Vol.2019-HPC-169 No.7
2019/5/10

不揮発性メモリ利用を最適化するユーザレベルファイルシステム

吉村　剛[1,a)]　千葉　立寛[1,b)]　堀井　洋[1,c)]

概要：不揮発性メモリは極めて低いレイテンシを提供するものの，その結果ファイルシステムのレイテンシが無視できなくなっている．特に，システムコールやハードウェア割り込みによるユーザ・カーネル空間のコンテキストスイッチ，ファイルシステムの高度な機能が相対的に大きなレイテンシとなっている．そこで，ユーザアプリケーションはユーザレベルのストレージ向けフレームワークや、mmap などを用いることで，それらレイテンシを回避することができる．しかし，多くの既存のアプリケーションは POSIX インターフェイスを前提としており，またアプリケーション開発者はデバイスレベルのチューニングの労力が必要となってしまう．本研究では，そういったアプリケーション開発者の労力を減らすためのユーザレベルファイルシステムを提案し，現段階での実装および予備実験結果を示す．提案するファイルシステムはユーザレベルのストレージスタック上に構築することでコンテキストスイッチ回数を最小限にし，またページキャッシュやダイレクト I/O などの複雑なファイル I/O 処理を非同期的に処理することで，レイテンシの増加を最小限にする．実験ではノンブロッキング I/O のレイテンシが 700 ナノ秒に到達したこと，またブロッキング I/O をジャーナリングを無効にした EXT4 に比べて 20 マイクロ秒削減することができたことを示す．

1. はじめに

不揮発性メモリ (NVM) の登場により，ファイルシステムのレイテンシがハードウェアに対して相対的に増大し，性能面で無視できないものとなっている．Non-volatile memory express (NVMe) SSD は現在 7 マイクロ秒の読み出しレイテンシ，18 マイクロ秒の書き出しレイテンシを示している [1]．Non-volatile main memory (NVMM) は 20 から 85 ナノ秒の読み出しレイテンシ，10 から 1000 ナノ秒の書き出しレイテンシに到達するとされている [2]．より大きなレイテンシを示す NAND フラッシュ SSD やハードディスクドライブと比べ，NVM のレイテンシはハードウェア割り込みやシステムコールによるコンテキストスイッチが大きな性能劣化につながる [2]．ユーザレベルアプリケーションはこのようなコンテキストスイッチのオーバヘッドを最小限にするため，NVM 向けのユーザレベルストレージフレームワーク (例: Storage Performance Development Kit (SPDK) [3]， NVMeDirect [4]) を利用して，アプリケーション内で直接 NVM 領域を読み書きすることができる．

しかし，既存のアプリケーションの多くは POSIX インターフェイスを利用している [5] にもかかわらず，SPDK や NVMeDirect は POSIX をサポートするファイルシステムを提供していない．その結果，アプリケーションはストレージへの読み書きをデバイスレベルで細かく制御する必要があり，NVM の性能を十分享受することは難しい．特に，NVM は DRAM と比べて書き出しレイテンシの大きさや，耐摩耗性において制約があることが課題となる [6–8]．HiNFS [6] は NVM 向けのファイルシステムにおいてもページキャッシュによる性能向上が見られることを示している．一方で，他の NVM 向けのファイルシステムでも示されているように，読み出しに関しての NVM の性能は DRAM に十分近く，ページキャッシュの排除の効果は大きくなると予想される．アプリケーション開発者はこうしたデバイスの特性やワークロードの特性を加味したうえで，さらにデバイスレベルのチューニングを行って始めて NVM の性能を最大限活用できるようになる．NVMM による `mmap` を利用する場合も，CPU キャッシュとストレージデータの一貫性制御を追加で行う必要があり，制御を誤れば性能の劣化や信頼性の低下が起きてしまう．

本研究では，NVM 利用を最適化する，ユーザレベルの POSIX ファイルシステムである *EvFS* を提案する．*EvFS*

[1] IBM 東京基礎研究所
[a)] tyos@jp.ibm.com
[b)] chiba@jp.ibm.com
[c)] horii@jp.ibm.com

はユーザレベルのストレージスタックの上で動作することでコンテキストスイッチを最小限にする．*EvFS* はダイナミックリンクライブラリとして提供され，既存のアプリケーションコードやバイナリの変更を必要としない．ライブラリは SPDK のユーザレベルストレージスタックで構成されており，システムコールの呼び出しを最小限にした POSIX インターフェイスを提供する．SPDK は NVMe と NVMM 両方のドライバをサポートしており，*EvFS* は将来増加していくデバイスへの対応のための拡張も可能である．また，*EvFS* はページキャッシュを提供している一方で，それをバイパスする direct I/O もアプリケーション開発者に提供している．これにより，読み出しの割合の大きなワークロードや，キャッシュを簡単に提供できるようなケースでは，direct I/O を使うことで NVM の性能をアプリケーションに最大限活かすことも可能になっている．

