JAIST Repository
https://dspace.jaist.ac.jp/

Title	WebAssembly 向けホスト非依存ファイルシステムの設計と実装 [課題研究報告書]
Author(s)	早坂, 絢子
Citation	
Issue Date	2026-03
Type	Thesis or Dissertation
Text version	author
URL	https://hdl.handle.net/10119/20521
Rights	
Description	Supervisor:宇多　仁, 先端科学技術研究科, 修士(情報科学)

JAPAN ADVANCED INSTITUTE OF SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

課題研究報告書

WebAssembly 向けホスト非依存ファイルシステムの設計と実装

岩崎 絢子
学生番号 2330025

指導教員　宇多 仁

北陸先端科学技術大学院大学

2026 年 1 月

Abstract

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 directories (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.

概要

はじめに

WebAssembly（Wasm）は，スタックベースの仮想マシン向けのバイナリ命令形式として登場し，当初は Web ブラウザ内での高性能なコード実行を目的として設計された．しかし，その適用範囲はブラウザ環境を超えて拡大している．クラウドコンピューティングの分野では，WebAssembly は従来のコンテナ技術を置き換える可能性を持つ軽量な実行単位として注目されており，特にサーバレスコンピューティングやエッジコンピューティングにおいて関心が高まっている．この普及を支えているのは，WebAssembly が備えるサンドボックス機構と，プラットフォーム間での移植性である．

これらの新しい領域で WebAssembly の採用が進むにつれて，アプリケーションは永続データや設定ファイル，一時ファイルを扱うためにファイルシステムへのアクセスを必要とするようになっている．ホストオペレーティングシステムのファイルシステムを WebAssembly アプリケーションに直接公開する従来の手法には課題がある．第一に，この手法は WebAssembly のサンドボックス特性を損ない，設定ミスやランタイム実装の欠陥を通じてセキュリティ上のリスクを招く恐れがある．第二に，ホストへの直接的な依存は，WebAssembly の長所である移植性を損なう．これは，アプリケーションが実行環境固有のディレクトリ構造やパス区切り文字といった，ホストの環境差異を吸収するための複雑なロジックを内包しなければならなくなるからである．

本論文では，ホストファイルシステムから独立しつつ，WebAssembly アプリケーションからアクセス可能な仮想ファイルシステムの設計と実装を提示する．本研究では，3 つの異なるアーキテクチャを提案する．Component Model を用いた静的合成，ホストトレイトを介した共有，および Remote Procedure Call（RPC）を介した動的接続である．さらに，データの永続性を確保するために，Amazon S3 互換のリモートストレージをバックエンドとして採用する同期機構について述べる．

背景と関連研究

WebAssembly System Interface（WASI）は，WebAssembly アプリケーションがファイルシステムやネットワーク接続などのシステムリソースにアクセスするための標準インタフェースを提供する．WASI はケーパビリティベースのセキュリティモデルを採用して

おり，アプリケーションはホストファイルシステム全体ではなく，明示的に許可されたディレクトリ（プリオープンディレクトリ）にのみアクセスできる．しかし，この手法でもホスト環境への依存は避けられない．

既存の仮想化手法はこれらの制限への対処を試みてきた．wasi-vfs ツールは，もともと Ruby や Python などのインタプリタ言語を WebAssembly 上で実行するために開発されたもので，WASI Preview 1 の関数をフックし，ファイル操作を WebAssembly モジュール内に埋め込まれた仮想ファイルシステムへリダイレクトする．wasi-virt ツールは，Component Model を活用してコンポーネント合成による仮想化を実現している．しかし，いずれの手法も現時点では読み取り専用操作に限定されており，複数の WebAssembly アプリケーション間でのファイル共有や外部ストレージシステムとの同期には対応できない．

提案アーキテクチャ

本研究では，ホストファイルシステムに依存しない，WebAssembly アプリケーションから利用できるファイルシステムを実現するための 3 つの手法を提案する．それぞれが異なるユースケースと要件に適している．

静的合成アプローチ

第 1 の手法は，Component Model の合成機能を利用して，アプリケーションコンポーネントとファイルシステムアダプタコンポーネントを静的にバンドルするものである．生成された単一の WebAssembly バイナリには必要なファイルシステム機能が含まれ，ホスト側の実装を必要としない．この手法は高い移植性を実現し，生成されたバイナリは追加の設定なしに WASI Preview 2 準拠の任意のランタイムで実行できる．

アーキテクチャは 3 つの主要コンポーネントで構成される．第 1 は，wasi:filesystem インタフェースをインポートする WebAssembly コンポーネントとしてコンパイルされたユーザアプリケーションである．第 2 は，これらのインタフェースを実装しエクスポートする VFS アダプタコンポーネントである．第 3 は，Rust で実装されたインメモリファイルシステムのコアライブラリである fs-core である．合成は wac plug コマンドによって行われ，アプリケーションのインポートとアダプタのエクスポートが接続される．

ホストトレイト実装アプローチ

第 2 の手法は，ファイルシステムの実装をホスト側に配置し，同一プロセス内の複数のWebAssembly インスタンスが共通のファイルシステムを共有できるようにするものである．ホストプログラムは WASI filesystem インタフェースをネイティブコードとして実装し，WebAssembly コンポーネントのインスタンス化時にこれらの実装を提供する．

この設計では，スレッドセーフティを確保しながら高い並行性を維持するために，2 層の細粒度ロック機構を採用している．第 1 層では，inode およびファイルディスクリプタテーブルの管理に並行ハッシュマップである DashMap を使用する．第 2 層では，個々のinode を RwLock で保護し，複数の並行読み取りを許可しつつ書き込みには排他的アクセスを保証する．このアーキテクチャにより，異なるファイルへの並列操作が可能となり，同一ファイルへのアクセスには適切な同期が提供される．

RPC 動的接続アプローチ

第 3 の手法は，ファイルシステムを独立したサーバプロセスとして分離し，Protocol Buffers によるシリアライゼーションを用いて TCP 経由でリクエストを受け付けるものである．このアーキテクチャにより，プロセス境界を越えたファイル共有や，分散システム内の異なるマシン間でのファイル共有が可能となる．アプリケーションは実行時に既存のファイルシステムサーバへ動的に接続でき，Kubernetes 環境や CI/CD パイプラインなど，アプリケーションが動的に追加・スケールされる状況に適している．

リモートストレージ同期

データの永続性要件に対応するため，Amazon S3 互換のオブジェクトストレージとの同期機構を実装した．本実装では，一貫性要件とパフォーマンスのバランスを考慮し，複数の同期戦略をサポートしている．

読み取り操作については 2 つのモードを提供する．リードスルーモードはアクセスごとに S3 からデータを取得し，常に最新のデータを参照できる．インメモリキャッシュモードはメモリからデータを読み込み，読み取り性能を向上させる．書き込み操作については，ライトスルーモードが書き込みごとに S3 へ同期的にアップロードを行い，非同期モードは書き込み操作をキューに蓄積してバッチ処理する．さらに，起動時の S3 からの復元と，外部からの変更を検出するための ETag ベースのポーリングにも対応している．

評価

3 つの手法すべてについて性能評価を実施した．静的合成アプローチは，シーケンシャル読み取りで約 2,100〜2,200 MB/s，シーケンシャル書き込みで 1,200〜1,500 MB/s のスループットを達成し，ext4 などの物理ディスクベースのファイルシステムと同等の実用的な性能を示した．

既存の wasi-virt 実装との比較では，読み取り性能に遜色がないことを確認した．これにより，本手法が wasi-virt にはない書き込みサポートと永続化機能を追加しつつ，同等の速度を維持していることが検証された．

提案した 3 つの手法の中では，ホストトレイト実装が最も高い性能を示した．ネイティブコードによる実行により，小ファイルのシーケンシャル読み取りでは 13,000 MB/s を超える性能を達成した．RPC アプローチは，特にランダムアクセスパターンにおいて顕著なレイテンシオーバーヘッドが見られた．

S3 同期を伴うシナリオでは，ホストトレイト実装が従来の s3fs-fuse と比較して同等以上の性能を示した．この改善は，FUSE ベースの実装で避けられないカーネル・ユーザ空間間の往復処理を回避できることに起因する．

並行性評価では，本研究の 2 層ロック機構がスレッド数の増加に対して効果的にスケールすることを確認した．同一ファイルに対する読み取り操作は，共有ロックの特性により線形にスケールし，異なるファイルに対する書き込み操作は，テーブルレベルでのロックフリー並行ハッシュマップにより並列スループットを維持した．

おわりに

本論文では，サーバレスおよびエッジコンピューティングにおいて WebAssembly の利点とされるサンドボックス特性と移植性を維持しながら，WebAssembly アプリケーションにファイルシステム機能を提供する手法を提示した．提案した 3 つの手法は，性能，移植性，データ共有機能の間で異なるトレードオフを持ち，開発者は要件に応じて最適なアーキテクチャを選択できる．

静的合成アプローチは，自己完結型の単一バイナリにより移植性を最大化する．ホストトレイト実装は，最高の性能を発揮し，プロセス内での共有を可能にする．RPC アプローチは，ネットワークレイテンシを許容する代わりに，プロセスおよびマシン境界を越えた共有を実現する．S3 同期機能と組み合わせることで，これらの手法は，WebAssembly プ

ラットフォームのセキュリティと移植性を損なうことなく，永続ストレージを必要とするWebAssembly アプリケーションを構築するための柔軟な基盤となる．

今後の課題として，利用可能なメモリを超えるサイズのファイルを扱うためのリモートストレージとの直接同期や，キャッシングとプリフェッチによる RPC 通信の最適化，およびマルチノード書き込み協調のための分散ロック機構の開発が挙げられる．

目次

第 1 章　序論 1

第 2 章　WebAssembly の概要 3

2.1 WebAssembly とは 3

2.2 ブラウザ外への適用拡大とサーバレス・エッジコンピューティング . . . 4

第 3 章　WebAssembly アプリケーションからのファイルシステムアクセス 7

3.1 ファイルシステムアクセスにおける課題 7

3.2 WASI の概要とバイナリ構造 8

3.3 WASI Preview 1 におけるファイルシステムアクセス 9

3.4 WASI Preview 2 と Component Model 9

第 4 章　ホスト非依存なファイルシステムの先行研究とその課題 12

4.1 wasi-vfs と Wizer による仮想化 12

4.2 wasi-virt によるコンポーネントベースの仮想化 13

4.3 既存手法の課題整理と本研究の目的 15

第 5 章　ホスト非依存な仮想ファイルシステムの設計と実装 16

5.1 Component Model における仮想ファイルシステム実装戦略 16

5.2 静的合成アプローチのシステム構成 17

5.3 fs-core の設計と実装 18

5.4 vfs-adapter と wac plug による合成 19

5.5 検証と課題 20

第 6 章　複数アプリケーション間におけるファイルシステム共有の実現 24

6.1 単一コンポーネント内完結の限界 24

6.2　ホスト側でのインターフェース実装 24
6.3　検証と課題 26

第 7 章　RPC を用いたアプリケーションの動的追加の実現　32
7.1　プロセス境界を超えた共有の必要性 32
7.2　RPC を用いたファイルシステムアクセス時のアーキテクチャ 32
7.3　通信プロトコルと実装の詳細 34
7.4　検証と課題 35

第 8 章　リモートストレージとの同期機能の実装　37
8.1　永続性の確保と S3 同期 37
8.2　同期メカニズム 37
8.3　検証と課題 39

第 9 章　評価　42
9.1　静的合成アプローチの性能試験 42
9.2　静的合成アプローチと既存実装の性能試験 44
9.3　静的合成アプローチ・ホストトレイトアプローチ・RPC ベースアプローチの性能評価 45
9.4　リモートストレージ同期機能の評価 46
9.5　排他制御に伴うオーバーヘッド 48

第 10 章　考察とまとめ　50
10.1　各手法の特性比較と選定指針 50
10.2　実装上の制限事項と今後の展望 52
10.3　結論 53

第 11 章　謝辞　54

付録 A　fs-core 詳細設計仕様　55
A.1　データ構造定義 55
A.2　クラス構造図 59
A.3　インターフェース仕様 59
A.4　処理シーケンス詳細 62

参考文献 64

図目次

3.1 WASI Preview 1 におけるファイルシステムアクセスの概観 10
3.2 WASI Preview 2 におけるファイルシステムアクセスの概観 11

4.1 wasi-vfs 概観 13
4.2 wasi-virt 概観 14

5.1 静的リンク時の概観 17
5.2 ファイルシステム操作確認時のスクリーンショット 21
5.3 ファイルシステム同梱確認時のスクリーンショット 22

6.1 ホストトレイト実装アプローチの概観 26
6.2 ホストトレイト実装アプローチの動作確認 28
6.3 同時書き込み時の挙動確認 29
6.4 HTTP キャッシュサーバ実装で，キャッシュにヒットしていることの確認 31

7.1 RPC 動的追加アプローチの概観 33
7.2 RPC を経由して別プロセスから同じファイルシステムが操作できることを確認 36

8.1 仮想ファイルシステムから S3 への同期確認 39
8.2 S3 から仮想ファイルシステムへの同期確認 40
8.3 再起動時に S3 から復元しているログ 40

A.1 fs-core クラス構成図 (UML Class Diagram, thread-safe feature 有効時) 59
A.2 書き込み処理のシーケンス図 63

表目次

2.1 WebAssembly モジュールの主要なセクション構成 4

9.1 実験環境 42
9.2 静的合成アプローチの性能試験結果 (括弧内はスループット) 43
9.3 提案手法と wasi-virt の比較結果 (括弧内はスループット) 44
9.4 3 つの提案手法における性能比較（括弧内はスループット MB/s） . . . 45
9.5 S3 同期における各手法の性能比較（実行時間 [ms]） 47
9.6 スレッド並列度とアクセス対象による性能比較 [ops/sec] 48

10.1 提案する 3 つのアプローチの特性比較 50

A.1 型エイリアス定義 55
A.2 オープンフラグ定義 56
A.3 Fs 構造体定義 56
A.4 Inode 構造体定義 57
A.5 Metadata 構造体定義 57
A.6 FileHandle 構造体定義 58
A.7 BlockStorage 構造体定義 58
A.8 主要メソッド仕様一覧 60

第 1 章

序論

WebAssembly (Wasm) は，スタックベースの仮想マシン用のバイナリ命令フォーマットとして登場して以来，ブラウザ内での高速なコード実行を可能にしてきた [1]．しかし現在，その適用範囲はブラウザの外へと広がっている．特にクラウドコンピューティングの分野では，従来のコンテナ技術に代わる軽量な実行単位として，あるいはエッジデバイス上での効率的なアプリケーション実行環境として，安全性，高速性，ポータビリティを兼ね備えた WebAssembly への期待が高まっている [2, 3]．

WebAssembly の利用拡大に伴い，アプリケーションが永続化データや設定ファイル，あるいは一時ファイルなどを扱うためにファイルシステムへアクセスする必要が生じている．ブラウザ外で動作する Wasm アプリケーションからファイルシステムへアクセスする手段として，オペレーティングシステムが提供するファイルシステムを直接利用する仕組みが提供されている．しかしこれは，隔離された環境であるはずの Wasm からホスト環境への直接的なアクセス経路を開くことになり，設定ミスやランタイムの実装不備によるセキュリティリスクを招く要因となる [4, 5]．

また，ホスト環境への直接的な依存は，セキュリティだけでなく，WebAssembly の利点であるポータビリティをも損なう．実行環境ごとに異なるディレクトリ構造やファイルパスの差異は，アプリケーションの移植性を低下させる [6]．

したがって，WebAssembly のセキュリティとポータビリティを維持したまま，アプリケーションに対してファイルシステム機能を提供するための新たなアーキテクチャが必要とされている．本稿では，WebAssembly アプリケーションから利用できる，ホスト環境のファイルシステムに依存しないファイルシステム機構の設計と実装について詳述する．

本稿の構成は以下の通りである．第 2 章では，WebAssembly の概要，および特にサーバやエッジ環境に着目してユースケースについて解説する．第 3 章では，WebAssembly ア

プリケーションからのファイルシステムアクセスに関する課題と，WebAssembly System Interface (WASI) の技術的背景，WASI Preview 1 および Preview 2 の具体的なメカニズムについて述べる．第 4 章では，既存の仮想ファイルシステムに関する先行研究とその課題について整理する．続く第 5 章から第 7 章にかけて，本研究で提案する 3 つの実装手法について解説する．第 5 章ではアプリケーションとファイルシステムを同梱する形式，第 6 章ではファイルシステムを複数アプリケーションから共有する形式，第 7 章では RPC を用いてアプリケーションを動的に追加する形式について論じる．さらに第 8 章では，リモートストレージとの同期機能について触れる．第 9 章で性能評価を実施したのち，第 10 章で考察を行い，本稿のまとめとする．

第 2 章

WebAssembly の概要

2.1 WebAssembly とは

WebAssembly (Wasm) は，スタックベースの仮想マシンで動作するバイナリ命令フォーマットであり，2017 年に主要ブラウザベンダーの協力の下，ウェブにおける低レベルコードの標準として提案された [1]．WebAssembly は C や C++，Rust などの言語からのコンパイルターゲットとして設計されており，ネイティブに近い実行速度と安全な実行環境を提供する．現在では Chrome，Firefox，Safari，Edge といった主要なウェブブラウザすべてが WebAssembly の実行をサポートしている [7].

