低アクセスレイテンシを実現する 不揮発メモリ向けユーザー空間ファイルシステム

松沢 敬一[1,2]　品川 高廣[2]

概要：メモリインターフェースを備える不揮発メモリの高いアクセス性能を活用するため，近年ストレージソフトウェアスタックの再考が進められている．本研究では，不揮発メモリの一部をプロセスで占有し，ユーザー空間上でライブラリとしてファイルシステムを構築する構成を提案する．本構成では，ファイルアクセス処理をユーザーモードで完結させることで，コンテキストスイッチを不要とし性能改善を図るとともに，既存アプリケーションを無変更で適用可能とする．DRAM を仮想不揮発メモリとみなし提案手法を適用したところ，従来ファイルシステム比較において，filebench ベンチマークで最大 73%の性能向上効果を確認した．

User-space Filesystem Leveraging Non-Volatile Memory for Low Access Latency

Keiichi Matsuzawa[1,2]　Takahiro Shinagawa[2]

1. はじめに

従来の計算機において，ストレージへのアクセス時間の支配的な要因はディスクのアクセス応答待ち時間であり，ディスクアクセス回数を削減するためソフトウェアでは I/O のスケジューリングやキャッシュ管理を行うことでアクセス性能の向上を図ってきた．しかし，メモリインターフェースでアクセスできる不揮発メモリ (Non-Volatile Memory, NVM) の技術開発が進み，ハードウェアの性能が向上すると，相対的にソフトウェア処理のオーバヘッドがボトルネックとして顕在化するようになった [1].

データベースやメールサーバ，ソフトウェア開発などのアプリケーションは，多数のファイルを参照・変更し，大量の小サイズ I/O を発行する [2],[3]．そのためアプリケーションの処理性能改善にはこれらファイルアクセスの性能向上が不可欠である．NVM の高いメディアアクセス性能を活かし，ファイルアクセスの処理性能を改善するには，ストレージアクセスのソフトウェアスタックにおけるオーバヘッドを削減する必要がある．加えて，NVM を実用的にかつ広範囲なアプリケーションで活用するためには，アプリケーションの改変を要せず適用可能なソフトウェアスタックでなければならない．

NVM を仮想的なブロックデバイスとみなしてファイルシステムを構築する手法 [4] は，従来のファイルシステムがディスク向けにセクタ単位のアクセスを前提として設計されているため，バイト単位でアクセスできる NVM の特性を活かせずデータ入出量が増加し処理効率が悪い．また，ファイルアクセス時逐一コンテキストスイッチによるオーバヘッドが生じる．NVM の性能特性に適したファイルシステムとして，BPFS [5]，SCMFS [6]，NOVA [7] が提案されている．これらはデータ入出力量を削減して処理効率を改善しているが，コンテキストスイッチは軽減されない．コンテキストスイッチ回数を削減する手法として，NVM 上のファイルデータブロックをプロセスのアドレス空間にマップして，ユーザーモードのプログラムで直接参照する Direct Access(DAX) [8] がある．DAX はファイルのメタデータのアクセスには性能改善効果がなく，またア

[1] (株) 日立製作所 研究開発グループ
Hitachi, Ltd. R&D Group.
[2] 東京大学
The University of Tokyo

プリケーション側に改変を要する．ファイルアクセスを用いず，DAX でマップした領域に，固有のデータ構造を構築するライブラリ [9],[10] やプログラミングのフレームワーク [11] も研究されている．これらは高い性能向上効果が期待できるものの，アプリケーションの改変を要し，かつその作業は一般的なアプリケーションプログラマには難しいといわれている [11].

本研究では，バイト単位アクセスが可能な NVM の性能特性を活かし，かつアプリケーションの改変を不要とするストレージソフトウェアスタックの構成方式を提案する．本方式では，多くのアプリケーションでファイルを排他的に利用していることに着目し，NVM 上にプロセスで占有するファイルシステムを提供する．本ファイルシステムでは，NVM 上の領域をプロセスの仮想アドレス空間に割り当てることで，コンテキストスイッチ不要でアクセス可能とし，性能向上効果を高める．また，アプリケーションの改変を不要とするため，ファイルシステムの機能をユーザーモードライブラリとして提供する．本ライブラリは，従来の POSIX インターフェース [12] の関数呼び出しの延長で動作するため，アプリケーションの改変や再コンパイルを要せず適用可能である．

本研究では，提案方式を Linux 及び glibc を適用した．DRAM を仮想 NVM として利用し，filebench ベンチマークで性能評価を行ったところ，仮想ブロックデバイスおよび従来ファイルシステムの組み合わせと比較して提案手法は最大で 73%の性能改善効果を得ることができ，提案手法の有用性を確認した．