EvFS はイベント駆動型のアーキテクチャをとっており，非ブロッキング I/O をナノ秒レベルのレイテンシで処理することを可能にする．ブロッキング I/O についてもユーザレベルストレージスタックの効果により数十マイクロ秒のレイテンシを削減する．その結果，*EvFS* は少ないスレッド数でも DRAM および NVM の帯域幅を十分活用することができる．

本研究はファイル I/O のためのマイクロベンチマークである FIO による予備実験結果も報告する．その結果では *EvFS* は 64 バイトの非ブロッキング書き込みに対し 700 ns のレイテンシを示し，ブロッキング I/O に対してはジャーナリングを無効にした EXT4 と比較して 20 マイクロ秒のレイテンシを削減していた．

2. 研究の動機

本節ではこれまでユーザアプリケーションの高速化に向け，提案されてきたユーザレベルファイルシステムについて議論し，本研究で解決する課題についてまとめる．

2.1 関連研究

Moneta-D [9] および Arrakis [10, 11] はハードウェア支援による仮想化を利用することで，ユーザレベルファイルシステムを実現している．それらファイルシステムはセキュリティチェックなどのソフトウェアの複雑性をハードウェアに移譲することで，性能を向上させている．具体的には，近年の高機能化されたストレージの特徴である，デバイス内の DRAM メモリによるキャッシュ，メモリ保護機能，flash translation layer による耐摩耗性の向上などを前提としている．しかし，そうした前提となる機能は多くはコストの観点から限定的になる傾向にある．例えばデバイス内のキャッシュはメインメモリに比べてかなり小さいかそもそも提供されていない場合が多い．その場合，ハードウェアキャッシュはソフトウェアでエミュレートするなどして提供する必要があり，ソフトウェアの複雑性の増加は避けられない．

Aerie [12] は NVMM の利用を前提としたファイルシステムを提供している．特に，Aerie は NVMM の高い性能を前提としてページキャッシュを排除しているものの，別のファイルシステムである HiNFS [6] は NVMM の高い書き出しレイテンシを削減するために，NVMM の特性 (CPU キャッシュラインのサイズなど) を考慮したページキャッシュを導入することによって性能を向上させている．Aerie は他の NVMM 向けのカーネルレベルファイルシステム [6, 13–16] と同様に，NVMM 領域の `mmap` を可能にしている．しかし，その場合はクラッシュ対策において課題を残してしまっている．NOVA-Fortis [16] はクラッシュ対策することで，ページフォルトによる性能劣化を招いてしまっている．

ScaleFS [17] と Strata [18] はユーザレベルのロガーを利用することで，プロセス単位でファイル更新を行い，カーネルファイルシステムで集約することを可能にしている．彼らはカーネル内のページキャッシュをユーザプロセス内へ移動しており，キャッシュへの読み書きはコンテキストスイッチを必要としない．ログは `fsync` の際にまとめて集約されるため，ユーザは複数の書き込みを集約することになり，その結果全体のスループットが向上する．しかし，`fsync` は最終的にカーネルファイルシステムに書き出しを行うため，コンテキストスイッチが避けられない．

FUSE はユーザプロセス内で動作するファイルシステムの実装を支援する．FUSE アプリケーションは OS へマウントポイントを簡単に公開することができると同時に実装において柔軟な実装を可能にする．その結果，GlusterFS や HDFS などの分散ファイルシステムの実装に FUSE は利用される傾向にある．しかし，FUSE はプロセス間通信のコストが必要であり [19, 20]，NVM にとって相対的に大きなレイテンシを引き起こしてしまう．

SPDK は RocksDB の拡張のための簡素なユーザレベルファイルシステムである BlobFS [21] も提供している．我々は初期段階では BlobFS を拡張していたものの，*EvFS* は初期の設計から大きく変わっている．例えば，BlobFS は POSIX API を提供しておらず，ページキャッシュはシーケンシャルアクセスのみ許している．その結果，*EvFS* は一部のユーティリティを除き，設計レベルで作り直ししている．

2.2 課題まとめと手法比較

NVM 向けのファイルシステムの特徴は，いかにリッチなハードウェア機能や先進的なストレージソフトウェアスタックを利用し，ファイル I/O 処理に関する OS の介在を削減することを目指すかにある．特に，ページキャッシュの介在による冗長なコピーおよびカーネルへのコンテキス

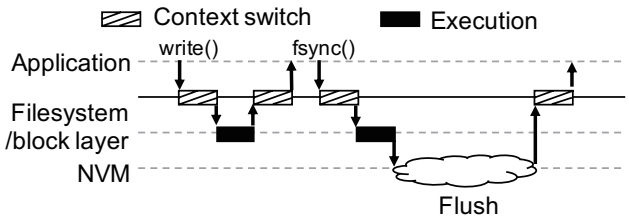

図 1　カーネルファイルシステムへの書き込み．図はページキャッシュへのコピーや I/O 発行などのインメモリ処理を “Exectuion” としてまとめている．図の複雑化を避けるためジャーナリングは除外している．

手法名	Full user stack	Page cache	Direct I/O
BPFS [13] PMFS [14] NOVA [15, 16]	-	-	✓
Arrakis [10, 11]	✓	-	✓
Moneta-D [9]	✓	-	✓
Aerie [12]	✓	-	✓
HiNFS [6]	-	✓	✓
ScaleFS [17] Strata [18]	-	✓	-
EvFS	✓	✓	✓

