2024 年度

筑波大学情報学群情報科学類

卒業研究論文

題目

ユーザ空間並列ファイルシステムのための システムコールフックライブラリの設計と評価

主専攻　情報システム主専攻

著者　宮内 遥楓

指導教員　建部 修見

要旨

ユーザー空間並列ファイルシステムは，ストレージシステムの性能を向上させるために開発されてきた．一方，POSIX インタフェースは，標準として長い間アプリケーションに使用されてきた．ユーザ空間ファイルシステム上でアプリケーションを書き換えずに動作させるためには POSIX インタフェースへの対応が必要であるが，既存手法の FUSE やプリロードライブラリには問題がある．FUSE は設計上パフォーマンスの低下が避けられず，既存のプリロードライブラリでは標準ライブラリ (glibc) を利用しないアプリケーションを動作させることができない．

本研究では，アプリケーションがファイルシステムにアクセスする際は必ずシステムコールを発行する点に着目し，アプリケーションが呼び出すシステムコールをユーザ空間ファイルシステムの API 呼び出しに置き換えるシステムコールフックライブラリを設計する．ライブラリの実装においてはバイナリ書き換えに基づくシステムコールフック機構である zpoline を利用する． 実装したシステムコールフックライブラリに対する評価実験の結果，ユーザ空間ファイルシステムの API を直接呼び出した場合と比較して性能がほぼ低下しないこと，既存手法の FUSE を使用した場合と比較して大幅に性能が向上することを確認した．

目 次

図 目 次

コード目次

第1章　序論

1.1　概要

近年のアプリケーションは，CPU の演算能力がボトルネックとなる演算指向に代わり，データの量や複雑さ，変化の速度が課題となるデータ指向のものが多くを占める．スーパコンピュータの主な用途である科学計算シミュレーションや AI においても，膨大な量のデータセットを処理することが要求されつつある．一例として，2020 年に OpenAI が発表した言語モデルである GPT-3 は 570GB 以上のテキストデータを学習データセットとして使用している [1]．CPU や GPU がデータを処理するためにはメモリにデータを載せる必要があるが，メモリの容量以上のデータセットを全部載せることはできないため，前述のワークロードにおいては何度もストレージからのデータ読み出しを行うことになる．また計算途中の状態をチェックポイントとして記録するため，ストレージへのデータ書き込みも頻発する．そのためストレージ性能の向上はますます重要になる．

ストレージ性能の向上のために，ファイルシステムの研究が進められている．ファイルシステムはアプリケーションがストレージにアクセスする際のインタフェースであり，ファイルシステムの設計を工夫することでストレージ性能の向上を図ろうとしている．

新しく提案されるファイルシステムの多くがユーザ空間で実装されている．ユーザ空間ファイルシステムをアプリケーションから利用するには，ファイルシステム API の直接呼び出し，FUSE，プリロードライブラリの利用の 3 つの方法がある．このうちファイルシステム API の直接呼び出しについては，アプリケーションのソースコードの変更と再コンパイルが必要になり，ソースコードが手に入らないアプリケーションを動かすことができない．もしソースコードがあったとしても，ファイルシステムごとに独自の API の仕様を把握し，適切に API を呼び出すよう変更を加えるのは容易ではない．現在主流となっているのが FUSE である．FUSE を利用することでユーザ空間ファイルシステムを手軽にマウントすることができ，通常のファイルシステムと同じようにアプリケーションから I/O 操作を実行できる．しかし FUSE は設計上ストレージ性能の低下を避けられないため，演算性能だけでなくストレージ性能も重要になる HPC アプリケーションではあまり利用されない．

そこで本研究ではプリロードライブラリの設計を提案する．プリロードライブラリは，LD_PRELOAD 環境変数を利用してアプリケーション実行時に読み込まれることで，通常のファイルシステムと同じようにアプリケーションから I/O 操作を実行できることを目的としている．いくつかのユーザ空間ファイルシステムでは既にプリロードライブラリを提供しており，標準ライブラリ (glibc)、または標準ライブラリ内システムコールの I/O 関連の関数をファイルシステム API 呼び出しに置き換えている．I/O 関連の関数を完全にファイルシステム API 呼

び出しに置き換えられることができれば，ファイルシステム API の直接呼び出しとほぼ同等のパフォーマンスを実現することができる．しかし既存のプリロードライブラリは I/O 関連の関数の完全な置き換えには至っておらず，アプリケーションによっては実行できない．この問題を解決するため，全てのシステムコールをファイルシステム API 呼び出しに置き換えるシステムコールフックライブラリを本研究で提案・実装する．

1.2 本稿の構成

本稿の構成は以下のとおりである． 第 2 章では，本研究の背景について述べ，提案手法の理解のために重要であるユーザ空間ファイルシステムについて解説する． 第 3 章では，アプリケーションがユーザ空間ファイルシステムを利用するための既存の手法について整理する．第 4 章では，提案手法の設計と実装について説明する． 第 5 章では，提案手法の評価実験とその結果について述べる． 第 6 章では，評価実験の結果を踏まえた提案手法の考察と今後の展望をまとめる．

第2章　背景

2.1　ファイルシステム

ファイルシステムは，アプリケーションがデータにアクセスするための共通のインタフェースである．

アプリケーションがファイルシステムを利用するときの一般的な流れを，図 2.1 を用いて説明する．アプリケーションがファイルを開くとき，Linux 標準ライブラリ (libc) の open 関数を実行する．open 関数は open システムコールのラッパー関数であり，実行するといくつかの前処理を行った後に open システムコールを呼び出す．ここでシステムコールはアプリケーションがカーネル空間のリソース (メモリ，ストレージ) にアクセスするための手段で，呼び出されると割り込みが発生し，カーネル空間に制御が移る．カーネル空間に制御が移ると，VFS(Virtual File System) によって実際のファイルシステムへのファイルオープン要求に変換される．VFS は複数のファイルシステムを共存させるためのインタフェースで，これによりファイルシステムごとの細かな違いを意識することなく共通の方法でファイルやディレクトリに対する基本的な操作を行うことができる．VFS によって適切なファイルシステムドライバにファイルオープン要求が送られ，実際のストレージへのアクセスが行われる．

