High Velocity Kernel File Systems with Bento

Samantha Miller Kaiyuan Zhang Danyang Zhuo† Thomas Anderson

University of Washington †Duke University

Abstract

High development velocity is critical for modern cloud systems. However, rapid development and release cycles have mostly skipped operating systems. Modifications to behavior in Linux, the most widely used server operating system in the cloud, must be done slowly to minimize risk of introducing bugs, be limited in scope, or be implemented in userspace with a potential performance penalty.

We propose Bento, a framework for high velocity development of Linux kernel file systems. Bento is inspired by the recent availability of type-safe, non-garbage collected languages like Rust. It interposes a thin layer between kernel calls to the file system and file system calls back to the kernel, exposing alternative interfaces to enable kernel file systems written in safe Rust. Future work will provide support for online upgrades, userspace debugging, and composable file systems. We evaluate Bento by using it to implement the xv6 file system and comparing against baselines written using the kernel VFS layer and FUSE. We find that the Bento file system achieves comparable performance to the VFS version and much better performance than the FUSE version. We also evaluate against ext4 on the macrobenchmarks and find that ext4 performs between 33% and 3.2× better than the Bento xv6 file system.

1 Introduction

High development velocity has become a widespread talisman for cloud software development [28]. Many popular cloud systems roll out new software releases on a weekly or even daily basis, to give users faster access to new features, to gain insight into priorities for further development, and to reduce integration costs. While this design pattern may seem inappropriate for mission critical software, cloud vendors have shown it is practical to use short release cycles for many high reliability services, including databases [8] and network stacks [10, 13, 22].

Rapid release cycles have largely skipped the operating system kernel development community, however. Linux is the most widely used server operating system for the cloud, but new versions drop only every few months, with major changes limited to once every few years. Of course, Linux is open source, and so anyone is free to iterate more rapidly, at the cost of the later pain of reintegration with the mainline development tree.

The Linux community has adopted several approaches to improving feature velocity, none entirely successful. One approach is to try to future-proof the kernel by adding features before they are needed. We can see that in action with the popular Docker container manager [11]. Docker leverages several recently-added Linux kernel features, but in the process exposed a number of potentially critical flaws in those kernel services that could compromise the security of the entire operating system. Alternately, we can move kernel services to user level, such as with the FUSE file system abstraction [14] and Open vSwitch [26]. However, these can impose a prohibitively high performance penalty [1, 32], necessitating a kernel caching layer [5] that poses its own set of tradeoffs. Certain Linux kernel interfaces can be rapidly reconfigured with eBPF [30] scripts, but only for small linear snippets of code. The widespread belief that in the future, all high performance operations must "bypass the kernel" is an illustration of how operating systems are losing the race [3, 25, 36].

Our goal is to enable high velocity development for high-performance, general-purpose operating system kernel extensions. Our trust model is that of a slightly harried kernel developer, rather than an untrusted application developer. We want to provide a way for kernel developers to add kernel features in a manner that isolates bugs to within the extension and also allows for dynamic replacement of that functionality without the need to restart applications [35]. To be as widely applicable as possible, we focus on enabling rapid development of Linux kernel code, rather than to assume a new code base designed for extensibility, such as Exokernels [12], Spin [4], or Barrelfish [35]. To be concrete, we restrict ourselves, at first, to file system extensibility. We leave dynamic

replacement of file systems to future work.

Our approach is inspired by the recent availability of type-safe, non-garbage collected, performant languages like Rust. Writing kernel extensions in Rust eliminates a class of cross-module bugs that could compromise kernel security, without the performance overhead of running at user-level or the restrictions on extension behavior imposed by eBPF. However, supporting compatibility with existing operating systems and features for high development velocity such as dynamic module replacement, debugging, and code reuse, is challenging.

For this work, we focus on high velocity kernel file systems. We have built a framework, called Bento, for injecting general-purpose file systems and file systems extensions, written in Rust, into Linux. Surprisingly, Linux's existing pluggable file system interface, VFS, is poorly suited to our needs, as it assumes shared data structures can pass freely across the extension interface, complicating compile-time type checking. Instead, Bento interposes a thin layer for calls into the file system, and calls from the file system back into the kernel, providing safety, high performance, generality, and compatibility with Linux. While our architecture is designed to be compatible with graceful online upgrades of running file systems along with support for other features for high development velocity, we leave that for future work.

We have used Bento to implement the xv6 file system to run in the Linux kernel. We have additionally implemented baseline versions, one written in C against the VFS layer and one (written in Rust) running in userspace using FUSE. We found that our framework has performance very similar to, and sometimes better than, the VFS C version while the FUSE version performed much worse than both.

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 present techniques for allowing safe Rust code to run in the Linux kernel and access kernel functionality.
- We implement a file system using Bento and evaluate its performance characteristics.

2 Background

2.1 Bug Analysis

One of the existing barriers to fast evolution in Linux comes from buggy code. New code often introduces bugs, disincentivizing fast evolution for mission-critical pieces of code like operating systems. Kernel code is particularly affected by this because kernel bugs are often difficult to find and can have severe non-local consequences. In particular, memory bugs, such as memory reuse and dangling pointers, can have catastrophic consequences on the reliability of the system, potentially even leading to security violations.

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: Count of analyzed bugs with effects of each bug, categorized as memory, concurrency, or type.

To understand the properties of bugs in existing Linux kernel extensions, we analyzed bug reports for three extensions used by Docker: AppArmor for security, Open vSwitch Datapath for networking, and Overlay FS for file system support. We analyzed all bug-fix git commits from 2014-2018 and categorized them by the type of bug that was fixed.

Our analysis focused on what we call low-level bugs: bugs that are unrelated to the specific logic of the extensions. These bugs can be caught without knowing specific correctness properties needed by the extension. This is opposed to semantic bugs which are caused by violations of high-level correctness properties. Low level bugs made up 50% of the total bugs. We divided the low-level bugs into three categories: memory bugs, concurrency bugs, and type errors. Memory bugs refer to incorrect usage of memory, including `NULL` pointer dereferences, out-of-bounds errors, and memory leaks. Concurrency bugs are caused by incorrect concurrency patterns, such as deadlocks and race conditions. Type errors are caused by incorrect usage of kernel types, most often by interpreting error values as valid data.

The results of the analysis are shown in Table 1. We found that 68% of these bugs were memory bugs. Of the memory bugs, 50% were a type of memory leak. Many of the bugs occurred along error handling pathways, often due to incorrect checking of returned values (unchecked error values) or missing cleanup (memory leaks, `NULL` pointer dereferences, etc.). Based on our analysis of these low-level bugs, 93% would be prevented by using Rust. The remaining 7% of low-level bugs were primarily deadlocks.