図 2　NVM 向けファイルシステムの特徴比較

スイッチが問題視されてきている．

図 1 は既存のファイルシステムが最適化を目指してきた原因となったファイル書き出しの概要を端的に表している．カーネルファイルシステムはファイル I/O に関わる複雑な処理を同期的に実行しており，その実行にはシステムコールとプロセスのカーネルコンテキストが介在している．本研究ではファイルシステムのレイテンシの原因をコンテキストスイッチと同期的なファイル I/O 処理の実行の二つに分け，それぞれ対策を行う．

図 2 は *EvFS* と既存の NVM 向けファイルシステムの比較となる．ページキャッシュの排除が多くの NVM 向けのファイルシステムの主要な特徴であった一方で，HiNFS のようにページキャッシュの導入によるメリットも示されており，Arrakis などで前提とされていたハードウェアレベルのキャッシュにも限界がある．そのため，*EvFS* は direct I/O を導入し，ワークロードやデバイスによってページキャッシュを有効にするかどうかを選択できるようにする．NVM 向けファイルシステムとして，ユーザスタックで構成され，ページキャッシュと direct I/O ともにサポートするものは *EvFS* が最初のものとなる．

3. *EvFS*

EvFS はコンテキストスイッチを削減し，非同期的にファイル I/O を実行することでユーザアプリケーションから見てレイテンシを最小限にする．図 3 はカーネルファイルシステムと *EvFS* を比較している．本研究では，システムコールやハードウェア割り込みによるコンテキストスイッチを減らすために SPDK のユーザレベルストレージスタックを利用している．また，非同期的なファイル I/O 処理を実現するために必要となる，イベント駆動型アーキテクチャを構築するために SPDK のユーティリティを利用している．

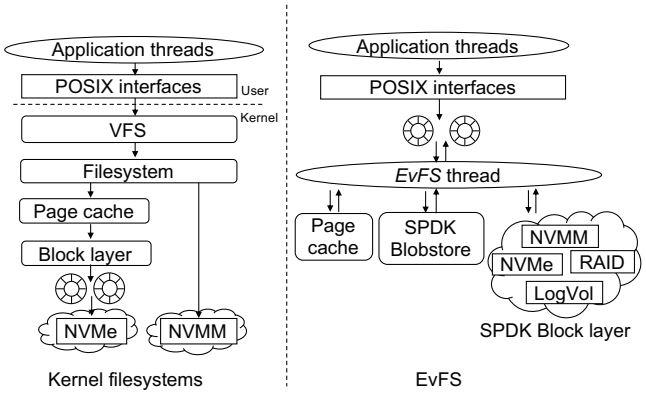

図 3　カーネルファイルシステムと ***EvFS*** の比較．*EvFS* の実装には SPDK のストレージスタックとユーティリティが使用されている．*EvFS* は非同期的なファイル I/O 処理とブロックレベル I/O のために必要となるポーリング用のスレッドを備えている．

3.1 ユーザレベルのストレージスタック

3.1.1 ライブラリ

EvFS は LIBC と同様の POSIX ファイル API (例：open) を持つ共有ライブラリとして提供されている．LIBC より先にロードすることで，*EvFS* は POSIX API をフックすることが可能になる．この設計により，既存の Linux アプリケーションは改変せずに *EvFS* を利用することができる．システムコールはライブラリの関数コールにすりかえられるため，それに伴うコンテキストスイッチを削減することができる．

3.1.2 ユーザレベルのブロックレイヤ

バックエンドのストレージスタックとして SPDK を利用することで NVMe や NVMM の多様なデバイスをブロックレイヤでサポートしている．SPDK は自前のユーザレベル NVMe ドライバに加えて NVMM のための外部ライブラリである PMDK をサポートしている．これにより，*EvFS* は将来のストレージに対しても，新たに SPDK のブロックドライバの追加をすることで対応することができる．SPDK はさらにスケーラブルなスレッドライブラリや，イベントベースの I/O 処理，さらにソフトウェア RAID や論理ボリュームなどの拡張ストレージドライバを提供している．*EvFS* は将来にわたって SPDK の高機能なストレージスタックを利用することで，今後の新しいストレージに対しても簡単に拡張・適応していくことができ

ると考えている．

EvFS は SPDK の Blobstore [22] を利用することで，NVM の名前空間の中で UNIX のファイル・ディレクトリ構造を構築する．Blobstore はフラッシュストレージにおいて簡素な論理ブロックレイアウトを構築・管理している．スーパーブロックでは *BLOB* と呼ばれる，ストレージのブロックページの集まりであるクラスタで構成されたひとつのチャンクを表すもののメタデータを管理している．Blobstore は BLOB に対する読み出し，書き出し，リサイズなどのインターフェイスを提供しており，*EvFS* は BLOB を inode とみなす．しかし， Blobstore 自体にはディレクトリ構造はないため，*EvFS* では簡単な UNIX のディレクトリ構造を BLOB を使ってエミュレートしている．そこでは単にサブディレクトリと他のファイルへのポインタを BLOB 内に書き出している．