WebAssembly バイナリファイルのセクション構成を表 2.1 に示す [8]．WebAssembly のセキュリティ設計の中核は，ホスト環境から隔離されたサンドボックス構造にある．WebAssembly のメモリ空間は「線形メモリ（Linear Memory）」と呼ばれる，ホストのメモリ空間とは論理的に隔離された，連続したバイト配列として抽象化されている．Wasm プログラムはこの線形メモリの範囲内でのみ読み書きが許可されており，ホストプロセスのメモリ領域や，OS が管理するカーネル領域へ直接アクセスすることは構造的に不可能である．これにより，仮に Wasm モジュール内でバッファオーバーフローなどのメモリ破壊が発生したとしても，その影響はサンドボックス内に封じ込められ，ホスト環境のメモリを侵害しない [9]．加えて，コード領域とデータ領域は厳格に分離されており，実行中のコードが自身の命令を書き換えることはできない構造となっているため，コードインジェクション攻撃に対しても堅牢性を有している [10].

表 2.1: WebAssembly モジュールの主要なセクション構成

セクション名	説明
version	モジュールのバージョン情報を示す．
Type Section	モジュール内で使用される関数のシグネチャ（引数と戻り値の型）を定義する．
Import Section	モジュール外部から提供される関数，メモリ，グローバル変数などを定義する．
Function Section	Type Section で定義されたシグネチャと Code Section の実装を結びつける．
Code Section	実際の関数の命令列（バイトコード）が格納される．
Export Section	モジュール内部で定義された関数やメモリを外部（ホストや他のモジュール）に公開する．
Data Section	線形メモリの初期化データを定義する．

2.2 ブラウザ外への適用拡大とサーバレス・エッジコンピューティング

WebAssembly はブラウザ向け技術として出発したが，その特性は Web 以外のドメインにおいても有用である．現在では IoT [11]，ブロックチェーン [12]，サーバレスコンピューティング [3]，エッジコンピューティング [2, 13] など，多岐にわたる分野での利用が進んでいる．

2.2.1 サンドボックス特性と軽量性を活用したユースケース

従来のクラウドコンピューティングや FaaS（Function as a Service）基盤では，アプリケーションの実行基盤として Docker コンテナや軽量 VM が用いられてきた．これらは高い隔離性を提供する一方で，プロセスの起動や初期化に伴うオーバーヘッドが避けられない．また，1 つのサーバ上で数千から数万の顧客関数を同時に稼働させる場合，各コンテナが独自の OS イメージやライブラリを持つため，メモリフットプリントが肥大化し，集積率の向上が困難となる [14]．

対して WebAssembly は，OS のプロセスや VM を新たに立ち上げるのではなく，単一

のランタイムプロセス内で論理的にメモリ空間を分離する．これは「ナノプロセス」モデルとして提唱された [15] 軽量な実行単位であり，以下の利点をもたらす．

- **起動速度**: 一般的なコンテナが数秒から数百ミリ秒を要するのに対し，WebAssembly モジュールはマイクロ秒オーダーでの起動が可能である [3].
- **高集積率**: メモリオーバーヘッドが小さいため，同じハードウェアリソースでコンテナの数倍から数十倍のインスタンスを稼働させることができる [16].
- **セキュリティ**: サンドボックス化されたメモリ空間により，モジュール間の干渉を防ぎつつ，マルチテナント環境での安全性を確保できる [17].

Fastly の「Compute」プラットフォームでは，リクエストごとの起動時間を極小化するために WebAssembly を採用している [18]．Fermyon は，常時起動しているコンテナをオンデマンドな WebAssembly 関数に置き換えることで，アイドル時のコストを削減する「Scale to Zero」の実践を提唱している [19] ほか，WebAssembly 向けのフレームワーク Spin を開発している．Spin はサーバレスアプリケーションやマイクロサービスの構築に特化したフレームワークである．Spin 自体は HTTP サーバ機能を持つ常駐プロセス（Host Process）として動作する．HTTP リクエストを受信すると，Spin はそのプロセス内部のメモリ空間に，対象となる Wasm コンポーネントの新たなインスタンスを生成する．このインスタンス化は数ミリ秒からマイクロ秒オーダーで完了する [20]．処理が完了しレスポンスを返却すると，インスタンスは即座に破棄される．これにより，単一のホストプロセス上で，数千の同時リクエストをそれぞれ隔離されたメモリ空間で高速に並行処理することが可能になる [21].

2.2.2 ブラウザ外での実行方法

CLI ツールによるスタンドアロン実行

最も基本的な実行方法は，WebAssembly ランタイムが提供するコマンドラインインターフェース（CLI）を利用するものである．代表的なランタイムである Wasmtime [22] や WAMR (WebAssembly Micro Runtime) [23] は，`wasmtime run app.wasm` のようなコマンドを提供しており，ユーザはコンパイル済みの Wasm バイナリをシェルから直接起動できる．このモデルでは，ランタイム自体がホスト OS のプロセスとして起動し，Wasm モジュールをインスタンス化して実行する．

WebAssembly は Kubernetes などのコンテナオーケストレーション環境においても実行できる．近年普及しつつある runwasi [24] は，ユーザがコマンドを実行する代わりに，

コンテナランタイム（containerd）がバックグラウンドで Wasm ランタイムを呼び出す仕組みである．

ホストアプリケーションへのライブラリ埋め込み

WebAssembly ランタイムの実装をライブラリとして利用し，独自のホストアプリケーションの一部として Wasm モジュールを実行することもできる．これは「Embedded Wasm」とも呼ばれる．例えば, Go 言語で記述されたアプリケーションであれば `wazero`, Rust 言語であれば `wasmtime` クレート（ライブラリ）を利用することで，プログラムコード内で Wasm ランタイムのインスタンスを生成できる．

このモデルの最大の特徴は，ホスト側（Go や Rust のコード）とゲスト側（Wasm）が連携できる点である．これにより，ホスト側のコードで定義された関数の呼び出しや，ホスト側のメモリデータの共有といった制御が可能となる．

埋め込みモデルの応用例として，先述の Fermyon Spin が挙げられる．また，既存のミドルウェアやアプリケーションに対するプラグイン機構にも用いられている．Envoy Proxy [25] や Redpanda [26]，SingleStore [27] といったシステムでは，ユーザが独自のロジック（フィルタリング，認証，データ変換など）を組み込むための手段として WebAssembly を採用している．

第 3 章

WebAssembly アプリケーションからのファイルシステムアクセス

3.1　ファイルシステムアクセスにおける課題

WebAssembly の利用拡大に伴い，アプリケーションが永続化データや設定ファイル，あるいは一時ファイルなどを扱うためにファイルシステムへアクセスする必要が生じている．ブラウザ外で動作する Wasm アプリケーションからファイルシステムへアクセスする手段として，オペレーティングシステムが提供するファイルシステムを直接利用する仕組みが提供されている．具体的には，ホスト側（ランタイム）が起動時に特定のディレクトリをオープンし，そのファイルディスクリプタをアプリケーションに渡すことでアクセスを許可する preopen という仕組みが存在する [14].

しかし，Wasm アプリケーションからホストのファイルシステムへのアクセス権限を付与することは，サンドボックスによる隔離性を意図的に弱体化させる行為に等しく，この例外的なアクセス経路は，厳密に管理されなければ重大なリスクとなる．例えば，ランタイム側でのパス解決ロジックにおいて，親ディレクトリへの移動やシンボリックリンクの扱いが不適切であった場合，WebAssembly アプリケーションがサンドボックスの制限を回避し，ホストの `/etc` などにアクセス可能となるリスクが存在する [5]．実際に，Wasm ランタイムの 1 つである Wasmer の脆弱性 CVE-2023-51661 では，サンドボックスのファイルシステム制限が正しく適用されず，Wasm コードがホストのファイルシステムに不正アクセスできてしまう問題が報告されている [4].

また，ホストのファイルシステムへの直接依存は，セキュリティだけでなく，WebAssembly の利点であるポータビリティをも損なう．例えば Spin においては，Windows

環境でのパス区切り文字の違いといった環境依存のトラブルが報告されている [6]．開発者のローカル環境，クラウド上のサーバ，工場内のエッジデバイスなど，実行環境ごとにディレクトリ構造やファイルパスが異なれば，アプリケーションはその差異を吸収するための複雑なロジックを内包しなければならず，WebAssembly の本来の特性であるポータビリティを維持することが困難となる．

したがって，WebAssembly 本来の隔離性を損なうことなくファイル操作機能を提供するためには，ホスト OS のファイルシステムから論理的に分離された新たなアーキテクチャが必要となる．具体的解決策の提示に先立ち，次節以降では，WebAssembly アプリケーションからのファイルシステムアクセスの現状と，その基礎となる技術仕様について整理する．

3.2 WASI の概要とバイナリ構造

WebAssembly 自体は計算を行うための命令セットであり，それ単体ではファイルシステムやネットワークといった外部リソースにアクセスする手段を持たない．ブラウザ外で WebAssembly を実行する場合，これらのホスト機能を呼び出すための統一的なシステムインターフェースが必要となる．そのために策定されたのが WebAssembly System Interface (WASI) である [28]．

WASI は, WebAssembly ランタイムとその基盤となるオペレーティングシステム（OS）との間の中間層として機能する標準インターフェースである．ホスト OS の機能を利用する場合，OS ごとのシステムコールの違い（Linux，Windows，macOS など）を直接 WebAssembly バイナリに記述してしまうと，ポータビリティが損なわれる．WASI はこの問題を解決するため，抽象化されたシステムコール群を定義している [7]．

WebAssembly モジュールのバイナリ構造において，WASI の関数呼び出しは通常の外部関数呼び出しと同様に扱われる．具体的には，モジュールの Import Section に WASI の関数名（例: `fd_read`, `path_open`）が記述され，実行時にランタイムがこれらのインポート定義に対して，ホスト側の実装をリンクする仕組みとなっている．これにより，WebAssembly アプリケーションはコンパイル時に特定の OS を意識することなく, WASI という統一されたインターフェースに対してコードを記述できる [28]．

3.3　WASI Preview 1 におけるファイルシステムアクセス

現在広く利用されている WASI のバージョンは Preview 1 と呼ばれ，wasi_snapshot_preview1 というモジュール名で定義されている [29].

WASI を利用する WebAssembly アプリケーションの開発において，ファイルシステムへのアクセス方法は言語によって異なる．C/C++ で記述されたプログラムの場合，ターゲットとして wasm32-wasi を指定してビルドすると，コンパイラは wasi-libc [30] をリンクする．wasi-libc は C 言語の標準ライブラリ libc の WASI 向け実装であり，fopen や fread といった標準関数の呼び出しを WASI のシステムコールに変換する．Python や Ruby などのインタプリタ言語を WebAssembly で動作させる場合も，それらのランタイムが C 言語で実装されているため，同様に wasi-libc を経由する．

一方，Rust の場合は wasi-libc を使用しない．Rust の標準ライブラリは WASI ターゲット向けに独自の実装を持っており，std::fs などの API は直接 WASI のシステムコールを呼び出す.

いずれの場合も，最終的には wasi_snapshot_preview1.path_open などの WASI 関数が呼び出される (図 3.1)．この呼び出しに対応する実装は Wasm モジュール内には存在しないため，外部からの依存として import section に記載される．

Wasmtime などの WebAssembly ランタイムは，Wasm モジュールをロードしてインスタンス化する際，import section に記載された path_open をホスト側（ランタイム側）に存在する wasi_snapshot_preview1.path_open 実装に解決する．この実装は，呼び出しをホスト OS の実際のファイルシステム操作へと変換して実行する．この際，セキュリティを確保するために「Capability-based Security」モデルが採用されている．WebAssembly アプリケーションは，ホストのファイルシステム全体にアクセスできるわけではなく，実行時に明示的に許可されたディレクトリ（Preopened Directory）に対してのみアクセスが可能である．ランタイムは内部的に，WebAssembly 側のファイルディスクリプタとホスト側のファイルディスクリプタの変換テーブルを保持し，許可された範囲内での操作であることを検証した上で，ホストのシステムコールを発行する [31].

3.4　WASI Preview 2 と Component Model

2024 年に正式発表された WASI Preview 2 では，アーキテクチャが根本的に刷新され,「Component Model」という新たな概念が導入された．Preview 1 がモノリシックな

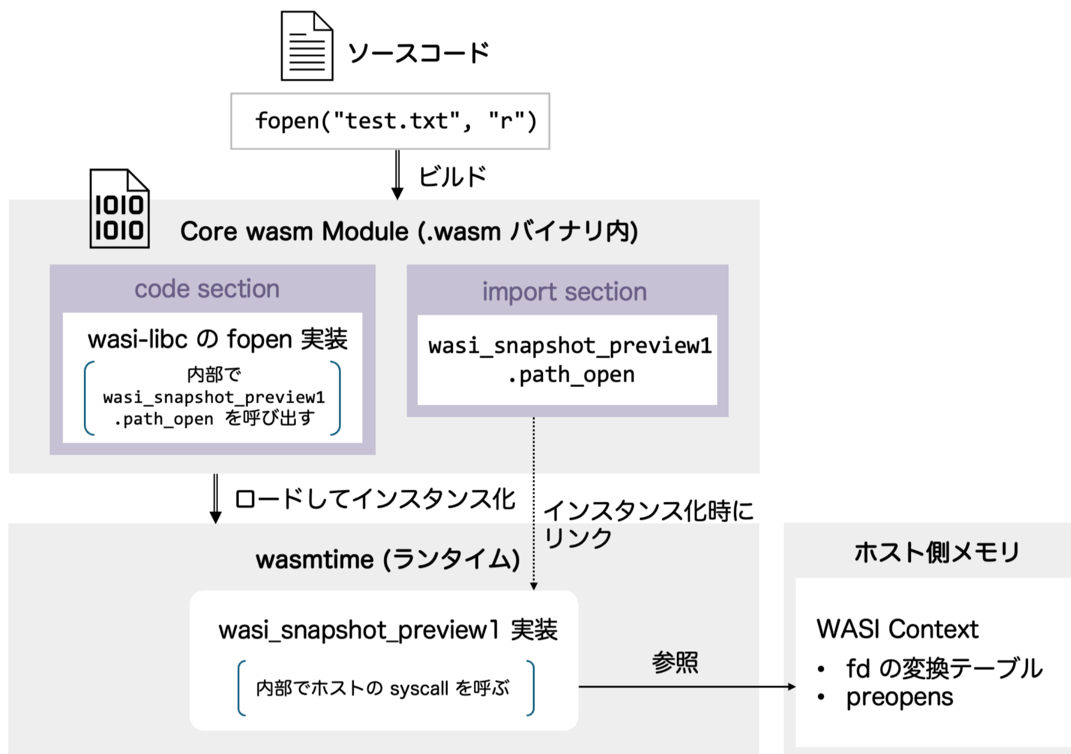

図 3.1: WASI Preview 1 におけるファイルシステムアクセスの概観

システムコール群であったのに対し，Preview 2 では機能がモジュール化され，インターフェース定義言語である WIT (Wasm Interface Type) を用いて厳密に型定義されるようになった [32].

Component Model は，複数の WebAssembly バイナリ（コンポーネント）を組み合わせて一つのアプリケーションを構築するための仕組みである [33]．各コンポーネントは「World」と呼ばれる環境定義に従って動作し，必要な機能をインポートし，提供する機能をエクスポートする．これにより，言語の壁を超えたコンポーネント間の連携や，機能の再利用性が向上した．

ファイルシステムアクセスに関しても，`wasi:filesystem` や `wasi:io` といった独立したインターフェース（WIT パッケージ）として定義されるように変更された．アプリケーションを Component Model 対応のターゲット `wasm32-wasip2` でビルドすると，生成されるコンポーネントは，従来の数値ベースのシステムコールではなく，WIT で定義された高レベルなインターフェース関数をインポートするようになる (図 3.2).

