Vol.2022-HPC-184 No.3
2022/5/11

不揮発性メモリを用いた分散オブジェクトストレージの設計

巨畠 和樹[1,a)]　建部 修見[2]

概要：
ビッグデータや機械学習アプリケーションのような I/O が全体の処理のボトルネックとなることが指摘されているワークロードでは計算性能だけでなくストレージ性能が特に求められている．近年登場した不揮発性メモリは, 計算ノードとストレージノードの間のホットストレージ層としての利用が検討されている．しかし I/O ノードはユーザからその数を指定することはできないため, アプリケーションに十分なスケーラビリティを得られない可能性がある．また, 不揮発性メモリを用いたキーバリューストアである pmemkv を利用したファイルシステムが存在するが, キーの数が増えると挿入操作のスループットが低下する問題がある．本研究では不揮発性メモリを利用した分散オブジェクトストレージである PXNO を提案する．PXNO は計算ノード上の不揮発性メモリを活用することで, ユーザがストレージのノード数を指定できるようにする．さらに分散オブジェクトストレージを構築することで, pmemkv のスループット低下を防ぐ．筑波大学計算科学研究センターのスーパーコンピューターである Cygnus による評価では, 32 ノードまで性能がスケールすることが分かった．また, YCSB ベンチマークによる評価では read heavy なワークロードで既存のストレージよりも最大で 17.8 倍の性能を達成した．

1. はじめに

ハイパフォーマンスコンピューティング（HPC）の分野では計算能力の向上だけでなく, I/O 性能の向上が求められている [15]．近年のビッグデータや機械学習のワークロードの増加に伴い, I/O が HPC システム全体のボトルネックとなっていることから, I/O 性能の向上はこのような分野の問題解決にかかる時間を短縮する点で重要である．特に大規模データ解析や機械学習などのアプリケーションは, 従来の書き込みのシミュレーションやワークフローの多かった科学技術計算のアプリケーションとは異なり, 読込と少量の書き込みを伴うアクセスパターンを持つ読込集中型が多い [16]．このようなワークロードでは多数のメタデータ要求によって性能が制限される並列ファイルシステムよりもバーストバッファ [21] のような, 計算ノードとストレージノードの中間となるストレージ層を用いることで, より高速な I/O を提供するシステムを用いるほうが有効である．従来のバーストバッファでは SSD が利用されていたが, SSD よりも高帯域幅・低レイテンシであり, バイト単位でのアクセスが可能な不揮発性メモリ [8] の登場によりこれをバーストバッファで利用することが可能となった．不揮発性メモリデバイスをそのまま用いる場合, 従来のブロックアクセスに最適化されたソフトウェア層が無駄なレイテンシを生み出し, 不揮発性メモリの性能を十分に発揮できていないことから, 不揮発性メモリを用いた新たなソフトウェアスタックの設計が行われている [11]．不揮発性メモリを用いたソフトウェアを開発する場合, 突然の電源喪失によりデータが壊れる可能性を考慮しなければならない．PMDK のトランザクション機能 [17] はデータが破損しないように読み書きするためのアトミックな操作を提供する．トランザクション機能を用いた pmemkv というキーバリューストアを用いたファイルシステムが提案されているが [18], キーの数が増えるとキーバリューの挿入操作のスループットが低下する問題がある．また, バーストバッファと同様の役割を果たす I/O ノードに不揮発性メモリを設置するシステムも登場している [13]．I/O ノードを用いるシステムでは, 並列ファイルシステムと同様にストレージのスケーラビリティをユーザが指定できないため, 計算ノードの増加に対してスケーラビリティを妨げてしまう．

本研究では, 計算ノード上のストレージデバイスを用いることでユーザがスケーラビリティを変更できる分散オブジェクトストレージを設計する．なお, ミドルウェアで利用できるシンプルなインターフェースについてその設計を行い, オブジェクトストレージの上で構成される I/O ミドルウェアについては設計を行わない．本研究のオブジェク

[1] 筑波大学大学院理工情報生命学術院システム情報工学研究群情報理工学位プログラム
[2] 筑波大学計算科学研究センター
[a)] obata@hpcs.cs.tsukuba.ac.jp

トストレージを PXNO と呼び, その設計目標について以下にまとめる.

- キーバリューの増加による挿入操作の性能低下を防ぐ
- ユーザがスケーラビリティを選択できるストレージを構築する
- オブジェクトストレージとしてシンプルな API を提供する

2. 関連研究

分散オブジェクトストレージに関する研究で, 特に不揮発性メモリを用いたストレージシステムとして DAOS [4], [11], [13] が挙げられる. DAOS はメタデータなどの小さなデータを不揮発性メモリに格納し, サイズの大きなデータは NVMe SSD に保存する. DAOS は計算ノードと並列ファイルシステムの中間に位置する I/O ノードに配置される. DAOS の I/O ノードには並列ファイルシステムなどの共有ストレージにデータを転送する役割やデータのフォーマット変換, さらにオンラインデータ解析の機能も含まれるが, 本研究ではストレージとしてのデータアクセス性能のみに着目し, このような機能については設計・実装を行わない. PXNO では DAOS のようにバージョニングによるストレージ容量の圧迫とバージョンの圧縮といったバックグラウンド処理を行わないような設計を行うことで, ストレージに必要な資源を効率的に利用しながら高い性能を提供する.

CHFS [18] は不揮発性メモリに特化したアドホックファイルシステムである. 不揮発性メモリに最適化された KV ストアである pmemkv を利用してコンシステントハッシングで分散 KV ストアを構築し, その上にファイルシステムを実装している. PXNO でもコンシステントハッシングを用いるが, pmemkv のハッシュマップエンジンの性能上の問題の解決のための設計を提案する. またフラットな名前空間でのデータ管理を行うオブジェクトストレージを構築することでファイルシステム実装をミドルウェアに移行し, データ管理や API をより単純なものにした.

MOSIQS [10] は不揮発性メモリプールを用意して, 各計算ノードから高速なインターコネクトを介して直接不揮発性メモリプール上のオブジェクトにアクセスできるようにするためのフレームワークを提供している. さらにデータアクセスを効率化するために永続インデックスデータ構造を導入し, 検索処理時間を短くするシステムを提案している. MOSIQS は不揮発性メモリがメモリプールとして集中管理されている場合にのみ利用でき, 分散した環境では利用できない. PXNO は計算ノードに分散した不揮発性メモリをひとつのストレージとして公開する役割を担う.