Blobstore の制約として，書き出しの前に必ず書き出すオフセットより BLOB の大きさが大きくなければならないというものがある．したがって，*EvFS* は BLOB のサイズを追跡しており，書き出しがそれよりも大きければリサイズをするようにしている．それぞれの BLOB は BLOB の名前とサイズをメタデータに保持しており，それらはインメモリで管理されている．そのため，BLOB の書き込みを確定するためには BLOB のメタデータ更新をストレージに同期する API を明示的に呼ぶ必要がある．Blobstore 自体にはページ単位でのクラッシュ保護機構があるものの，*EvFS* はその同期 API を使ってファイルレベルの一貫性を管理する必要がある．

3.1.3 ユーザレベルのページキャッシュ

SPDK のストレージスタックに加ええ，ひとつの BLOB への複数の読み書きを集約する，ユーザレベルページキャッシュを導入する．ページキャッシュを利用することで，ユーザアプリケーションはメモリ利用率が低いときは DRAM レベルのスピードでファイル読み書きが可能となり，メモリ利用率が高い時は NVM レベルのスピードでの読み書きとなる．また，ユーザアプリケーションは必要に応じて `open` の際のフラグを変更することで， direct I/O を利用しページキャッシュを使わない選択もできる．それにより，ページキャッシュ利用で発生する冗長なメモリコピーを減らすことができる．

EvFS ではページキャッシュ内のメモリ利用量に制限をつけることができる．その場合，汚れたページキャッシュの割合があらかじめ与えられた閾値を超えたときにメモリ上のデータが NVM へ追い出されることになる．また，ページキャッシュに書き出すためのメモリ領域が足りない場合，*EvFS* はユーザの書き出し要求を遅延させることがある (3.2.2 節)．その時に書き出し要求中のスレッドは他の進行中の書き込みが完了し，そのメモリ領域が解放されるまでスリープすることになる．

種別	APIs
ファイル	`open`, `read`, `write`, `pread`, `pwrite`, `lseek`, `close`, `__xstat`, `__lxstat`, `__fxstat`, `posix_fadvise`, `fsync`, `unlink`, `unlinkat`, `stat`, `access`
ディレクトリ	`opendir`, `readdir`, `closedir`, `mkdir`, `truncate`, `ftruncate`, `creat`
フラグ	`O_SYNC`, `O_DIRECT`
スレッド	`pthread_create`, `__libc_start_main`

図 4 ***EvFS*** **でサポートされている API (Ubuntu 18.04 LTS).** 図では 64 ビット版の API, 例えば open64 などを省略している． `__libc_start_main` は最終的にアプリケーションの main 関数を呼び出す．API の正確な名称は OS や LIBC, CPU の種別，その他の環境に依存している．

3.1.4 プライベートマウントポイント

ユーザアプリケーションは POSIX インターフェイスを介して NVM 名前空間へ排他的にアクセスすることができる．ライブラリがロードされた段階で，*EvFS* はアプリケーション専用のマウントポイントを構築する．そのマウントポイントは現状では OS カーネルや他のユーザプロセスからアクセスされることや読み出されることはない．今後，既存研究 [11] でされたように，マウントポイントを公開するためのインターフェイスを提供することを考えている．また，本研究では NVM 名前空間はハードウェアによって提供されたものを前提としている．理論的には SPDK がすでに提供している論理ボリュームや既存の仮想ボリューム [23] を利用することで，名前空間のサポートのない NVM でも *EvFS* と他のファイルシステムと単一 NVM 内で共存させることができる．

3.1.5 POSIX インターフェイス

図 4 は現在の *EvFS* がサポートしている API を示している．*EvFS* は `open` の呼び出しで inode, つまり BLOB とファイルディスクリプタ (FD) を対応づける．FD の番号は他のファイルシステムで開いたファイルとの衝突を避ける必要があるため，特別なファイル (Linux では `/dev/null`) を一度オープンし，その番号を再利用する．`O_SYNC` など，オープン時に指定できるフラグを制御することで，ユーザはブロッキングの可否などを制御することができる．例えばユーザは `O_SYNC` を指定しないようにすることで非ブロッキング書き出しが可能になり，`O_DIRECT` を指定することでページキャッシュの介在を明示的に避ける direct I/O が可能になる．*EvFS* は他の API と同様，将来的に `mmap` をサポートする予定となっている．

EvFS は他にも，スレッド開始関連の API (`pthread_create` and `__libc_start_main`) をフックすることで，アプリケーションスレッドごとに I/O チャネルを関連づけている．スレッドごとのチャネルを利用することで，同期コストや並列ブロック I/O によるコスト [24, 25] を避けることが可能になる．頻繁なメモリ確保

はカーネルへのコンテキストスイッチや，ページテーブル更新などによる性能劣化が発生し，さらにはスケーラビリティの問題が発生する恐れもある [26].