2. 関連研究

本章では，メモリインターフェースで利用可能な NVM の利用方法の観点で従来研究を分類する．表 1 に方式の一覧を記載した．

2.1 仮想ブロックデバイス方式

本方式は，NVM を仮想的なブロックデバイスとして扱い，XFS など従来のディスク向けのファイルシステムを介して利用する方式である．本方式の利点として，主要な OS では NVM 向けの仮想ブロックデバイスドライバが標準で提供されており，また，アプリケーションからは従来のファイルシステムのインターフェースを用いてアクセスするため，OS・アプリケーションともに改変不要である．反面，本方式は下記要因が性能オーバヘッドとして残る．

a) 従来ファイルシステムは，ディスクにセクタ単位でデータアクセスを行う前提で設計されており，バイト単位アクセスが可能な NVM の特性を活かせず，アクセス効率が低い．
b) データの読み書きにページキャッシュを用いるため，NVM と DRAM 間でページ単位のデータコピーが生じる．
c) ファイルアクセス時はカーネルモードへのコンテキストスイッチが生じる．

2.2 NVM 向けファイルシステム方式

本方式は，NVM 向けの専用ファイルシステムを構築する方式である．これらの方式は，NVM がバイト単位アクセス可能であるという特性に着目し，データ構造やデータ処理順の効率化を図る．これにより，2.1 で挙げた性能オーバヘッド要因 a) の解消を図る．本方式に該当する先行研究には，BPFS [5]，SCMFS [6]，NOVA [7] などが挙げられる．

2.3 DAX 方式

Direct Access(DAX) 方式は，メモリ上にファイルシステムを構築することを想定した OS 内のメモリ管理方式で，以下の 2 点の特徴がある [8].

- ファイルデータにアクセスする場合，ページキャッシュを介さない．
- `mmap` 関数実行時，NVM が直接アプリケーションのアドレス空間にマッピングする．

前者の特徴により，2.1 における性能オーバヘッド要因 b) を解消し，後者の特徴により性能オーバヘッド要因 c) を解消できる．DAX は近年の Linux および Windows 及びその主要ファイルシステムで標準サポートされている．また，DAX 対応を備えた NVM 向けのファイルシステムとして，UMFS [13] などの先行研究もある．

DAX の利点は，アプリケーションに対して従来のファイルアクセスインターフェースを維持できる点にある．ただし `mmap` 関数を使用しない場合，データアクセス時のコンテキストスイッチが残るため性能改善効果は限定的である．そのため性能改善効果を最大化するには，`mmap` 関数を用いるようアプリケーションの対応が必要である．もう一つの問題として，ディレクトリツリーの名前空間やファイルのメタデータ操作はカーネルへのコンテキストスイッチが依然として生じ，性能改善効果が得られない．

2.4 固有データ構造構築方式

NVM 上で，アプリケーション固有のデータ構造を構築する方式である．多くは DAX と併用され，`mmap` によりアドレス空間にマッピングした領域を用いる．構築するデータ構造によって様々な方式が提案されている．例えば不揮発ヒープを提供する方式には pVM [9] や HEAPO [10] があり，Key Value Store の構築例として pmemkv [14] がある．また，NVM を前提としたデータ構造を構築するためのプログラミングのデザインパターンも提案されている [11]. これらはそれぞれ性能や一貫性の保持において，ファイルシステムよりも優位な点を要するが，それぞれ独自のイン

表 1　NVM 活用方式の分類

方式	プロセス間データ共有	性能	アプリケーションの改変
仮想ブロックデバイス方式	✓	低	不要
NVM 向けファイルシステム	✓	中	不要
DAX 対応ファイルシステム	✗	中	要
DAX + 固有データ構造構築	✗	高	要
DAX + ライブラリファイルシステム	(実装依存)	高	(実装依存)
提案方式 (DAX + ライブラリファイルシステム)	✗	高	不要

ターフェースを用いており対応のためにアプリケーションの改変を要する．

2.5　ライブラリファイルシステム方式

本方式は，ファイルシステムをユーザーモードのライブラリとして実装することで，ファイルアクセスにおけるコンテキストスイッチを削減する．本方式におけるプロセス間のデータ共有可否やアプリケーションの改変要否は，実装依存である．

ユーザーモードでのストレージアクセス機能を提供するライブラリ SPDK [15] はファイルシステムのサブセット BlobFS を提供している．ただし BlobFS は専用インターフェースで提供されるためアプリケーションの改変を要する．また，プロセス間でデータ共有はできない．

pmemfile [16] は，POSIX インターフェースを用いるアプリケーションに対しファイルシステムを提供する．pmemfile では，標準 C ライブラリのシステムコール呼び出し部に動的にバイナリパッチを充てることで，アプリケーション側を無改造で利用可能としている．pmemfile の設計方針や制限は，提案方式と共通部分が多い．ただし pmemfile は提案後継続的なメンテナンスがされておらず，また性能評価も小規模なマイクロベンチマークでのみしかなされていない．実際の実用アプリケーションに対する定量的な評価は不十分である[*1]．