GekkoFS [19] は分散ファイルシステムのひとつで, RocksDB [5] をメタデータの格納に利用している. RocksDB は SSD に最適化された組み込みデータベースであり, LSM ツリーを基に構成される. RocksDB は LSM ツリーのコンパクション操作について性能が低下することが知られている. 本研究ではシングルノードにおける RocksDB との性能の比較を行い, 様々なアクセスパターンで評価することで PXNO の性能の特性を明らかにする. YCSB ベンチマークによる評価では, コンパクションが発生していない場合でも, 多くのワークロードで PXNO の方がスケーラビリティに優れており, より高い性能を発揮することを示す.

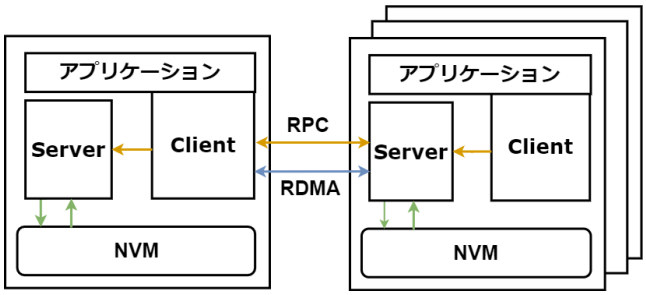

図 1: PXNO の概要図

3. 設計

HPC アプリケーションでは POSIX I/O のセマンティクスのすべてを利用しているわけではない [20] ため, いくつかの POSIX I/O についてはその制限を緩和することでストレージとしてのスケーラビリティを高める試みが行われている [2], [11], [18], [19] . オブジェクトストレージは POSIX I/O セマンティスクを緩和することにより高いスケーラビリティを可能にする [12]. また階層的なストレージで多く用いられるメタデータサーバを使わず, データをオブジェクトとしてフラットな名前空間に格納するためメタデータアクセスがボトルネックとなりにくく, データの検索を効率的に行うことができる. データをオブジェクトとして抽象化しているので, オブジェクトを用いたミドルウェアを設計することで様々な I/O インターフェースに対応することもできる. したがって, 本研究では計算ノードのストレージを用いたオブジェクトストレージとして PXNO を設計する.

図 1 に PXNO の構成の概要図を示す. PXNO ではサーバ・クライアントモデルを採用する. サーバは計算ノードに常駐するプロセスで, 不揮発性メモリを利用してデータをオブジェクトとして保存する. クライアントはサーバに対して Remote Procedure Call (RPC) を発行し, データの分割や転送などの処理を行う. アプリケーションはクライアントライブラリを用いてデータの受け渡しや受け取りを行う.

不揮発性メモリを利用する場合, 唐突な電源喪失より一貫性のない中途半端な状態でデータが保存される可能性がある. この問題を解決するために PMDK [7], [17] の

トランザクション機能を利用する．PMDK は Persistent Memory Development Kit のことで, 不揮発性メモリに対して直接アクセスする方法を提供するライブラリ群である．PMDK の libpmemobj ライブラリではトランザクション機能を提供しており, 停電などの障害が発生してもコミットされていない操作をロールバックしてデータの破損を防ぐ．libpmemobj ライブラリを用いた KV ストアである pmemkv はトランザクション機能を利用して突然の電源損失などの障害によりデータが破損しないようにキーバリューの挿入や取得といった操作を提供する．さらにゼロコピーでデータアクセス可能な API を提供しているので, 直接不揮発性メモリ上のデータに読み書きできる．pmemkv では複数のストレージエンジンが実装されており, 例えば B-tree や Radix tree といったツリーベースのエンジンやハッシュマップベースのストレージエンジンがある．特に同時実行をサポートしていてデータが不揮発なエンジンは実験導入されているエンジンを除くと執筆現在（2022 年 4 月）ではハッシュマップを利用した `cmap` のみである．前節で挙げた CHFS ではこの `cmap` エンジンを用いて分散キーバリューストアを構築している．ハッシュマップではキーの配置をハッシュで決定する．ハッシュが衝突した場合は再度キーの配置先を決める操作が必要となることから, キーの数が多くなると性能が低下することが知られている．

pmemkv のハッシュマップ実装である `cmap` の性能について調査を行った結果を図 2 に示す．実験環境は表 1 に示す計算機を利用した．fsdax（Filesystem-DAX）モードは I/O パスからページキャッシュを取り除き mmap() で不揮発性メモリに直接アクセスできるようにする機能である．対応するファイルシステムで不揮発性メモリデバイスをフォーマットし, DAX マウントすることでこの機能を利用できる．この実験では不揮発性メモリデバイスを xfs でフォーマットし, DAX マウントしてその上にデータストアを構築する．計測は pmemkv で作成したデータベースに対して 1,000 個のキーバリューの挿入 (put) 操作を 10^6 回以上繰り返すベンチマークを 8 スレッドを使って行った．キーのサイズは 20 バイト, バリューのサイズは 256 バイトとした．

実験の結果からキーの数が 3,000 個のときに IOPS が最大となり, その後段階的に性能が低下していることがわかる．キーの数が 4,000 個や 8,000 個のときに性能が急落しているのは, ハッシュテーブルの拡張に伴って不揮発性メモリの新たな領域を確保する操作で時間がかかっているためである [14]．ハッシュテーブルの拡張は $2^N - 1$ 個（N は 8 以上の整数）のキーバリューが挿入された時に行われる．また, キーが 10,000 個を超えると性能は最大値の半分ほどとなることがわかった．CHFS ではファイルはチャンクサイズによって分割されキーバリューとして pmemkv に保存される．また, ディレクトリエントリ作成についてもそのフルパスをキーとしてエントリを pmemkv に追加する．したがって pmemkv を各ストレージノードで単一の KV ストアとして扱う場合, チャンクサイズの小さい場合やファイルやディレクトリ数が多い場合にはキーバリューの数が増えるため, ストレージとしての性能が低下してくと考えられる．