I/O 操作を行うときは標準ライブラリの関数を通常呼ぶが，インラインアセンブラを利用してシステムコールを直接発行することでファイルシステムにアクセスすることもできる．例えば Go 言語は libc を利用せずアセンブリコードとしてシステムコールの呼び出しを実装している．

2.2　ユーザ空間ファイルシステム

従来，ファイルシステムはカーネルコードの書き換えやカーネルモジュールの追加によって OS カーネル内に実装されてきた．しかしあらゆるアプリケーションに対応するため，ファイルシステムの複雑性は年々増している．例として並列ファイルシステムの GPFS [2] のソースコードは百万行を超える [3]．カーネルコードにバグがあった場合，データの破損や脆弱性など様々な問題を引き起こす可能性があるが，年々複雑になるファイルシステムを保守し続けるのは容易ではない．そこでファイルシステムをカーネル内で実装するのではなく，ユーザ空間上のプロセスとして動作するファイルシステムを開発することで，この問題を回避することができる．

ユーザ空間でファイルシステムを開発する利点として，開発の容易さ，信頼性，移植性，パ

フォーマンスがあげられる．ユーザ空間でバグが発生してもシステム全体をクラッシュさせたり破壊したりすることがなく，開発者にとって安全性が高い．また，カーネル開発が通常 C または C++に限定されるのに対し，ユーザ空間では任意のプログラミング言語を使用できる．さらに，カーネルで実行されるコードを減らすことで，カーネルのバグが本番システム全体をクラッシュさせる可能性を減らすことにも繋がる．このほか，ユーザ空間のファイルシステムを他の OS に移植するのはカーネルコードよりも簡単に行うことが可能である．パフォーマンスの面においても，ユーザ空間でより効率的なアルゴリズムを実装した新しいライブラリを使用できる．

またファイルシステムをユーザ空間で提供することは，開発者だけでなくユーザにもメリットがある．ファイルシステムの中には特定のアプリケーションに特化したものがあり，それらのファイルシステムをアプリケーションから利用できれば大幅な性能向上につながる可能性がある．しかしファイルシステムの追加はカーネルの変更に当たるため，特に複数人が利用する計算機システムにおいて，自由にファイルシステムを追加することはできない．対してユーザ空間ファイルシステムであればカーネルの変更なくアプリケーションから利用することができるため，画期的な実装のファイルシステムを試すことが容易に可能である．

このような背景から，多くの新しいファイルシステムがユーザ空間で開発されるようになった．特に，産業や学術研究において，ファイルシステム設計の新しいアプローチのプロトタイプ作成や評価を迅速に行うために，ファイルシステムがユーザ空間で開発されている．またカーネルの機能のうち，プロセス管理，スケジューラ，メモリ管理といった必要最低限の機能を実装するマイクロカーネルアーキテクチャでは，ファイルシステムをユーザ空間で提供する仕組みが採用されている．

2.3 並列ファイルシステム

HPC アプリケーションは計算の高速化のため並列化を行うことが多く，その際のデータ読み出しで多数の I/O リクエストが発生する．通常のファイルシステムでは 1 つのサーバがリクエストに対応するが，リクエストの量によっては対応しきれずボトルネックの原因になる．そこで複数のサーバ，または複数のストレージを仮想的に 1 つのストレージとして扱えるようすることで高いストレージ性能を達成する並列ファイルシステムが HPC 分野ではよく使用される．

2.4 POSIX

POSIX（Portable Operating System Interface）は UNIX 系 OS 間でアプリケーションの移植性を高めるために定義された IEEE の標準規格である．POSIX に準拠して開発されたアプリケーションは，POSIX 準拠の OS ならばどれでも同じように動作させることができる．