ファイルシステムの処理を，ユーザーモードとカーネルモードで分担する手法も提案されている．SplitFS [17] や ZoFS [18] は，データアクセスはユーザーモード内で処理を完結させ，メタデータに関する処理のみカーネルモードで実行している．これらの手法は，プロセス間でデータ共有を行いたい場合，ディレクトリの名前空間やアクセス管理についてカーネルのファイルシステムと同程度の機能を提供することができる．一方で，メタデータアクセスが多いアプリケーションに対してはソフトウェアオーバヘッド削減効果が減少する．

3.　設計

データベースやメールサーバ等のアプリケーションでは，ディレクトリ階層のツリーの一部に，同一の所有者・アクセス権で構成されたファイル群としてアプリケーションのデータを配置することが知られている [18]．これは，これらのアプリケーションが実質的にそのツリー下を排他的に使用することを意味している．そこで本手法では，アクセス毎にカーネルや他プロセスとの排他処理を不要とすることで，低オーバヘッドでアクセスできるプロセス固有のディレクトリツリーを NVM 上に構築する．

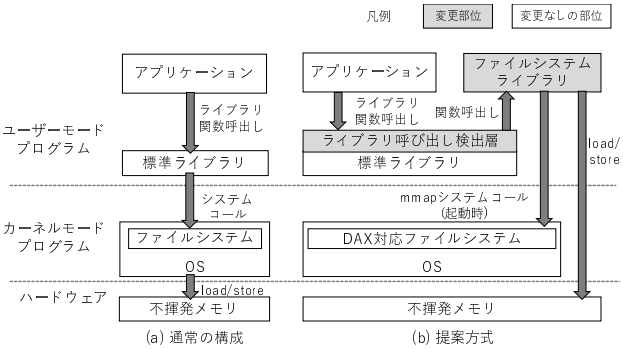

図 1　ソフトウェア構成

本手法は，性能オーバヘッドの削減とアプリケーションの改変が不要であることを両立するため，ライブラリファイルシステム方式を採用する．図 1 にソフトウェア構成を示す．(a) は，NVM 上にファイルシステムを構築した場合の一般的な構成で，仮想ブロックデバイス方式に相当する．アプリケーションが標準ライブラリが提供する POSIX インタフェースの関数を呼び出すと，標準ライブラリがシステムコールでカーネル内のファイルシステムを呼び出す．最終的にカーネル内のファイルシステムが NVM 上のデータを Load/Store することでアクセスが完了する．

(b) は提案方式の構成である．(b) ではアプリケーションが標準ライブラリの関数を呼び出すと，標準ライブラリはファイルシステムライブラリを呼び出す．このファイルシステムライブラリが，NVM 上に構築したデータを直接 Load/Store することでアクセスを行う．この際にカーネルモードへのコンテキストスイッチが不要であるため，本方式はオーバヘッドを削減できる．

3.1　アプリケーションにおける適用方法

本方式を組み込んで動作させるアプリケーションでは，ファイルシステムコールライブラリを呼び出すよう改変さ

[*1] MongoDB, MariaDB, filebench の pmemfile 上での動作を試みたが，いずれも起動に至らなかった

れた標準ライブラリを用いて起動する．アプリケーションからは，特定ディレクトリ以下がファイルシステムライブラリによって提供されるファイルシステムとして認識される．アプリケーションはそのディレクトリ配下に対し，これまで通り POSIX インターフェースの関数を呼び出すことで，NVM 上に構築されたファイルを読み書きできる．このディレクトリ下のデータは同一プロセス内のスレッド間でのみ共有され，他プロセスおよびカーネルからは認識されない．

3.2 標準ライブラリ検出層

標準ライブラリ検出層では，POSIX インターフェースのうちファイルアクセスに関する関数呼び出しを捉えて，ファイルシステムライブラリを呼び出す．ファイルシステムライブラリによってアクセス要求が処理された場合，その場で処理は完了とし標準ライブラリの機能は呼び出されない．アクセス対象がファイルシステムライブラリの提供範囲外のディレクトリである場合，継続して標準ライブラリが通常の処理を行う．

3.3 ファイルシステムライブラリ

ファイルシステムライブラリは，標準ライブラリ検出層からの呼び出しに対応して，ファイルアクセスの処理を行う．ファイルシステムライブラリは DAX 対応ファイルシステム上に単一の大サイズファイルを作成し，そのファイルを自身のアドレス空間上に `mmap` したうえで，NVM 上に inode やディレクトリエントリと言ったファイルシステムのデータ構造を作成・配置・管理する．以後，この `mmap` した一連のメモリ領域をプールと呼ぶ．