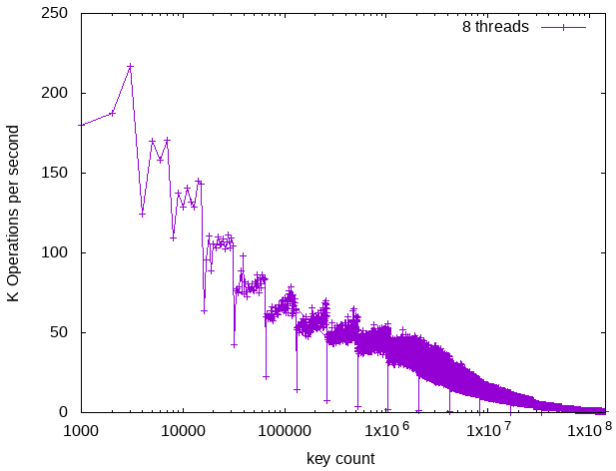

図 2: pmemkv における put の性能評価

3.1 オブジェクト

PXNO ではキーバリューの増加に伴う挿入操作のスループット低下を回避するために複数の KV ストアを構築してこれらを不揮発性メモリで管理する．オブジェクト内のキーバリューの保存例を図 3 に示す．図の例では `key0` から `key5` までのキーバリューをオブジェクト `obj1, obj2` のそれぞれに格納される．各ノードで単一の KV ストアを複数の KV ストアに分割することで, KV ストアあたりの操作回数を減らすことが可能となる．これにより挿入操作におけるキーの衝突を減らすことができるため, スループットの低下を防ぐことができる．図 3 の例では `node1, node2` それぞれのノードで KV ストアを 2 つに分割している．また, CHFS ではメタデータをキーバリューに組み込んでいるが, オブジェクト単位で KV ストアを公開することによって, メタデータをデータとは別のオブジェクトで管理できるため, バリューにデータとメタデータを両方埋め込む必要がなくなる．ファイルシステムなどのミドルウェアは本研究では取り扱わないが, キーバリューの挿入や取得といった単純な API から構築可能である．

性能をスケールさせるためにオブジェクトに対するアクセスを分散させる必要がある．データを分散する際に, ノードの増減によってデータ移動が発生する場合がある．ノードの障害時やその復旧時にデータ移動を最小限に抑える方法としてコンシステントハッシングがある．PXNO は CHFS と同様にリングベースのコンシステントハッシングを用いてオブジェクトの分散を行い, すべてのノードはリ

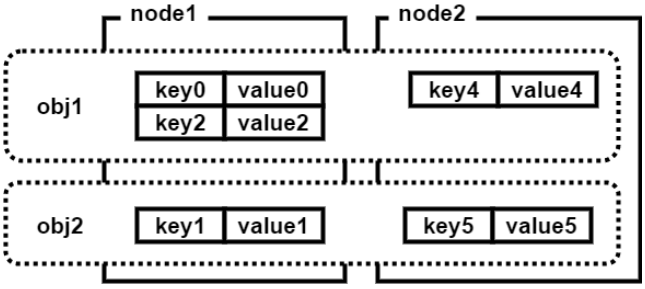

図 3: オブジェクトの構成の例

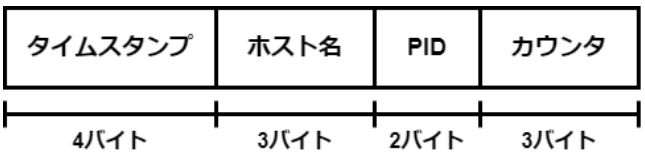

図 4: オブジェクト ID の構成

ングに参加しているノードのアドレスを共有する．各ノードはアドレスのハッシュ値でリング状に分散され, リング上で直前のノードから自ノードまでをそれぞれのノードの担当範囲としてオブジェクトを保持する．

3.2 オブジェクト ID

オブジェクト ID は各オブジェクトに対してアクセスする方法を提供するための識別子で, システムで一意となるようにクライアントで作成される．PXNO では並列環境でオブジェクト ID が重複しないように MongoDB のオブジェクト ID の生成方法を利用する．具体的なオブジェクト ID の構成を図 4 に示す. MongoDB のオブジェクト ID は 96 ビットの 12 バイトから成り, 生成にタイムスタンプやホスト名の MD5 ハッシュ値の上位 3 バイト, プロセス ID, カウンタを用いて組み合わせることでシステムでの一意性を確保している [6].

オブジェクトの分散にはシャードを用いる．シャードはオブジェクトを分散する先の各サーバのことで, シャード値はオブジェクトを分散する先のサーバの台数を表す．オブジェクト ID とシャードを用いてオブジェクトの分散先のサーバをコンシステントハッシングで決定する．

以下にクライアントがオブジェクトを作成する手順を示す.

(1) オブジェクト ID を生成する.
(2) オブジェクト ID のハッシュ値からコンシステントハッシングによりオブジェクトを作成するサーバを決定する .
(3) (2) でオブジェクトを作成したサーバからリング上でシャード値分先のサーバまで, すべてのサーバにオブジェクトを作成する.

オブジェクト ID とシャードの一覧を I/O のたびにサーバから取得するのは, 通信が多く発生してしまうため適切でない. そこで PXNO ではこの対応をクライアントでキャッシュする．オブジェクトの分散先がオブジェクトの作成操作などによって決定された後は, このキャッシュを参照する．キャッシュにオブジェクト ID との対応がない場合や, シャードにアクセスできなければ再度シャードの計算を行うことで分散先のサーバを特定する．キーバリューの操作先のサーバはシャードの一覧から選択される．シャードの一覧上でキーのハッシュ値を基にシャードを選び, 選択されたシャードに対してキーバリュー操作が行われる．

オブジェクトとオブジェクト ID の対応をサーバで管理しておく必要がある．そこでオブジェクト ID はオブジェクト ID 管理用の KV ストアで管理する．オブジェクト ID 管理用の KV ストアではオブジェクト ID から各オブジェクトへアクセスするためのポインタを検索することができる．オブジェクト ID とオブジェクトの紐付けが管理用の KV ストアにない場合は, オブジェクトに最初にアクセスする際にオブジェクトを作成して紐付けられる．各オブジェクトはオブジェクト ID を用いて, この KV ストアに格納されたオブジェクトへのポインタを通じてアクセスされる．

3.3 インターフェース

PXNO の提供するインターフェースにはクライアントとサーバで分かれており, アプリケーションはクライアントのインターフェースを呼び出し, クライアントは内部でサーバのインターフェースを呼び出す.