3.2 イベント駆動アーキテクチャ

3.2.1 イベント記述子

EvFS では，専用のスレッドがイベントキューをポーリングし，発行されたイベントを適時実行している．それぞれのイベントはイベント記述子によって構成されており，それぞれがターゲットとなる BLOB，オフセット，サイズ，ユーザバッファへのアドレス などの API 引数と対応するコールバック関数へのアドレスを持つ．このファイル I/O 処理では，SPDK のイベント駆動型の実行モデルを踏襲している．

イベントの処理フローを NVMe をマウントしたファイルシステムに対して `read` を呼び出した場合を例に説明する．その場合，`read` ははじめにイベント記述子を確保し，そこに *EvFS* 内部の `read` 用のコールバックと `read` のための引数および I/O 完了通知のためのセマフォをコピーする．そして，イベント記述子をイベントキューに挿入し，ポーリングしているスレッドが挿入されたイベント記述子を取り出してコールバックを渡された引数で呼び出す．そのコールバックはさらに別のイベント記述子を確保し，Blobstore の読み出し用のコールバックと完了ハンドラのコールバック，そして読み出しのための引数をコピーし，イベントキューへ挿入する．ポーリングスレッドが再度そのイベントを読み出して Blobstore の読み出しを実行する．

このような連続したイベント挿入は最終的に SPDK のユーザレベル NVMe デバイスドライバを呼び出す．NVMe ドライバはメモリマップト I/O を通じてデバイスと通信し，DMA によるデータ転送を発行する．残念ながら，`read` の引数にあるユーザバッファが DMA 可能であることを保証できないため，直接ユーザバッファを使って NVMe とのデータ転送はできない．代わりに，*EvFS* は DMA 可能としているページキャッシュを経由して転送するか，direct I/O の場合は一時的に DMA 可能なメモリを確保してデータ転送を行う．ただし，NVMM のような DMA の不要なストレージで，かつ direct I/O の場合はユーザバッファを直接使ってデバイスとのデータ転送を行う．

DMA 要求の後，*EvFS* のスレッドは NVMe のドアベルレジスタをポーリングし，I/O の完了待ちをする．完了を検知した場合，スレッドがそのまま上位レベルの完了ハンドラを同期的に呼び出す．例えば，Blobstore の読み出しのための完了ハンドラが呼び出され，そのハンドラはさらにイベント記述子にあるセマフォに対して完了通知を行い，最初のユーザスレッドへ完了通知を行う．同期的な I/O 待ちはハードウェア割り込みを排除し，それにより *EvFS* のレイテンシを削減する．

POSIX API はブロッキング・非ブロッキングファイル I/O 両方サポートしている．ブロッキング I/O では，ユーザスレッドは I/O の完了を必ず待たなければならない．その場合，*EvFS* はユーザバッファのアドレスをそのままイベント記述子にコピーし，DMA バッファから直接データをコピーする．非ブロッキングファイル I/O の場合，ユーザアプリケーションはイベント挿入してすぐさま呼び出し元へリターンすることができる．その場合，ユーザバッファが更新される可能性があるため，*EvFS* は一時的なメモリ確保をしてそこへユーザバッファの内容をコピーし，そのアドレスをイベント記述子へコピーする．この時に *EvFS* はできるだけメモリ確保によるカーネル呼び出しを減らすために，簡単なメモリプールを持ち，そこから確保したメモリを一時メモリとして利用している．

3.2.2 依存関係のあるイベント処理の遅延

EvFS は時々複雑なイベント依存関係を対処する必要がある．その場合，イベント記述子を依存しているオブジェクトのキューへつなげ，そのオブジェクトに対する処理が完了するのを待ってイベントが処理できるようにする．例えば，BLOB 書き出しがそのサイズを超えたオフセットで起きた場合，書き出しはリサイズの後に実行する必要がある．その場合，書き出しイベントを依存しているリサイズのイベントのキューへつなげる．また，*EvFS* はページキャッシュをサポートしているため，書き出しが NVM へのライトバック中に起こった場合の依存関係も同様に処理する必要がある．その場合，その書き出しイベントを依存しているライトバック中のページキャッシュページに対してつなぎ，ライトバックが終わるまでイベント処理を遅延させる．他にもページキャッシュメモリが逼迫した場合や `fsync` が呼ばれたときに汚れていないページを解放する必要がある．`fsync` が完了した後，ユーザは `fsync` が呼ばれる前の API が完了していることを期待するため，*EvFS* は現在進行中のイベントを管理するキューへバリアイベントをつなぎ，進行中のイベントが全て完了したときに同期的にキャッシュの追い出しを開始できるようにする．

4. 実験

本節では，*EvFS* の FIO によるマイクロベンチマークの実験結果を示す．特に，*EvFS* と EXT4 の性能を比較する．FIO は POSIX ファイル I/O を実行するベンチマークで，実験結果として 10 回実行した結果を示す．それぞれの実行では，FIO スレッドは 40GB のランダム読み出しや書き出しを行う (混合させた読み書きは行わない)．Power9 AC922 (160 Power9, 3.8 GHz 論理 CPU コア，1TB メモリ, 6.4TB Samsung NVMe 172Xa) 上の Ubuntu 18.04 LTS で評価する．結果の分析を簡単にするため，ブロックレイヤの先行読み出しや EXT4 のジャーナリングは無効