ファイルシステムを構成するデータは，NVM 上と DRAM 上に分けて配置する．この配置については 3.4 にて述べる．

3.3.1 サポートする機能

本ライブラリでは，POSIX インターフェースのうち基本的なファイルアクセス関数のみ対応する．

ファイルデータアクセス系 ファイルの `open`, `close` や，データ読み書きを行う関数が相当する．

ファイル・ファイルシステム情報 `stat`, `statvfs` 系列の情報取得に関する関数が相当する．

ディレクトリツリー操作 ディレクトリエントリの参照や，`rename`, `unlink` に関する関数が相当する．

ファイルロック `fcntl`・`flock` が相当する．

本設計及び実装では，データ入出力及びメタデータの操作は同期的に行う．これは，ディスク入出力と異なり NVM の Load/Store は同期的に行うため，ファイルシステム側で非同期処理を行うことでコンテキストスイッチのオーバヘッドが生じることを防ぐためである．

主要なアプリケーションでは，非同期 IO 機能の利用はオプションで切り替えられることが多く，実用上は問題とならない．

例外的に，`unlink` はディレクトリツリーから inode を削除することは同期的に行うが，inode が NVM 上から削除する処理は遅延して行う．これは Linux における，`open` 中ファイルでスクリプタが存在する限り inode を削除しないという仕様に合わせたためである．

3.3.2 サポートしない機能

以下の拡張的な機能は，使用するアプリケーションや条件が限定的であるためサポートしていない．

拡張属性 `getxattr`, `setxattr` 等が相当する．

非通常ファイル `symlink` や `mknod` 関数が作成する非通常ファイルはサポートしない．本ファイルシステムはファイルデータへのアクセス高速化を図るものであるため，非通常ファイル作成は目的と合致しない．

memory mapped file メモリマッピングに対応するアプリケーションは，直接 DAX 対応ファイルシステムを用いればよいため，本ライブラリでは対応しない．

セキュリティコンテキスト `getfilecon` 等が相当する．本ファイルシステムはプロセス内で閉じたディレクトリを提供するため，ファイルシステムレベルで強化されたセキュリティ機能を提供する必然性がない．

ファイルアクセス監視機能 `inotify` 等のファイルアクセス監視機能は対応しない．こちらもプロセス間でデータを共有できないため，非実用的なためである．

3.3.3 対象ファイル判定

アプリケーションがファイルアクセスを実行する際，本ライブラリが提供するファイルのみ処理対象とし，それ以外は通常のアクセス処理を行うようにしなければならない．

この判断は，各関数の引数を検証し，以下の通り行う．検証すべき引数は関数毎に個別に指定している．

ファイルパス 処理対象のファイルがパス名で指定される関数に関しては，パス名を絶対パスに変換したうえで，その接頭辞が提供パスと一致するかどうかで判定する．

ファイルデスクリプタ 処理対象のファイルがファイルデスクリプタで指定される関数に関しては，ライブラリ内で事前に予約しておいたデスクリプタと一致するかどうかで判定する．

3.4 データ配置

NVM は DRAM と性能特性が異なる．例えば Intel DCPMM は DRAM と比較してアクセスレイテンシ・帯域いずれも劣るとしている [19]．そのため，ファイルシステムの性能を高めるためには，ファイルシステムの管理データのうち，不揮発性を要するデータのみ NVM 上に配置し，参照・更新頻度が高く不揮発性を要しないデータは DRAM 上に配置することが望ましい．

3.4.1 NVM 上に配置するデータ

NVM 上に配置すべき不揮発性を要するデータは，他の

情報から再生成できないデータであり，おおよそ一般的なファイルシステムでディスク上に配置される下記のデータが相当する．

- ファイルシステム全体の設定情報
- ファイルメタデータ (inode 及びブロックマップ)
- ディレクトリエントリ
- ファイルデータ
- inode 番号の割り当て状況
- プール内の空き領域情報

ファイル数等の統計情報は NVM には配置しない．一般的なファイルシステムでは，統計情報もディスク上スーパーブロックに格納している．しかしこれらの情報は，プロセス起動時にファイルシステム内の全ファイルを辿ることで再生できる．本ファイルシステムは NVM 上に構築されおり，その容量の上限から全ファイルの探索は現実的な時間内で完了すると考える．そのため，本方式では統計情報等は DRAM 上のみに配置することで，起動時の時間と引き換えに通常アクセス時の NVM 更新の削減を図る．

inode 番号の割り当て状況を NVM 上に保持するのは，`unlink` 関数実行後 inode 削除までの間に電断が発生した場合でも，それら inode を再起動時に削除するためである．

NVM 上のデータ構造において，ファイルのブロックマップなどのようにプール内の他の位置を指し示す情報がある．この時，ブロックの位置は通常のアドレス参照を用いることができない．これはプロセス再起動後，`mmap` 関数呼び出しの際にプールが同じアドレスにマップされることが保証されないためである．そのため本データ構造では，NVM 内の位置を指し示す際にプールの先頭位置からのオフセットの形で保持する．