サーバのインターフェースにはオブジェクト操作のためのインターフェースとリング管理のためのインターフェースがそれぞれ必要である．まず, リングを構成するには隣接するノードへのサーバ一覧の受け渡しやリングへの参加・脱退の処理が必要となる．リング管理の操作については CHFS と同様のインターフェースを用いることでリングの構築・管理を可能にする．次に, オブジェクトの作成や削除, オブジェクト内のデータへアクセスを提供するためのインターフェースも必要となる．オブジェクト操作のためのインターフェースを図 5 に示す．オブジェクトを作成する前にオブジェクトを管理するためのデータベースを構築する必要があるため, Create で管理用の KV ストアを構築する．オブジェクトは KV ストアで構成されるので, KV ストアにアクセスするためのデータ構造の作成は Open で行う．Open で作成されたオブジェクトへのポインタを破棄するための処理は Close で行う．ただし, Close で破棄するのはオブジェクトへのポインタのみで不揮発性メモリ上の KV ストアは Destroy で破棄する．オブジェクト内の各キーバリューに対する操作も提供する必要がある．オブジェクト内のキーバリューの挿入は Put, キーバリューの取得は Get, 各キーバリューの削除は Remove で行う．キーバリューの取得について, オブジェクト内のすべてのキーまたはキーバリューを取得する際は RPC の回数をで

きるだけ減らすことでネットワークのレイテンシを最低限に抑えることができる．したがって, オブジェクト内のキーまたはキーバリューの一覧を取得する List を導入する．これによりキーまたはキーバリューの一覧を取得するための RPC の回数を抑える．Put・Get についてはバッファサイズが大きい場合にはデータコピーがボトルネックとなる可能性があるため RDMA によるデータ転送が有効であると考えられる．よって RDMA による転送を行うためのインターフェースとして PutRDMA と GetRDMA を導入し, バッファサイズが大きい場合の転送の最適化を図る．

クライアントのオブジェクト操作はサーバインターフェースを用いて, 対象のオブジェクトの適切なシャードに転送する役割を持つ．クライアントインターフェースを図 6 に示す．オブジェクトのシャードを配置するサーバはあらかじめ設定された数だけ Open で作成され, そのオブジェクトのハンドラを得る．ハンドラにはオブジェクト ID やシャード値といったオブジェクトの各操作で必要な情報がおり, ハンドラを通じて各操作を行うことで, それぞれの操作で必要となる情報を共有する．各シャードについて Close や Destroy ではサーバ上のオブジェクトへのポインタ及び不揮発性メモリ上のオブジェクトの破棄を行う．クライアントにおけるデータ操作ではひとつのキーバリューに対して並列に処理を行えるようにすることが望ましい．PXNO ではひとつのバリューを一定の長さのチャンクで分割して複数のキーバリューに分けてそれぞれのチャンクに対して Put や Get といった操作を行うことで, サーバで各チャンクの処理の完了を待たずに並列に処理を可能にする．したがって, チャンクの分割やシャードへの分散を行うためのインターフェースが必要となる．Write ではキーバリューの挿入を行うが, バリューをチャンクに分割する操作と各シャードに分散する処理が含まれる．Read では各シャードに対してサーバの Get を呼び出し, すべてのチャンクを取得する．Remove では各シャードに対して指定されたキーに対応するキーバリューの削除を行う．バリューの途中からの読取や書き込みについては Put と Get で行う．先頭からのオフセットサイズを指定することで適切なシャードを計算し, そのシャードに対してキーバリューの挿入や取得操作を行う．Put・Get・Write・Read ではチャンクサイズによって PutRDMA や GetRDMA を用いて RDMA 転送を行う．

4. 実装

サーバプロセスは RPC プロセスを起動時に生成し, クライアントからの RPC 要求を受け付ける．RPC は thallium ライブラリで実装される．thallium は Mochi-margo の C++バインディングであり, HPC に最適化された RPC を提供するフレームワークである Mercury と軽量スレッドを扱うための Argobots を組み合わせたライブラリである．

```
Open(server, oid)
Close(server, oid)
Destroy(server, oid)
Put(server, oid, key, offset, size, value)
Get(server, oid, key)
PutRDMA(server, oid, key, off, size)
GetRDMA(server, oid, key, off, size)
Remove(server, oid, key)
List(server, oid, is_key_only)
```

図 5: サーバインターフェース

```
Open(server, oid, label)
Close(server, obj_handler)
Destroy(server, obj_handler)
Write(obj_handler, key, value, size)
Read(obj_handler, key, value, size)
Put(obj_handler, key, offset, value)
Get(obj_handler, key, offset, value)
Remove(obj_handler, key)
```

図 6: クライアントインターフェース

Mercury は libfabric プラグインを実装しており, `tcp` や `verbs`, `psm2`, `gni` といったネットワークハードウェアの抽象化を利用することができる．RDMA も Mercury でサポートされており, thallium から利用できる．

4.1 オブジェクトとオブジェクト ID

不揮発性メモリ上には 2 種類の KV ストアを構築する．ひとつはオブジェクト管理のためにオブジェクト ID とオブジェクトをなす KV ストアへの永続ポインタを保持するための KV ストアで, もうひとつはオブジェクトをなす KV ストアである．永続ポインタは不揮発性メモリ上でデータにアクセスする方法を提供するためのデータ構造である．永続ポインタには不揮発性メモリプールの UUID とプールの先頭からのオフセットが含まれる．この永続ポインタを用いることで, 不揮発性メモリ上に保存されたデータにアクセスすることができる．オブジェクト管理のための KV ストアについては libpmemobj ライブラリにおいて他の不揮発性メモリ上のデータにアクセスするためのエントリポイントとなるルートオブジェクトと呼ばれる部分に KV ストアへの永続ポインタを保存する．各オブジェクトを構成する KV ストアへのアクセスは各 KV ストアの作成時に得られる永続ポインタをオブジェクト管理のための KV ストアに保存することで可能となる．不揮発性メモリ上でのオブジェクト ID と各 KV ストアへの永続ポインタの保存の例を図 7 に表す．この例では `obj1` にアクセスする時には, まずルートオブジェクトである `pool_mgr` KV ストアにアクセスする．その後, `obj1` のオブジェクト ID である `OID001` を検索して永続ポインタを取得する．最終的に取得した永続ポインタから実際のオブジェクトの KV ストア

