readrandom') workloads on the four file systems. Due to caching, Bento-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.

5.4 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

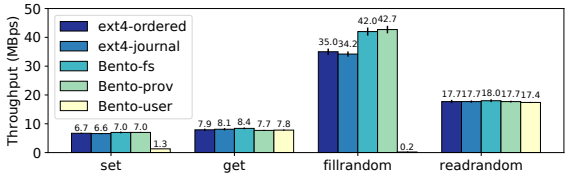


Figure 4: Performance results for ext4 in `data=ordered` mode and `data=journal` mode, Bento-fs, Bento-fs with provenance , and a userspace version of Bento-fs on Redis 'set' and 'get' and RocksDB 'fillrandom' and 'readrandom'. Bento-user performs much worse on write benchmarks.

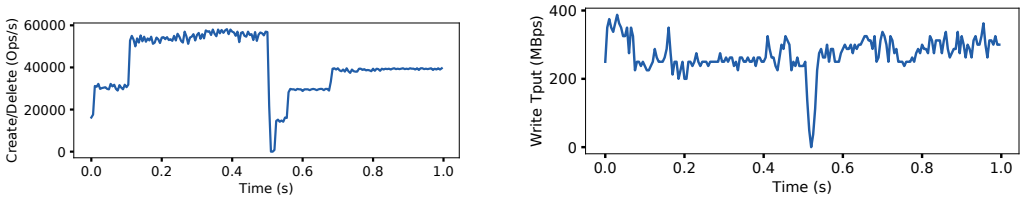
that initially contained 400,000 files. In the first, we executed a single thread that repeatedly created and deleted files. In the second, we executed 10 threads that repeatedly wrote and synced 64Kb writes to random files; we used 10 threads because with too many threads any service interruption caused 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 5a and Figure 5b.

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 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.

6 RELATED WORK

Using safe languages for kernel development. Several systems, including Pilot [40], SPIN [6], Singularity [20], Biscuit [13], Redox [42], and Tock [23] write the entire operating system, including the kernel, in a high-level language. SPIN leverages type safety to allow application-specific customization of kernel behavior. We are also not the first to integrate Rust into the Linux kernel [24, 27]. The Berkeley Packet Filter (eBPF) [32] is a type safe language for safe extensibility in Linux. Users can insert eBPF programs at predefined kernel locations, and the kernel verifies the safety of the inserted programs before running them. ExtFUSE [8] has enabled writing parts of a stackable file system using eBPF. Compared to these, Bento shows that it is possible to develop feature-rich file systems in a safe language to allow continuous integration of new features into a commodity operating system.

Software fault isolation and verification. An alternative approach is to allow development in an unsafe language (e.g., C) but do additional compile-time and runtime checks to prevent memory errors from affecting the rest of the system. Software fault isolation (SFI) [9, 30, 49] is a technique for sandboxing the impact of faults in C modules to the module itself; SFI has been widely used for protecting kernel device



(a) Create+delete operations in ops/s.

(b) Synced writes with 10-threads in MBps.

Figure 5: 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.

drivers. We chose to use Rust instead as it has lower runtime overhead and provides the additional benefit of bug prevention in addition to sandboxing errors. Software verification is a powerful tool for producing bug-free kernel code, and it has been shown that a simple, single-threaded file system can be verified [45]. Extending that work to handle concurrency and high performance file systems is still ongoing.

Moving kernel features to userspace. Microkernel design, where kernel services run in userspace, is another way to speed operating system development [1, 25] especially when safety and/or development velocity are more important than raw performance. Filesystem in Userspace (FUSE) is a good example in the Linux file system context. Many file systems have been developed in FUSE; when people need performance, they often re-implement the system inside the kernel [10, 19] using VFS. With Bento, developers no longer need to choose between performance and development velocity.

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 [4, 7, 22, 31]. To date, this approach has only been proposed and not implemented for file systems [28]. Performance can often be competitive with an equivalent kernel implementation, except when processors need to busy wait or when the system needs page remapping 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 [21] and Linux as the libOS [46] and Linux Kernel Library [39] projects; similarly, User Mode Linux [15] enables running a Linux kernel as a userspace process.

OS live upgrade. There are three main commercially available tools for live upgrade of Linux systems: ksplice [3, 36], kpatch [38], or kGraft [37]. 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. Most similar to Bento's design is K42 [5], 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 [18], another research operating system, also supports live upgrade of modular components. DynAMOS [29] and LUCOS [11] 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.

Stackable file systems. Stackable designs construct complex file systems by stacking layers of functionality on top of simple base file systems, enabling high velocity development. File system stacking is natively supported by VFS and is used by the overlay file system and eCryptfs, but these file systems still suffer from the velocity problems caused by kernel C code. FiST [50] proposed a framework for development of portable stackable file systems written in a new high-level language, augmented with C code. This improves velocity by reducing the complexity of code written by developer, but cannot support complex file system data structures and cannot provide safety guarantees about the C code.

7 CONCLUSION

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.

Acknowledgements. We would like to thank Remzi Arpaci-Dusseau for his helpful feedback on earlier drafts of this paper. We would also like to thank our anonymous reviewers and our shepherd, Rob Ross, for their helpful comments and feedback. This work is partially supported by the National Science Foundation grant CNS-1856636 AM04. This work was also supported by Google and Huawei.

REFERENCES

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A New Kernel Foundation For UNIX Development. In *Summer USENIX*, 1986.

[2] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R. Ganger, and George Amvrosiadis. File Systems Unfit as Distributed Storage Backends: Lessons from 10 Years of Ceph Evolution. In *SOSP*, 2019.