3.4.2 DRAM 上に配置する情報

DRAM 上には，プロセス実行中のみ必要なデータや，ファイルアクセス時の処理高速化に有用なデータを保持する．前者は，ファイルデスクリプタとその内容が相当する．後者の高速化に要するデータは，下記が相当する．

inode のメタデータの一部　inode は NVM 上にも格納しているが，パスの Lookup 処理の高速化のため事前に inode のフルパスを DRAM 上に保持している．また，データアクセス高速化のため，ブロックマップについて事前にプール内のオフセットから仮想アドレスに変換したアドレスを DRAM 上に保持している．

POSIX 互換形式のディレクトリエントリ　POSIX インターフェースでは，ディレクトリエントリ一覧は `readdir` 関数の規定するフォーマットで返す必要がある．しかしこのフォーマットは，可変長の項目をシングルリンクリスト形式で保持したもので，検索や更新の処理効率に優れず，この形式のまま NVM に配置することは性能的に好ましくない．そこで本方式では，NVM 上ではディレクトリエントリは固定長エントリの配列で格納し，DRAM 上では `readdir` 関数向けに再構成したデータをキャッシュとして保持する．

ファイルパスの Lookup テーブル　ファイルパスの指すファイルを Lookup を高速化するため，DRAM 中にパス名と inode を対応付けるハッシュテーブルを保持する．

ファイルシステム統計情報　`statfs` 関数では，ファイルシステム中のファイル数やブロック数の情報を返す必要がある．これらの情報はファイルアクセス中に高頻度で更新されるうえ，全ファイルを辿ることで再構成可能なため，DRAM 上のみに配置する．

4. 実装

4.1 標準ライブラリ検出層

本実装では，`glibc` のソースコードを一部改変・再コンパイルし，3.3.1 で述べた関数毎にファイルシステムライブラリ呼び出しコードを付与した．

POSIX インターフェースにおけるファイルアクセスには，ファイルデスクリプタと I/O ストリームの二つのインターフェースが提供されている．`glibc` では，前者を低レベル I/O とよび，後者の I/O ストリーム機能は内部で低レベル I/O を呼び出す実装となっている．ライブラリ呼び出しをフックし，機能を加える手法として `LD_PRELOAD` を用いる手法 [20] があるが，I/O ストリーム機能から低レベル I/O の呼び出しはライブラリ内で静的にリンクされており，`LD_PRELOAD` で捉えることはできない．そのため，本実装では低レベル I/O の処理部を改変し，標準ライブラリ検出層を挿入することで，後者の I/O ストリームへの対応も実現した．

本実装では，Ubuntu 19.10 付属の `glibc` を改変し，また生成したライブラリは Ubuntu のパッケージファイルとしてパッケージングされる．そのため，通常の Ubuntu のパッケージの更新と同じ手順で差し替えが可能である．これは，アプリケーションの改変不要に加え，デプロイの容易化に有効な措置である．

4.2 ファイルシステムライブラリ

ファイルシステムライブラリは，C++で実装した．トランザクション処理を備えるために，Persistent Memory Development Kit (PMDK)[14] が提供する NVM 向けのオブジェクト管理ライブラリ `libpmemobj` を用いた．

本ファイルシステムは空のプールまたは以前他プロセスが使用したプールを利用することができる．以前に使用されたプールを用いる場合，起動時にプール内のデータを辿り，3.4.2 に示す DRAM 上に配置するデータを再構築する．もしディレクトリツリーで参照されない inode が残っていた場合，それは `unlink` 処理後遅延処理が行われていない inode なので，起動時に削除する．

表 2　ソフトウェア構成

項目	バージョン・設定
OS	Ubuntu Server 19.10 (Kernel 5.3.0)
ファイルシステム	XFS
glibc	2.30
PMDK	1.7
libpmemobj-cpp	1.8
filebench[3]	1.5-alpha3

反対に，プロセスの終了時は，遅延されたファイル削除処理を除き何も行わない．ファイルの更新は同期的に行われるため，データの一貫性に問題は生じない．

5. 評価

提案手法を用いて，従来方式に対する性能評価を行った．

5.1 実行環境

利用したサーバは，CPU に Intel Xeon E5-2690(8core)，DRAM 64GB を搭載している．評価に際し NUMA の影響を排除するため，プロセッサとメモリは 1 ソケット分のみ使用した．そのため使用可能な DRAM は 32GB であり，そのうち 24GB を仮想 NVM とみなし実験を行った．

ソフトウェアの構成を表 2 に示す．性能評価には，多数のファイルのデータ・メタデータを参照・更新する例として，filebench の `fileserver` ワークロードを用いた．このワークロードでは，10000 個のファイルに対し，20 ワーカースレッドがファイルの作成・削除・読み書き・追記を 1 分間繰り返す．平均ファイルサイズは，64KB・128KB・256KB の 3 通りの構成で測定した．測定時には，ファイル処理の秒間実行回数，各処理の平均応答時間，CPU 使用率を収集した．

性能比較対象は，DRAM 上に構築した XFS で，DAX の有効・無効を両方測定した．提案手法は DAX を有効化した XFS 上にプールを作成して測定した．

性能測定にあたり，PMDK に変更を加えており，プールファイルの新規作成時，ファイルの全領域にゼロデータを書き込むようにした．これは，XFS の unwritten extent を消すことで，アプリケーション動作中に，が生じ性能が不安定になることを防ぐためである．