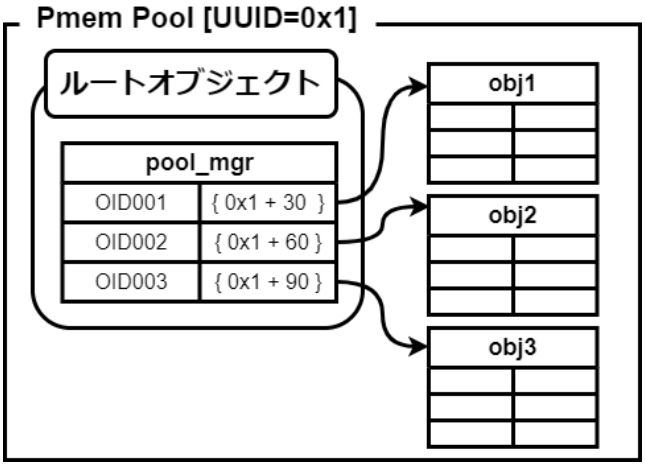

図 7: 不揮発性メモリ上でのオブジェクトの保存例

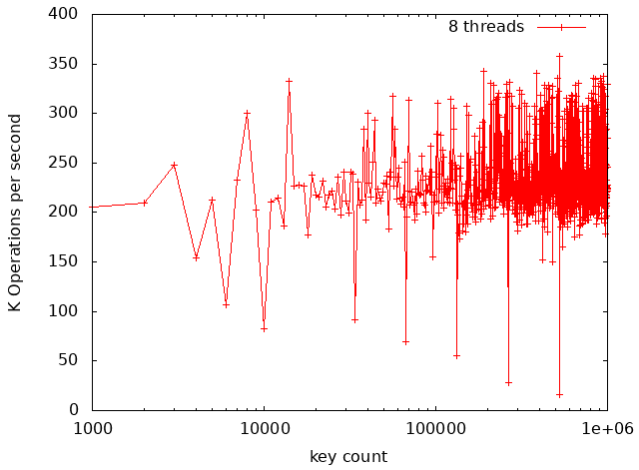

図 8: PXNO の Put のスループット

にアクセスすることができる.

インターフェースの実装は pmemkv の API を用いる. Put・Remove 操作は pmemkv の pmem::kv::put と pmem::kv::remove を用いる. Get 操作は pmemkv の pmem::kv::get を用いて, 与えられたキーに対してコールバック関数を呼び出すことで不揮発性メモリ上のデータを取得する. List 操作は pmem::kv::get_all を用いて, オブジェクト内の各キーに対してコールバック関数を呼び出して, すべてのキーバリューを取得する. クライアントの実装ではキーバリューを分割してシャードに分配してリクエストする場合がある. シャードはオブジェクト ID からコンシステントハッシュによって最初に決められたサーバからシャードの個数分先のサーバをシャードとして決定され, シャードの一覧はクライアントで保持される. Read や Write ではこのシャードの一覧を用いてキーのハッシュ値を基に各サーバにキーバリューの分配を行う. Read の場合, 実際のバリューのサイズがわからない場合がある. このような場合には指定されたキーの分割されたエントリが見つからなくなるまでシャードから探す.

5. 性能評価

PXNO の性能評価のために以下のような評価実験を行う.

- KV の put のスループット計測
- 単一ノードにおける YCSB ベンチマークでの評価
- Cygnus を用いたスケーラビリティの評価

評価は表 1 で示す研究室のクラスタおよびスーパーコンピュータ Cygnus で行う. Cygnus の計算ノードにはスクラッチ領域としてノードローカル SSD を利用することができる. 不揮発性メモリは fsdax モードとして ndctl コマンドでデバイスファイルを作成し, このモードに対応しているファイルシステムである xfs にフォーマットした上で DAX マウントして利用する. 研究室のクラスタの不揮発メモリは fsdax モードで DAX マウントしたファイルシステムを利用する. 不揮発性メモリは 2 分割されており, 実験ではその片方の 373GB の領域を用いる.

5.1 PXNO の Put のスループット計測

pmemkv の `cmap` でキーバリューが増えた時のスループットが低下する問題について, PXNO のオブジェクトを用いて改善されるかを確認する. 計測は図 2 と同様にキーバリューを 1000 個ずつ挿入する操作を 8 スレッドで実行するベンチマークで行う. キーのサイズは 20 バイト, バリューのサイズは 256 バイトとし, 1000 個のオブジェクトに対して各イテレーションですべてのオブジェクトにキーバリューを挿入する操作を行う.

実験結果を図 8 に示す. pmemkv で計測した結果である図 2 と比較すると, 個数が多くなった場合でも平均で 227.9KIOPS となっていることから, Put でキーバリューの個数が増えた時の性能低下を防ぐことができていることがわかる.

5.2 YCSB ベンチマークによる評価

YCSB ベンチマーク [1], [3] はキーバリューストアの評価に用いられるベンチマークで, 様々な読み書きのワークロードにおけるストレージ性能の評価を行うことができる. YCSB ベンチマークにより PXNO の各ワークロードにおける性能の特性を明らかにする. 比較対象は GekkoFS で用いられているキーバリューストアである RocksDB と分散オブジェクトストアである DAOS とした. YCSB の RocksDB コネクタについては既存の Java コネクタを利用した. DAOS では不揮発性メモリのデバイスファイルを指定することで, 内部でフォーマットした上で DAX マウントして不揮発性メモリを利用している. RocksDB については DAX マウントされたファイルシステム上のディレクトリをデータファイルの保存先として指定した. DAOS と PXNO については既存のコネクタが実装されていなかったため, YCSB の C バインディングである YCSB-C [9] を利

表 1: 実験に使用した計算ノードの構成

CPU	Intel ®Xeon ®Gold 5218 CPU 2.30GHz × 1
メモリ	96GiB (DDR4-3200 16GiB × 6)
ネットワーク	InfiniBand HDR100 × 1
OS	CentOS 7.9
Persistent Memory	Intel®Optane™Persistent Memory 128GB Module × 6 (fsdax mode)

用してコネクタを実装し, 評価を行った. 対象となるワークロードは表 2 に示すもので, 範囲クエリに関するワークロードについては PXNO でハッシュマップに対する範囲クエリが効率的にできないため対象から除外した. ワークロード F の Read-modify-write はレコードを読んで書き戻すワークロードである. 各ワークロードにおいてクライアントスレッド数を増やした時の読み IOPS によって評価する. サーバで利用できるスレッドするを一律にするために, 各ストレージについて以下の用途でスレッドを利用する.