Many of the bugs could have serious impacts on the integrity of the system. Of the identified low-level bugs, 26%

	Safety	Performance	Generality	Online Upgrade
VFS	✗	✓	✓	✗
FUSE	✓	✗	✓	✗
eBPF	✓	✓	✗	✗
Bento	✓	✓	✓	tbd

Table 2: A comparison of Linux file system extensibility mechanisms. None of Linux's existing mechanisms provide all the desired features.

of the bugs caused a kernel `oops` which either kills the offending process or panics the kernel. An additional 34% of the analyzed bugs bugs would result in a memory leak, potentially leading the system to run out of memory and opening up the system to DoS attacks.

2.2 File System Extensibility Today

Linux has several existing techniques to support rapid evolution of file system functionality. These include the Virtual File System (or VFS) layer built into Linux, FUSE for userspace file systems, and eBPF for running small portions of a user space code safely in the kernel. However, none of these approaches provide all of the properties we need for high velocity development. A summary is shown in Table 2, and details are discussed below. Note that compatibility with existing Linux code is implicit in all of these approaches and in Bento.

VFS: Linux provides a mechanism for adding new file systems called the Virtual File System (or VFS) layer. This layer defines a set of function pointers to be implemented by new file system modules and calls these functions inside related system calls. It is used by all major file systems in Linux.

This interface prioritizes generality and performance, allowing file systems maximum flexibility when interacting with core kernel components. The resulting interface is complex and has few guardrails, making it difficult for developers to implement new functionality without introducing bugs. While a new file system can be loaded dynamically, an existing file system cannot be modified except by mount/unmount and quiescing application use of the file system. Likewise, debugging support is limited.

FUSE: Filesystem in Userspace, or FUSE [14], enables running file system code in userspace, via a small kernel VFS layer that forwards operations to the userspace implementation. Thus, FUSE is able to achieve safety and generality, along with the ability to use normal user-level debuggers. This comes at a cost, however. All file system operations pass through VFS and the FUSE kernel driver before being packaged up and copied to userspace, reducing performance by up to 83% [32]. Despite this slowdown, FUSE is frequently used for prototyping new file systems, especially in circumstances where performance is not critical. FUSE does not provide a mechanism for transparent online modification of running file systems, although such a system could theoretically be implemented at user level.

eBPF: Another approach to safe extensibility in Linux is the eBPF (extended Berkeley Packed Filter) [23], an in-kernel virtual machine that allows short extensions with limited control flow and written in a restricted language to be run at predefined points in the kernel. While the main-line Linux kernel doesn't support eBPF for file systems, a project (ExtFUSE [5]) has provided support for parts of a FUSE file system to be run in the kernel using eBPF. For kernel code that can fit within thee eBPF model, this provides safe extensibility without significant performance overhead. However, the restrictions placed on eBPF extensions make it very difficult to implement whole file systems or even significant file system extensions using eBPF. ExtFUSE does not support dynamic reconfiguration.

3 Goals and Challenges

The goal of Bento is to provide for high-velocity development of Linux file systems. To make our design goals concrete, consider the OverlayFS extension to Linux used by Docker. OverlayFS allows for the name space of a file system to be layered on top of another, allowing containers to be configured with a base file system plus changes. Or consider improving the support for non-volatile memory (NVM) to Linux. Systems such as Strata [17] have shown that prepending an operation log stored in NVM can dramatically improve write performance while reducing vulnerability to application-level bugs. These operation logs can be replicated for high availability [2].

Finally, consider what would be needed to add data provenance to Linux - the ability to track all of the data sources and executable images that could have affected a particular output file [31]. If a data source becomes invalid (e.g., because of a change to sensor calibration), provenance can be used to track down what derived data needs to be regenerated. Further, old versions of data files may need to be retained (and later garbage collected) if they are part of the provenance of live output files.

In all three cases, the functionality needs to work with existing, unmodified Linux binaries, has complex internal logic and data structures, is performance-sensitive, benefits from ongoing development, and to be deployable, must not compromise the security of the rest of the operating system. We assume the developer is well-intentioned but a bit clumsy - it is not our intent to prevent malicious insider attacks for newly developed code.

Thus, our framework must support several, seemingly conflicting, goals:

- **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. These bugs should be kept to a minimum.
- **Performance:** Performance should be similar to that achievable by the same functionality implemented directly in the kernel.
- **Generality:** There is a large variety of file system designs that developers might want to implement. The framework should not limit the types of file systems that can be developed.
- **Compatibility:** New functionality should be deployable to existing, unmodified Linux binaries without recompiling or relinking, and without substantial changes to Linux's internal architecture.
- **Development velocity:** The framework should support dynamic upgrades to running file system code, transparently to applications, except for a small delay. Further, code should be easily migratable between user level and the kernel, to enable use of modern debugging and software analysis tools. This last goal is supported architecturally by our approach, but experimental demonstration is beyond the scope of this paper.

Our high level approach for Bento is to enable writing file systems in a safe, non-garbage collected language, specifically Rust. This is able to provide the first three goals detailed above. Rust's strict type system is able to provide safety, eliminating certain classes of bugs such as `NULL` pointer deferences or use-after-free bugs. Since Rust is compiled like C and does not use garbage collection, it has performance similar to C and does not suffer from performance unpredictability caused by garbage collectors. Rust is a general purpose programming language and provides the necessary generality to enabling writing a wide variety of file systems.

To realize this approach, we need to address several challenges. Compatibility with existing operating systems and online upgrades, the other two goals for this work, are not inherently provided by writing file systems in Rust. Bento must provide additional support in order to achieve these properties. However, challenges arise when trying to provide that support.

3.1 Compatibility Challenges

In order for a Rust file system to execute in the Linux kernel, there must be a way for the Rust file system to interact with the C kernel. A naive approach is just compiling the Rust file systems into a binary format and load it into the kernel. Rust is designed to interface with code written in other languages, particularly C, easily using its Foreign Function Interface. Rust code can call functions written in C and vice versa, and Rust data structures can be tagged so they use C-style memory layout. In fact, without considering any other factors, running Rust code in the Linux kernel is fairly straightforward.

However, this naive approach does not maintain the safety of the Rust file systems. Rust code that calls external functions or dereferences raw pointers must be tagged as unsafe. Rust's type system is not able to provide the same guarantees about unsafe code, e.g. `NULL` pointer dereferences and out-of-bounds accesses are possible, so unsafe code cannot provide the safety we require for Bento. Simple techniques for introducing safety, such as wrapping C functions in safe wrappers or replacing pointers with references, are not enough to fully provide safety due to fundamental challenges caused by kernel design patterns, which we now describe. We assume that the kernel is correct.

3.1.1 Challenge 1: Memory Management

One challenge is caused by memory management for data structures passed across the boundary between the file system and the kernel. Rust is able to provide memory safety and automatic memory management by doing compile-time tracking of data structures. However, the VFS interface requires that some data structures created by the file system be passed across the kernel/file system boundary and back again. Since the Rust compiler is not able to analyze the code outside the file system, it is not able to verify the safety of taking ownership of data structures from the kernel. Therefore, the VFS file system interface cannot be implemented in safe Rust.