5.2 実行結果

図 2 に，各構成におけるファイルアクセス処理の秒間実行回数 (折れ線) と CPU 使用率 (積み立て棒グラフ) を示す．CPU 使用率の `user` はユーザーモード，`sys` はカーネルモード，`idle` はアイドル中の動作時間の割合を示す．

いずれのファイルサイズにおいても，提案手法は DRAM 上に構築した XFS よりも高い性能値を示した．特にファイルサイズが小さいときに効果が大きく，平均ファイルサイズ 64KB の測定では DAX 無効の XFS に対し 73%大きい．CPU 使用率を比較すると，XFS の場合は実行時間の大半をカーネルモードで占め，また CPU がアイドルの割合も多い．一方提案手法では CPU 時間を 95%以上使い切り，かつそのほとんどをユーザーモードで使用されており，カーネルの関与が少ないことがわかる．なお，提案手法におけるカーネルモード動作は，filebench 自体がワーカースレッド間の排他を行うために `futex` システムコールを呼ぶためのもので，ファイルアクセスとは関係しない．加えて XFS では，DAX の有効無効によらず CPU の実行時間が使い切れていない．これはジャーナル処理等，複数スレッドの直列化を要する処理時間が長いためである．

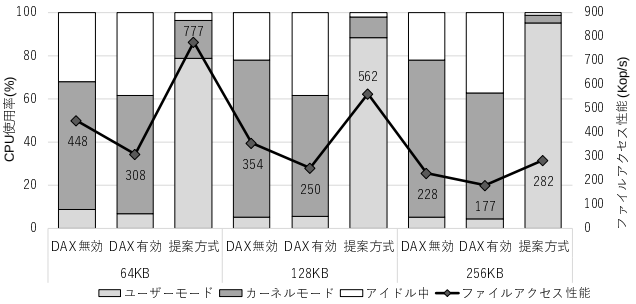

図 2　filebench 実行結果

図 3 に，ファイルアクセス種別ごとの応答時間を示す．提案手法では，`delete` 関数の inode 削除処理を遅延させ，他の処理の呼び出し時に行っている．そのため，本来ファイルサイズに依存しない処理においても，平均的には処理時間がファイルサイズに依存して上積みされている．

まず，`open`，`close`，`stat` に関しては手法毎に大きな差はない．`create` は空ファイルを作成する処理で，提案手法は XFS に比べ高速に処理を終えている．これは XFS におけるファイル作成は，複数の B-Tree やジャーナルを含め複数のデータ構造の変更を要するためである．`create` はファイルのデータブロックを参照しないため，本来 DAX 有効・無効は性能に影響しないはずである．それにも関わらず顕著な性能差があるのは，他の処理によってメタデータ更新の待ち時間が増えているためと考えられる．`write` 後に実行した `close` では，DAX 無効時のみファイルサイズに比例して若干応答時間が伸びている．これは XFS の Delayed Allocation の動作により，`close` 時に extent 情報の更新が行われるためである．

次にデータ I/O 性能を比較する．`read-all` は filebench がファイルの全データを読み込む処理である．XFS において，DAX 有効化時はページキャッシュが存在しないことでデータコピーが少ない分，DAX 無効時よりも短い時間で応答している．提案手法は小ファイルサイズにおいては XFS より短い時間で応答しているが，ファイルサイズが増加すると，線形以上に応答時間が伸びている．ただし，こちらは既に CPU 実行時間が飽和していることから，メモリ帯域の上限に達したことでメモリアクセス自体が遅延し