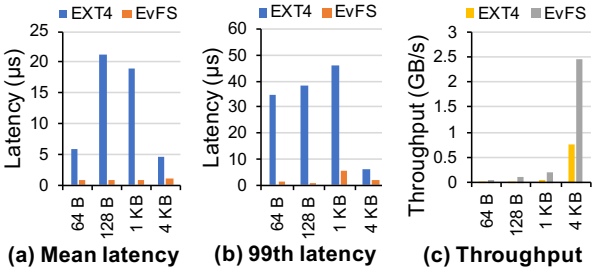

図 5 非ブロッキング書き出しのレイテンシ．(a) と (b) は *EvFS* と EXT4 において異なるサイズの I/O を行なった場合の平均と 99th パーセンタイルのレイテンシを示している．それぞれのワークロードのスループット結果は (c) に示されている．

にしている．メモリは逼迫していない状況での結果を示している．他のファイルシステムと同様に，*EvFS* はメモリが逼迫している時により高いレイテンシ，低いスループットを示す．ただし，メモリが十分なときでも，汚れたページキャッシュが閾値を超えた時に *EvFS* は他のファイルシステムと同様にキャッシュの追い出しを行なっている．実験ではその閾値として 20% を設定している．

4.1 非ブロッキング I/O

図 5 (a) と (b) は EXT4 と *EvFS* において非ブロッキング書き出しをした場合の平均と 99 パーセンタイルのレイテンシを示している．*EvFS* は EXT4 に比べて全ての場合で低いレイテンシを示している．このワークロードでは，EXT4 の書き出しレイテンシはカーネルページキャッシュとコンテキストスイッチのオーバヘッドとなる．*EvFS* はユーザバッファをスレッドごとのイベントバッファにコピーしているものの，コンテキストスイッチや同一ページへの書き込みによる待ちを 3.2.2 節の遅延手法により回避している．その結果，*EvFS* による 64 バイトの非ブロッキング書き込みは 700 ナノ秒に到達した．

ナノ秒レベルのレイテンシの結果，シングルスレッドでの高いスループットにつながっていた．図 5 (c) は 4KB の書き込みにおいて大きなスループットの増加を示している．

関連手法として，LIBAIO と POSIXAIO による非同期 I/O による実験も行っている．しかし，それらはカーネルの介在やスレッド生成などの複雑な処理の結果，EXT4 と比べても良いレイテンシは示さなかった．非同期 I/O の実装を *EvFS* が提供することで，さらに性能改善が見込める可能性がある．

4.2 Direct I/O

図 6 は direct I/O と buffered I/O それぞれの平均および 99 パーセンタイルのレイテンシを示している．この実験では buffered I/O として `write` と `fsync` のペアを実行している．*EvFS* は direct I/O による読み書き両方に

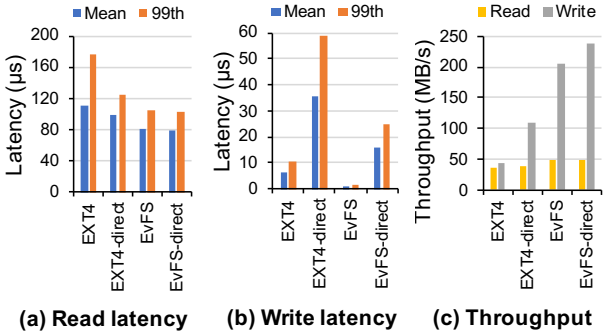

図 6 **direct/ buffered I/O の性能**．(a) と (b) は EXT4, EXT4 への direct I/O (EXT4-direct), *EvFS*, *EvFS* への direct I/O (*EvFS*-direct) の平均および 99 パーセンタイルのレイテンシを示している．それぞれのワークロードのスループット結果は (c) に示されている．

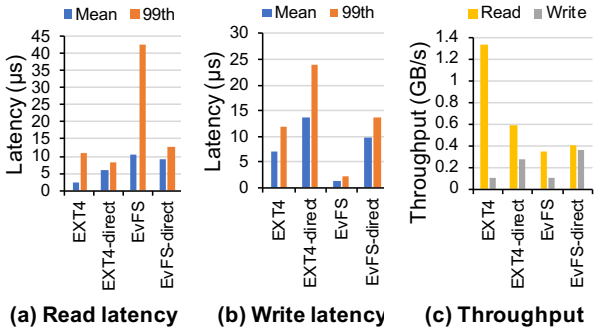

図 7 **Ramdisk 上での direct/ buffered I/O の性能**．(a) と (b) は EXT4, EXT4 への direct I/O (EXT4-direct), *EvFS*, *EvFS* への direct I/O (*EvFS*-direct) の平均および 99 パーセンタイルのレイテンシを示している．それぞれのワークロードのスループット結果は (c) に示されている．