3.1.2 Challenge 2: Accessing Kernel Services

Another challenge stems from the file system's need to access services provided by the kernel. However, the interfaces exposed by kernel services are not designed for Rust's safety guarantees, so kernel services cannot necessarily be exposed safely to Rust file systems without modifications. To allow the file system to use kernel services safely, Bento must translate the unsafe kernel-provided interfaces into interfaces that can be used by the file systems safely.

3.2 Online Upgrades Challenges

Online upgrades, updating a file system without bringing it offline, also is not provided by writing the file systems in a safe language. In Linux today, file system module upgrades is done by shutting down all services relying on the file system, unmounting the file system, removing the module, inserting the new module, mounting the new file system, and

then restarting all services. In order to support online upgrades, additional functionality must be added to enable updating to a new version of the file system without requiring the file system or services running on top of it to be shut down. Trying to implement that functionality in Linux gives rise to the following challenges.

3.2.1 Challenge 3: Shared Data Structure Lifetime

The memory management pattern described in §3.1.1, where data structures created by the file system are passed to the kernel, also introduces challenges for online upgrades. Since the kernel holds data structures backed by file system memory, and the file system has no way to control when that memory should be reclaimed. If the file system were updated when there were outstanding data structures held by the kernel, those kernel pointers would become invalid. To avoid this case, the file system must wait for the kernel to have completed all operations on the file system and have returned all shared-ownership data structures to the file system. There is no guarantee of this happening until the file system is unmounted, so upgrades cannot be done online.

3.2.2 Challenge 4: Tracking In-Use Structures

Another challenge is caused by the need to track data structures that the file system is currently using, both data structures from kernel services and in-memory data used by the file system. For example, a running file system will execute block I/O or possibly network operations and may be using kernel data structures for those operations when the upgrade occurs. The file system could also have internal, in-memory state such as which blocks need to be written to a commit log or a cache of on-disk data structures. If the file system updates without transferring any of its in-use data structures, potentially bad behavior can occur. In the best case, caches of on-disk data structures need to be rebuilt, and performance temporarily suffers. In the worst case, correctness conditions could be violated if the file system requires long-lived state. Since the existing techniques for upgrades in Linux assume that the file system will be completely shut down during the upgrade, there are no mechanisms to transfer data structures.

3.3 Debugging Challenges

The ability to quickly and effectively debug code is critical for fast development in practice. Kernel code is notoriously difficult to debug because of the often non-local effects of kernel bugs and the potential for a buggy operating system to interfere with the process of debugging. In order to enable effective debugging, we propose allowing file systems written using Bento to be run in userspace without requiring code modifications.

3.3.1 Challenge 5: Debugging API Design

To support running the same code in the kernel and in userspace, we must provide an API that can be implemented in both. All APIs, both for Bento to call file system functions and for the file system to access necessary services, must be the same in both the kernel and userspace. Providing compatibility with Linux will not necessarily provide this because the interfaces provided by kernel services may not be compatible with the system call interface.

3.4 Code Reuse Challenges

The ability to reuse code is also important for development velocity. This is particularly relevant for file systems because there are many circumstances when a user would want to modify the behavior of an underlying file system, such as enabling encryption or tracking data provenance. In Linux today, developers can implement these types of file systems by stacking layers of file systems (e.g., the `ecryptfs` file system can be layered on top of another file system to add encryption). The higher layer file systems call top-level VFS functions to access the lower file systems as if the relevant system call had been executed. This support for stackable, or composable, file systems allows developers to provide services as file system modules that can be used with any existing file system.

3.4.1 Challenge 6: Composable File System Support

Linux's existing model for composable file systems can be supported by exposing the top-level VFS functions to Bento file systems. However, it is not clear that this is the best solution. Calling top-level VFS functions can add overhead to each call to a lower file system, resulting in potentially large overhead if several file systems are layered on top of one another. Bento may be able to provide a different interface for supporting composable file systems that does not introduce this overhead but still provides the necessary flexibility.

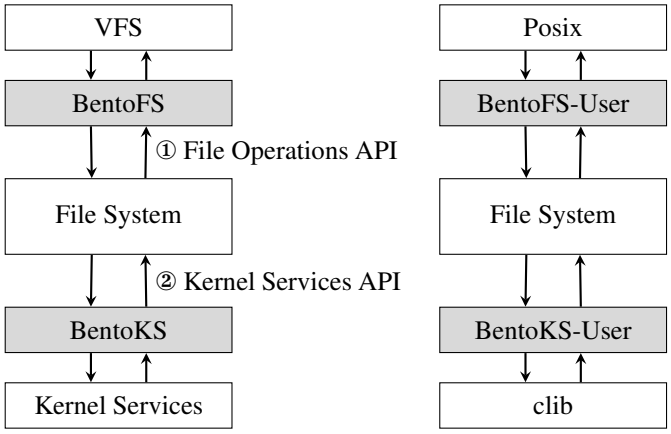
4 Bento

4.1 System Overview

The design of Bento is shown in Figure 1. Shaded portions are the framework. The framework runs as a thin layer that sits between the unmodified Linux kernel and kernel-level file systems designed for our framework. The Linux kernel is unmodified other than the introduction of Bento. Like the VFS layer, Bento defines a set of function calls that file systems must implement and provides a mechanism for file systems to register themselves with the framework by providing the necessary function pointers. Unlike the VFS layer, Bento is designed to support file systems written in Rust, a type-safe language that provides memory safety and data race freedom.

Challenge	Solution	Problem Description	Detailed Solution
Unsafe Shared Memory Management	Restricted Memory Sharing	§3.1.1, §3.2.1	§4.3
Unsafe Kernel Interfaces	Safe Abstractions Around Kernel Services	§3.1.2	§4.5
Transferring Objects During Upgrade	Online Upgrade Component	§3.2.2	§4.8

Table 3: Summary of Challenges and the Associated Solutions



(a) Kernel Bento

(b) Userspace Bento for Debugging

Figure 1: The design of Bento

Table 3 shows a summary of the challenges and solutions in Bento. Bento currently consists of two components. One component of the framework interposes between the VFS calls and the file system, handling calls into the file system. This component provides the file operations API, translating from the VFS interface. The other component interposes below the file system, handling calls out of the file system into the kernel. This component provides wrappers around kernel data structures and functions, allowing the Rust file system to safely access relevant kernel functionality. For file systems, this primarily handles block I/O.

4.2 Usage

To write a file system using Bento, developers write a safe Rust kernel module using the provided APIs and insert that module into their running Linux kernel like any other kernel module. File system functions are exposed to the operating system by implementing the file operations API and providing those function pointers to Bento when the file system is inserted. When file system functions need to access kernel functionality, they can do so by calling the safe Rust functions provided by the kernel services API.