[3] Jeff Arnold and M. Frans Kaashoek. Ksplice: Automatic Rebootless Kernel Updates. In *EuroSys*, 2009.

[4] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *SOSP*, 2009.

[5] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W. Wisniewski, and Jeremy Kerr. Providing Dynamic Update in an Operating System. ATEC, 2005.

[6] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility Safety and Performance in the SPIN Operating System. In *SOSP*, 1995.

[7] Brian Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. User-level Interprocess Communication for Shared Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(2), May 1991.

[8] Ashish Bijlani and Umakishore Ramachandran. Extension Framework for File Systems in User Space. In *USENIX ATC*, 2019.

[9] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast Byte-granularity Software Fault Isolation. In *SOSP*, 2009.

[10] Ceph. Ceph kernel module. https://github.com/ceph/ceph-client.

[11] Haibo Chen, Rong Chen, Fengzhe Zhang, Binyu Zang, and Pen-Chung Yew. Live Updating Operating Systems Using Virtualization. In *VEE*, 2006.

[12] Russ Coxx, Frans Kaashoek, and Robert Morris. Xv6, a simple Unix-like teaching operating system, 2020.

[13] Cody Cutler, M. Frans Kaashoek, and Robert T. Morris. The benefits and costs of writing a POSIX kernel in a high-level language. In *OSDI*, 2018.

[14] Jeff Dean and Sanjay Ghemawat. LevelDB: A Fast Persistent Key-Value Store, 2011.

[15] J. Dike. A user-mode port of the Linux kernel. In *Annual Linux Showcase & Conference*, 2000.

[16] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An Operating System Architecture for Application-level Resource Management. In *SOSP*, 1995.

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

[18] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Safe and Automatic Live Update for Operating Systems. In *ASPLOS*, 2013.

[19] GlusterFS. Glusterfs kernel module. https://staged-gluster-docs.readthedocs.io/en/release3.7.0beta1/Features/libgfapi/.

[20] Galen C. Hunt and James R. Larus. Singularity: Rethinking the Software Stack. *SIGOPS OSR*, 2007.

[21] Antti Kantee. Rump File Systems: Kernel Code Reborn. In *USENIX Annual Technical Conference*, 2009.

[22] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas E. Anderson. TAS: TCP Acceleration as an OS Service. In *EuroSys*, 2019.

[23] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kB Computer Safely and Efficiently. In *SOSP*, 2017.

[24] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C.S. Lui. Securing the Device Drivers of Your Embedded Systems: Framework and Prototype. In *ARES*, 2019.

[25] Jochen Liedtke. On Microkernel Construction. In *SOSP*, 1995.

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

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

[28] Jing Liu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Sudarsun Kannan. File Systems as Processes. In *HotStorage*, 2019.

[29] Kristis Makris and Kyung Dong Ryu. Dynamic and Adaptive Updates of Non-Quiescent Subsystems in Commodity Operating System Kernels. In *EuroSys*, 2007.

[30] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Software Fault Isolation with API Integrity and Multi-principal Modules. In *SOSP*, 2011.

[31] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, and et al. Snap: A Microkernel Approach to Host Networking. In *SOSP*, 2019.

[32] Steven McCanne and Jacobson Van. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Winter USENIX*, 1993.

[33] Jeffrey C. Mogul and John Wilkes. Nines are Not Enough: Meaningful Metrics for Clouds. In *HotOS*, 2019.

[34] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing. In *OSDI*, 2018.

[35] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo Seltzer. Provenance-Aware Storage Systems. In *ATEC*, 2006.

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

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

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

[39] O. Purdila, L. A. Grijincu, and N. Tapus. LKL: The Linux kernel library. In *9th RoEduNet IEEE International Conference*, pages 328–333, 2010.

[40] David D. Redell, Yogen K. Dalal, Thomas R. Horsley, Hugh C. Lauer, William C. Lynch, Paul R. McJones, Hal G. Murray, and Stephen C. Purcell. Pilot: An Operating System for a Personal Computer. *Commun. ACM*, 23(2):81–92, February 1980.

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

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

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

[44] Marc Rozier, Vadim Abrossimov, François Armand, I. Boule, Michel Gien, Marc Guillemont, F. Herrmann, Claude Kaiser, S. Langlois, Pierre Leonard, and W. Neuhauser. CHORUS Distributed Operating Systems. *Computing Systems*, 1988.

[45] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button Verification of File Systems via Crash Refinement. In *OSDI*, 2016.

[46] H. Tazaki, Ryo Nakamura, and Y. Sekiya. Operating System with Mainline Linux Network Stack. 2015.

[47] Amin Vahdat and Thomas E. Anderson. Transparent result caching. In *1998 USENIX Annual Technical Conference, New Orleans, Louisiana, USA, June 15-19, 1998*. USENIX Association, 1998.

[48] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To FUSE or Not to FUSE: Performance of User-Space File Systems. In *FAST*, USA, 2017. USENIX Association.

[49] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-based Fault Isolation. In *SOSP*, 1993.

[50] Erez Zadok and Jason Nieh. FiST: A Language for Stackable File Systems. *ACM SIGOPS Operating Systems Review*, 34, 03 2002.

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, lock_owner, reply)	Called on each close of a file
bento_release(&self, req, ino, fh, flags, lock_owner, 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, lock_owner, start, end, typ, pid, reply)	Test for a file lock
bento_setlk(&self, req, ino, fh, lock_owner, 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 5: The full 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.