ついて，EXT4 に比べて平均のレイテンシを 20 マイクロ秒に削減している．その結果，EXT4 に比べて書き出しのスループットは 2.2 倍，読み出しは 1.3 倍となっている．*EvFS* の direct I/O による読み出しは buffered I/O による読み出しに比べて 2 マイクロ秒削減している．この結果は *EvFS* は非常に低い読み書きレイテンシを持つ将来の NVM に対して十分な性能を引き出すことが期待される．レイテンシに敏感なアプリケーションは自前でキャッシュを持つべきではあるものの，direct I/O を使うことで性能を向上させられると予想される．

4.3 PMDK

4.1、4.2 節では Samsung NVMe 172Xa をストレージとして用いていた結果を示したものの，*EvFS* は NVMe 以外の種別のストレージにも利用ができる．例えば，インメモリファイルシステム (tmpfs) 上のファイルをマッピングした PMDK を利用することで，NVMM 上での *EvFS* の動作をエミュレートすることができる．ここでは，分析を簡単にするため，性能のノイズやメモリコピーの追加のレ

イテンシは特に入れず，NVM が DRAM と全く同じレイテンシを出せた場合のエミュレーションをする．

図 7 は PMDK 上でのエミュレーション環境での *EvFS* と RAMdisk 上で動作している EXT4 に対し，4.2 節と同様に，direct/buffered I/O を用いて比較した結果を示している．この実験からは，direct/buffered I/O の間の性能特性の差と，PMDK による性能の影響について大まかな考察をすることができる．RAMdisk の場合でも direct I/O を選択しない場合にページキャッシュへ読み書きされるため，direct I/O との性能特性が異なる．ただし，RAMdisk 自体の性能に依存する実験となるため，この結果がそのまま将来的に利用できるようになる NVM と全く同じ性能特性にはならないことに注意する必要がある．

RAMdisk は NVM と同様に極めて低いレイテンシで読み出すことができ，NVMe を利用した場合との最大の違いになる．一方で，書き出しに関しては十分最適化されておらず，レイテンシは読み出しよりも大きくなる傾向にある．図 7 (b) では EXT4 に対して *EvFS* がより低いレイテンシを示していることから，より小さいレイテンシの NVM においても *EvFS* が高い書き込み性能を出すことが予想される．ただし，読み出し性能に関しては依然最適化を進める必要があることがわかる．図 7 (a) について，*EvFS* と EXT4 を比べると，EXT4 のほうがレイテンシが小さい．この差は PMDK 内の NVM ブロックドライバと，SPDK 内の Blobstore が同じようなページ単位の読み書きの最適化や信頼性の向上を行っており，ストレージスタック内で冗長な処理が含まれていることが要因になっていると考えている．この最適化には PMDK 側をより簡素化し，Blobstore 側でできるだけ処理を集約していく必要があると考えている．

5. まとめと今後の課題

本研究では，NVM 利用を最適する，ユーザレベル，イベント駆動，POSIX ファイルシステムである *EvFS* を提案した．イベント駆動アーキテクチャは低いレイテンシと高いスループットの I/O を可能にする．*EvFS* は DRAM を活用することで，NVMM の制限のある帯域幅の問題を抑制する可能性があった．しかし，*EvFS* は全ての POSIX API を提供しておらず，クラッシュ保護機能も十分ではない．実験結果から，20 マイクロ秒のブロッキング I/O のレイテンシ削減や，700 ナノ秒の非ブロッキング書き出しを示していることから，イベント駆動型のアーキテクチャはそういった複雑な機能を持つ将来の *EvFS* や NVM のレイテンシを最小限にすることを可能にすると考えている．ただし，4.3 節の結果から，PMDK を利用して NVMM へアクセスする場合，PMDK と SPDK の連携をより最適化していくことがさらに必要になると考えられる．

将来の POSIX API の中で特に，`mmap` をどのように実装するかを十分検討していく必要があると考えている．特に，NVMM 領域を直接マッピング可能にするべきか，それともページキャッシュ領域をマッピングさせるべきかが焦点になる．

前者の方法では，他のファイルシステムと同様の提供方法となるものの，NVM の複雑性をアプリケーションに押し付けることになる．アプリケーションは適切にメモリフェンスを貼らなければ，簡単に性能劣化や一貫性の破壊を引き起こしてしまう．また，遅い書き出しレイテンシがアプリケーションの性能を劣化させる可能性もある．

ページキャッシュ領域のマッピングはアプリケーションからみて，他のストレージと同様にマッピング領域を扱うことが可能になる点が利点となる．また，マッピングした領域の読み書きは DRAM のスピードで扱える点も利点となり，NVM のサイズが少なくとも，DRAM を使ってより巨大な領域に `mmap` することが可能になる．しかし，このアイディアは同時に DRAM と NVM の間のページ交換をうまく行う必要がある点が最大の課題となる．最適なページ交換をしない限り，頻繁なページフォルトによりアプリケーションの性能劣化につながる可能性が高い．