4.3 BentoFS & File System Operations API

The VFS layer and the patterns it introduces cause fundamental challenges to safety when handling memory management of shared data structures, in particular inodes. Inodes are allocated and destroyed using functions implemented in the file system and called by the kernel. When the kernel needs a new inode, it requests one from the file system which allocates the inode using its own memory pool. When the kernel is finished with the inode, it returns the inode to the file system so the memory can be reclaimed. Giving ownership from Rust to C can be implemented in Rust by leaking the memory behind the data structure; this is safe because leaking memory does not violate Rust's memory safety, but is not ideal. Taking ownership from C to Rust cannot be implemented safely. Rust must trust that the data structure will not be used anymore and was originally allocated by Rust. Since these properties cannot be validated by the Rust compiler, this is inherently unsafe.

In order to enable safe file systems, Bento must provide a different interface than the VFS layer for file system operations. Calls from the VFS layer are intercepted by BentoFS and translated into this new interface, shown in Figure 1 at ①. This interface calls from BentoFS to the file system, so the interface must be designed so it can be implemented safely. To support this, we define a model that our interface must follow.

4.4 Ownership model

Our interface follows what we call an "ownership model", borrowing the terminology from Rust. In this model, ownership of an object can never be passed across the interface, but objects can be "borrowed". For each object, one side of the interface is responsible for both the lifetime management (tracking when the object is no longer needed) and memory management. To share an object, the caller passes a reference to the object to the callee. This does not pass ownership (the callee has no control over the underlying memory) but does allow the callee to access the object. This is analogous to a borrow in Rust and similarly can be mutable or immutable, allowing modification of the object or not, respectively. To the file system developer, this is just writing typical Rust code.

This model implies a contract between the caller and the callee. The caller is responsible for ensuring that the object is not freed while it has been borrowed, that the object is valid, and that only one mutable borrow exists at one time. The

callee is responsible for only accessing an object during the borrow window, accessing objects correctly (i.e., no pointer arithmetic), and only mutating objects during a mutable borrow.

In this case, the callee is the file system, written in Rust. All of the callee's responsibilities are checked by the Rust compiled when using safe Rust, so the file system is guaranteed to uphold the model. Our framework is the caller and must be carefully designed to fulfill its side of the contract.

This ownership model can be viewed as a relaxed version of what is needed across address space boundaries where no memory can be shared. This observation led us to leverage the FUSE kernel module and the FUSE low-level API when developing BentoFS and the file operations API. The file operations API is a Rust version of FUSE low-level API augmented with a reference to the `super_block` data structure needed for file system block operations.

This model should not introduce significant performance overhead. This loan/borrow model is only used to check compile time properties, so does not add performance overhead at runtime. The performance impact of the interface change is more difficult to predict, but should still be low. The design interface does not increase the functionality needed to implement a file system, it just splits the behavior implemented by a VFS file system between BentoFS and the file system.

4.5 BentoKS & Kernel Services API

File systems need access to kernel functionality implemented outside the file system, such as block I/O for access to the underlying storage device. These kernel interfaces, like those in the VFS layer, are not designed to abide by type-safety properties and so cannot be directly used in the file system. In order to enable use of necessary kernel services, BentoKS provides safe abstractions around kernel data structures and functions.

As an example, we will focus on the kernel block I/O functions. File systems in Linux access block devices using the buffer cache. In this API, a file system that needs to read or write to a block device calls `sb_bread`, passing in a pointer to the `super_block` data structure and a block number. 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` and the file system can 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.

The widespread use of pointers and pointer manipulation in the Linux kernel make this challenging. Safe Rust disallows dereferencing raw pointers because the compiler cannot check the validity of the memory being pointed to. Rust instead relies on typed references that cannot be offset, cast to nonequivalent types, or `NULL` safely. However, many kernel interfaces rely on pointers, so these interfaces cannot be used by the file system safely.

4.6 Capability Model

In order to access kernel functionality, the file system must be able to use kernel data structures, both for calling kernel functions and for making use of objects provided to the file system by the file operations API described above. The kernel operates on pointers, but 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 to the superblock, no guarantees can be made about the memory layout underlying that pointer.

We use a capability-based model to safely expose kernel pointers to the file system where pointers are replaced by capability-style types defined in Bento. These types give the file system the right to access to the fields of the data structure and to call functions that are exposed by that type. Creation of these capability-types is limited; they cannot be safely cast from other types, and initialization is predefined and sometimes entirely disallowed. Bento converts between the capability type and the analogous kernel type. For example, the file system often receives the `SuperBlock` capability type from the file operations API to represent the kernel `super_block` data structure. It can use the `SuperBlock` capability type to read fields of the kernel `super_block` and call kernel functions like `sb_bread` for block I/O that require a kernel `super_block`. The `SuperBlock` type cannot be created by the file system, so having this type is proof that the file system has access to a valid kernel `super_block`. Bento can then safely convert the capability type to a pointer and directly access kernel functions.

The capability types are compile-time wrappers around pointers so the Rust compiler can enforce correctness properties at compile time. It is assumed that the kernel passes in valid pointers, so no properties need to be checked at runtime and no runtime overhead is added.

4.7 Wrapping Abstractions

Bento must also provide wrapping abstractions around kernel services so they can be used safely by the file system. To enable file systems written in safe Rust, Bento must provide safe abstractions wrapping kernel services. These abstractions can be used by the file system like any other Rust data structures and functions.

To be concrete, we address the example discussed above. We provide a safe abstraction to wrap the kernel `buffer_head`. We implement a method on the `BufferHead` wrapper to convert the separate pointer and size fields for the contained memory region into a sized memory region that can be used safely. That method must use unsafe code to make a 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, which is called when the wrapper goes out of scope. With this, buffer management has the same properties as memory management in Rust: memory leaks are possible but difficult.

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 will most likely need to perform a runtime check to ensure that the requirements are held. This overhead should be small since checks are not performed often and are simple.

4.8 Online Upgrades API

In order to enable online upgrades, Bento will provide a mediating layer that maintains any state that needs to be preserved through the upgrade, such as long-lived kernel data structures like a network connection for a networked file system or internal file system state like an in-memory cache of on-disk data structures. Bento is already a runtime in the kernel, so it can easily be extended to include the necessary functionality.

This component will need to have a data structure transfer mechanism so important data structures can be passed from the old version of the file system to the new version during the upgrade. Kernel data structures can already be tracked by Bento through the kernel services API, and functionality can be added to support transferring these data structures. To transfer file system internal data structures, the online upgrade component will extend Bento's interface with new functions for storing in-memory state and initializing from that provided state. When the old version of the file system is about to be stopped, the online upgrade component will call the file system's provided function. This function will perform any necessary shutdown, such as flushing state, and will return in-memory state that should be transferred. This state will then be passed to the new version of the file system when it starts up so it can restore the necessary in-memory state.