実行時の仕組みは Preview 1 と同様であり，ホスト側のランタイムがこれらのインター

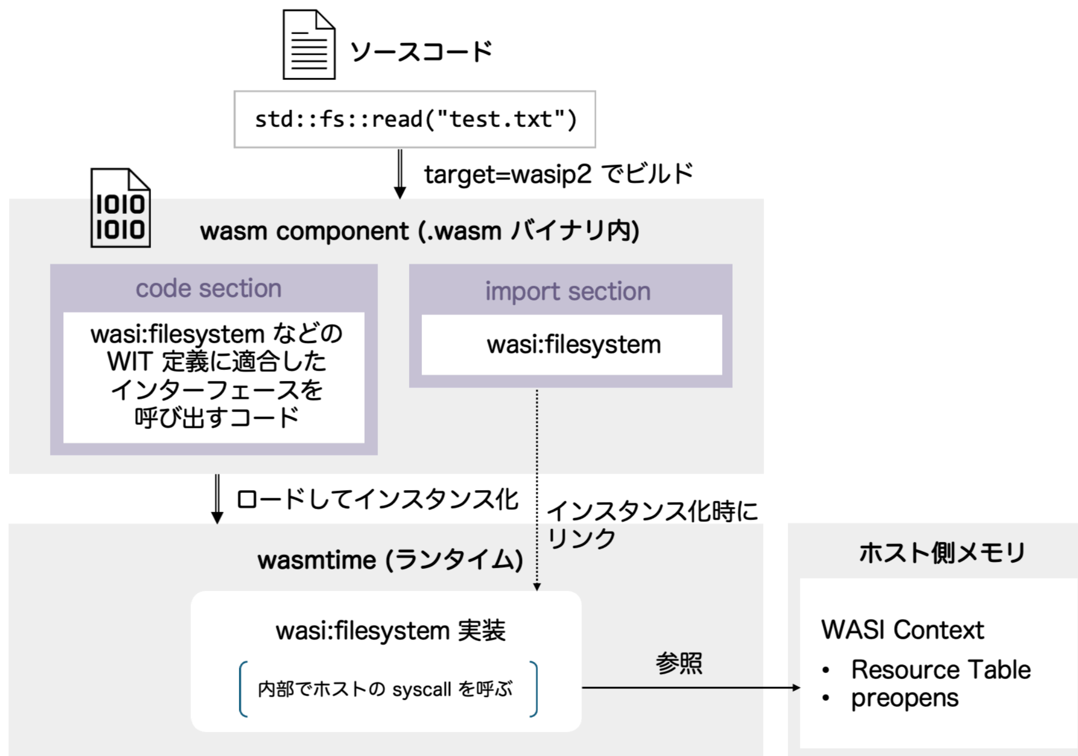

図 3.2: WASI Preview 2 におけるファイルシステムアクセスの概観

フェースの実装を提供する．しかし，Component Model の導入により，ホスト側の実装を別の WebAssembly コンポーネントへ置換することや，既存の実装をラップする構成を容易に実現できるようになった．これは，本研究で提案するホスト非依存ファイルシステムを実現する上で重要な特性となる．

第 4 章

ホスト非依存なファイルシステムの先行研究とその課題

WebAssembly アプリケーションにファイルシステム機能を提供しつつ，ホスト環境への依存を断ち切る試みはいくつか行われてきた．本章では代表的な先行事例として `wasi-vfs` [34] および `wasi-virt` [35] について解説し，それぞれの課題を指摘する．

4.1 wasi-vfs と Wizer による仮想化

`wasi-vfs` [34] は，元々 Ruby や Python といったインタプリタ言語を WebAssembly 上で動作させるために開発された VFS（Virtual File System）である．これらの言語ランタイムは，標準ライブラリをロードするために多数のファイルを必要とするため，ホストにこれらのファイルを配置しなければ動作しないという問題があった．

`wasi-vfs` のアプローチは，WASI Preview 1 の関数をフックし，WebAssembly モジュール内部に埋め込まれた仮想ファイルシステムに対して操作を行うというものである．具体的には，`wasi-vfs` が `wasi_snapshot_preview1` の `path_open` などの関数の実装を提供し，ビルド時にリンクさせることによって，import セクションにそれらが記載されないようにする．この仕組みにより，ファイルシステム呼び出しはホスト側の実装を呼び出すことなく，Wasm モジュール内部で処理が完結するようになる．

ファイルの埋め込みには `Wizer` というツールが活用される．Wizer は WebAssembly モジュールの事前初期化ツールであり，モジュールを一度実行し，初期化処理が完了した時点でのメモリ状態（スナップショット）を新たな WebAssembly バイナリとして出力する．`wasi-vfs` はこれを利用し，必要なファイルをメモリ上に展開した状態のスナップ

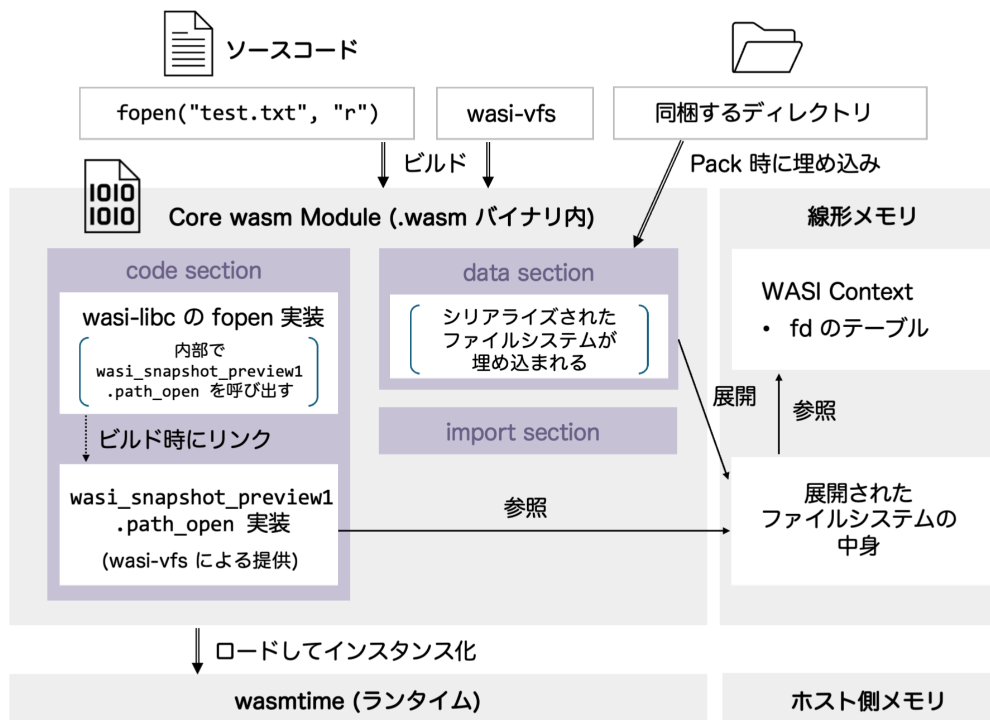

図 4.1: wasi-vfs 概観

ショットを作成することで，実行時にはあたかもファイルが存在するかのように振る舞う (図 4.1).

しかし，この手法にはいくつかの課題がある．第一に，WASI Preview 1 および `wasi-libc` に強く依存しており，`wasi-libc` を使用しない Rust などの言語では利用が難しい．第二に，`wasi-vfs` の実装は読み取り専用であり，実行中に生成されたデータの永続化および動的なファイルの追加はサポートされていない．第三に，ファイルシステムがアプリケーションと一体化してしまうため，複数のアプリケーション間でファイルを共有できない．

4.2 wasi-virt によるコンポーネントベースの仮想化

`wasi-virt` [35] は，Component Model の特性を活かした仮想化ツールである．これは，`wasi:filesystem` インターフェースをエクスポートする WebAssembly コンポーネントとして実装されており，アプリケーションコンポーネントと合成（Compose）することで機能する．

WebAssembly Composition（wac）などのツールを用いて合成を行えば，アプリケーションがインポートする `wasi:filesystem` を，`wasi-virt` が提供する実装に差し替えることができる．これにより，アプリケーションコードを一切変更することなく，ファイルアクセス先を仮想環境に向けることが可能となる (図 4.2)．具体的には，アプリケーションからビルドされるコンポーネントは `wasi:filesystem` を import するが，`wasi-virt` からビルドされるコンポーネントはそれを export するため，両者を組み合わせて結合したバイナリでは依存が解決され，import セクションに `wasi:filesystem` は残らなくなる．そのため，ロードしてインスタンス化された際もホスト側の実装が呼び出されることはない．

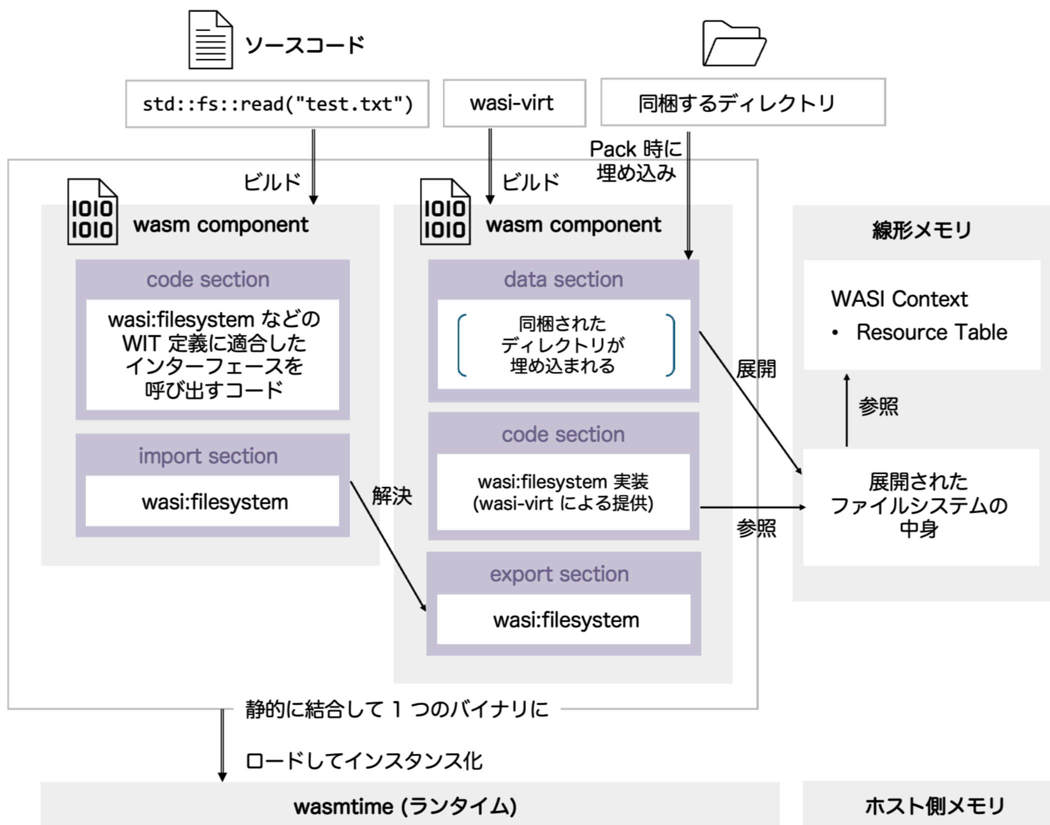

図 4.2: wasi-virt 概観

`wasi-virt` はポータビリティの観点では優れているが，現状の実装では機能が限定的であり，2026 年 1 月時点では読み取り専用と記載されている [35]．また，`wasi-vfs` 同様，基本的には単一のアプリケーションに対する静的な構成であり，複数アプリケーショ

ン間での動的なファイル共有や，外部ストレージとの同期といった高度な要件には対応できない．

4.3 既存手法の課題整理と本研究の目的

WASI を介したホストファイルシステムへのアクセスは，サンドボックスの隔離性低下というセキュリティ上の懸念と，OS 間の差異によるポータビリティの阻害という課題を抱えている．既存の仮想化手法である `wasi-vfs` や `wasi-virt` は，複数の WebAssembly アプリケーション間で同一の仮想ストレージを共有することや，外部のネットワークストレージと透過的に同期するといった柔軟な運用が考慮されていない．

本研究の目的は，これらの課題を解決し，WebAssembly のサンドボックス性とポータビリティを維持しつつ，多様な実行形態に対応可能な仮想ファイルシステムを設計・実装することである．具体的には，以下の 3 つの要件を満たすシステムを構築する．

1. ホスト OS のファイルシステムからの論理的隔離
2. WASI Preview 2 インターフェースとの互換性による，既存アプリケーションの無修正での動作
3. ユースケースに応じた柔軟なデプロイメント（静的合成，複数アプリケーション間での共有，RPC による動的接続）の実現

次章以降では，これらの要件を実現するための 3 つの実装アプローチについて詳述する．

第 5 章

ホスト非依存な仮想ファイルシステムの設計と実装

本章から第 7 章にかけて，本研究で提案する 3 つの異なる構成によるホスト非依存ファイルシステムの実装について解説する．まず本章では，Component Model の合成機能を用いた，アプリケーションとファイルシステムを静的に同梱する形式について論じる．

5.1 Component Model における仮想ファイルシステム実装戦略

Component Model において，コンポーネントが必要とする機能（Import）が解決されない場合，その実行は失敗する．通常，ファイルシステム操作（`wasi:filesystem`）の実装はランタイム（ホスト）によって提供されるが，これを独自の WebAssembly コンポーネントによって解決させることで，ホスト非依存な環境を構築できる．

このアプローチには大きく分けて，静的な合成と動的なリンクの 2 つの方法がある．本章で採用する静的な合成は，ビルド時にアプリケーションコンポーネントとファイルシステムコンポーネントを結合し，単一の WebAssembly バイナリ（`.wasm`）を生成する，wasi-virt に類似した手法である．これにより，ランタイム側に追加の構成やプラグインを必要とせず，WASI Preview 2 に対応したランタイムがあればどこでも動作するポータビリティの高いバイナリを作成できる．以下では本アプローチを「静的合成アプローチ」と呼称する．

5.2 静的合成アプローチのシステム構成

静的合成アプローチの構成を図 5.1 に示す.

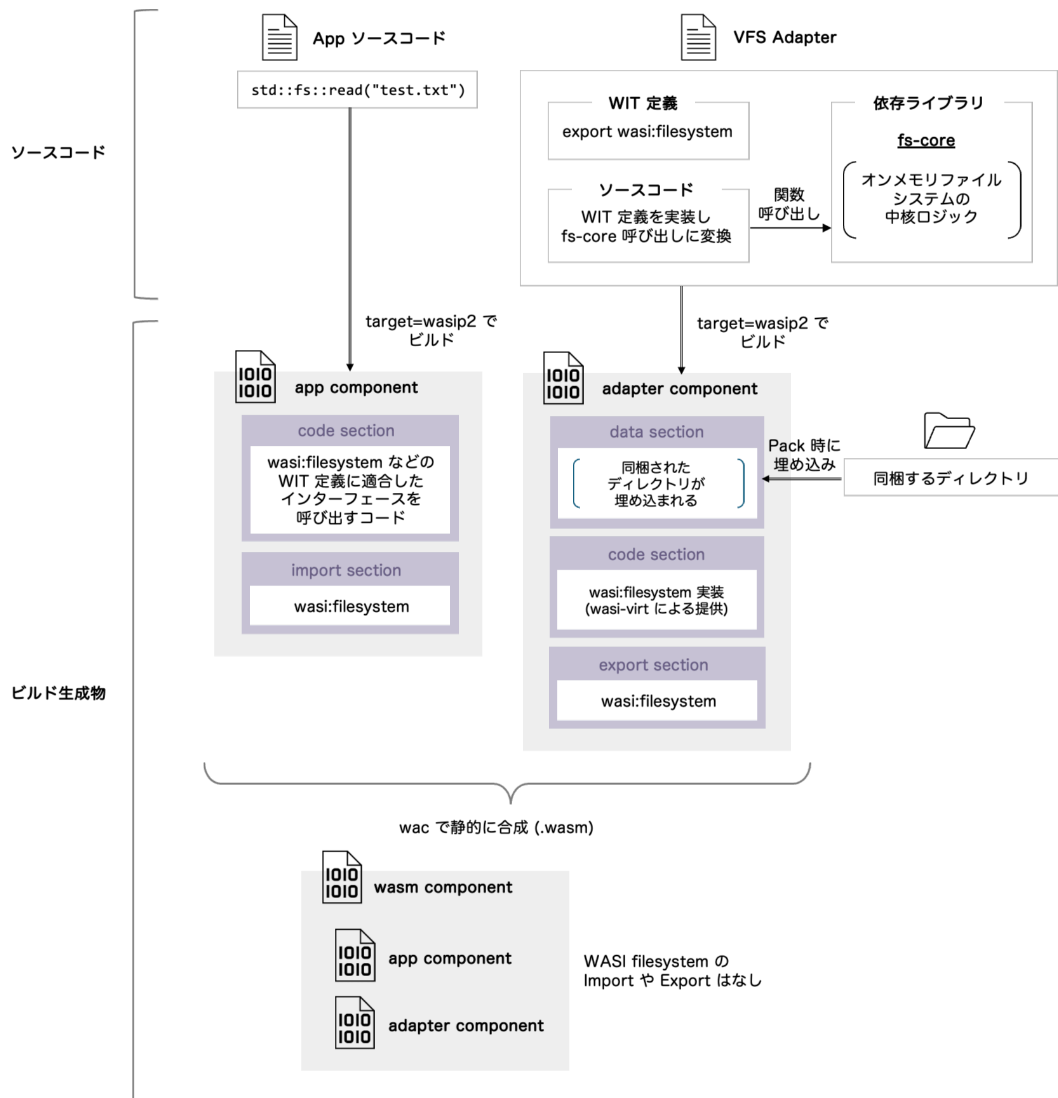

図 5.1: 静的リンク時の概観

1. **App**: ユーザが記述した WebAssembly アプリケーションであり，Rust の std::fs や C 言語の標準入出力関数といった，プログラミング言語固有の標準ライブラリを使用して記述される．コンパイル時には，ツールチェーンによってこれらの標準的