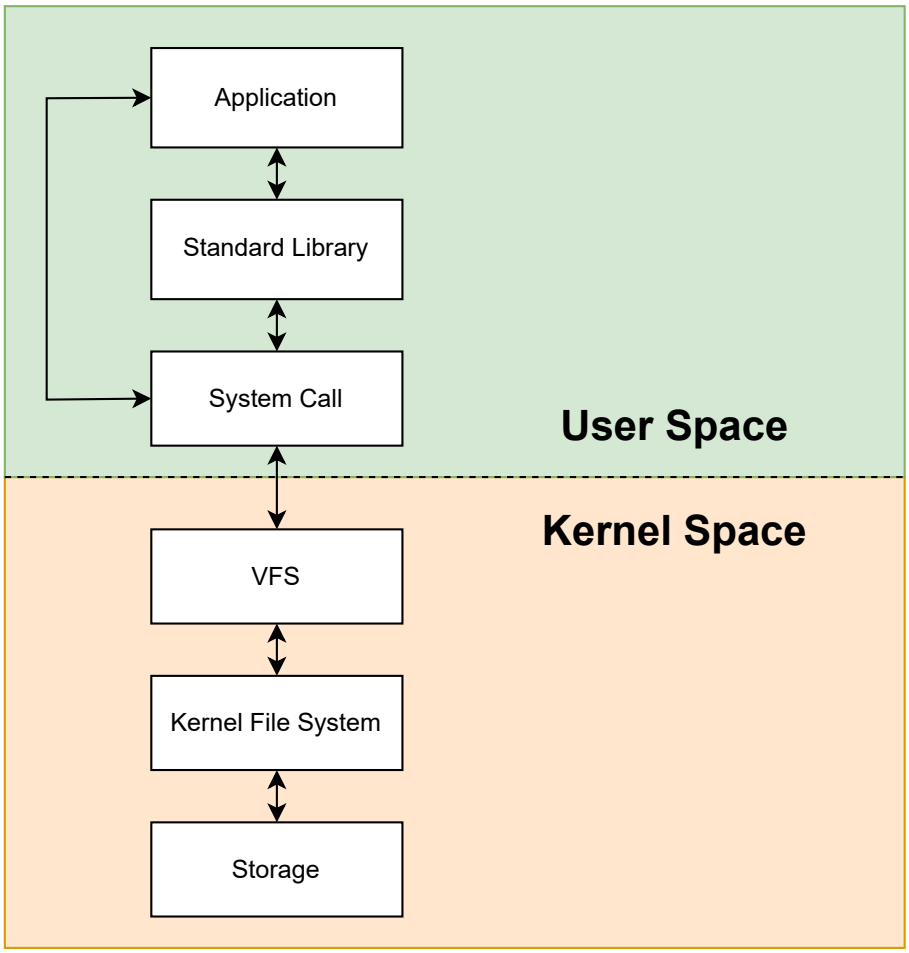

図 2.1: ファイルシステム概念図

第3章　既存手法

前章で説明したように，アプリケーションがファイルに何らかの操作を実行するときには必ずシステムコールを呼び出し，制御がカーネル空間に移った後 VFS を経由してファイルシステムにアクセスする．カーネル空間ファイルシステムと違い，ユーザ空間ファイルシステムは OS に登録することが簡単にはできないため，アプリケーションの書き換えなしにはユーザ空間ファイルシステムを利用できない．この問題の既存の解決方法として，FUSE とプリロードライブラリが挙げられる．

3.1　FUSE

FUSE(Filesystem in Userspace) はユーザ空間ファイルシステムを開発するときに最も使用されているフレームワークである．FUSE を利用することで容易にユーザ空間ファイルシステムをマウントすることができるため，SSHFS [4]，GlusterFS [5]，ZFS [6] など数多くの FUSE ベースのファイルシステムが開発されており，その数は 100 を超える．

アプリケーションが FUSE を介してユーザ空間ファイルシステムにアクセスする過程を図 3.1 に示す．アプリケーションが標準ライブラリを通じてシステムコールを発行し，VFS に到達するまでは通常のファイルシステムアクセスと同じである．VFS には FUSE ドライバが登録されており，このドライバとユーザ空間ファイルシステムが/dev/fuse ブロックデバイスを介して通信する．この仕組みによりカーネルに変更を加えることなくファイルシステムを追加することができる．

しかしながら，アプリケーションと FUSE プロセスとの通信コストや，コンテキストスイッチによるオーバーヘッドが原因の性能低下により，特にパフォーマンスが求められる HPC 分野での利用は適していないと論じられている [7]．そのため FUSE から派生したより効率的なユーザ空間ファイルシステムフレームワークの研究も多くされている [8, 9, 10].

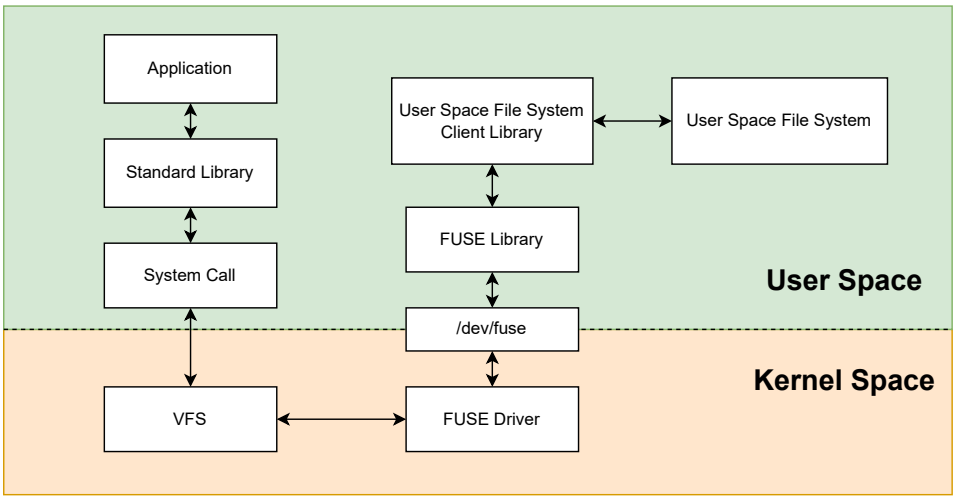

図 3.1: FUSE アーキテクチャ

3.2 プリロードライブラリ

Linux には共有ライブラリの関数をフックできる LD_PRELOAD 環境変数が存在する．LD_PRELOAD に共有ライブラリのパスを指定すると，プログラム起動時に動的リンカが対象ライブラリを優先的にリンクする．これにより，ある関数が LD_PRELOAD に指定したライブラリに存在する場合，他のライブラリの実装より優先して呼び出されるため，既存関数の実装差し替えや，main 関数実行前に任意の関数を実行することが可能になる．各ユーザ空間ファイルシステムが提供するプリロードライブラリを LD_PRELOAD で指定してアプリケーションを実行することで，ソースコードの変更や再コンパイルなしにユーザ空間ファイルシステムを利用することが可能になる．しかしアプリケーションが呼び出す I/O 関数を網羅的にフックすることは容易ではないため，プリロードライブラリの作成には次に示すライブラリが使用されることが多い．

Gotcha [11] は任意の関数をラップするライブラリで，LD_PRELOAD に似ているが，プログラム可能な API を介して動作する．UnifyFS [12] は Gotcha を利用して libc(C 標準ライブラリ) の I/O 関連の関数をフックし，UnifyFS の API に置き換えることで POSIX インタフェースの対応を行っている．しかしアプリケーションが libc を経由せず直接システムコールを呼び出す場合，例えばインラインアセンブラを利用して呼び出されたシステムコールに対応できない．

syscall_intercept [13] は，メモリにロードされた glibc のテキストセグメント中の syscall 命令を検知し，jmp 命令に置き換えることで任意の関数呼び出しを可能にするライブラリである．syscall_intercept を使用して POSIX インタフェース対応をしているユーザ空間ファイルシステムには GekkoFS [14] がある．glibc の関数をフックする場合と比較して，少数のシステムコールフックを定義するだけで済むため，ファイルシステムが提供するプリロードライブラリの実装は相対的に容易である．しかし syscall_intercept は Linux の glibc にしか対応していないため，Linux 以外での OS では動作しない可能性がある．