参考文献

[1] : Intel® Optane™ Memory Series (32GB, M.2 80mm PCIe 3.0, 20nm, 3D Xpoint) Product Specifications, https://ark.intel.com/content/www/us/en/ark/products/series/99743/intel-optane-memory-series.html.
[2] Zhang, X., Feng, D., Hua, Y. and Chen, J.: Optimizing File Systems with a Write-Efficient Journaling Scheme on Non-Volatile Memory, *IEEE Transactions on Computers*, Vol. 68, No. 3, pp. 402–413 (2019).
[3] Yang, Z., Harris, J. R., Walker, B., Verkamp, D., Liu, C., Chang, C., Cao, G., Stern, J., Verma, V. and Paul, L. E.: SPDK: A Development Kit to Build High Performance Storage Applications, *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom '17)*, pp. 154–161 (2017).
[4] Kim, H.-J., Lee, Y.-S. and Kim, J.-S.: NVMeDirect: A User-space I/O Framework for Application-specific Optimization on NVMe SSDs, *Proceedings of the 8th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage '16)*, pp. 41–45 (2016).
[5] Tsai, C.-C., Jain, B., Abdul, N. A. and Porter, D. E.: A Study of Modern Linux API Usage and Compatibility: What to Support when You'Re Supporting, *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*, pp. 16:1–16:16 (2016).
[6] Ou, J., Shu, J. and Lu, Y.: A High Performance File System for Non-volatile Main Memory, *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*, pp. 12:1–12:16 (2016).
[7] Boukhobza, J., Rubini, S., Chen, R. and Shao, Z.: Emerging NVM: A Survey on Architectural Integration and Research Challenges, *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, Vol. 23, No. 2, pp. 14:1–14:32 (2017).
[8] Zhang, X., Feng, D., Hua, Y. and Chen, J.: Optimizing

File Systems with a Write-Efficient Journaling Scheme on Non-Volatile Memory, *IEEE Transactions on Computers*, Vol. 68, No. 3, pp. 402–413 (2019).

[9] Caulfield, A. M., Mollov, T. I., Eisner, L. A., De, A., Coburn, J. and Swanson, S.: Providing Safe, User Space Access to Fast, Solid State Disks, *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*, pp. 387–400 (2012).

[10] Peter, S., Li, J., Woos, D., Zhang, I., Ports, D. R. K., Anderson, T., Krishnamurthy, A. and Zbikowski, M.: Towards High-performance Application-level Storage Management, *Proceedings of the 6th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage '14)* (2014).

[11] Peter, S., Li, J., Zhang, I., Ports, D. R. K., Woos, D., Krishnamurthy, A., Anderson, T. and Roscoe, T.: Arrakis: The Operating System is the Control Plane, *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI '14)*, pp. 1–16 (2014).

[12] Volos, H., Nalli, S., Panneerselvam, S., Varadarajan, V., Saxena, P. and Swift, M. M.: Aerie: Flexible File-system Interfaces to Storage-class Memory, *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*, pp. 14:1–14:14 (2014).

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

[14] Dulloor, S. R., Kumar, S., Keshavamurthy, A., Lantz, P., Reddy, D., Sankaran, R. and Jackson, J.: System Software for Persistent Memory, *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*, pp. 15:1–15:15 (2014).

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

[16] Xu, J., Zhang, L., Memaripour, A., Gangadharaiah, A., Borase, A., Da Silva, T. B., Swanson, S. and Rudoff, A.: NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System, *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, pp. 478–496 (2017).

[17] Bhat, S. S., Eqbal, R., Clements, A. T., Kaashoek, M. F. and Zeldovich, N.: Scaling a File System to Many Cores Using an Operation Log, *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, pp. 69–86 (2017).

[18] Kwon, Y., Fingler, H., Hunt, T., Peter, S., Witchel, E. and Anderson, T.: Strata: A Cross Media File System, *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, pp. 460–477 (2017).

[19] Vangoor, B. K. R., Tarasov, V. and Zadok, E.: To FUSE or Not to FUSE: Performance of User-Space File Systems, *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*, pp. 59–72 (2017).

[20] Zhu, Y., Wang, T., Mohror, K., Moody, A., Sato, K., Khan, M. and Yu, W.: Direct-FUSE: Removing the Middleman for High-Performance FUSE File System Support, *Proceedings of the 8th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS '18)*, pp. 6:1–6:8 (2018).

[21] : SPDK: Blobstore Filesystem, https://spdk.io/doc/blobfs.html.

[22] : SPDK: Blobstore Programmer's Guide, https://spdk.io/doc/blob.html.

[23] Nanavati, M., Wires, J. and Warfield, A.: Decibel: Isolation and Sharing in Disaggregated Rack-scale Storage, *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation (NSDI '17)*, pp. 17–33 (2017).

[24] Bjørling, M., Axboe, J., Nellans, D. and Bonnet, P.: Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems, *Proceedings of the 6th International Systems and Storage Conference (SYSTOR '13)*, pp. 22:1–22:10 (2013).

[25] Min, C., Kashyap, S., Maass, S., Kang, W. and Kim, T.: Understanding Manycore Scalability of File Systems, *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference (USENIXATC '16)*, pp. 71–85 (2016).

[26] Clements, A. T., Kaashoek, M. F. and Zeldovich, N.: RadixVM: Scalable Address Spaces for Multithreaded Applications, *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*, pp. 211–224 (2013).