- RocksDB： LSM ツリーのコンパクションに用いるスレッド
- DAOS： 各ターゲットの RPC・I/O スレッド
- PXNO：サーバの RPC スレッド

PXNO におけるチャンクサイズは 1MiB とし, 各ストレージのスレッド数は 4 とする. また RocksDB の memtable はデフォルトの 64MB ととした. 各ワークロードで扱うレコード数, オペレーション数を 100,000 として, クライアントスレッド数を 1 から 8 まで変化させたときの実験結果を各ワークロードについてワークロード A から順に, 図 9, 図 10, 図 11, 図 12, 図 13 に示す. 結果から, RocksDB についてはクライアントスレッドを変えてもスループットに変化はなかったが, PXNO, DAOS はデータ操作が並列に処理されるためクライアントスレッドを増やすとスループットも高くなった. read heavy なワークロードであるワークロード B では, クライアントスレッドが 8 スレッドの時に DAOS と比較して 17.8 倍のスループットを達成した. PXNO と DAOS を比較すると, すべてのワークロードでスループットもクライアントスレッドを増やした時のスループットの増加幅も PXNO の方が大きいことが分かる. DAOS ではキーの一覧を取得してから各キーに対してバリューを取得するが, PXNO は List によりキーバリューを同時に取得できる. そのため PXNO は DAOS よりもキーバリューのペアの取得のレイテンシが小さくできるため, 高いスループットとなったと考えられる. RocksDB については, 定期的にログに発行されるコンパクションステータスからコンパクションをしていないことが分かっている. つまり, コンパクションが発生しない状況でもクライアントスレッド数によっては RocksDB よりも高いスループットを発揮できることが分かる. また, ワークロード D については RocksDB よりもスループットが低くなっている. これは pmemkv で挿入の完了を待ってから読取を行うため, その分のレイテンシが影響していると考えられる. しかし, ワークロード D のように値を書き込んだ後にすぐに読むワークロードは HPC アプリケーションでは少なく, アプリケーションへの影響は小さいと思われる.

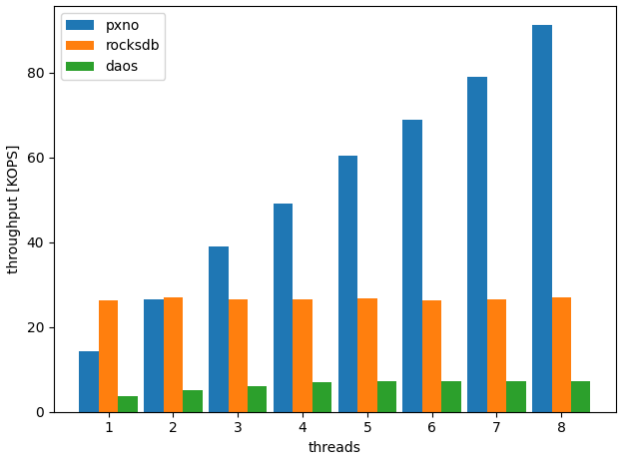

図 9: YCSB ベンチマークによる workload A の評価

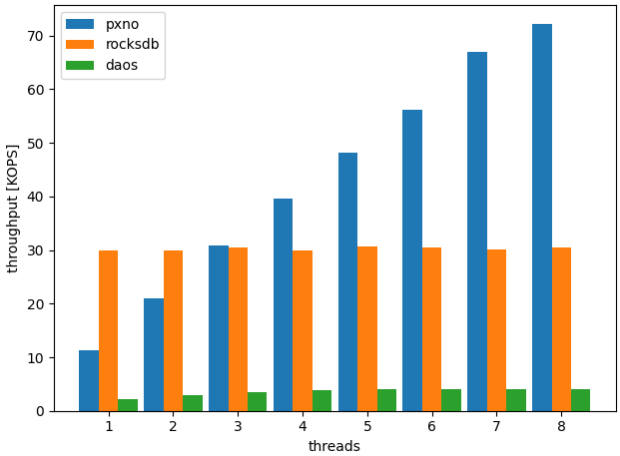

図 10: YCSB ベンチマークによる workload B の評価

5.3 スケーラビリティの評価

PXNO のスケーラビリティを確認するために Cygnus で評価を行う. Cygnus では計算ノードに不揮発性メモリは設置されていないため, 代わりに計算ノードのローカル SSD を利用する. 予備実験として pmemkv から計算ノードの SSD を利用したときのスループットを計測する. 計測は 20B のキーと 128B から 8MiB までバリュー変化させなが

表 2: YCSB のワークロード

ワークロード	説明
A	(update heavy) 10K のレコードの 50% 読取 / 50% 更新
B	(read heavy) 10K のレコードの 95% 読取 / 5% 更新
C	(read only) 10K のレコードの 100% 読取
D	(read latest) 10K のレコードの 95% 読取 / 5% 挿入
F	(read-modify-write) 10K のレコードの 50% 読取 / 50% Read Modify Write