第4章　提案手法

本研究では，ユーザ空間ファイルシステムにおける POSIX インタフェース対応のために，前章で既存手法として挙げたプリロードライブラリの新しい設計を提案する．既存手法として挙げたライブラリはどちらもアプリケーションが glibc を利用して I/O 操作を行うことを前提としており，アプリケーションの実装によっては I/O 関数をフックできずに正常な動作が行われない可能性がある．一方，標準ライブラリを使用しないアプリケーションであっても，I/O 操作は最終的にはカーネルのリソースを利用することになるため，必ずシステムコールを発行する．そのため I/O 操作に関連するシステムコールをファイルシステムの API に置き換えることができれば，全てのアプリケーションを動作させることができると考えられる．

この考察を基にして，ユーザ空間ファイルシステム上で動作するアプリケーションによって呼び出されるシステムコールをフックするライブラリを設計する．また提案手法の評価のため，ユーザ空間並列ファイルシステムである CHFS [15] の POSIX インタフェース対応を目的としたシステムコールフックライブラリを実装する．ライブラリの実装においては，システムコールフックメカニズムの zpoline [16] を利用する．

4.1　システムコールフックライブラリの設計

システムコールフックライブラリの設計を図 4.1 を用いて説明する．アプリケーションがユーザ空間ファイルシステムにアクセスするエントリポイントとして，仮想的なマウントポイント/virtual_mount_point を仮定する．アプリケーションから呼び出されたシステムコールに含まれるパス名が/virtual_mount_point で始まる場合，そのファイルはユーザ空間ファイルシステムの管理下にあると判断し，システムコールの呼び出しをユーザ空間ファイルシステムのクライアントライブラリの API 呼び出しに置き換える．そうでない場合は本来のシステムコールを呼び，通常通り VFS を経由してカーネル空間のファイルシステムにアクセスする．

システムコールがフックされた場合，本来のシステムコールは呼ばれないためコンテキストスイッチが発生しない．そのため FUSE で発生しているようなコンテキストスイッチに起因するオーバヘッドを削減することができ，ユーザ空間ファイルシステムを効率的に利用することが可能になる．

4.2 zpoline

zpoline は安形らによって提案された，システムコールを高速にフックする手法である．既存手法で例に挙げた syscall_intercept には glibc 内で呼び出されるシステムコールしかフックできないという問題があったが，zpoline ではバイナリ書き換えとトランポリンコードを組み合わせることでこの問題を解決し，全てのシステムコールをフックすることを可能にした．

zpoline がシステムコールをフックする仕組みを図 4.2 を用いて説明する．zpoline は LD_PRELOAD でロードされることを想定したライブラリとして実装されており，アプリケーションの main 関数が実行されるよりも前にシステムコールフックの準備を行う．

プログラムのバイナリがメモリにロードされると，zpoline はまずロードされたバイナリの書き換えを実行する．ユーザ空間プログラムがカーネルの機能を利用する際には必ずシステムコールを発行する必要がある．そのために使われる syscall / sysenter 命令は 2byte の機械語命令であり，システムコールを他の関数呼び出しに置き換えるには syscall / sysenter 命令を別の機械語命令に書き換えてやればよい．しかし 3byte 以上の機械語命令で書き換えてしまうと syscall / sysenter 命令の次の命令を上書きしてしまい，プログラムが正常に動作しなくなるため，2byte という非常に厳しい制約のもと命令を書き換える必要がある．zpoline は callq *%rax という 2byte の命令で syscall / sysenter 命令を書き換える．callq *%rax は%rax レジスタに入ったアドレスにジャンプし，ジャンプ元のアドレスをスタックへプッシュするという命令である．ここで Linux のシステムコールの呼出規約では%rax レジスタにシステムコール番号を入れておく必要がある．システムコール番号はカーネルの定義により高々500 に収まる数のため，callq *%rax が実行されるとアドレス 0 ~500 にジャンプする．

次に zpoline はメモリアドレス 0 から最大のシステムコール番号+1 の位置に，トランポリンコードと呼ばれる任意のフック関数にジャンプするためのコードを用意する．まずメモリアドレス 0 から最大のシステムコール番号までを nop 命令で埋める．nop 命令は何もせず次の命令を実行する命令である．その後最後の nop 命令の次の位置に，任意のフック関数があるアドレスにジャンプするためのコードを埋め込む．

zpoline によるシステムコールフックの準備が完了すると，通常通りプログラムの main 関数が実行される．プログラム内でシステムコールが呼ばれると，バイナリ書き換えにより syscall / sysenter 命令から置き換えられた callq *%rax 命令が実行される．callq *%rax 命令によりメモリ上ではシステムコール番号のアドレスに移動する．移動先はトランポリンコードの設置により nop 命令で埋められており，nop 命令が繰り返し実行された後，任意のシステムコールフック関数があるアドレスにジャンプする．これによりシステムコールに対し任意の関数呼び出しが実行される．

4.3 CHFS

CHFS は建部らの開発するスーパコンピュータ向けの並列ファイルシステムである．近年のスーパコンピュータは各計算ノードにローカルストレージを持つことが多く，CHFS はそれらのストレージをまとめて 1 つのストレージであるかのように扱えることを目的としてい

る．CHFS はスーパコンピュータ向けの I/O 性能の世界ランキング IO500 の 2023 年 6 月の 10 ノード研究部門において 21 位を記録しており，高性能計算分野における今後の利用が期待される．