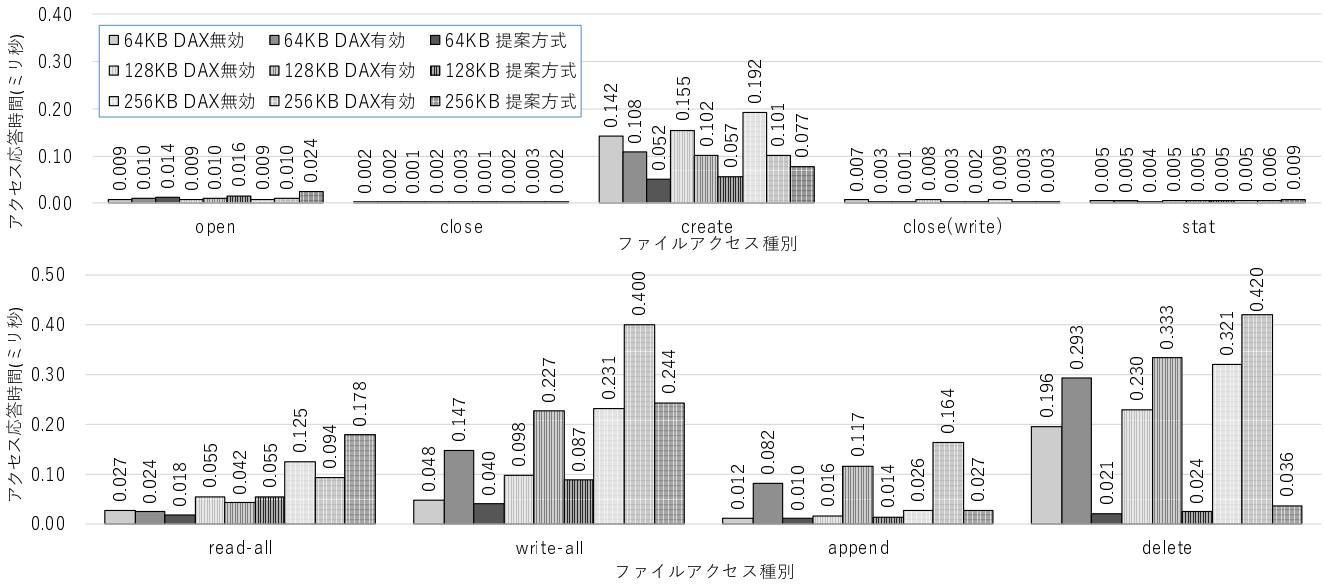

図 3 ファイルアクセス種別毎の応答時間

ている可能性がある．こちらはより大きなメモリ帯域を持つ実行環境を用いて今後精査していく．

`write-all` は，作成直後の空ファイルに対し，指定したファイルサイズの全データを書き込む処理である．こちらは DAX 無効時の XFS と提案手法が同程度の応答時間で，DAX 有効時の XFS がそれより時間がかかっている通常 XFS では Delayed Allocation の動作により割り当て作業を遅延処理させることができるが，DAX 有効時のファイル追記は，その時点でファイルへのブロック割り当てを要するため，応答時間が伸びている．`append` は，filebench が既存ファイルにデータを追記する処理である．追記サイズは `write-all` より小さいため，応答時間が短い点以外は `write-all` と同じ傾向を示す．

`delete` は，filebench がファイルを削除する処理である．提案手法は inode 削除を遅延させているため，XFS より大幅に短い時間で処理を終えている．XFS において DAX が有効である方が応答時間が遅い．これは DAX 無効時はブロックの割り当てが遅延され，ファイル削除時点までに割り当てが行われないと，削除処理も短縮されるためと考えられる．

6. 制限事項

従来ファイルシステムが提供する機能のうち，本方式及び実装での制限事項について論ずる，

6.1 マルチプロセス対応

本実装では，二つの理由でマルチプロセスによるデータ共有・同時アクセスへの対応を行っていない．

一つ目の理由は性能トレードオフである．本方式では，ファイルシステムを構成する一部の管理データを DRAM に配置している．これらの管理データをプロセス間で共有するには，アドレス空間を共有し，かつ適宜排他処理を挿入する必要がある．一方で，プロセス毎に管理データを持つ場合，1 プロセスが加えた操作に対し，全プロセスが同期して管理データを更新する必要があり性能オーバヘッドが大きい．いずれにしても，性能オーバヘッドが増大する．

もう一つの理由は，プロセスの異常終了への対応である．あるプロセスがファイルハンドルをロックしたまま不正終了した場合，そのロックは解除されず，他のプロセスはそのファイルのロックを永久に取得できない．通常の OS では，プロセス終了時に OS がロックを解除するが，この処理を OS を介さず行うと，プロセスの死活監視など実装の複雑化とオーバヘッドの増加を要する．

前述の通り，多くのアプリケーションではプロセス内でディレクトリ内のデータを排他的に使用するが，ソフトウェアのビルドなど複数プロセスが並列に動作し同じディレクトリのデータを参照するアプリケーションもある．これらの用途に関しては，今後対応を行い性能改善効果の評価を続ける．

6.2 標準ライブラリ不使用アプリケーション

提案手法は標準ライブラリを改変することで，アプリケーションのファイルシステムアクセスを捉えている．そのため標準ライブラリを用いないアプリケーションには適用できない．これらのアプリケーションは，実行バイナリの解析によりシステムコール呼び出し部を書き換えることで，本提案手法と同等の処理を行える．しかしその場合，対象のコンピュータアーキテクチャやシステムコールの呼び出し規約毎に解析部を準備する必要がある．

7. おわりに

本稿では，不揮発メモリの高性能を活かし，アプリケーションのファイルアクセス性能を向上させるソフトウェアスタックの構成手法について述べた．不揮発メモリの性能を実用アプリケーションで活かすためには，ファイルアクセスにおけるソフトウェアオーバヘッドを削減し，またアプリケーションを改変せず適用できる構成でなければならない．提案手法では，多くのアプリケーションがディレクトリツリーの一部を排他的に使用することに着目し，不揮発メモリ上にライブラリファイルシステムを構築し，プロセスで占有できるディレクトリツリーを提供する．本構成ではファイルアクセスに対しカーネルモードへのコンテキストスイッチが不要であり，またプロセス間のデータ共有を行わないことで排他処理を削減し，ソフトウェアオーバヘッドを削減できる．また，従来のインターフェースを保持し，標準ライブラリ以降の処理に不揮発メモリ対応を組み込むことで，標準ライブラリを用いるアプリケーションに無変更で適用可能とした．