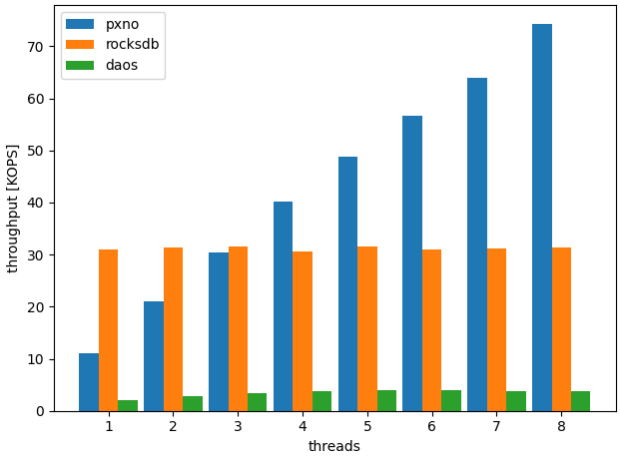

図 11: YCSB ベンチマークによる workload C の評価

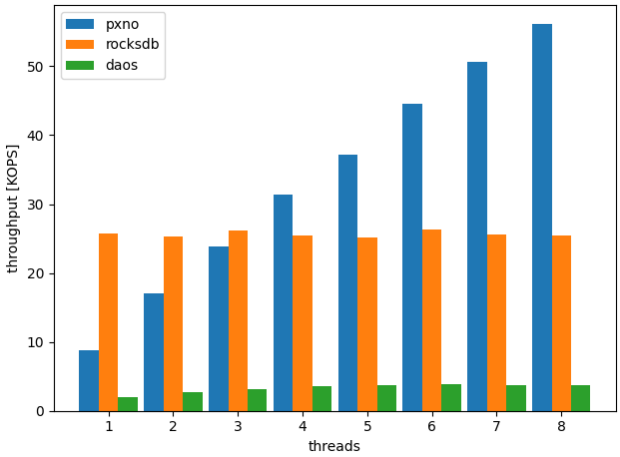

図 13: YCSB ベンチマークによる workload F の評価

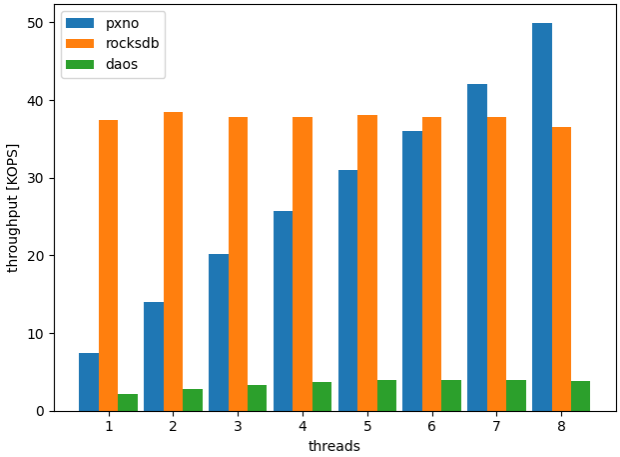

図 12: YCSB ベンチマークによる workload D の評価

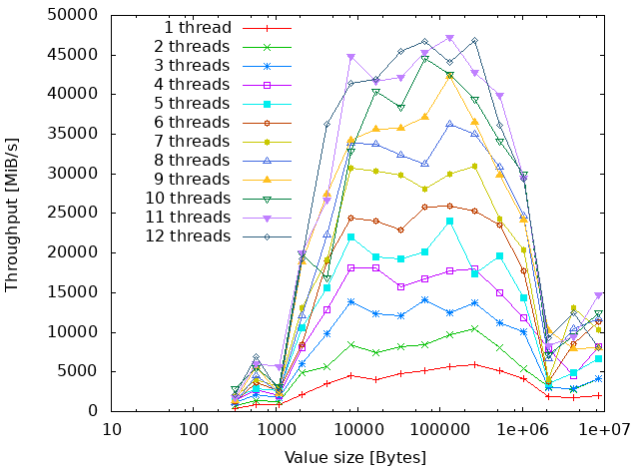

図 14: 計算ノードの SSD を pmemkv で利用したときの Get のスループット

ら, それぞれのバリューサイズについて 40 秒間挿入と取得するベンチマークを実行することで行った. スレッドは 1 から 12 スレッドを利用する. 12 は Cygnus の計算ノードの 1 ソケットで利用できる上限のコア数である. Put, Get それぞれの結果を図 14 と図 15 に示す. これらのでは SSD 上での pmemkv のスループットの最大値は Get では 12 スレッドで 256KiB の時に 46.8GiB, Put では 3 スレッドで 2MiB の時に 800MiB であった.

予備実験の結果を踏まえて, チャンクサイズは Put・Get ともにできるだけスループットの高かった 1MiB とする. またサーバスレッドは 12 とし, クライアントプロセスも 12 として, それぞれ別々のソケットに配置する. バリューは 4MiB として単一のオブジェクトに対してキーバリューの挿入を行うベンチマークを各プロセスで 256 回行い, サーバ・クライアント共に 1 から 32 ノードを用いた時のスループットの最大値を計測する. 実験結果を図 16 に示す. ノード数が 32 の場合で Get は 162.2Gib/sec, Put は 8.7GiB となった. 図 16 より Put・Get 共に線形にスケールしていることが分かる.

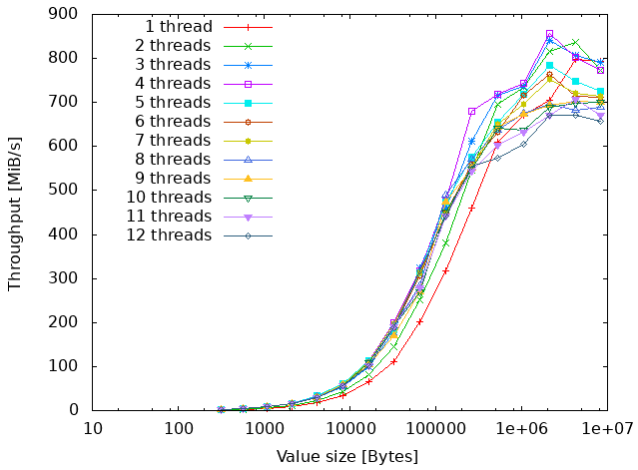

図 15: 計算ノードの SSD を pmemkv で利用したときの Put のスループット

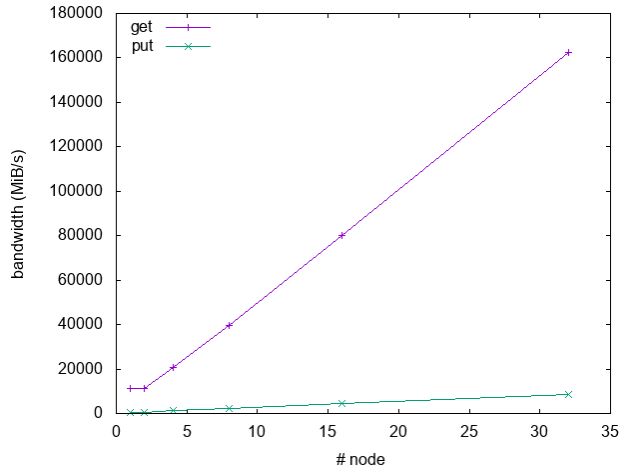

図 16: PXNO のスケーラビリティの評価

6. まとめ

本研究では, 不揮発性メモリを利用した分散オブジェクトストレージの提案を行った. 特にユーザがスケーラビリティを変更でき, キーバリューの数が増えた場合でも Put のスループットが低下しないための設計を行った. 性能評価の結果, Put のスループットは平均で 227.9KIOPS となり, キーバリューが増えてもスループットが低下しないことを示した. また, YCSB ベンチマークでは既存の分散オブジェクトストレージよりも read heavy なワークロードで最大で 17.8 倍高い性能を達成した. Cygnus スーパーコンピューターによる評価ではノードが増えた場合でも性能がスケールすることを示した.