しかしながら IO500 で示された CHFS の性能は，ベンチマークプログラムのファイル I/O 部分の関数を CHFS の API に置き換えて実行することで測定されたものであり，実際のアプリケーションの複雑なソースコードを同様に書き換えることは現実的ではない．そのためアプリケーションを CHFS 上で動作させるには FUSE を利用したマウントが必要になるが，前章で述べた通り通信コストやコンテキストスイッチによる性能低下が予想される．

4.4 ライブラリ実装

本研究では提案した設計に基づき，CHFS のシステムコールフックライブラリを実装する．/chfs を仮想的なマウントディレクトリと設定し，/chfs から始まるパス名を対象とするシステムコールを，zpoline を利用して CHFS の API 呼び出しに置き換える．次章の実験では IO500 で使用されるベンチマークプログラムである IOR [17] を用いて性能評価を行う．IOR ベンチマークが呼び出すシステムコールに対応するため，実装するライブラリでは read，write，open，close，stat，lstat，lseek，pread64，pwrite64，openat，fsync，newfstatat システムコールをフックする．

システムコールフックの例として，open，close システムコールのフック実装の疑似コードをそれぞれコード 4.1，コード 4.2 に示す．

zpoline により，アプリケーションの open，write システムコールの呼び出しはそれぞれ hook_open，hook_write 関数の呼び出しに置き換えられる．まずファイルオープン操作が CHFS によって処理されるべきかどうかを判定する．パスの引数をチェックし，ファイル名が/chfs/で始まる場合，CHFS によって処理されるべきであると判定し，CHFS の API 呼び出しに置き換える．パスから/chfs/を削除し，フラグに応じて CHFS のファイル作成関数 chfs_create，ファイルオープン関数 chfs_open のどちらかを呼び出す．それらの関数の返り値はファイルディスクリプタであり，open システムコールの結果として返す前にファイルディスクリプタの 31 ビット目を 1 にセットしておく．

ファイルへの書込み操作が CHFS によって処理されるべきかどうかを判定するには，ファイルオープンで得られたファイルディスクリプタの 31 ビット目を確認する．ファイルディスクリプタの 31 ビット目が 1 であれば，CHFS のファイルディスクリプタと判定し，システムコール呼び出しを CHFS の API 呼び出しに置き換える．ファイル書き込み関数 chfs_write にファイルディスクリプタを引数として渡す前に，31 ビット目を 0 に戻す．

コード 4.1: open システムコールフック

```
hook_open(const char *path, int flags, …)
{
    if path[:6] == "/chfs/" {
        int ret;
        path += 6;
        if (flags & O_CREAT) {
            ret = chfs_create(path, flags, …);
        } else {
            ret = chfs_open(path, flags);
        }
        if (ret < 0) return ret;
    return ret | 1<<30;
    } else {
        return next_sys_call(path, flags, …);
    }
}
```

コード 4.2: write システムコールフック

```
hook_write(int fd, void *buf, size_t count)
{
    if (*fd & 1<<30) {
        return (chfs_write(fd ^ 1<<30, buf, count));
    } else {
        return next_sys_call(fd, buf, count);
    }
}
```