4.9 Userspace Debugging

To support easy debugging, Bento will enable developers to run the same code in userspace and in the kernel and so use userspace debuggers. To enable this, Bento will provide alternate implementations of the BentoFS and BentoKS components that interface with userlevel interfaces, specifically the POSIX API instead of VFS and C library functions instead of kernel services. Since the interfaces exposed by the kernel and by userspace libraries are different, it is not obvious that the APIs written for the kernel will be able to be implemented without modification. We will analyze and implement this as part of our future work.

5 Implementation

Bento is built in Linux kernel version 4.15. It is implemented as a Linux kernel module in 1409 lines of Rust code for BentoKS and 7409 lines of C code for BentoFS.

5.1 Kernel-Mode Rust

Writing a kernel module in Rust is different than writing userspace Rust code. The basic structure of our kernel module is borrowed from tsgates/rust.ko on Github. The kernel module is compiled as a static library which is then linked with any required C code to generate the kernel module (a .ko file). This kernel module can then be inserted into the kernel as normal by any sudo user. Kernel code in Rust, like all kernel code, cannot use the standard library, but the Rust core library can still be used. We found that we had to additionally limit the Rust implementation to code that can't cause a panic.

The Rust portions of the Bento kernel module must interface with C code. Rust data structures can be tagged with `#[repr(C)]` to force the memory layout to match the C layout of the same structure, allowing the data structure to be passed across the language boundary. Rust functions can be called from C as long as they are tagged with `#[no_mangle]`, preventing the Rust compiler from mangling the name of the function. Rust's FFI (Foregin Function Interface) enables Rust code to seamlessly call functions implemented in C. The Rust code only needs to define the function interface in an `extern` block and the functions will be linked at compile time. The Rust bindgen tool can be used to automatically generate these bindings from C header files.

5.2 VFS Interposition

One of the primary jobs of Bento is to interpose between the VFS layer and the file system. As part of this translation, the file operations component of Bento must handle the interactions with core kernel data structures that are expected of a file system written against the VFS layer.

We use the FUSE kernel module and the FUSE low-level interface as starting points for BentoFS and the file operations API. The FUSE kernel module must implement much of the same functionality as BentoFS, so we use a modified version of it to implement BentoFS.

Unlike in FUSE, the file operations layer and the file system reside in the same address space and trust domain. Bento can therefore communicate with the file system using function calls. Our framework implements this like the VFS layer; function pointers to file system operations are stored in a data structure that is provided to Bento when the file system is mounted and upgraded.

5.3 Implementing Safe Wrappers

The Rust capability types are implemented as a Rust type with one field: a pointer to the relevant kernel type. This enables BentoFS to pass a pointer to the kernel data structure to the file system functions with no overhead. BentoKS implements methods of these capability types that the file system can use to safely access kernel functionality. These functions can be called from the Rust file system on the capability type data structures even though these were originally allocated as C data structures.

6 Evaluation

We evaluate the performance of Bento to determine what, if any, overheads exist to using it. For this, we have implemented the file system from the xv6 teaching operating system and two variants: one written in C, running in the kernel using the VFS layer and one written in Rust, running in userspace using FUSE. By comparing against the VFS layer, we can determine the overhead Bento introduces. By comparing against FUSE, we can quantify the benefits of Bento relative to a purely userlevel file system.

Since xv6 is a toy operating system, it is missing optimizations that a commercial-grade optimizations would have. This can heavily impact the FUSE baseline because the unoptimized operations may be particularly expensive from userspace. The VFS baseline is also less optimized than Bento because Bento inherits optimizations from the FUSE kernel module while the VFS baseline was just written for this evaluation. Therefore, the xv6 evaluation could be somewhat unfairly optimistic to Bento when compared to the same evaluation on a commercial-grade file system. We therefore also compare against ext4 on the macrobenchmarks. Ext4 is more optimized than the xv6 file system, but the performance results can still be compared to understand ballpark performance differences. Relatively small differences can indicate that our results may be similar to those we would achieve on a commercial-grade file system. We mount ext4 with the `data=journal` option so it logs file data in the journal like the xv6 file system.

6.1 Xv6 File System Changes

In order to write the xv6 file system in Rust and run the benchmarks, some changes needed to be made to the file system design. In all versions of the file system, we needed to add locks around inode and block number allocations due to race conditions on the block device. We also added double indirect blocks to all three versions of the file system so files up to 4GB could be created. In general, the Rust versions include more locks than the C version and official xv6 repository [34], specifically on global mutable variables that are only modified during initialization. Otherwise, the Rust file systems are nearly identical to the C file systems.

6.2 Baseline Implementations

As a baseline, we implement a VFS file system written in C. It is implemented in 1862 lines of C code. This filesystem is as close to our framework's version as possible to enable accurate comparison between the two approaches. This baseline allows us to analyze any overhead that Bento may introduce over the VFS layer.

The other baseline is a userlevel version using FUSE. This is 1744 lines of Rust and uses a Rust reimplementation of the FUSE userspace library [15] with minor changes such as enabling the writeback cache. The code for this version is nearly identical to the code written using our framework. Minor changes to the code are needed to swap out kernel services for Rust user-level services, such as using the Rust standard library mutex instead of the kernel semaphore. Additionally, block I/O from userspace is done by opening the Linux disk file using the `O_DIRECT` flag. We note future work will be able to run the same code in Bento and at userlevel.

6.3 Test Setup

The benchmarks were run on a machine with 8 $\times$ Intel Core i7 CPU, 31 GiB DDR4 RAM, and a 512GB Samsung PM981 NVMe SSD. All benchmarks were run using the SSD as the backing device. Due to the present health concerns, the benchmarks were run through a virtual machine using PCIe passthrough to access the SSD.

6.4 Benchmarks

From filebench, we run the single-threaded and 32-threaded read, write, file creation and file deletion microbenchmarks and the varmail and fileserver macrobenchmarks. To this, we also measure untaring the Linux kernel. These benchmarks were run on an NVME SSD and executed for one minute in all cases. The varmail macrobenchmark simulates a mail server. It repeatedly generates file creates, file deletes, file reads and writes, and appends to an operation log, syncing after writing. The fileserver macrobenchmark simulates a file serving application.