な機能呼び出しが WASI へのインポートへと変換され，結果として wasi:filesystem をインポート要求に含むコンポーネントとして構成される．

2. **VFS Adapter**: `wasi:filesystem` インターフェースを実装し，エクスポートする Adapter コンポーネント．内部で `fs-core` ライブラリを利用し，外部からのファイル操作リクエストを `fs-core` の操作へと変換する．
3. **fs-core**: 本研究で実装したオンメモリファイルシステムの中核ロジック．Rust で実装され，Inode 管理，ブロック割り当て，ディレクトリエントリの管理など，仮想ファイルシステムの基本的な機能をメモリ上で提供する．

Listing 5.1: fs-core における Inode 構造体

```
pub struct Metadata {
    pub size: u64,
    pub created: u64,
    pub modified: u64,
    pub permissions: u16,
    pub is_dir: bool,
}

pub enum FileContent {
    File(BlockStorage),
    Dir(BTreeMap<String, InodeId>),
}

pub struct Inode {
    pub id: InodeId,
    pub metadata: Metadata,
    pub content: FileContent,
}
```

5.3 fs-core の設計と実装

`fs-core` は，UNIX ファイルシステムにおける inode やブロック管理の概念を参考に，簡略化した構造を持つ．すべてのデータは WebAssembly の線形メモリ上に保存される．

ファイルシステムのメタデータは `Inode` 構造体によって管理される．リスト 5.1 に fs-core の Inode 構造体を示す．`Inode` には，ファイル種別（ファイルまたはディレクト

リ），サイズ，パーミッション，作成日時などの属性情報が含まれる．ファイルの実データは，固定長のバイト配列（ブロック）のリストとして管理され，ファイルの伸長に応じて動的にメモリが割り当てられる．

ディレクトリは，ファイル名と Inode 番号のペアを保持する特殊なファイルとして実装されている．これにより，階層的なディレクトリ構造をメモリ上で表現している．

fs-core の実装は wasi-virt の VFS 実装とその着想を同じくしているが，本研究の目的を達成するために独自の設計に基づき新規に実装を行った．既存の wasi-virt におけるファイルシステム実装は，主にビルド時における静的資産の同梱を主眼に置いており，書き込み操作をサポートしていない一方で，本実装では書き込みを含むファイル操作を提供する．さらに，後述する複数コンポーネント間での共有を見据え，スレッドセーフな実装へと容易に切り替え可能な設計を採用した．具体的には，メモリ管理に Rust のスマートポインタを活用し，単一スレッド向けの最適化と，複数スレッド間での排他制御をコンパイル時に切り替え可能とすることで，実行形態に応じた柔軟な構成を実現している．詳細な設計については Appendix A を参照.

5.4 vfs-adapter と wac plug による合成

`vfs-adapter` は, Component Model の WIT 定義に従い, `wasi:filesystem/preopens` や `wasi:filesystem/types` といったインターフェースを実装する．例えば，`open-at` 関数が呼び出されると，Adapter はパスを解析し，`fs-core` 内の該当する Inode を探索し，ファイルディスクリプタを生成して返却する．

最終的なバイナリの生成には，`wac plug` コマンドを使用する．これにより，App Component と Adapter を合成し，App Component の import を vfs-adapter の export で解決する．

Listing 5.2: wac コマンドによるコンポーネントの合成

```
1 wac plug app.wasm --plug vfs-adapter.wasm -o bundled-app.wasm
```

生成された `bundled-app.wasm` は，外部への `wasi:filesystem` インポートを持たず，内部で完結しているため，どのようなホスト環境でも同一の挙動を示す．

5.5 検証と課題

5.5.1 動作検証

一連のファイルシステム操作を確認

ファイルシステム操作を行うアプリケーションを Rust で記述し，先述の手順に沿って wasm バイナリファイルにビルドした上で wasmtime CLI を用いて実行した．実行したアプリケーションのコードをリスト 5.3，実行結果を図 5.2 に示す．アプリケーションコードは標準のファイルシステム API を利用して，ディレクトリの作成，ファイルの一覧表示，メタデータの取得，書き込み，読み込み，削除を実行している．実行結果より，書き込んだファイルのメタデータの取得およびファイル内容の読み出しが可能であることが確認できた．

Listing 5.3: ファイルシステム操作を行うアプリケーションコード

```
1  use std::fs;
2  use std::io::Write;
3
4  fn main() {
5      let directory_path = "/testdir";
6      fs::create_dir(directory_path);
7
8      let nested_directory_path = "/testdir/sub1/sub2";
9      fs::create_dir_all(nested_directory_path);
10
11     let file_path = "/testdir/test.txt";
12     let content = "Hello␣from␣Component␣Model!";
13     fs::write(file_path, content);
14
15     let metadata = fs::metadata(file_path);
16     println!("File␣metadata:␣{:?}", metadata);
17
18     let read_content = fs::read_to_string(file_path);
19     println!("Read␣content:␣{}", read_content.unwrap());
20
21     fs::read_dir(directory_path).unwrap().for_each(|entry| {
22         let entry = entry.unwrap();
```

```
23          println!("Entry:␣{}", entry.path().display());
24      });
25
26      fs::remove_file(file_path);
27
28      fs::read_dir(directory_path).unwrap().for_each(|entry| {
29          let entry = entry.unwrap();
30          println!("Entry␣after␣file␣remove:␣{}", entry.path().
                display());
31      });
32  }
```

```
File metadata: Ok(Metadata { file_type: FileType { is_file: true, is_dir: false, is_s
ymlink: false, .. }, permissions: Permissions(FilePermissions { readonly: false }), l
en: 27, modified: SystemTime(1767250776s), accessed: SystemTime(1767250776s), created
: SystemTime(1767250776s), .. })
Read content: Hello from Component Model!
Entry: /testdir/sub1
Entry: /testdir/test.txt
Entry after file remove: /testdir/sub1
```

図 5.2: ファイルシステム操作確認時のスクリーンショット

同梱したファイルシステムにアクセスできることを確認

起動時に設定ファイルを読み込むユースケースを想定し，本システムはビルド時に特定のディレクトリを data section に埋め込み，起動時にメモリ上の仮想ファイルシステムに展開する CLI を提供している．CLI を利用すれば，指定したディレクトリを埋め込んだ wasm バイナリをリスト 5.4 のように作成できる．

Listing 5.4: 本研究で開発した CLI ツールによるビルド時のディレクトリ埋め込み

```
1 halycon-pack -- embed /tmp/composed.wasm --mount /data=/tmp/vfs-
      test-data -o /tmp/final.wasm
```

このようにして生成されたバイナリを wasmtime で実行すれば，アプリケーションからは同梱したディレクトリの中身を読み込むことができる (図 5.3).

```
Root directory contents:
  [DIR] /data

/data directory contents:
  [DIR] /data/config
  [FILE] /data/hello.txt

Reading /data/hello.txt:
  Content: Hello from embedded file!

Reading /data/config/settings.json:
  Content: {"version": "1.0", "name": "test"}
```

図 5.3: ファイルシステム同梱確認時のスクリーンショット

5.5.2 ユースケース

この構成は，設定ファイルを同梱して読み込む用途のほか，書き込みの特性を活かして，一時的な作業領域を必要とするアプリケーションや，単体テスト環境での利用にも適している．例えば，コンパイラや画像処理ツールなど，入力ファイルを変換して出力するようなバッチ処理において，中間ファイルをホストのファイルシステムに保存することなく処理を行うことが可能となる．

ここでは，入力データの読み込み，中間ファイルの生成，最終出力の作成を行う画像処理パイプラインのユースケースを解説する．アプリケーションを利用するソースコードの抜粋をリスト 5.5 に示す．このように，画像データを加工して書き出したのち，それを読み出して後続の処理を行うことができる．

こうした一時ファイルを介したデータ連携を伴うバッチ処理において，静的合成アプローチは有用な手法となる．

Listing 5.5: 画像処理パイプラインのソースコード

```
1  use std::fs;
2
3  fn main() {
4      // 中略
5      fs::write("/input/photo.raw", b"RAWIMG_DATA").unwrap();
6
7      let raw_data = fs::read("/input/photo.raw").unwrap();
8      let resized_data = process_resize(raw_data); // 実装は省略
9      fs::write("/work/resized.dat", resized_data).unwrap();
10
11     let tmp_data = fs::read("/work/resized.dat").unwrap();
```

```
12      let png_data = process_convert(tmp_data); // 実装は省略
13      fs::write("/output/photo.png", png_data).unwrap();
14  }
```

5.5.3 課題

本アプローチのデメリットとして，ファイルシステムの状態がインスタンスごとに独立しており，プロセスの終了とともにデータが消失する点が挙げられる．また，複数のアプリケーションから同一のファイルシステムを参照することはできない．

第 6 章

複数アプリケーション間における ファイルシステム共有の実現

6.1 単一コンポーネント内完結の限界

前章で提案したアプリケーションとファイルシステムを静的に合成する手法は，高いポータビリティとセキュリティを提供する一方で，データ共有の観点では大きな制約が存在する．各 WebAssembly インスタンスが独立した fs-core を持つため，あるインスタンスが書き込んだファイルを，別のインスタンスから読み取ることができない．

実際のサーバレス環境やエッジコンピューティングの現場では，前段のアプリケーションが生成した中間ファイルを後段の別アプリケーションが読み出すことで一連のタスクを完遂するパイプライン処理や，共通の設定ファイルやセッション情報を参照するケースが存在する．このようなユースケースにおいて，ファイルシステムの状態がインスタンスごとに隔離されていることは問題となり得る．

そこで本章では，Component Model の柔軟性を活かし，ファイルシステムの実装をホスト環境側に移譲することで，同一プロセス内で動作する複数の WebAssembly アプリケーション間でのファイルシステム共有を実現する手法について論じる．

6.2 ホスト側でのインターフェース実装

Component Model において，コンポーネントがインポートする `wasi:filesystem` などのインターフェース実装は，必ずしも WebAssembly コンポーネントである必要はない．ホスト側のプログラムを記述する Rust や Go などの言語で直接実装し，インスタン

ス化の際にこれらの実装をコンポーネントのインポート要求に割り当てることが可能である.

この手法では，ホストプロセス内に共有可能な状態（Shared State）として fs-core のインスタンスを保持する．Rust 言語を用いる場合，`wasi:filesystem` などの WIT 定義から `wit-bindgen` によって生成されたホスト側のトレイトを実装することで，各ファイル操作に対する具体的な処理を記述できる.

アーキテクチャは以下の構成要素からなる (図 6.1)．以下ではこのアプローチを「ホストトレイト実装アプローチ」と称する.

1. **VFS Host**: Trait を実装した Rust のライブラリであり，内部で先述の fs-core を呼び出す．本アプローチではファイルシステムの実体はホストに存在し，マルチスレッドで動作するため，fs-core をスレッドセーフなロック機構（RwLock）によって保護されたモードに指定して利用する.
2. **Runner**: ソースコード中で Wasm のバイナリのパスを指定し，WebAssembly アプリケーションをインスタンス化する．その際，Linker に対して VfsHostState への参照を渡す．これにより，アプリケーションからの `wasi:filesystem` 呼び出しは，ホスト内の VfsHostState のメソッドへとディスパッチされる．Runner は VFS Host と同梱され，ネイティブバイナリとしてビルドされる.
3. **複数の App Component**: それぞれ独立してインスタンス化されるが，wasi:filesystem インポートは VFS Host によって解決され，fs-core 呼び出しへ変換される．結果的に， 複数インスタンスが同一のファイルシステムを共有する.

本研究で実装した fs-core は，マルチスレッド環境における並行性とデータの整合性を両立するため，二層構造の細粒度ロック機構を採用している．第一層として，Inode およびファイルディスクリプタを管理するテーブルに，細粒度ロックによる高並行性ハッシュマップである `DashMap` を導入することで，テーブル全体を単一のロックで保護する際のボトルネックを排除し，異なるエントリに対する検索や挿入，削除における高い並列性を実現した．第二層では，個々の Inode の実体を `Arc<RwLock<Inode>>`で保護し，同一ファイルに対するアクセスの粒度を制御している．RwLock の採用により，複数のスレッドによる同時読み取りを許容しつつ，書き込み時には排他的なアクセスを強制することで，データの整合性を担保している．また，Inode ID やファイルディスクリプタの採番には整数型を扱う Atomic 型の変数によるロックフリーカウンタを用いることで，高頻度なファイル生成時における競合を回避している．これらの設計により，異なるファイルへの同時操作を並行化する一方で，同一ファイルに対しては排他制御を提供できる.

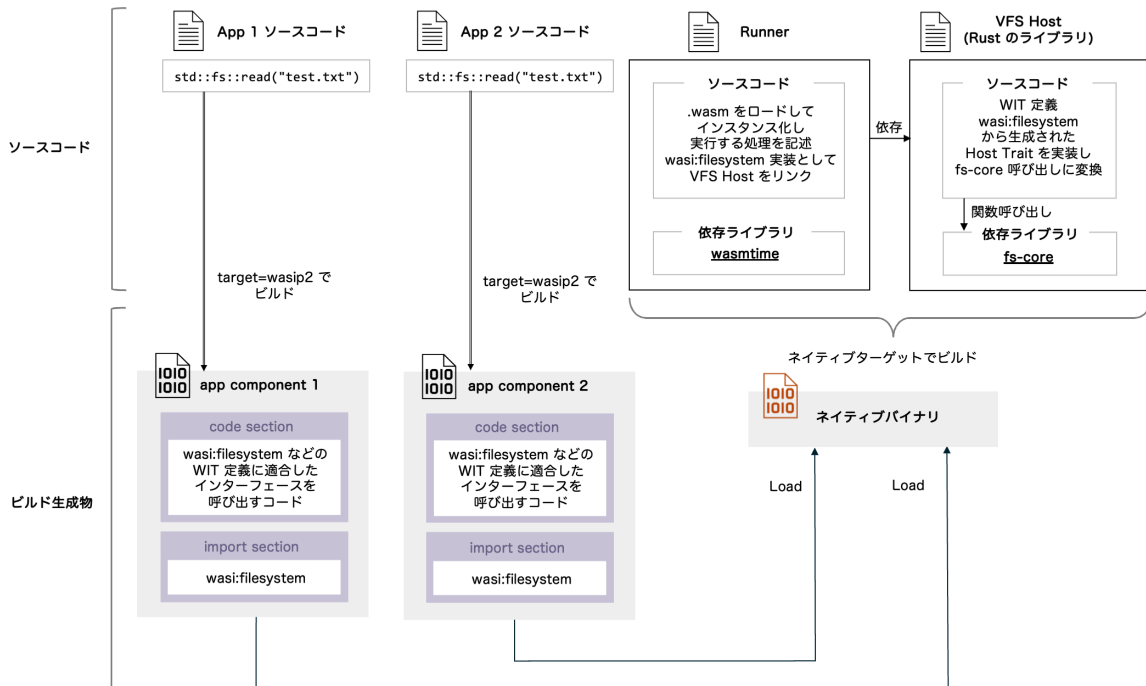

図 6.1: ホストトレイト実装アプローチの概観

6.3 検証と課題

6.3.1 動作検証

本構成の実証として，書き込み用アプリケーション（Writer）と読み込み用アプリケーション（Reader）の 2 つを用意した．これらのアプリケーションは，両者をロードして実行する Runner を介して，同一のホストプロセス上で順次実行される．Runner, Writer, Reader のソースコードをそれぞれリスト 6.1, 6.2, 6.3 に示す．Runner が Writer の wasm ファイルをロードして起動させ，テキストを書き込んで終了した後，同様に Runner が reader の wasm ファイルをロードして起動させると，当該ファイルを正しく読み取れることが確認できる (図 6.2).