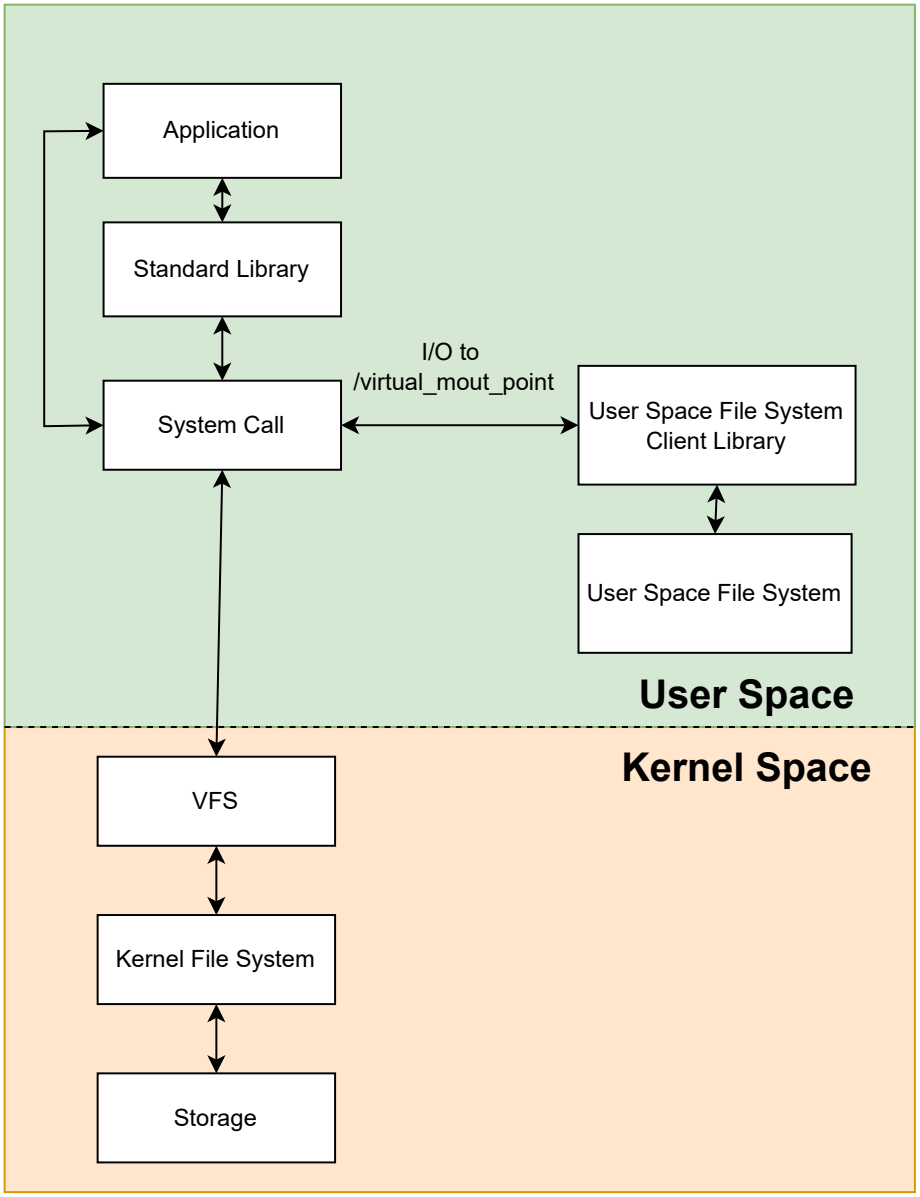

図 4.1: システムコールフックによるファイルシステムアクセス

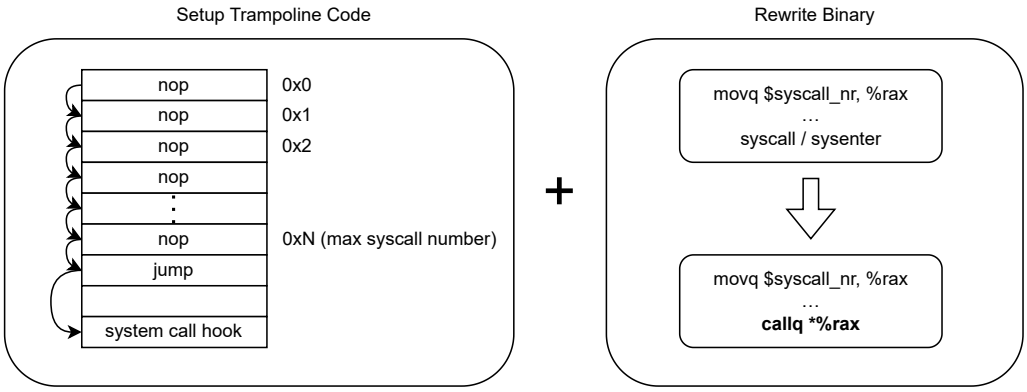

図 4.2: zpoline メカニズム

第5章　評価実験

前章で実装したシステムコールフックライブラリの評価実験を行う．本実験の目的としては，ユーザ空間ファイルシステムが提供するクライアントライブラリを使用した場合と比べてどの程度ストレージアクセス性能が保たれるか評価すること，また既存手法の中で最もよく用いられる FUSE との性能を比較することである．

5.1　実験環境

実験環境として筑波大学計算科学研究センターが運用する Pegasus スーパーコンピュータを利用する．Pegasus は各計算ノードに専用のストレージを保持しており，NVMe SSD と Intel Optane Persistent Memory(PMEM) を利用できる． 計算ノード間は InfiniBand NDR 200 で接続されており，ネットワーク帯域幅は 200Gbps である．

5.2　予備実験: FUSE 性能評価

最初に予備実験として，既存手法である FUSE を利用した際のストレージ性能の低下がどの程度発生するかを調査する．予備実験では Pegasus システムの計算ノードを 1 台使用し，ローカルの SSD を使用する．ローカル SSD には，/scr としてマウントされている XFS ファイルシステムを通じてアクセスできる．/scr に直接ファイル読み書きを実行した場合と，ファイル読み書きの要求をそのまま/scr に流すだけのダミーのファイルシステムを FUSE を使用してマウントし，ファイル読み書きを実行した場合の性能を IOR ベンチマークを使用して比較する．IOR はプロセス数を 1，file-per-process 方式で実行する．

実験結果を図 5.1 に示す．FUSE を利用した帯域幅性能は，直接読み書きした場合に比べ，読み込み性能が 1/7，書き込み性能が 1/3 という結果になった．

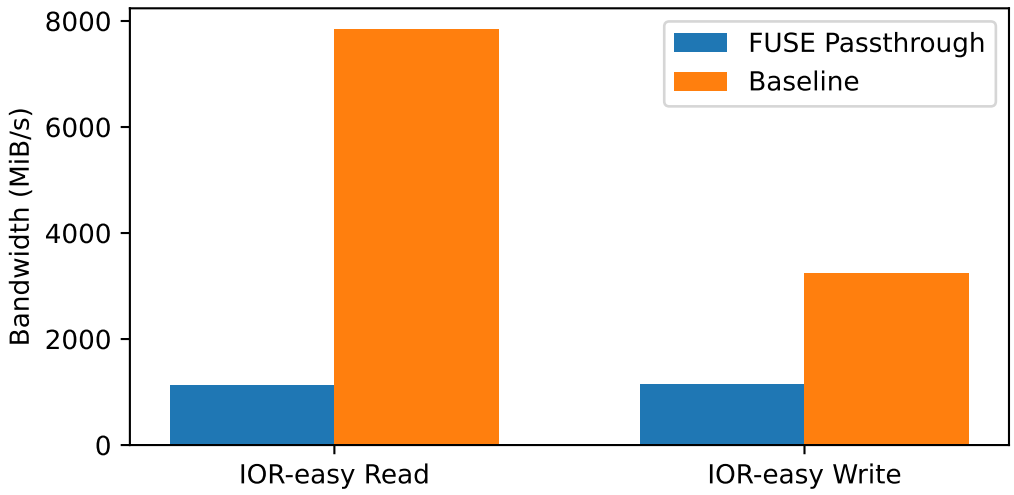

図 5.1: ネイティブファイルシステムと FUSE の性能比較

5.3 実験方法

本実験では IOR ベンチマークを実行して CHFS に対するファイル読み込み/書き込み帯域幅を測定する．提案手法を含む以下の 3 種類の条件下でベンチマークを実行し，その性能を比較する．

5.3.1 ネイティブ API

ファイルシステムが提供するクライアントライブラリの API を利用してファイルにアクセスする．本実験では IOR ベンチマークのソースコードを変更し，CHFS クライアントライブラリの関数を呼び出すことで CHFS 上のファイルへの読み書きを行う．