We see significant slowdown because of slow block I/O from userspace, even through the `O_DIRECT` file interface. Each block operation from userspace must pass across the user/kernel boundary and through the VFS layer before reaching the disk, adding 200-400ns to each operation. On top of that, the file interface imposes additional overheads. The file system must occasionally sync blocks to disk (such as during log operations), but the file interface provides no way to sync parts of a file, so the whole disk file must be

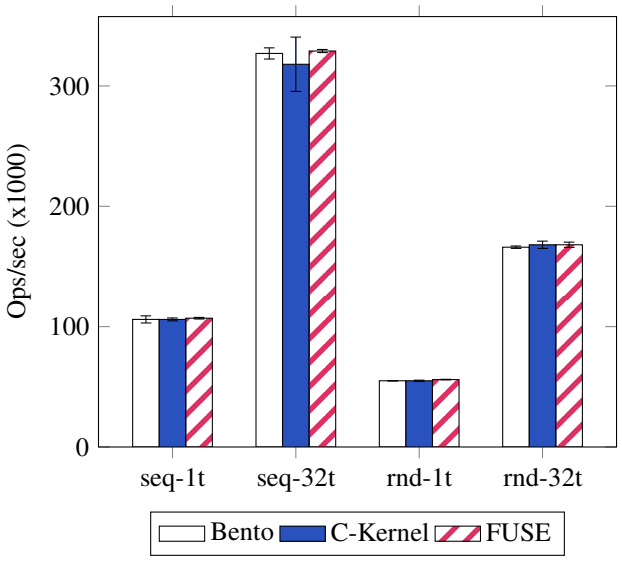


Figure 2: Read Performance (4KB), Ops/sec

synced every time one block needs to be synced, making fsyncs very costly.

6.5 Microbenchmarks

6.5.1 Read

The performance results for the read microbenchmarks are shown in Figure 2 and Figure 3. All figures include single-threaded and 32-threaded benchmarks for both sequential read and random read. Figure 2 shows performance for 4KB reads in operations per second. The other graphs show performance for 32KB, 128KB, and 1024 KB reads in throughput, measured in MBps.

All three versions of the file system show very similar performance results for all sizes of reads for both random and sequential reads. The similarity in performance is due to in-kernel caching and the small size of the file. All three versions of the file system use the same technique for caching read requests, implemented in the file system in the C-kernel version, in the file operations layer in Bento, and in FUSE kernel module for the FUSE version. Since the file is small and read requests are fast, the file is cached very quickly. After this, all requests hit the same in-kernel cache, and all versions execute the exact same code. The xv6 file system cannot support files larger than 4GB, so we cannot run a benchmark that evaluates the differences.

6.5.2 Write

The performance results for the write microbenchmarks are shown in Figure 4. The graphs include single-threaded sequential writes and single-threaded and 32-threaded random writes for 32KB, 128KB and 1024KB writes. Our evaluation does not include 4KB writes because these often triggered a segmentation fault in Filebench. Performance is measured in throughput in MBps. The performance of the FUSE file system was so low, these bars are nearly flush with the bottom of the graphs.

The Bento file system shows similar performance to the C version of the file system, and both perform much better than the FUSE version of the file system. The versions written in Rust using Bento and in C in the kernel implement nearly identical behavior, so it is expected for them to have similar performance. The Bento file system performs somewhat better than the VFS file system on large writes because Bento, which inherits from the FUSE kernel module, uses a more optimized technique for writing pages. Bento uses the `writepages` method instead of `writepage`, allowing sequential pages to be batched.

We see significant slowdown because of slow block I/O from userspace, even through the `O_DIRECT` file interface. Each block operation from userspace must pass across the user/kernel boundary and through the VFS layer before reaching the disk, adding 200-400ns to each operation. On top of that, the file interface imposes additional overheads. The file system must occasionally sync blocks to disk (such as during log operations), but the file interface provides no way to sync parts of a file, so the whole disk file must be synced every time one block needs to be synced, making `fsyncs` very costly.

	1 Thread	32 Threads
Bento	1126	1072
C-Kernel	933	881
FUSE	24	24

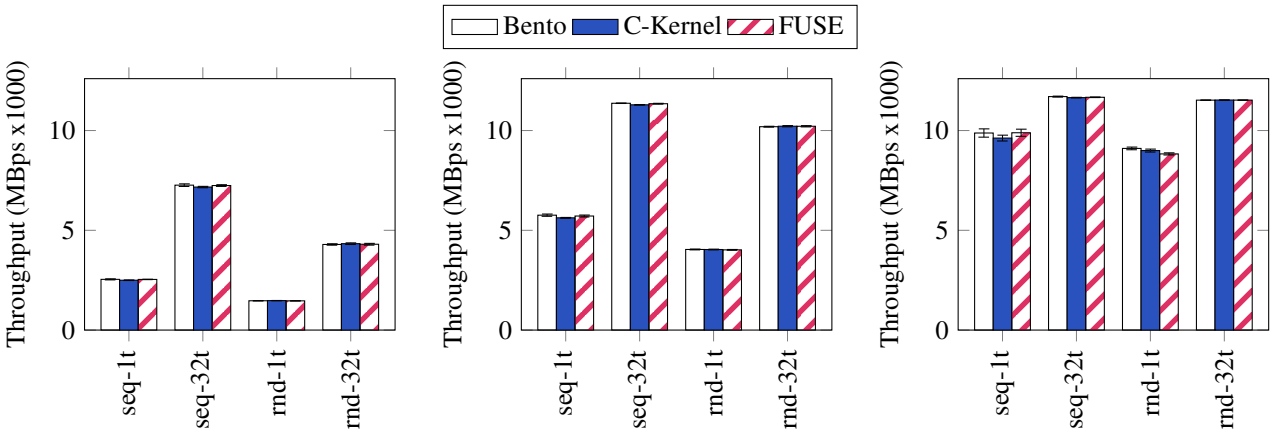
Table 4: Create Microbenchmark Performance (Ops/sec)

6.5.3 Create

Performance for the file creation microbenchmark on an SSD is shown in Table 4 for both single threaded and 32-threaded creates. In these benchmarks, Bento shows competitive performance to the C version in the kernel and much better performance than the FUSE version. File creation involves many small writes (and so syncs in the log), so the FUSE performance is heavily impacted by slow syncs. FUSE shows less slowdown for creates than it does for writes. This occurs because the create microbenchmarks spend a smaller percentage of the time executing slow disk operations.