Listing 6.1: Runner のソースコード (抜粋)

```
1 let vfs_host_state1 = vfs_host::VfsHostState::new(engine,
      vfs_adapter_path)
2     .context("Failed to create VfsHostState")?;
3
```

```
let mut store1 = Store::new(engine, vfs_host_state1);
let mut linker1 = wasmtime::component::Linker::new(engine);
vfs_host::add_to_linker_with_vfs(&mut linker1)?;

let writer_path = "../../../target/wasm32-wasip2/debug/demo-writer.wasm";
let writer_component =
    Component::from_file(engine, writer_path).context("Failed␣to␣load␣demo-writer.wasm")?;

let writer_command = Command::instantiate(&mut store1, &writer_component, &linker1)
    .context("Failed␣to␣instantiate␣demo-writer")?;

// 起動して実行
match writer_command.wasi_cli_run().call_run(&mut store1) {
    Ok(Ok(())) => println!("demo-writer␣executed␣successfully"),
    //... 中略
}

// 同様に reader も instance 化
```

Listing 6.2: Writer のソースコード (抜粋)

```
use std::fs;

fn main() {
    println!("===␣VFS␣Demo␣App␣1:␣File␣Writer␣===");
    let message = "Hello␣from␣App1!";
    println!("\nWriting␣file:␣/message.txt");
    match fs::write("/message.txt", message) {
        Ok(()) => println!("␣␣Wrote␣{}␣bytes", message.len()),
        Err(e) => {
            eprintln!("␣␣Failed␣to␣write:␣{}", e);
            return;
        }
    }
}
```

Listing 6.3: Reader のソースコード (抜粋)

```
use std::fs;

fn main() {
    println!("===␣VFS␣Demo␣App␣2:␣File␣Reader␣===");

    println!("\nGetting␣file␣metadata:␣/message.txt");
    match fs::metadata("/message.txt") {
        Ok(meta) => {
            println!("␣␣File␣size:␣{}␣bytes", meta.len());
        }
        // 省略...
    }
    println!("\nReading␣file:␣/message.txt");
    match fs::read_to_string("/message.txt") {
        Ok(content) => {
            println!("␣␣Content␣({}␣bytes):", content.len());
            println!("␣␣\"{}\"", content);
        }
        // 省略...
    }
}
```

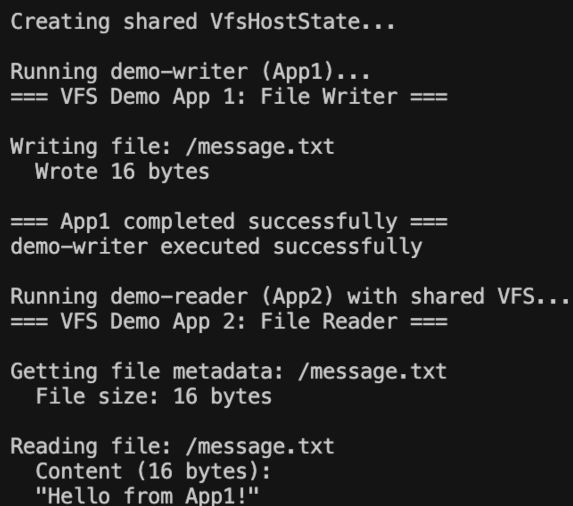

図 6.2: ホストトレイト実装アプローチの動作確認

さらに，並列実行時の試験として，複数のアプリケーションを同時にインスタンス化して実行し，同じファイルに対して同時に追記させる動作確認を行った．具体的には，Wasm アプリケーション 4 つを，それぞれ異なる CLIENT ID (**CLIENT_001** 形式) を付与した上で同時に起動し，同一ファイルに 1 行ずつ，シーケンス番号 (**SEQ_00000** 形式) を含めて 200 回追記を行った．作成されたファイルの冒頭を図 6.3 に示す．一つのシーケンス番号について CLIENT 001-004 の書き込みが全て含まれており，同時書き込みによるデータの欠落や破損が発生していないことが確認できる．また，作成されたファイルの行数は書き込みの合計回数と一致していた．以上より，適切なロック制御が行われているため，データ破損を起こすことなくファイルの共有が可能であることを確認した．

```
[1768658561205] CLIENT_002:SEQ_00000
[1768658561205] CLIENT_003:SEQ_00000
[1768658561205] CLIENT_001:SEQ_00000
[1768658561205] CLIENT_004:SEQ_00000
[1768658561307] CLIENT_004:SEQ_00001
[1768658561307] CLIENT_001:SEQ_00001
[1768658561307] CLIENT_003:SEQ_00001
[1768658561307] CLIENT_002:SEQ_00001
[1768658561409] CLIENT_001:SEQ_00002
```

図 6.3: 同時書き込み時の挙動確認

6.3.2 ユースケース

ホストトレイトアプローチの最大の利点は，同一のファイルシステム実体（fs-core）を，複数の WebAssembly インスタンス間で共有できる点にある．想定されるユースケースとしては，例えばエッジデバイス上でのデータ処理パイプラインがある．センサデータを受信して一時ファイルに書き出す軽量な Ingest モジュールと，そのファイルを定期的に読み出して分析・圧縮を行う Process モジュールを別々の WebAssembly アプリケーションとして実装し，連携させることができる．このように機能をモジュール単位で分離することは，セキュリティにおける最小権限の原則を適用する上で有効である．各モジュールを独立したコンポーネントとして隔離し，実行に必要な最小限の権限のみを個別に付与することで，システム全体の堅牢性を向上させることができる．

そのほかに想定されるユースケースとして，リクエストごとに WebAssembly インスタンスを生成して処理を行う Fermyon Spin のような構成において，ファイルシステムを介してリクエスト間のデータを共有するキャッシュサーバの実装が挙げられる．キャッシュサーバのソースコードをリスト 6.4 に示す．ここでは，リクエストが来るたびに handler

(wasm アプリケーション) をインスタンス化して立ち上げている．それらのインスタンスが共通のファイルシステムを共有するため，同じ path へのアクセスが TTL 以内に再び届けば，キャッシュヒットとみなしてファイルシステムから結果を迅速に返却できる (図 6.4).

Listing 6.4: キャッシュサーバ のソースコード (抜粋)

```
1  #[tokio::main]
2  async fn main() -> Result<()> {
3      // 中略
4      let engine = Engine::new(&config)?;
5      let handler_component = Component::from_file(&engine,
           handler_path).context("Failed␣to␣load␣handler␣component")
           ?;
6      let initial_vfs_state = vfs_host::VfsHostState::new().context
           ("Failed␣to␣create␣VfsHostState")?;
7      let shared_vfs = initial_vfs_state.get_shared_vfs();
8      let state = Arc::new(AppState {engine, handler_component,
           shared_vfs,});
9      let app = Router::new()
10         .route("/api/*path", get(handle_api_request)) // この内部
               で Wasm をインスタンス化して実行
11         .route("/health", get(|| async { "OK" }))
12         .with_state(state);
```

```
[SERVER] Handling request: /api/users
=== HTTP Cache Handler ===
Request: /api/users

[CACHE MISS] /api/users
[API FETCH] /api/users (mock)
[CACHE WRITE] /cache/api_users.cache

Response:
{
  "data": {
    "items": [
      1,
      2,
      3,
      4,
      5
    ],
    "message": "Response for /api/users"
  },
  "path": "/api/users",
  "timestamp": 1768645025
}
[SERVER] Handling request: /api/users
=== HTTP Cache Handler ===
Request: /api/users

[CACHE HIT] /api/users (expires in 299s)
```

図 6.4: HTTP キャッシュサーバ実装で，キャッシュにヒットしていることの確認

6.3.3 課題

この手法において，共有の範囲はあくまで単一のホストプロセス内に限られており，プロセス外から動的に追加したアプリケーションに対してファイルシステムを共有することはできない．また，wasm ファイルだけでなくネイティブバイナリに依存するため，デプロイ時に複数のファイルを配布する必要がある．

第 7 章

RPC を用いたアプリケーションの動的追加の実現

7.1　プロセス境界を超えた共有の必要性

前章の手法はプロセス内での共有を実現したが，大規模な分散システムやマイクロサービスアーキテクチャにおいては，プロセス境界を超えたファイル共有が求められる場合がある．例えば，負荷に応じて WebAssembly の実行基盤自体がスケールアウトする場合や，異なる物理マシン上で動作するアプリケーション間でデータを共有したい場合である．

また，システムを停止させることなく新しいアプリケーションを動的に追加し，既存のファイルシステムにアクセスさせたいという要求もある．これらを満たすためには，ファイルシステム自体を独立したサーバプロセスとして切り出し，ネットワーク経由でアクセスする機構が必要となる．

7.2　RPC を用いたファイルシステムアクセス時のアーキテクチャ

本章では，Remote Procedure Call (RPC) を用いたホスト非依存ファイルシステムの設計について述べる．システム (図 7.1) は以下のコンポーネントで構成される．以下，この構成を「RPC 動的追加アプローチ」と称する．

1. **VFS RPC Server**: ファイルシステムの実体（fs-core）を管理する独立したサー

バプロセス．TCP ソケット上で Protocol Buffers を用いて外部からのファイル操作リクエストを受け付ける．

2. **RPC Adapter**: アプリケーションにリンクされる Adapter コンポーネント．`wasi:filesystem` インターフェースを実装するが，内部動作としてはファイル操作を RPC リクエストに変換し，VFS RPC Server へ送信する．

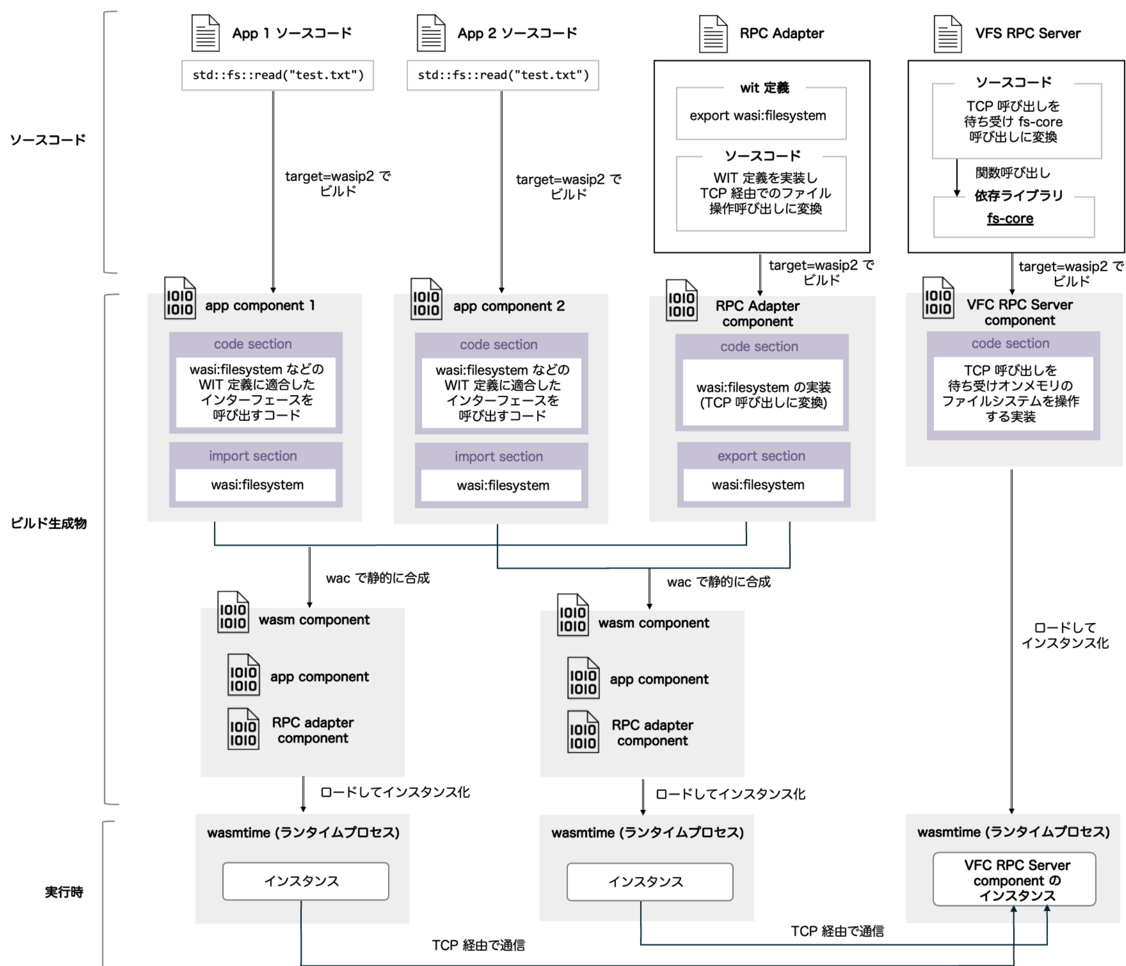

図 **7.1:** RPC 動的追加アプローチの概観

7.3 通信プロトコルと実装の詳細

通信には，WASI 環境でも利用可能な TCP ソケット通信を用いる．RPC Adapter は，ファイルオープン（open-at）のリクエストを受け取ると，サーバに対して Protocol Buffers でシリアライズした OpenPath リクエストを送信する．サーバはパス解決を行い，成功すればファイルディスクリプタ（fd）をクライアントに返す．以降の Read/Write 操作は，この fd とともにリクエストされ，サーバ側の fs-core で処理される．Protocol Buffers メッセージ定義の一部をリスト 7.1 に示す．

Listing 7.1: Protocol Buffer のスキーマ定義 (抜粋)

```
message RpcRequest {
  optional string session_id = 1;
  oneof request {
    Connect connect = 2;
    OpenPath open_path = 3;
    OpenAt open_at = 4;
    Read read = 5;
    Write write = 6;
    Close close = 7;
    Seek seek = 8;
    Ftruncate ftruncate = 9;
    Stat stat = 10;
    Fstat fstat = 11;
    Mkdir mkdir = 12;
    MkdirP mkdir_p = 13;
    Unlink unlink = 14;
    Readdir readdir = 15;
    ReaddirFd readdir_fd = 16;
    Rmdir rmdir = 17;
    AppendWrite append_write = 18;
  }
}

message RpcResponse {
  oneof response {
    Connected connected = 1;
```

```
27      Ok ok = 2;
28      Fd fd = 3;
29      Data data = 4;
30      Written written = 5;
31      Position position = 6;
32      MetadataResponse metadata = 7;
33      DirEntries dir_entries = 8;
34      Error error = 9;
35    }
36  }
37
38  message MetadataResponse {
39    uint64 size = 1;
40    uint64 created = 2;
41    uint64 modified = 3;
42    bool is_dir = 4;
43  }
```

この手法を採用する利点は，アプリケーションとファイルシステムのライフサイクルを分離できることである．アプリケーションが終了してもファイルシステムサーバは稼働し続ける．また，新しいアプリケーションを別プロセスで起動する際，設定として RPC サーバのアドレスを与えるだけで，即座に共有ファイルシステムに参加できる．

7.4 検証と課題

7.4.1 動作検証

動作検証として，ある Wasm アプリケーションから行った変更が，別プロセスとして後で立ち上げた Wasm アプリケーションから閲覧できることを確認した．具体的には，VFS RPC Server を立ち上げた状態で，Writer アプリケーションを別プロセスとして立ち上げて書き込みを行い，さらに Reader アプリケーションを立ち上げて内容を確認した（図 7.2）．Writer と Reader のアプリケーションコードはそれぞれリスト 6.2, 6.3 と同一である．起動は全て wasmtime の CLI で実施した．

```
ternbusty@parthenon halycon %  wasmtime run -S inherit-netw
ork=y ./target/wasm32-wasip2/debug/composed-demo-writer.was
m
=== VFS Demo App 1: File Writer ===

Writing file: /message.txt
  Wrote 16 bytes

=== App1 completed successfully ===
ternbusty@parthenon halycon % wasmtime run -S inherit-netwo
rk=y ./target/wasm32-wasip2/debug/composed-demo-reader.wasm

=== VFS Demo App 2: File Reader ===

Getting file metadata: /message.txt
  File size: 32 bytes

Reading file: /message.txt
  Content (32 bytes):
  "Hello from App1!"

=== App2 completed ===
```

図 7.2: RPC を経由して別プロセスから同じファイルシステムが操作できることを確認

7.4.2 ユースケース

この構成は，アプリケーションのインスタンス数が増減しうるシステム上で WebAssembly ワークロードを実行する際に特に有効である．各アプリケーションがステートレスであっても，共有の RPC サーバを介して状態を保持できる．

ユースケースとしては，継続的インテグレーション/継続的デプロイ（Continuous Integration / Continuous Delivery; CI/CD）パイプラインにおけるビルドキャッシュの動的共有が挙げられる．例えば CI 環境に VFS RPC Server を常駐させ，ジョブから RPC Adapter を介して接続することにより，Wasm アプリケーションにおけるビルドキャッシュの共有が可能となる．CI の開始により別プロセスでアプリケーションが立ち上がっても，後から RPC 経由で同じファイルシステムに接続可能である．