5.3.2 FUSE

CHFS サーバを起動した後，chfuse コマンドを使用して指定のディレクトリにマウントする．IOR ベンチマークのファイル読み込み・書き込みはマウントしたディレクトリに対して実行する．

5.3.3 システムコールフック (提案手法)

前章で実装したシステムコールフックライブラリを利用する．仮想的なマウントディレクトリ/chfs に対して IOR ベンチマークのファイル読み込み・書き込みを行う．

5.4 実験設定

Pegasus システムにおいて，計算ノード 10 台を CHFS サーバとして割り当て，さらに 1 台を IOR ベンチマークの実行に使用し，合計で 11 台の計算ノードを専有する．各計算ノードに設置されている PMEM を devdax モードで使用する．

CHFS はチャンクサイズを 16MiB に設定し，各ノード 46 プロセスで CHFS サーバを起動する．通信プロトコルに verbs，バックエンドに pmemkv を利用する．

IOR はプロセス数を 1，2，4，…と 16 まで増やしながら実行する．IOR では file-per-process 方式でアクセスし，各プロセスが別々のファイルに対して 1TiB 読み書きする．この操作を 5 回行い，帯域幅の平均を求める．

5.5 結果

実験結果を図 5.2 と図 5.3 に示す．どの条件でもプロセス数を増やすにつれ帯域幅が増加していることがわかる．読み込みの 16 プロセスにおいて性能の増加がないのは，Pegasus 計算ノード間の通信帯域幅 200Gbps に到達していると考えられる．

基準であるネイティブ API を使用した場合の性能に対して，提案手法のシステムコールフックは読み込み・書き込みともに同程度の性能を示した．また FUSE を使用した場合と比較して提案手法は 5.3 倍から 6.4 倍高い性能を示した．

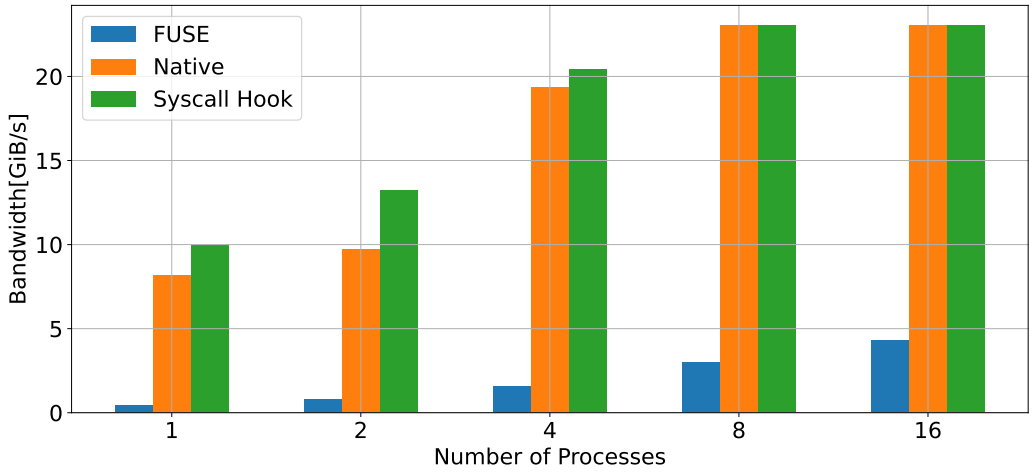

図 5.2: IOR 読込み性能

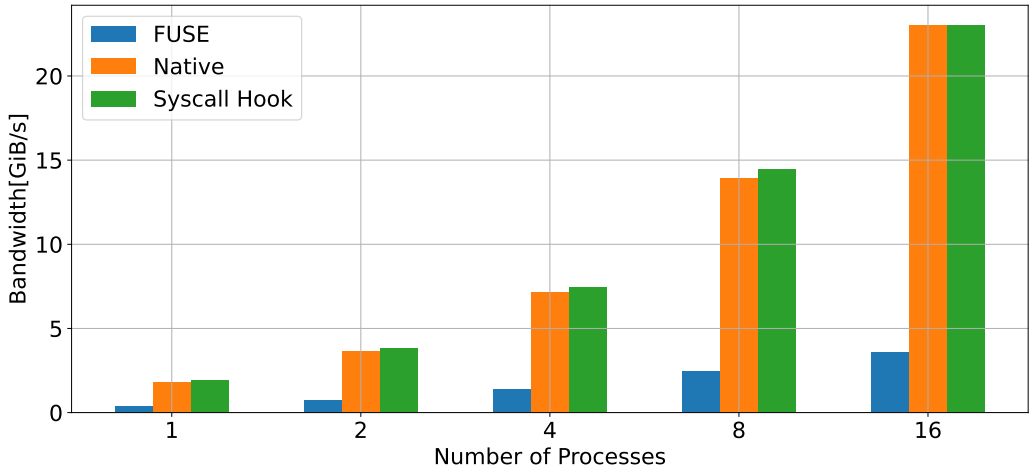

図 5.3: IOR 書込み性能

第6章　終論

6.1　まとめ

ユーザ空間ファイルシステムの POSIX インタフェース対応を目的とした手法はいくつか存在するが，既存手法には性能や汎用性の面で問題があった．本研究ではファイル操作に必要なシステムコールをユーザ空間ファイルシステムの API に置き換える，システムコールフックライブラリを設計した．高速なシステムコールフック機構である zpoline を使用したシステムコールフックライブラリを実装し，性能評価を行った．性能評価においてはストレージ性能を評価する IOR ベンチマークを用いて評価を行い，ユーザ空間ファイルシステムの API を直接呼び出した場合と比較してほぼ性能が低下しないこと，既存手法の FUSE を利用した場合と比較して 5.3 倍から 6.4 倍高い性能を示すことを確認した．

