Design and Implementation of a Host-Independent File System for WebAssembly

2330025 Ayako Iwasaki

Introduction

WebAssembly (Wasm) emerged as a binary instruction format for stack-based virtual machines, initially designed to enable high-performance code execution within web browsers. However, its application scope has expanded beyond the browser environment. In cloud computing contexts, WebAssembly is gaining attention as a lightweight execution unit that could potentially replace traditional container technologies, particularly in serverless computing and edge computing scenarios. This expansion is driven by WebAssembly's inherent characteristics, namely robust sandboxing and portability across diverse platforms.

As WebAssembly adoption grows in these new domains, applications increasingly require access to file systems for handling persistent data, configuration files, and temporary files. The conventional approach of directly exposing the host operating system's file system to WebAssembly applications presents significant challenges. First, this approach compromises WebAssembly's sandboxing properties, creating potential security risks through configuration errors or runtime implementation flaws. Second, direct host dependency undermines, as applications become tied to specific directory structures and file path conventions that vary across execution environments.

This paper presents the design and implementation of a virtual file system that is accessible to WebAssembly applications while remaining independent of the host file system. We propose three distinct architectural approaches, namely static composition using the Component Model, sharing via host traits, and dynamic connection via Remote Procedure Call (RPC). Furthermore, we describe a synchronization mechanism that employs Amazon S3-compatible remote storage as a backend to ensure data persistence.

Background and Related Work

The WebAssembly System Interface (WASI) provides a standardized interface for WebAssembly applications to access system resources such as file systems and network connections. WASI implements a capability-based security model where applications can only access explicitly permitted directo-

ries (preopened directories) rather than the entire host file system. However, this approach still creates a direct dependency on the host environment.

Existing virtualization approaches have attempted to address these limitations. The wasi-vfs tool, originally developed for running interpreter languages like Ruby and Python on WebAssembly, hooks WASI Preview 1 functions and redirects file operations to a virtual file system embedded within the WebAssembly module. The wasi-virt tool leverages the Component Model to provide virtualization through component composition. However, both approaches are currently limited to read-only operations and cannot support file sharing among multiple WebAssembly applications or synchronization with external storage systems.

Proposed Architecture

Our research proposes three complementary approaches for implementing a host-independent file system, each suited to different use cases and requirements.

Static Composition Approach

The first approach utilizes the Component Model's composition capabilities to statically bundle the application component with a file system adapter component. The resulting single WebAssembly binary contains all necessary file system functionality without requiring any host-side implementation. This approach offers strong portability, as the generated binary can execute on any WASI Preview 2-compliant runtime without additional configuration.

The architecture consists of three main components. The first is the user application compiled to a WebAssembly component that imports wasi:filesystem interfaces. The second is the VFS Adapter component that implements and exports these interfaces. The third is fs-core, the core in-memory file system library implemented in Rust. The composition is performed using the wac plug command, which connects the application's imports to the adapter's exports.

Host Trait Implementation Approach

The second approach moves the file system implementation to the host side, enabling multiple WebAssembly instances within the same process to share a common file system. The host program implements the WASI filesystem

interfaces as native code and provides these implementations when instantiating WebAssembly components.

This design employs a two-layer fine-grained locking mechanism to ensure thread safety while maintaining high concurrency. The first layer uses DashMap, a concurrent hash map, for managing inode and file descriptor tables. The second layer protects individual inodes with RwLock, allowing multiple concurrent readers while ensuring exclusive access for writers. This architecture enables parallel operations on different files while providing proper synchronization for accesses to the same file.

RPC Dynamic Connection Approach

The third approach separates the file system into an independent server process that accepts requests over TCP using Protocol Buffers serialization. This architecture enables file sharing across process boundaries and across different machines in a distributed system. Applications can dynamically connect to an existing file system server at runtime, making this approach suitable for scenarios where applications are added or scaled dynamically, such as in Kubernetes environments or CI/CD pipelines.

Remote Storage Synchronization

To address data persistence requirements, we implemented a synchronization mechanism with Amazon S3-compatible object storage. The implementation supports multiple synchronization strategies to balance consistency requirements with performance needs.

For read operations, we provide two modes. Read-through mode fetches data from S3 on every access to ensure freshness. In-memory cache mode loads data at startup for improved performance. For write operations, write-through mode synchronously uploads data to S3 on every write operation, while asynchronous mode queues write operations for batch processing. Additionally, the system supports cold-start restoration from S3 and ETag-based polling for detecting external modifications.

Evaluation

Performance evaluation was conducted across all three approaches. The static composition approach achieved throughput of approximately 2,100 to 2,200 MB/s for sequential reads and 1,200 to 1,500 MB/s for sequential

writes, demonstrating practical performance comparable to physical disk-based file systems such as ext4.

Comparison with the existing wasi-virt implementation showed equivalent read performance, validating that our approach maintains competitive speed while adding write support and persistence capabilities that wasi-virt lacks.

Among the three proposed approaches, the host trait implementation demonstrated the highest performance, achieving over 13,000 MB/s for small file sequential reads due to native code execution. The RPC approach showed significant latency overhead, particularly for random access patterns, but remained viable for scenarios prioritizing flexibility over raw performance.

For S3 synchronization scenarios, the host trait implementation demonstrated performance comparable to or better than the traditional s3fs-fuse approach. This improvement stems from avoiding the kernel-userspace round trips inherent in FUSE-based implementations.

Concurrency evaluation confirmed that our two-layer locking mechanism scales effectively with increasing thread counts. Read operations on the same file showed linear scaling due to shared lock semantics, while write operations on different files maintained parallel throughput through the lock-free concurrent hash map at the table level.

Conclusion

This paper presented a solution for providing file system functionality to WebAssembly applications while maintaining the sandboxing properties and portability that make WebAssembly attractive for serverless and edge computing scenarios. The three proposed approaches offer different trade-offs between performance, portability, and data sharing capabilities, allowing developers to select the most appropriate architecture for their specific requirements.

The static composition approach maximizes portability with a single self-contained binary. The host trait implementation provides the highest performance and enables sharing within a process. The RPC approach enables sharing across process and machine boundaries at the cost of network latency. Combined with S3 synchronization capabilities, these approaches provide a flexible foundation for building WebAssembly applications that require persistent storage without compromising the security and portability benefits of the WebAssembly platform.

Future work includes implementing direct synchronization with remote storage for handling files larger than available memory, optimizing RPC communication through caching and prefetching, and developing distributed

locking mechanisms for multi-node write coordination.