7.4.3 課題

本アプローチのデメリットとしては，ネットワーク通信に伴うレイテンシの増大が挙げられる．RPC の通信オーバーヘッドは無視できない．したがって，高頻度かつ低遅延なアクセスが要求される用途には不向きである可能性がある．

第 8 章

リモートストレージとの同期機能の実装

8.1 永続性の確保と S3 同期

ここまで論じてきた fs-core はオンメモリファイルシステムであり，電源断やプロセスの再起動によってデータは消失する．実運用においては，データの永続性は不可欠なケースがある．そこで，リモートストレージとして Amazon S3 互換オブジェクトストレージを利用し，ファイルシステムの状態を永続化する拡張機能を実装した．

利用者目線では，WebAssembly アプリケーションからは通常ファイルシステム操作と同じインターフェースを利用しつつ，書き込んだ内容が S3 へ同期される形である．これは Linux における Filesystem in Userspace (FUSE) を用いた s3fs などのマウントツールと似た概念であるが，特権なしでホストの環境を問わず動作させることができる．

S3 同期は，各アプローチについてそれぞれ実装を追加することで実現可能である．静的合成アプローチでは VFS Adapter に，ホストトレイト実装アプローチでは VFS Host に，RPC アプローチでは VFS RPC Server にそれぞれ実装を追加する．下記では，それらに共通して実装した同期メカニズムについて述べる．

8.2 同期メカニズム

同期戦略は，データの整合性要求と実行パフォーマンスのトレードオフに基づき，複数のモードを実装した．以下では，アプリケーションの操作に応じた読み込み・書き込み時の同期戦略と，アプリケーションの操作とは独立して動作するバックグラウンド同期につ

いて述べる．

8.2.1 読み込み時の同期戦略

アプリケーションがファイルを読み込む際に，S3 からどのようにデータを取得するかについて，以下の方式を導入した．

- **Read-through モード**: ファイルオープンまたはリード操作のたびに S3 へリクエストを行い，最新のデータを取得する方式である．常に最新の状態を参照する必要がある共有ストレージとしての利用に適している．
- **オンメモリキャッシュモード**: 読み出しをすべてメモリから行う方式である．ネットワーク遅延なしでアクセスできるため高速だが，外部プロセスによる S3 上のデータ更新を即座に反映できない制約がある．読み取り頻度が高い静的なアセットの参照や，低レイテンシが要求されるワークロードに適している．

8.2.2 書き込み時の同期戦略

アプリケーションがファイルに書き込んだ内容を，どのように S3 へ反映するかについて，以下の方式を導入した．

- **Write-through モード**: `write` 操作や `close` 操作が発生するたびに，S3 に対して同期的なアップロードを行う方式である．不慮の停止時におけるデータ損失リスクを最小限に抑えられるが，通信の往復遅延がアプリケーションの実行速度に直接影響する．高い耐障害性が求められる用途に適している．
- **非同期モード**: ファイル操作リクエストを内部のキューに蓄積し，非同期で S3 へ反映する方式である．アプリケーションは通信の完了を待機せずに処理を続行できるため，大量の書き込みが発生するバッチ処理や，スループットの最大化を優先するシナリオに適している．

8.2.3 バックグラウンド同期

アプリケーションの操作とは独立して，S3 と VFS 間の状態を同期する仕組みとして，以下の機能を実装した．

- **コールドスタート・リストア**: 仮想ファイルシステムの初期化プロセスにおいて，指定された S3 バケット内のオブジェクトツリーを走査し，メタデータおよび実データを一括でダウンロードする機能である．これにより，プロセス起動時に S3 上の状態を VFS に復元できる．
- **ETag ベースのポーリング更新**: 定期的に S3 上のオブジェクトの ETag を確認し，変更が検知された場合のみ該当するファイルまたはディレクトリの差分更新を行う．更新頻度は環境変数経由で設定可能である．

8.3 検証と課題

8.3.1 動作検証

ここでは，RPC 実装に追加した S3 同期機能を用いて，S3 互換のローカルサーバである LocalStack を用いた検証を行った．読み込み・書き込みはいずれも Write-through, Read-through モードに設定した．以下では，Wasm アプリケーションからファイルの作成・書き込みを行ったのち，S3 上で作成されたファイルにアクセスできることを確認した (図 8.1)．また，別途 S3 上で作成されたファイルを，アプリケーションから読みにいくことができることを確認した (図 8.2)．

```
ternbusty@parthenon halycon % wasmtime run -S inherit
-network=y ./target/wasm32-wasip2/debug/composed-demo
-writer.wasm
=== VFS Demo App 1: File Writer ===

Writing file: /message.txt
  Wrote 16 bytes

=== App1 completed successfully ===
```

(a) Wasm でファイルを Write

```
awslocal s3 cp s3://test-vfs-bucket/vfs/files/message.txt -
Hello from App1!
```

(b) Wasm で作成されたファイルが LocalStack 上の S3 から確認できる

図 8.1: 仮想ファイルシステムから S3 への同期確認

```
[INFO] [sync] Downloaded: /external-message.txt
[INFO] [sync] Inbound: 1 downloaded, 0 deleted
```

(a) S3 に追加したファイルが自動でダウンロードされている VFS RPC Server ログ

```
=== VFS Demo App 2: File Reader ===

Getting file metadata: /external-message.txt
  File size: 14 bytes

Reading file: /external-message.txt
  Content (14 bytes):
  "Hello from S3
"

=== App2 completed ===
```

(b) S3 に追加したファイルが Wasm アプリケーションから確認できる

図 8.2: S3 から仮想ファイルシステムへの同期確認

さらに，一度 VFS RPC Server プロセスを終了させた後，再度プロセスを立ち上げた際に，前のセッションで書き込んだファイルが S3 から正しく復元されていることを確認した (図 8.3).

```
[INFO] VFS RPC Server starting...
[INFO] S3 persistence enabled: bucket=test-vfs-bucket, prefix=vfs/
[DEBUG] [s3] Using custom endpoint: http://127.0.0.1:4566
[INFO] [sync] Found 4 files in S3
[INFO] [sync] Loaded: /external-message.txt
[INFO] [sync] Loaded: /external/s3file.txt
[INFO] [sync] Loaded: /message.txt
[INFO] [sync] Loaded: /shared/message.txt
[INFO] Sync mode: Batch (set VFS_SYNC_MODE=realtime for immediate sync)
[INFO] Socket bound to 127.0.0.1:9000
[INFO] VFS RPC Server listening on 127.0.0.1:9000
```

図 8.3: 再起動時に S3 から復元しているログ

8.3.2 ユースケース

本アプローチは，単にデータの永続化に役立つだけでなく，本来データ共有ができなかったアプローチでのファイルシステム共有を可能にする．例えば，静的合成アプローチではアプリケーションとファイルシステムが静的に同梱され共有は不可能であったが，共通の S3 と sync することにより同じファイルシステムを操作可能になる．ホストトレイト実装についても同様に，別プロセスのアプリケーションとも S3 を通じてファイルシステムを共有できるようになる．

具体的なユースケースとしては，地理的に分散したエッジノードで動作する WebAssembly ワークロードにおいて，各ノードでの処理ログやセンサーデータを集計するケースが挙げられる．各エッジノードはプロバイダが異なり，ホスト OS のファイルシステム構造や権限設定が統一されていないことも多い．

このような環境下においても，本手法が提供する仮想ファイルシステムは各プログラミング言語の標準ライブラリから利用可能な入出力インターフェースを提供するため，ファイルシステムへの出力を前提とした既存のログ出力ライブラリを一切変更することなくそのまま利用可能である．さらに，本手法ではシステム要件に応じて，読み込みや書き込みの同期タイミングをモードの切り替えのみで柔軟に変更できる．例えば，ログの欠損が許されない厳格な監視用途では Write-through モードによる即時同期を選択し，一方で高いスループットが要求されるデータ集計用途では非同期モードへと設定を変更するといった運用が可能である．このように，アプリケーション側のロジックを修正することなく，設定レベルでパフォーマンスと整合性のトレードオフを最適化できる点は，分散環境における開発効率を向上させる．

最終的に，ホスト側に特別なエージェントやマウント設定を必要とせず，分散ノード間でのログ共有・集約が完結する点は，運用管理コストの低減とセキュリティの向上に寄与すると考えられる．

第 9 章

評価

本章では，提案するホスト非依存仮想ファイルシステムの各アプローチについて性能評価を行う．

評価に使用した実験環境を表 9.1 に示す．一部の実験については，後述するように Linux 環境下のマシンで実行する必要があったため，同一マシン上に構築した仮想マシン上で実施した．

表 9.1: 実験環境

項目	**ホストマシン (macOS)**	**仮想マシン (Ubuntu)**
CPU	Apple M1 Pro (10 コア)	同左 (4 コア)
アーキテクチャ	arm64	arm64
メモリ	32 GB	4GB
ストレージ	APPLE SSD AP1024R	同左
OS	macOS Tahoe	Ubuntu 24.04.3 LTS
Wasm ランタイム	wasmtime 39.0.1	

9.1　静的合成アプローチの性能試験

静的合成アプローチの比較対象として，ホスト OS 上の物理ディスクを用いたファイルシステムである ext4 と，メインメモリ上に構築されたファイルシステムである tmpfs を用いた．本ファイルシステムはメモリ上で動作するため，tmpfs との比較が最も直接的な性能指標となる．ext4 での測定を実施するため，本項の試験は上述の仮想マシン上で

行った．

9.1.1 計測手法

以下の 3 つの項目について計測を行った．各計測は 5 回実施し，その中央値を採用した．ファイルの書き込みおよび読み込みは WebAssembly アプリケーションから実行した．

- **Sequential Write**: 1MB, 10MB, 100MB のデータを，先頭から順に書き込む際の実行時間とスループットを計測した．
- **Sequential Read**: 書き込み済みの 1MB, 10MB, 100MB のデータを，先頭から順に読み込む際の実行時間とスループットを計測した．
- **Random Read**: 1MB, 10MB, 100MB の各ファイルに対し，4KB の読み込みをランダムな位置に対して 1,000 回繰り返した際の実行時間とスループットを計測した．

9.1.2 評価結果

表 9.2: 静的合成アプローチの性能試験結果 (括弧内はスループット)

項目	**サイズ**	**提案手法: 静的合成アプローチ**	**tmpfs**	**ext4**
Sequential Read	1MB	0.48ms (2,105 MB/s)	0.29ms	1.55ms
	10MB	4.67ms (2,143 MB/s)	1.88ms	7.68ms
	100MB	45.3ms (2,206 MB/s)	31.5ms	46.9ms
Random Read	1MB	2.56ms (1,524 MB/s)	1.29ms	2.59ms
	10MB	2.57ms (1,520 MB/s)	1.60ms	1.65ms
	100MB	2.76ms (1,417 MB/s)	1.85ms	1.88ms
Sequential Write	1MB	0.84ms (1,196 MB/s)	0.32ms	0.31ms
	10MB	7.06ms (1,416 MB/s)	2.61ms	3.83ms
	100MB	67.4ms (1,484 MB/s)	28.9ms	33.8ms

9.1.3 考察

計測結果から，提案手法の静的合成アプローチはホスト OS のネイティブなファイルシステムと比較して一定のオーバーヘッドがあるものの，1GB/s 以上のスループットを維持していることが確認された．

Sequential Read においては，提案手法は 2,100〜2,200 MB/s のスループットを記録した．これは tmpfs の 50〜60% 程度の性能であるが，物理ディスクを用いる ext4 の 100MB 読み込みとほぼ同等の性能であった．Sequential Write においては，提案手法のスループットは tmpfs の約 40% 程度（1,200〜1,500 MB/s）に留まっていた．

tmpfs よりも性能が低くなった理由としては，提案手法においては，Wasm の線形メモリ内でのデータコピーに加え，Component Model を介した関数呼び出しのオーバーヘッドが操作のたびに発生することが考えられる．

9.2 静的合成アプローチと既存実装の性能試験

既存実装である wasi-virt と提案手法の性能比較を行った．wasi-virt は 2026 年 1 月現在読み取り専用の実装であるため，比較項目は読み込み操作に限定した．計測は先述の macOS 環境で実施し，5 回の試行における中央値を採用した．

9.2.1 評価結果

表 9.3: 提案手法と wasi-virt の比較結果 (括弧内はスループット)

項目	**サイズ**	**提案手法: 静的合成アプローチ**	**wasi-virt**
Sequential Read	1MB	0.38ms (2,660 MB/s)	0.44ms
	10MB	4.31ms (2,321 MB/s)	4.43ms
	100MB	51.8ms (1,930 MB/s)	50.5ms
Random Read	1MB	2.19ms (1,785 MB/s)	2.21ms
	10MB	2.49ms (1,571 MB/s)	2.25ms
	100MB	2.45ms (1,592 MB/s)	2.45ms

9.2.2 考察

計測結果から，提案手法は先行研究である wasi-virt と比較して，読み込み性能においてほぼ同等の性能を達成していることが確認された．両手法ともに WebAssembly の線形メモリ上に展開されたデータにアクセスする構造であるため，スループットに決定的な差が生じなかったものと考えられる．

性能面で同等であることを確認した一方で，機能面では大きな差異がある．先行研究である wasi-virt が読み取り専用のファイルシステムに限定されているのに対し，提案手法は書き込み操作および永続化を考慮した設計となっている．したがって，性能を維持しつつ書き込みをサポートした点は，提案手法の重要な優位性であると言える．

9.3 静的合成アプローチ・ホストトレイトアプローチ・RPC ベースアプローチの性能評価

本研究で提案した 3 つの実装手法について，先述の macOS 環境下での性能比較を行った．

9.3.1 評価結果

表 9.4: 3 つの提案手法における性能比較（括弧内はスループット MB/s）

項目	**サイズ**	**静的合成**	**ホストトレイト実装**	**RPC 動的接続**
Seq. Read	1MB	0.45ms (2,205.28)	0.07ms (13,430.21)	90.79ms (11.01)
	10MB	4.64ms (2,156.08)	0.90ms (11,072.15)	112.81ms (88.64)
	100MB	52.88ms (1,890.95)	15.13ms (6,609.71)	420.24ms (237.96)
Seq. Write	1MB	0.62ms (1,605.89)	0.28ms (3,524.23)	48.53ms (20.61)
	10MB	6.36ms (1,573.10)	2.80ms (3,572.28)	51.91ms (192.65)
	100MB	63.39ms (1,577.64)	28.09ms (3,559.52)	176.73ms (565.85)
Rand. Read	1MB	2.37ms (1,650.47)	1.06ms (3,701.58)	24,663.78ms (0.16)

9.3.2 考察

ホストトレイト実装は，全項目において最も高い性能を記録した．特に 1MB の Sequential Read において 13,000 MB/s を超える高いスループットを示している．これは，ファイルシステムのロジックがホスト側のネイティブコードとして実行されるためと考えられる．

RPC 動的接続は, 他の手法と比較して大幅な性能低下が見られた．特に Random Read においては，通信遅延の累積によりスループットが 0.16 MB/s まで低下している．一方で，100MB の Sequential Write においては 500 MB/s を超える速度を記録しており，データサイズが増大するほど通信オーバーヘッドの影響が相対的に小さくなる傾向が確認された．

9.4 リモートストレージ同期機能の評価

本項では，外部ストレージとの同期を想定したユースケースにおいて，本研究で提案した各手法が既存手法と比較してどの程度の性能を示すかを評価する．比較対象として，ホスト OS 上で FUSE を用いて S3 をマウントする手法として広く用いられている s3fs-fuse [36] を用いた．計測は，Wasm アプリケーションから各サイズのファイル読み書きを 5 回実施し，処理に要した時間を測定し中央値を取得した．リモートストレージには，Google Cloud Storage (asia-northeast1 リージョン，レプリケーションなし) を利用した．計測時，ホストトレイト実装・静的合成・RPC 動的合成それぞれは先述の Read Through, Write Through 設定とし，読み書き時に s3fs-fuse (書き込み時に同時に S3 へ書き込み，読み込み時には S3 から読み込む) と同等の挙動になるようにした．性能評価は先述の VM 環境下で実施した．

9.4.1 評価結果

各手法における計測結果を表 9.5 に示す．

表 9.5: S3 同期における各手法の性能比較（実行時間 [ms]）

項目	サイズ	既存手法: s3fs-fuse	ホストトレイト実装	静的合成	RPC 動的接続
Seq. Write	1KB	156ms	143ms	224ms	449ms
	10KB	173ms	151ms	252ms	446ms
	100KB	160ms	144ms	249ms	496ms
	1MB	229ms	165ms	282ms	547ms
Seq. Read	1KB	61ms	86ms	136ms	502ms
	10KB	71ms	85ms	154ms	500ms
	100KB	71ms	88ms	181ms	567ms
	1MB	110ms	99ms	202ms	728ms