6.2　今後の展望

IOR ベンチマークを用いた性能評価により，提案手法の有用性を示した．今後は実アプリケーションの I/O 性能の向上に向け，より多くの実アプリケーションを動作させることを目指す．実アプリケーションは今回使用した IOR ベンチマークよりも多くのシステムコールを使用するため，システムコールのフックを増やす必要がある．現在はファイルディスクリプタを複製する dup・dup2，子プロセスを生成する fork システムコール等の対応が完了している．その結果，Linux の ls，touch 等ファイルシステムに関連するコマンドや，与えられた画像データセットをニューラルネットワークで分類する簡単な PyTorch プログラムを CHFS 上で動作させることに成功している．

今後はより大規模かつ実際に使用されているものに近いアプリケーションを動作させる．第 1 章で説明したように，近年の科学計算シミュレーションや AI ワークロードでは大量のデータをストレージから読み込む必要があるため，CHFS を利用できれば学習をより短時間で完了できる可能性がある．そのため現在は機械学習ワークロード向けのストレージベンチマークである MLPerf Storage ベンチマーク [18] を動作させることを目標にシステムコールのフックを進めている．

謝辞

1 年間親身かつ丁寧に研究を指導してくださった筑波大学計算科学研究センターの建部修見教授に深く感謝申し上げます．加えて，研究テーマの相談段階から論文作成にいたるまで研究にご指導ご尽力いただき，また国際会議の発表において共著者としてご協力いただいた HPCS 研究室の小山創平氏に大変感謝申し上げます．また，計算科学研究センターの研究員である平賀弘平氏，HPCS 研究室の杉原航平氏にも研究を進めるうえで多大なアドバイスをいただきました．感謝申し上げます．同時に，計算科学研究センターの職員の皆様，特に同センター秘書の桑野洋子氏には事務的な面で研究をサポートしていただきありがとうございました．HPCS 研究室の皆様には，研究を進めるにあたり議論を通じてご協力いただきました．そして共同研究先である富士通研究所の皆様には学外としての立場から大変ありがたいご支援をいただきました．ありがとうございました．最後に，これまで大学生活を共にした友人と，学生生活を支えてくれた家族に心から感謝致します．

参考文献

[1] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In Proceedings of the 34th International Conference on Neural Information Processing Systems, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc.

[2] Frank Schmuck and Roger Haskin. GPFS: A Shared-Disk file system for large computing clusters. In Conference on File and Storage Technologies (FAST 02), Monterey, CA, January 2002. USENIX Association.

[3] Marc-André Vef, Vasily Tarasov, Dean Hildebrand, and André Brinkmann. Challenges and solutions for tracing storage systems: A case study with spectrum scale. ACM Trans. Storage, Vol. 14, No. 2, April 2018.

[4] Matthew E Hoskins. SSHFS: super easy file access over SSH. Linux Journal, Vol. 2006, No. 146, p. 4, 2006.

[5] Alex Davies and Alessandro Orsaria. Scale out with GlusterFS. Linux Journal, Vol. 2013, No. 235, p. 1, 2013.

[6] Ohad Rodeh and Avi Teperman. zFS-a scalable distributed file system using object disks. In 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies, 2003.(MSST 2003). Proceedings., pp. 207–218. IEEE, 2003.

[7] André Brinkmann, Kathryn Mohror, Weikuan Yu, Philip Carns, Toni Cortes, Scott A Klasky, Alberto Miranda, Franz-Josef Pfreundt, Robert B Ross, and Marc-André Vef. Ad hoc file systems for high-performance computing. Journal of Computer Science and Technology, Vol. 35, pp. 4–26, 2020.

[8] Kyu-Jin Cho, Jaewon Choi, Hyungjoon Kwon, and Jin-Soo Kim. RFUSE: Modernizing userspace filesystem framework through scalable Kernel-Userspace communication. In 22nd

USENIX Conference on File and Storage Technologies (FAST 24), pp. 141–157, Santa Clara, CA, February 2024. USENIX Association.

[9] Yue Zhu, Teng Wang, Kathryn Mohror, Adam Moody, Kento Sato, Muhib Khan, and Weikuan Yu. Direct-FUSE: Removing the Middleman for High-Performance FUSE File System Support. In Proceedings of the 8th International Workshop on Runtime and Operating Systems for Supercomputers, pp. 1–8, 2018.

[10] James Lembke, Pierre-Louis Roman, and Patrick Eugster. DEFUSE: An interface for fast and correct user space file system access. ACM Trans. Storage, Vol. 18, No. 3, September 2022.

[11] LLNL. Gotcha. `https://github.com/LLNL/GOTCHA`.

[12] Michael J. Brim, Adam T. Moody, Seung-Hwan Lim, Ross Miller, Swen Boehm, Cameron Stanavige, Kathryn M. Mohror, and Sarp Oral. UnifyFS: A user-level shared file system for unified access to distributed local storage. In 2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 290–300, 2023.

[13] pmem. Syscall_intercept. `https://github.com/pmem/syscall_intercept`.

[14] Marc-André Vef, Nafiseh Moti, Tim Süß, Tommaso Tocci, Ramon Nou, Alberto Miranda, Toni Cortes, and André Brinkmann. GekkoFS - A Temporary Distributed File System for HPC Applications. In 2018 IEEE International Conference on Cluster Computing (CLUSTER), pp. 319–324, 2018.

[15] Osamu Tatebe, Kazuki Obata, Kohei Hiraga, and Hiroki Ohtsuji. CHFS: Parallel Consistent Hashing File System for Node-local Persistent Memory. In International Conference on High Performance Computing in Asia-Pacific Region, pp. 115–124, 2022.

[16] Kenichi Yasukata, Hajime Tazaki, Pierre-Louis Aublin, and Kenta Ishiguro. zpoline: a system call hook mechanism based on binary rewriting. In 2023 USENIX Annual Technical Conference (USENIX ATC 23), pp. 293–300, Boston, MA, July 2023. USENIX Association.

[17] hpc. IOR. `https://github.com/hpc/ior`.

[18] mlcommons. MLPerf storage benchmark suite. `https://github.com/mlcommons/storage`.