DRAM を仮想不揮発メモリとみなして本手法を適用したところ，filebench ベンチマークを用いたファイルアクセス性能評価において，同じく仮想不揮発メモリ上に構築した XFS と比較して最大 73%の性能改善効果を得ることができ，本手法の効果を確認できた．

今後の課題として，多様な実用アプリケーションや，実際の不揮発メモリによる性能評価がある．また，その評価結果と不揮発メモリの性能特性を踏まえて，DRAM と不揮発メモリ上へのデータ構造の配置の改善を行っていく．

参考文献

[1] Strassmaier, M.: Intel Optane DC Persistent Memory Performance Review, *Presented at Storage Developers Conference 2019*, SDC '19 (2019).

[2] Standard Performance Evaluation Corporation: SPEC SFS 2014, (online), available from ⟨https://www.spec.org/sfs2014/⟩ (accessed 2019-11-12).

[3] Tarasov, V., Zadok, E. and Shepler, S.: Filebench: A Flexible Framework for File System Benchmarking, *;login: The Usenix Magazine*, Vol. 41, No. 1, pp. 6–12 (2016).

[4] Storage Networking Industry Association: NVM Programming Model, (online), available from ⟨https://www.snia.org/tech_activities/standards/curr_standards/npm⟩ (accessed 2019-11-12).

[5] Condit, J., Nightingale, E. B., Frost, C., Ipek, E., Lee, B., Burger, D. and Coetzee, D.: Better I/O Through Byte-addressable, Persistent Memory, *In Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pp. 133–146 (2009).

[6] Wu, X. and Reddy, A. L. N.: SCMFS: A file system for Storage Class Memory, *In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'11, pp. 1–11 (2011).

[7] Xu, J. and Swanson, S.: NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories, *In Proceedings of 14th USENIX Conference on File and Storage Technologies*, FAST '16, pp. 323–338 (2016).

[8] Bates, S. and Christensen, N.: PM Support in Linux and Windows, *Presented at SNIA Persistent Memory Summit*, PM SUMMIT '18 (2018).

[9] Kannan, S., Gavrilovska, A. and Schwan, K.: pVM: Persistent Virtual Memory for Efficient Capacity Scaling and Object Storage, *In Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pp. 13:1–13:16 (2016).

[10] Hwang, T., Jung, J. and Won, Y.: HEAPO: Heap-Based Persistent Object Store, *Trans. Storage*, Vol. 11, No. 1, pp. 3:1–3:21 (2014).

[11] Ren, J., Hu, Q., Khan, S. and Moscibroda, T.: Programming for Non-Volatile Main Memory Is Hard, *In Proceedings of the 8th Asia-Pacific Workshop on Systems*, APSys '17, pp. 13:1–13:8 (2017).

[12] IEEE: *IEEE 1003.1-2017 - IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7* (2017).

[13] Chen, X., Sha, E. H.-M., Zhuge, Q., Wu, T., Jiang, W., Zeng, X. and Wu, L.: UMFS: An efficient user-space file system for non-volatile memory, *Journal of Systems Architecture*, Vol. 89, pp. 18 – 29 (2018).

[14] PMDK team: Persistent Memory Development Kit, (online), available from ⟨https://pmem.io/⟩ (accessed 2019-11-12).

[15] SPDK: Storage Performance Development Kit, SPDK (online), available from ⟨https://spdk.io/⟩ (accessed 2019-11-12).

[16] Czyrylo, K.: Using Persistent Memory to Build a High-Performance, Fully User Space File System, *Presented at Open Source Summit Europe*, OSSEU '17 (2017).

[17] Kadekodi, R., Lee, S. K., Kashyap, S., Kim, T., Kolli, A. and Chidambaram, V.: SplitFS: Reducing Software Overhead in File Systems for Persistent Memory, *In Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pp. 494–508 (2019).

[18] Dong, M., Bu, H., Yi, J., Dong, B. and Chen, H.: Performance and Protection in the ZoFS User-space NVM File System, *In Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pp. 478–493 (2019).

[19] Intel: Intel 64 and IA-32 Architectures Optimization Reference Manual, (online), available from ⟨https://software.intel.com/en-us/articles/intel-sdm⟩ (accessed 2019-11-12).

[20] Shi, Z., Feng, D., Zhao, H. and Zeng, L.: USP: A Lightweight File System Management Framework, *In Proceedings of IEEE Fifth International Conference on Networking, Architecture, and Storage*, NAS 2010, pp. 250–256 (2010).