9.4.2 考察

計測結果から，本研究で提案したアプローチのうち，ホストトレイト実装は既存手法である s3fs-fuse と比較して同等あるいは特に Write については高い性能を発揮することが確認された．これは，FUSE が S3 へのアクセスだけでなくカーネル空間とユーザ空間の往復を伴うのに対し，WebAssembly のホストトレイトを介した実装では，S3 へのアクセス以外がランタイム内でのメモリ操作のみで完結するため，オーバーヘッドが小さいと考えられる．

静的合成でのアプローチは，ホストに存在するバイナリに依存することなく一定の速度を維持しており，ポータビリティを重視する用途において s3fs-fuse の代替となり得る性能を示した．一方，RPC 動的接続は，TCP のラウンドトリップタイムが大きく，他手法と比較して性能は低く抑えられていた．

以上の評価より，提案する仮想ファイルシステムは，ホスト OS のマウント権限や設定を必要とせずに，既存のリモートストレージマウント手法と同等以上の利便性とパフォーマンスを WebAssembly アプリケーションに提供できることが実証された．

9.5 排他制御に伴うオーバーヘッド

第 6 章で述べたように，ファイルシステムがマルチスレッドで動作する場合は，同一のファイルシステムに対して同時に複数の操作が発生しうるため，排他制御が必要になる．今回実装した fs-core では，thread safe モードとして，Read Lock (共有ロック) と Write Lock (占有ロック) をかける機構を用意している．ここでは，ホストトレイト実装において，複数スレッドから同じファイルを読み書きすることにより競合を発生させたケースと，複数スレッドから異なるファイルを読み書きすることにより競合を発生させないようにしたケースを比較し，排他制御のオーバーヘッドを計測する．

9.5.1 評価結果

計測結果を表 9.6 に示す．本実験では 1MB のデータを用い，スレッド数を 1, 4, 8 と変化させた際のスループット（ops/sec）を測定した．「同一ファイル」は全スレッドが単一の Inode を対象とするケース（RwLock の競合が発生）を，「別ファイル」はスレッドごとに異なる Inode を対象とするケース（DashMap による並行アクセス）を指す．計測は先述の macOS 環境を利用して実施した．

表 9.6: スレッド並列度とアクセス対象による性能比較 [ops/sec]

操作	対象範囲	1 thread	4 threads	8 threads
Read	同一ファイル	1,213	2,321	2,938
	別ファイル	1,027	2,262	2,877
Write	同一ファイル	990	1,660	1,452
	別ファイル	1,124	2,095	2,673

9.5.2 考察

計測結果から，本研究で採用した DashMap と RwLock による二層構造の排他制御が，並列度の向上に対して効果的に機能していることが確認された．

Read 操作におけるスケーラビリティ: Read 操作においては，同一ファイルおよび別ファイルのいずれのケースにおいても，スレッド数の増加に伴いスループットが良好に向上している．これは，Read 操作では Inode 単位の RwLock において共有ロックを取得

するため，同一ファイルに対する同時読み取りがブロックされず，マルチコア CPU の性能を享受できていることを示している．

Write 操作における競合の影響: Write 操作の同一ファイルケースでは，4 スレッドから 8 スレッドへ増大させた際にスループットの低下が確認された．これは占有ロックの取得待ちによるオーバーヘッドが顕在化したためと考えられる．対照的に，スレッドごとに別ファイルを操作するケースでは，8 スレッド時でもスループットが向上し続けている．これは，第一層の管理テーブル（DashMap）により細粒度のロックが実現されているため，ファイルシステム全体としての並行性が維持されていることを裏付けている．

第 10 章

考察とまとめ

10.1　各手法の特性比較と選定指針

本研究で提案した 3 つのアプローチについて，性能，ポータビリティ，データ共有範囲，および導入の容易性の観点から比較を行った結果を表 10.1 に示す．提案した各手法はそれぞれ異なる特性を有しており，ユーザはアプリケーションの要件に応じて最適な手法を選択することができる．

表 10.1: 提案する 3 つのアプローチの特性比較

評価軸	静的合成	ホストトレイト	RPC 動的接続
I/O 性能	○	◎	×
ポータビリティ	◎ (シングル Wasm バイナリ)	○ (ホストにバイナリが必要)	○ (あらかじめサーバを起動しておく必要がある)
データ共有範囲	× (S3 なしでは共有不可)	○ (S3 なしでは同一プロセス内のみ)	◎ (プロセス・ノード間)
永続化 (S3 同期)	○	◎	○
主な用途	一時ファイル利用 設定ファイル同梱	サーバレス環境での ファイルシステム共有	CI/CD キャッシュ 分散システム

10.1.1 静的合成アプローチ

本手法で特筆すべき点は，生成物が純粋な WebAssembly バイナリとなり，特定のランタイム機能に依存しないことである．

第 9 章の評価において，ネイティブファイルシステムと比較しても実用的な速度を維持していることが確認された．したがって，コンパイラの中間ファイル生成や，画像処理における一時バッファなど，完結したタスクをどこでも実行可能にしたい場合に最適な選択肢となる．

10.1.2 ホストトレイト実装アプローチ

提案手法 3 つの中では最も高い性能を示した．特に S3 同期時の Write 性能においては，既存の s3fs-fuse を上回るパフォーマンスを発揮している．

ユースケースとしては，Fermyon Spin のようなサーバレス基盤において，同一ノード上の複数のアプリケーションがデータを高速に共有するパターンに適していると考えられる．

10.1.3 RPC 動的接続アプローチ

性能評価においては，ネットワーク遅延の影響を大きく受け，特にランダムアクセスにおいて著しい低下が見られた．しかし，データ共有範囲がプロセス境界を超えてネットワーク全体に拡張されるため，Kubernetes 上で動的に増減する Pod 間でのデータ共有や，第 7 章で述べた CI/CD パイプラインのような，異なるライフサイクルを持つプロセス間での連携が可能となる．

10.1.4 リモートストレージ同期機能の有効性と FUSE との比較について

リモートストレージとの同期機能に関する評価において，特にホストトレイト実装アプローチは既存の s3fs-fuse と同等かそれ以上の性能を示した．FUSE を利用したファイルシステムは，アプリケーションからカーネル内の VFS を経由してユーザ空間の FUSE デーモンが処理を行う構造上，ユーザ空間とカーネル空間の往復を余儀なくされる．この過程では，システムコールの呼び出しやコンテキストスイッチに加え，メモリ空間を跨ぐデータコピーや FUSE プロトコル特有の処理が累積的に発生し，これが大きなオーバー

ヘッドとなる．対して本研究のホストトレイト実装は，WebAssembly ランタイムとホスト関数が同一プロセス内のユーザ空間で完結しており，コンテキストスイッチのコストが大幅に低減されている．また，FUSE と異なり，設定にホストの特権は必要ない．

また，静的合成アプローチにおいては，OS へのマウント権限を持たない環境であっても，アプリケーション自身が S3 への接続機能を内包することで，実質的な永続化ストレージを利用可能にした．

10.2 実装上の制限事項と今後の展望

10.2.1 オンメモリ構造に起因するメモリ制約

現在の fs-core の実装は，すべてのファイルデータをメモリ上に保持する設計となっている. そのため，いずれのアプローチにおいても，ホストの搭載メモリ容量を超える巨大なファイルの扱いや，大量のファイルを同時に展開するワークロードにおいては，メモリ不足による強制終了のリスクがある．

静的合成アプローチおよび RPC 動的接続アプローチでは，これに加えて Wasm 固有の制約が生じる．これらのアプローチでは fs-core が WebAssembly モジュールとして動作し，ファイルデータが Wasm の線形メモリ上に保持されるため，wasm32 環境における線形メモリの最大サイズである 4GiB の制限を受ける．なお，WebAssembly 3.0 で標準化された memory64 拡張によりこの制限は緩和される [37] が，WASI との併用は 2026 年 1 月現在困難であるほか，利用可能なメモリ量は結局ホストの物理メモリに依存する．ホストトレイト実装アプローチでは，fs-core がネイティブコードとして動作するためこの追加制約は適用されないが，物理メモリによる制限は同様に存在する．

これらの課題に対する解決策としては，全データをメモリに読み込まずに S3 と直接ストリーミングを行う機構の導入や，S3 の特定のディレクトリのみを同期するオプションの追加が考えられる．

10.2.2 RPC 通信におけるパフォーマンス最適化

評価結果が示す通り，RPC 動的接続アプローチにおけるランダムアクセスの性能低下は著しい．これは，微細な読み書き操作のたびにネットワークラウンドトリップが発生していることに起因する．この課題に対しては，Linux のページキャッシュのような機構を RPC Adapter 側に実装することが有効であると考えられる．そのほか，同一ディレクトリ内のデータを一括して取得するなどのプリフェッチを実装する等の方法で，通信回数を

削減し，ネットワーク遅延の影響を最小限に抑える改善が求められる．

10.2.3　分散環境における排他制御の強化

現在の S3 同期機能は，単一の書き込み主体に対しては整合性を保つが，分散した複数のノードから同一ファイルに対して同時に書き込みが行われた場合の競合解決策が不十分である．分散システム向けのロックマネージャを導入するほか，書き込み権限の集約による役割の分離や，クォーラムベースの合意形成アルゴリズムの適用など，分散環境下で厳密なデータ整合性を担保するための機構構築が今後の課題となる．

10.3　結論

本稿では，WebAssembly のサンドボックス特性とポータビリティを損なうことなく，柔軟なファイルシステムアクセスを提供するアーキテクチャを設計・実装した．提案したホスト非依存ファイルシステムは，WebAssembly をサーバレスやエッジコンピューティングといった領域に適用する上での有用な基盤技術となりうる．今後，WebAssembly エコシステムの成熟とともに，このような抽象化レイヤの重要性はますます高まっていくと考えられる．

第 11 章

謝辞

本研究の遂行にあたり，多大なるご支援をいただいた方々に深く感謝いたします．

まず，指導教員である宇多准教授には，研究の方向性に関する的確なご示唆をいただいたほか，議論を通じて複雑な課題を整理する機会を賜りました．先生の深い知見と熱心なご指導がなければ，本論文を完成させることはできませんでした．厚く御礼申し上げます．

また，副査を務めていただいた諸先生方には，本論文の査読において有益なコメントや改善に向けた貴重なご助言をいただきました．審査を通じて得られた知見は，本研究の質を向上させる上で極めて重要なものとなりました．

さらに，研究室のメンバーには，ゼミでの議論や実装上の課題に対するアドバイスをいただきました．心より感謝いたします．

最後に，本研究を支えてくださったすべての方々に，この場を借りて深く感謝の意を表します．

付録 A

fs-core 詳細設計仕様

A.1 データ構造定義

A.1.1 型定義

fs-core で使用される基本型を表 A.1 に示す.

表 A.1: 型エイリアス定義

型名	実体	説明
`Fd`	`u32`	ファイルディスクリプタ. 0–2 は標準入出力用.
`InodeId`	`u64`	Inode の一意識別子.
`InodeRef`	`Arc<RwLock<Inode>>`*	スレッドセーフな Inode 参照.
`FdTable`	`DashMap<Fd, FileHandle>`*	ファイルディスクリプタテーブル.
`InodeTable`	`DashMap<InodeId, InodeRef>`*	Inode テーブル.

* thread-safe feature 無効時は `Rc<RefCell>` および `BTreeMap` を使用.

A.1.2 オープンフラグ

ファイルオープン時に指定可能なフラグを表 A.2 に示す.

表 A.2: オープンフラグ定義

定数名	値	説明
O_RDONLY	0x0	読み取り専用モード.
O_WRONLY	0x1	書き込み専用モード.
O_RDWR	0x2	読み書き両用モード.
O_CREAT	0x40	ファイルが存在しない場合は作成.
O_TRUNC	0x200	オープン時にファイルを空にする.
O_APPEND	0x400	追記モード．書き込みは常にファイル末尾.

A.1.3 Fs 構造体

ファイルシステム全体を管理する中核構造体である．ファイルディスクリプタテーブルと Inode テーブルを保持し，すべてのファイル操作の起点となる.

表 A.3: Fs 構造体定義

フィールド名	型	説明
next_inode	AtomicU64	次に割り当てる Inode ID．アトミック操作により一意性を保証.
next_fd	AtomicU32	次に割り当てるファイルディスクリプタ番号．0–2 は標準入出力用に予約.
fd_table	FdTable	オープン中の全 FileHandle を管理する DashMap. キーは Fd.
inode_table	InodeTable	全 Inode を管理する DashMap．キーは InodeId.
root_inode	InodeId	ルートディレクトリの Inode ID（常に 0）.
time_provider	T	タイムスタンプ生成器．テスト時にモック可能.

A.1.4　Inode 構造体

ファイルシステム上のすべてのオブジェクト（ファイルおよびディレクトリ）を表現する基本構造体である．thread-safe feature 有効時は Arc<RwLock<Inode>>，無効時は Rc<RefCell<Inode>> としてラップされ，環境に適したアクセス制御が行われる．

表 A.4: Inode 構造体定義

フィールド名	型	説明
id	u64	ファイルシステム内で一意な識別子 (Inode ID).
metadata	Metadata	ファイルサイズ，作成日時，権限情報などの属性情報.
content	FileContent	ファイルの実体．列挙型により以下のいずれかを保持する． - File: ブロック単位で管理されるバイトデータ (BlockStorage) - Dir: 子エントリのマップ (BTreeMap<String, InodeId>)

A.1.5　Metadata 構造体

ファイルまたはディレクトリのメタ情報を保持する構造体である．

表 A.5: Metadata 構造体定義

フィールド名	型	説明
size	u64	ファイルサイズ（バイト単位）．ディレクトリの場合は 0.
created	u64	作成時刻（UNIX タイムスタンプ，ミリ秒）.
modified	u64	最終更新時刻（UNIX タイムスタンプ，ミリ秒）.
permissions	u16	UNIX パーミッション（例: 0o644）.
is_dir	bool	ディレクトリフラグ．true ならディレクトリ．

A.1.6　FileHandle 構造体

プロセスが開いているファイルの「状態」を管理する構造体である．同一の Inode に対して複数の FileHandle が存在しうる．

表 A.6: FileHandle 構造体定義

フィールド名	型	説明
`inode_id`	`u64`	操作対象となる Inode への参照 ID．
`position`	`AtomicU64`*	現在の読み書きカーソル位置．スレッドセーフ環境ではアトミック整数により並行アクセス時の整合性を担保する．
`flags`	`u32`	`O_RDONLY`, `O_APPEND` などのオープンフラグ．

* `thread-safe` feature 無効時は `Cell<u64>` を使用．

A.1.7　BlockStorage 構造体

ファイルの実データを管理する構造体である．4KB 単位のブロックで管理され，スパースファイル（穴あきファイル）をサポートする．

表 A.7: BlockStorage 構造体定義

フィールド名	型	説明
`blocks`	`Vec<Option<Box<Block>>>`	4KB ブロックの配列．`None` はスパース領域（読み取り時はゼロを返す）．
`size`	`usize`	論理ファイルサイズ．実際の割り当てサイズとは異なりうる．

ここで `Block` は `[u8; BLOCK_SIZE]` の型エイリアスであり，`BLOCK_SIZE` は 4096 バイト（4KB）である．

A.2 クラス構造図

本システムの中核となるファイルシステム **fs-core** のクラス構造を図 A.1 に示す.