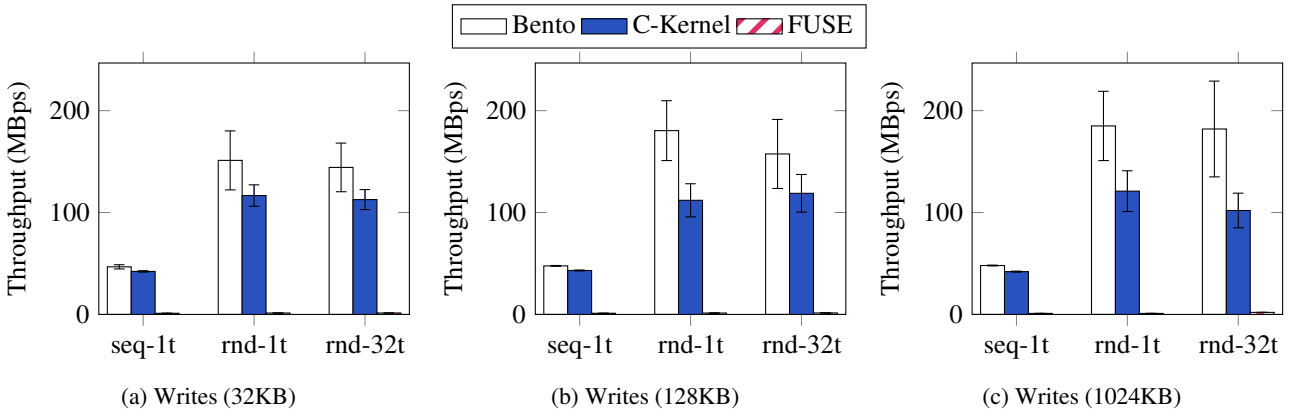
6.5.4 Delete

Performance results for the file deletion microbenchmark on an SSD are shown in Table 5 for both single-threaded and 32-threaded benchmarks. These results show similar trends as



(a) Reads (32KB) (b) Reads (128KB) (c) Reads (1024KB)

Figure 3: Read Performance (32KB-1024KB), Throughput MBps (x1000)



(a) Writes (32KB) (b) Writes (128KB) (c) Writes (1024KB)

Figure 4: Write Performance, Throughput (MBps)

	1 Thread	32 Threads
Bento	7499	7502
C-Kernel	7500	8253
FUSE	118	116

Table 5: Delete Microbenchmark Performance (Ops/sec)

	Varmail (ops/s)	Fileserver (ops/s)	Untar (s)
Bento	320	3860	19.8
C-Kernel	303	2947	31.6
FUSE	24	7	3404.9
Ext4	785	5172	6.2

Table 6: Macrobenchmark Performance

the file creation microbenchmarks because both are metadata heavy benchmarks so generate many small writes.

6.6 Macrobenchmarks

6.6.1 Varmail

Performance results for the varmail macrobenchmark on an SSD are shown in Table 6. The file system implemented using Bento and the C version in the kernel have very similar performance while the FUSE version shows much worse performance. Since this is a metadata-heavy macrobenchmark, the xv6 file system results are similar to the metadata-heavy microbenchmarks (file creation and deletion). The FUSE version performs comparatively better on the varmal benchmark than it does on the other benchmarks because varmail executes fsyncs on files. While fsyncs are slower for the FUSE version than they are for the other two xv6 file systems, the slowdown is not as large when whole files are being synced instead of individual blocks. On all three versions of the xv6 file system, the fsyncs take up the majority of the runtime, so the performance properties of the fsyncs are reflected in the overall performance numbers. For this benchmark, ext4 performs about 2.5x faster than either of the in-kernel xv6 implementations.

6.6.2 Fileserver

Performance results for the fileserver benchmark on the SSD are shown in Table 6. This benchmark involves many reads, writes, and file creates and deletes. Like the other benchmarks, these results show that the file system implemented using Bento and the version using the VFS layer in the kernel have very similar performance, and both outperform the FUSE version. This benchmark is particularly affected by the FUSE slowdowns because it involves many writes and creates, both of which introduce significant overhead. Ext4 only performs only 33% better than the xv6 file system written using Bento. At that point, ext4 appears to be bounded by the throughput of the SSD.

6.6.3 Untar Linux

This benchmark (shown in Table 6) untars the Linux kernel onto the relevant file system, generating many file creations and writes across many directories. Unlike the other benchmarks, this measures total execution time instead of operations per second, so lower is better. This benchmark shows somewhat more performance difference between the Bento file system and the VFS file system. This is likely caused by the same difference seen in the write microbenchmarks: Bento is able to batch sequential writes while the VFS implementation of xv6 is not.

6.7 Future Evaluation

As future work, we will demonstrate development velocity by implementing and evaluating real-world file systems using Bento. The simplicity of the xv6 file system was ideal for the proof-of-concept work, but a more full-featured file system will better demonstrate the velocity, generality, and low performance impact of Bento. We plan to focus on a research file system that shows promise for practical use, both proving that Bento can support a more complicated file system and providing a high-performance implementation of a file system that has demand in the community.

7 Related Work

7.1 FUSE

FUSE (Filesystem in Userspace) [14] is a framework that enables implementing a file system to run in userspace. The FUSE framework consists of two pieces, a kernel driver that translates VFS calls to FUSE-internal requests that are sent to userspace and libFUSE, a userspace library that interfaces with the user file system. Like Bento, FUSE targets safety and ease-of-development for Linux file systems. It's able to provide these properties quite well by running code in userspace and providing a simplified interface. However, running the file system in userspace introduces extra kernel crossings, leading to up to 83% overhead. This overhead is too severe for many applications, and production systems rarely employ FUSE. Bento takes advantage of the interface work done in FUSE, using the FUSE low-level interface and a modified version of the FUSE kernel driver.

7.2 eBPF

The extended Berkeley Packet Filter (eBPF) [23] is another technique for safe extensibility in Linux. It allows users to insert limited pieces of code to run in the Linux kernel at a set of predefined locations with restricted permissions. As

implied by the name, eBPF was originally aimed at network packet processing, but has since been expanded to support broader functionality. While the mainline eBPF code has no support for file systems, a project [5] has enabled writing parts of a stackable file system using eBPF. Broadly, eBPF provides a high level of safety and good performance but can't easily support the large modules with complex logic and data structures that Bento targets. eBPF programs are limited in size and type of operations. While eBPF programs can be chained together with tail calls, maintaining state across the tail calls is complicated at best.

7.3 Verification

Over the last few years, several papers have been published on verified operating systems and file systems [7, 24, 29]. These projects use formal verification to ensure that an implementation of a system abides by some defined correctness properties. By enforcing proofs of correctness properties, this technique is able to eliminate many bugs without necessarily adding performance overhead. However, verified file systems are still difficult to design and implement, requiring specialized knowledge. Additionally, there are currently no practical mechanisms to verifying concurrent code.

7.4 Software Fault Isolation

Software fault isolation is a technique for limiting the impact of faults in a module and has seen several implementations [6, 33], including one for Linux modules [20]. Using this technique, faults in a protected module are unable to impact the correctness of surrounding code. SFI can have significant performance overhead in many cases, ranging anywhere from $0\times$ to $4\times$ CPU overhead. This overhead manifests both while executing the isolated. module and while transitioning into and out of the module. Additionally, while SFI can address some of the safety concerns when developing Linux modules, it does not reduce the number of bugs in the module, only ensuring that bugs in the module will be isolated to the module.