謝辞 本研究の一部は，JSPS 科研費 22H00509，筑波大学計算科学研究センターの学際共同利用プログラム（Cygnus），国立研究開発法人新エネルギー・産業技術総合開発機構（NEDO）の委託業務（JPNP16007）および富士通との共同研究の助成を受けたものです.

参考文献

[1] Brian Cooper: YCSB benchmark, Yahoo Research (online), available from ⟨https://ycsb.site/⟩ (accessed 2022-03-29).

[2] Chandrasekar, R. R., Evans, L. and Wespetal, R.: An exploration into object storage for exascale supercomputers, *Proceedings of the 2017 Cray User Group*, p. 13 (2017).

[3] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R. and Sears, R.: Benchmarking Cloud Serving Systems with YCSB, *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, New York, NY, USA, Association for Computing Machinery, pp. 143–154 (online), DOI: 10.1145/1807128.1807152 (2010).

[4] daos-stack: DAOS documentation, daos-stack (online), available from ⟨https://daos-stack.github.io/⟩ (accessed 2022-03-29).

[5] Facebook: RocksDB, A persistent key-value store, Facebook (online), available from ⟨http://rocksdb.org/⟩ (accessed 2022-03-29).

[6] Hasija, H. and Kumar, D.: Compression & Security in MongoDB without Affecting Efficiency, *Proceedings of the Second International Conference on Information and Communication Technology for Competitive Strategies*, ICTCS '16, New York, NY, USA, Association for Computing Machinery, pp. 1–6 (online), DOI: 10.1145/2905055.2905155 (2016).

[7] Intel: Persistent Memory Development Kit, Intel (online), available from ⟨https://pmem.io/pmdk/⟩ (accessed 2022-03-29).

[8] Intel Corporation: Optane DC Persistent memory Brief, Intel Corporation (online), available from ⟨https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-dc-persistent-memory-brief.pdf⟩ (accessed 2022-03-29).

[9] Jinglei Ren: YCSB-C, Yahoo Research (online), available from ⟨https://github.com/basicthinker/YCSB-C⟩ (accessed 2022-03-29).

[10] Khan, A., Sim, H., Vazhkudai, S. S. and Kim, Y.: MOSIQS: Persistent Memory Object Storage With Metadata Indexing and Querying for Scientific Computing, *IEEE Access*, Vol. 9, pp. 85217–85231 (online), DOI: 10.1109/ACCESS.2021.3087502 (2021).

[11] Liang, Z., Lombardi, J., Chaarawi, M. and Hennecke, M.: DAOS: A Scale-Out High Performance Storage Stack for Storage Class Memory, *Asian Conference on Supercomputing Frontiers*, Springer, pp. 40–54 (2020).

[12] Liu, J., Koziol, Q., Butler, G. F., Fortner, N., Chaarawi, M., Tang, H., Byna, S., Lockwood, G. K., Cheema, R., Kallback-Rose, K. A., Hazen, D. and Prabhat, M.: Evaluation of HPC Application I/O on Object Storage Systems, *2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage Data Intensive Scalable Computing Systems (PDSW-DISCS)*, pp. 24–34 (online), DOI: 10.1109/PDSW-DISCS.2018.00005 (2018).

[13] Lofstead, J., Jimenez, I., Maltzahn, C., Koziol, Q., Bent, J. and Barton, E.: DAOS and Friends: A Proposal for an Exascale Storage System, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, IEEE Press, pp. 585–596 (online), DOI: 10.1109/SC.2016.49 (2016).

[14] Malakhov, A.: Per-bucket concurrent rehashing algorithms, *CoRR*, Vol. abs/1509.02235, p. 7 (online), available from ⟨http://arxiv.org/abs/1509.02235⟩ (2015).

[15] Paul, A. K., Faaland, O., Moody, A., Gonsiorowski, E., Mohror, K. and Butt, A. R.: Understanding HPC Application I/O Behavior Using System Level Statistics, *2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pp. 202–211 (online), DOI: 10.1109/HiPC50609.2020.00034 (2020).

[16] Paul, A. K., Karimi, A. M. and Wang, F.: Characterizing Machine Learning I/O Workloads on Leadership Scale HPC Systems, *2021 29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 1–8 (online), DOI: 10.1109/MASCOTS53633.2021.9614303 (2021).

[17] Scargall, S.: *Programming Persistent Memory: A Comprehensive Guide for Developers*, Apress (2020).

[18] Tatebe, O., Obata, K., Hiraga, K. and Ohtsuji, H.: CHFS: Parallel Consistent Hashing File System for Node-Local Persistent Memory, *International Conference on High Performance Computing in Asia-Pacific Region*, HPCAsia2022, New York, NY, USA, Association for Computing Machinery, pp. 115–124 (online), DOI: 10.1145/3492805.3492807 (2022).

[19] Vef, M.-A., Moti, N., Süß, T., Tocci, T., Nou, R., Miranda, A., Cortes, T. and Brinkmann, A.: GekkoFS - A Temporary Distributed File System for HPC Applications, *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 319–324 (online), DOI: 10.1109/CLUSTER.2018.00049 (2018).

[20] Wang, C., Mohror, K. and Snir, M.: File System Semantics Requirements of HPC Applications, *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '21, New York, NY, USA, Association for Computing Machinery, pp. 19–30 (online), DOI: 10.1145/3431379.3460637 (2021).

[21] Wang, T., Mohror, K., Moody, A., Sato, K. and Yu, W.: An Ephemeral Burst-Buffer File System for Scientific Applications, *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 807–818 (online), DOI: 10.1109/SC.2016.68 (2016).