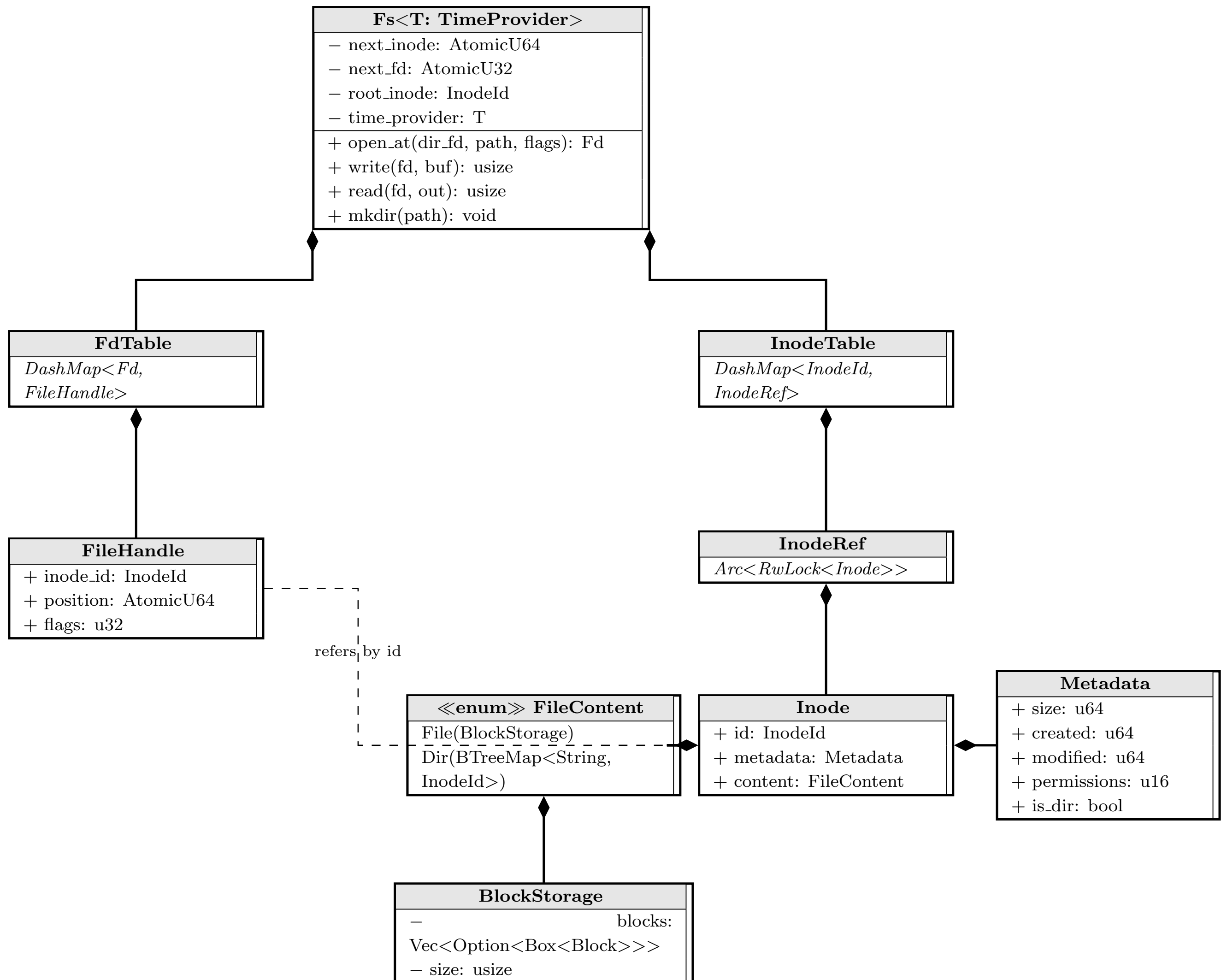

図 A.1: fs-core クラス構成図 (UML Class Diagram, thread-safe feature 有効時)

A.3 インターフェース仕様

fs-core が提供する主要な API 仕様を以下に示す.

表 **A.8:** 主要メソッド仕様一覧

メソッド名	シグネチャ・引数・機能説明
open_at	fn open_at(&self, dir_fd: Fd, path: &str, flags: u32) -> Result<Fd, FsError> **概要**: 指定されたディレクトリ記述子を起点としてパスを解決し，ファイルを開く． **引数**: dir_fd (起点), path (相対パス), flags (動作フラグ)
open_path_with_flags	fn open_path_with_flags(&self, path: &str, flags: u32) -> Result<Fd, FsError> **概要**: ルートディレクトリを起点としてパスを解決し，ファイルを開く．
write	fn write(&self, fd: Fd, buf: &[u8]) -> Result<usize, FsError> **概要**: ファイル記述子の現在のカーソル位置にデータを書き込む． **挙動**: 書き込み完了後，カーソル位置は更新される．
read	fn read(&self, fd: Fd, out: &mut [u8]) -> Result<usize, FsError> **概要**: 現在のカーソル位置からデータを読み込む． **戻り値**: 読み込んだバイト数 (0 は EOF を意味する)．
append_write	fn append_write(&self, fd: Fd, buf: &[u8]) -> Result<usize, FsError> **概要**: カーソル位置に関わらず，ファイル末尾にアトミックに追記する． **用途**: ログ出力など，複数プロセス間の競合回避に使用．
close	fn close(&self, fd: Fd) -> Result<(), FsError> **概要**: ファイル記述子を解放する．

次ページへ続く...

（前ページからの続き）

メソッド名	シグネチャ・引数・機能説明
seek	fn seek(&self, fd: Fd, offset: i64, whence: i32) -> Result<u64, FsError> **概要**: 読み書きカーソルを移動する． **引数**: whence は 0: 先頭, 1: 現在地, 2: 末尾 を指定.
ftruncate	fn ftruncate(&self, fd: Fd, size: u64) -> Result<(), FsError> **概要**: ファイルサイズを指定バイト数に変更（切り詰め/ゼロ埋め拡張）する．
stat	fn stat(&self, path: &str) -> Result<Metadata, FsError> **概要**: 指定されたパスのメタデータを取得する．
fstat	fn fstat(&self, fd: Fd) -> Result<Metadata, FsError> **概要**: 開いているファイル記述子のメタデータを取得する．
mkdir	fn mkdir(&self, path: &str) -> Result<(), FsError> **概要**: 新しいディレクトリを作成する．親が存在しない場合はエラー．
mkdir_p	fn mkdir_p(&self, path: &str) -> Result<(), FsError> **概要**: 中間ディレクトリを含めて再帰的にディレクトリを作成する．
unlink	fn unlink(&self, path: &str) -> Result<(), FsError> **概要**: 指定されたファイルを削除する (ディレクトリは不可).

次ページへ続く...

（前ページからの続き）

メソッド名	シグネチャ・引数・機能説明
rmdir	`fn rmdir(&self, path: &str) -> Result<(), FsError>` **概要**: 指定された空のディレクトリを削除する.
readdir	`fn readdir(&self, path: &str) -> Result<Vec<String>, FsError>` **概要**: ディレクトリ内のエントリ名一覧を取得する.
readdir_fd	`fn readdir_fd(&self, fd: Fd) -> Result<Vec<(String, bool)>, FsError>` **概要**: ディレクトリ記述子を用いてエントリ一覧を取得する. (種別情報付き)

A.4 処理シーケンス詳細

一例として，書き込み操作 (`write`) における内部処理フローとロック制御のシーケンスを図 A.2 に示す．デッドロックを回避するため，ファイル記述子テーブルの参照および Inode のロック取得を段階的に行う設計とした.

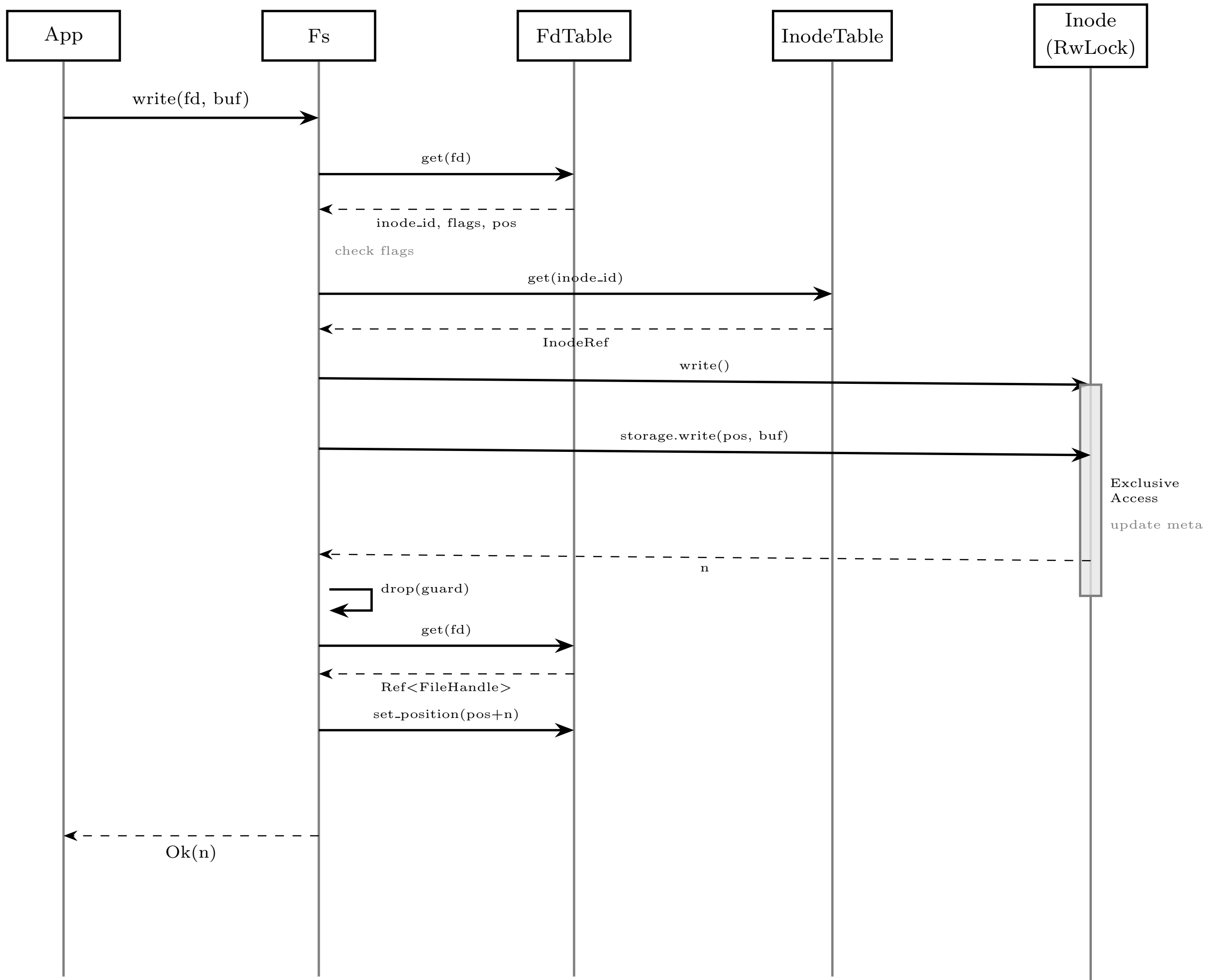

図 A.2: 書き込み処理のシーケンス図

参考文献

[1] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, and Alon Zakai. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*, pp. 185–200, 2017.

[2] Pankaj Mendki. Evaluating WebAssembly enabled serverless approach for edge computing. In *2020 IEEE Cloud Summit*, pp. 165–170, 2020.

[3] Phani Kishore Gadepalli, Gregor Peach, Ludmila Cherkasova, Rob Aitken, and Gabriel Parmer. Challenges and opportunities for efficient serverless computing at the edge. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, pp. 261–265, 2019.

[4] Natuonal Vulnerability Database. Cve-2023-51661 detail. `https://nvd.nist.gov/vuln/detail/CVE-2023-51661`. Accessed: 2025-12-31.

[5] eumonia-bpf org. WASI and the WebAssembly component model: Current status. `https://eunomia.dev/blog/2025/02/16/wasi-and-the-webassembly-component-model-current-status/`. Accessed: 2025-12-31.

[6] Http component sources do not work on windows. `https://github.com/spinframework/spin/issues/2112`, November 2023.

[7] Yixuan Zhang, Mugeng Liu, Haoyu Wang, Yun Ma, Gang Huang, and Xuanzhe Liu. Research on WebAssembly runtimes: A survey. *ACM Trans. Softw. Eng. Methodol.*, Vol. 34, No. 8, pp. 1–47, November 2025.

[8] WebAssembly Community Group. Modules — WebAssembly 3.0 (2025-12-08). `https://webassembly.github.io/spec/core/syntax/modules.html`.

Accessed: 2025-12-31.

[9] Daniel Lehmann, Johannes Kinder, and Michael Pradel. Everything old is new again: Binary security of WebAssembly. In *Proceedings of the 29th USENIX Security Symposium*, pp. 217–234, 2020.

[10] WebAssembly Community Group. Security - webassembly. `https://webassembly.org/docs/security/`. Accessed: 2025-12-31.

[11] Elliott Wen and Gerald Weber. Wasmachine: Bring IoT up to speed with a WebAssembly OS. In *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pp. 1–4. IEEE, March 2020.

[12] Rafael Belchior, André Vasconcelos, Sérgio Guerreiro, and Miguel Correia. A survey on blockchain interoperability: Past, present, and future trends. *ACM Comput. Surv.*, Vol. 54, No. 8, pp. 1–41, November 2022.

[13] Phani Kishore Gadepalli, Sean McBride, Gregor Peach, Ludmila Cherkasova, and Gabriel Parmer. Sledge: A serverless-first, light-weight wasm runtime for the edge. In *Proceedings of the 21st International Middleware Conference*, New York, NY, USA, December 2020. ACM.

[14] Philipp Gackstatter, Pantelis A Frangoudis, and Schahram Dustdar. Pushing serverless to the edge with WebAssembly runtimes. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pp. 140–149. IEEE, May 2022.

[15] Lin Clark. Announcing the bytecode alliance: Building a secure by default, composable future for WebAssembly. `https://bytecodealliance.org/articles/announcing-the-bytecode-alliance`, November 2019. Accessed: 2026-1-24.

[16] Mugeng Liu, Haiyang Shen, Yixuan Zhang, Hong Mei, and Yun Ma. WebAssembly for container runtime: Are we there yet? *ACM Trans. Softw. Eng. Methodol.*, Vol. 34, No. 6, pp. 1–22, July 2025.

[17] Zhen Wang, Jianda Wang, Zhendong Wang, and Yang Hu. Characterization and implication of edge WebAssembly runtimes. In *2021 IEEE 23rd Int Conf on High Performance Computing & Communications; 7th Int Conf on Data Science & Systems; 19th Int Conf on Smart City; 7th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*, pp. 71–80. IEEE, December 2021.

[18] M J Jones. How compute is tackling serverless cold starts, regional latency, and observability. https://www.fastly.com/blog/how-compute-edge-is-tackling-the-most-frustrating-aspects-of-serverless, December 2024. Accessed: 2025-12-31.

[19] Matt Butcher. The scale to zero problem. https://www.fermyon.com/blog/scale-to-zero-problem, April 2022. Accessed: 2025-12-31.

[20] Spin-executor/sleep-1ms/concurrency-1 - criterion.rs. https://fermyon.github.io/spin-benchmarks/criterion/reports/spin-executor_sleep-1ms/concurrency-1/index.html. Accessed: 2026-1-31.

[21] Radu Matei. Spin 1.0 — the developer tool for serverless WebAssembly. https://www.fermyon.com/blog/introducing-spin-v1, March 2023. Accessed: 2025-12-31.

[22] wasmtime: A lightweight WebAssembly runtime that is fast, secure, and standards-compliant. https://github.com/bytecodealliance/wasmtime.

[23] wasm-micro-runtime: WebAssembly micro runtime (WAMR). https://github.com/bytecodealliance/wasm-micro-runtime.

[24] runwasi: Facilitates running wasm / WASI workloads managed by containerd. https://github.com/containerd/runwasi.

[25] Wasm extensions. https://gateway.envoyproxy.io/docs/tasks/extensibility/wasm/. Accessed: 2025-12-31.

[26] How data transforms work. https://docs.redpanda.com/current/develop/data-transforms/how-transforms-work/, December 2025. Accessed: 2025-12-31.

[27] Code engine - powered by wasm · SingleStore helios documentation. https://docs.singlestore.com/cloud/reference/code-engine-powered-by-wasm/. Accessed: 2025-12-31.

[28] Introduction. https://wasi.dev/. Accessed: 2025-12-31.

[29] Wasihost_core::Wasi_snapshot_preview1 - rust. https://wasi.bchlr.de/wasihost_core/wasi_snapshot_preview1/. Accessed: 2025-12-31.

[30] wasi-libc: WASI libc implementation for WebAssembly. https://github.com/WebAssembly/wasi-libc.

[31] Preview_1.rs - source. https://docs.wasmtime.dev/api/src/wasi_common/snapshots/preview_1.rs.html. Accessed: 2025-12-31.

[32] Dan Gohman. WASI 0.2 launched. https://bytecodealliance.org/articles/WASI-0.2, January 2024. Accessed: 2025-12-31.

[33] Introduction - the WebAssembly component model. https://component-model.bytecodealliance.org/introduction.html. Accessed: 2025-12-31.

[34] Yuta Saito. wasi-vfs: A virtual filesystem layer for WASI. https://github.com/kateinoigakukun/wasi-vfs.

[35] WASI-virt: Virtual implementations of WASI APIs. https://github.com/bytecodealliance/WASI-Virt.

[36] s3fs-fuse: FUSE-based file system backed by amazon S3. https://github.com/s3fs-fuse/s3fs-fuse.

[37] WebAssembly/memory64. https://github.com/WebAssembly/memory64/blob/main/proposals/memory64/Overview.md. Accessed: 2026-1-24.