7.5 High-level Languages

Other projects have also employed a high-level language with a strict type system to provide safety in the operating system. Like Bento, the SPIN operating system [4] provides safe extensibility by combining a safe, modular interface with modules implemented in a safe language, though SPIN designed the whole operating system around extensibility. Bento applies these techniques and the associated benefits to Linux. Other operating systems projects, such as Singularity [16], Biscuit [9], and Redox [27], explore writing the entire operating system in a high-level language. Additionally, we're not the first group to integrate Rust code into the Linux kernel. Other projects have implemented device drivers in Rust [18, 19].

8 Future Work

The work we have done on Bento so far is the beginning of a larger project. Over the next year, we plan to further work both on Bento for file systems and for other interfaces across the Linux kernel. Over the next six months, we plan to conclude our work on the file system extensibility layer and submit a paper on the topic. After that, we intend to apply the concepts in this paper to other interfaces across the Linux kernel, starting with networking.

8.1 Fully Supporting File Systems

On top of addressing the remaining challenges and implementing future work discussed thus far (online upgrades, debugging API, composable file systems, real-world evaluation), we will update to more recent kernel versions to take advantage of new functionality. Recent versions of the Linux kernel have included a new abstraction for performing asynchronous I/O from userspace called `io_uring`. Using this interface for the I/O accesses from the FUSE version of the xv6 file system in the evaluation could result in better performance numbers, potentially decreasing the overhead seen by using FUSE. More interestingly, our framework could hook into the file I/O part of `io_uring` along with the VFS layer, allowing users to entirely bypass the VFS layer. We cannot currently use `io_uring` because our framework is built in Linux kernel version 4.15 and `io_uring` requires version 5.1. Updating to the new kernel version and incorporating `io_uring` is part of our future work on this project.

8.2 Beyond File Systems

While this project focuses on file systems, none of the goals, challenges, or techniques are unique to file systems; other types of extensions can also benefit from this design. We plan to first focus on networking, particularly the TCP/IP functionality (or more broadly, OSI layers 3 and 4). Recent work has shown demand for specialized, userspace network stacks [21, 22] to improve performance for specific applications. Similarly, exokernel operating systems [12] and kernel-bypass networking [25] seek to improve performance by using optimized network stacks and/or avoiding the overhead of the Linux kernel network stack. However, kernel bypass networking has downsides, requiring the whole NIC to be given to the userspace process and burning CPU cores for polling. Applying the concepts from this projects to kernel networking could enable running specialized network stacks in the kernel, enabling the performance gains of running a specialized network stack without the downsides of kernel bypass networking.

In the long term, we intend to apply the concepts from this project to interfaces across the Linux kernel. Many interfaces could benefit from improved safety and increased development velocity. Past work has used Rust to write drivers [18], but these drivers still have a significant amount of unsafe code. Other pieces of the kernel, such as scheduling algorithms or security modules, could be targets for future work.

9 Conclusion

In this paper we present Bento, a framework to improve development velocity in operating systems, focusing on Linux kernel file systems for this work. We've identified several properties an extensibility framework must provide for high development velocity: safety, performance, generality, compatibility with existing operating systems, and other features for fast development velocity such as online upgrades and code reuse. By taking advantage of Rust, a modern type-safe, non-garbage-collected language, and enforcing restricted memory sharing between the file system and the kernel, safe abstractions around kernel services, and isolated file system modules, Bento is able to provide the first four of these properties for Linux kernel file systems, with the fifth coming soon. We've implemented the xv6 file system using Bento and shown that it has similar performance to a VFS kernel file system using the same design. As future work, we'll continue development on Bento, adding more features such as support for online upgrades, and we'll use Bento to implement more full-featured file systems. Code will be available once this additional work has been completed.

References

[1] 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 *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 353–369, New York, NY, USA, 2019. Association for Computing Machinery.

[2] Thomas E. Anderson, Marco Canini, Jongyul Kim, Dejan Kostic, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N. Schuh, and Emmett Witchel. Assise: Performance and availability via NVM colocation in a distributed file system. *CoRR*, abs/1910.05106, 2019.

[3] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, Broomfield, CO, October 2014. USENIX Association.

[4] 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.

[5] Ashish Bijlani and Umakishore Ramachandran. Extension framework for file systems in user space. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 121–134, USA, 2019. USENIX Association.

[6] 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.

[7] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using Crash Hoare Logic for Certifying the FSCQ File System. In *SOSP*, 2015.

[8] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 261–264, Hollywood, CA, 2012. USENIX Association.

[9] 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.

[10] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCabooter, Marc de Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, and Amin Vahdat. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 373–387, Renton, WA, April 2018. USENIX Association.

[11] Docker. https://www.docker.com/, 2018.

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

[13] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, Renton, WA, April 2018. USENIX Association.

[14] Filesystem in Userspace. https://github.com/libfuse/libfuse, 2018.

[15] Rust FUSE. https://github.com/zargony/fuse-rs.

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

[17] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas E. Anderson. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 460–477. ACM, 2017.

[18] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C.S. Lui. Securing the device drivers of your embedded systems: Framework and prototype. In *Proceedings of the 14th International Conference on Availability, Reliability and Security*, ARES '19, New York, NY, USA, 2019. Association for Computing Machinery.

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

[20] 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.

[21] Ilias Marinos, Robert N.M. Watson, and Mark Handley. Network stack specialization for performance. *SIGCOMM Comput. Commun. Rev.*, 44(4):175–186, August 2014.

[22] 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 *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 399–413, New York, NY, USA, 2019. Association for Computing Machinery.

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

[24] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. Hyperkernel: Push-Button Verification of an OS Kernel. In *SOSP*, 2017.

[25] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, Broomfield, CO, October 2014. USENIX Association.

[26] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J. Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Jonathan Stringer, Pravin Shelar, Keith Amidon, and Martín Casado. The Design and Implementation of Open vSwitch. NSDI, 2015.

[27] Redox. https://www.redox-os.org/, 2018.

[28] Chuck Rossi. Rapid release at massive scale.

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

[30] William Tu, Joe Stringer, Yifeng Sun, and Wei Yi-Hung. Bringing the Power of eBPF to Open vSwitch. In *Linux Plumbers Conference*, 2018.

[31] 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.

[32] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To fuse or not to fuse: Performance of user-space file systems. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, FAST'17, page 59–72, USA, 2017. USENIX Association.

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

[34] xv6 OS. https://github.com/mit-pdos/xv6-public.

[35] Gerd Zellweger, Simon Gerber, Kornilios Kourtis, and Timothy Roscoe. Decoupling cores, kernels, and operating systems. In Jason Flinn and Hank Levy, editors,

11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014, pages 17–31. USENIX Association, 2014.

[36] Irene Zhang, Jing Liu, Amanda Austin, Michael Lowell Roberts, and Anirudh Badam. I'm not dead yet!: The role of the operating system in a kernel-bypass era. In *Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS 2019, Bertinoro, Italy, May 13-15, 2019*, pages 73–80. ACM, 2019.