Vol.2013-HPC-140 No.12
2013/7/31

不揮発性デバイス向けのObject Storageの設計

鷹津 冬将[1,3,a)]　平賀 弘平[1,3]　建部 修見[2,3]

概要：分散ファイルシステムのストレージノードで用いられる Object Storage としてハードディスクなどよりも高速で汎用的な API が提供されている不揮発性デバイスにより適した Object Storage の設計を行い，プロトタイプ実装の評価を行った．オブジェクト作成の性能評価において，スレッド数の増加に応じて高い性能を示し，16 スレッドの場合においては directFS の 3.34 倍の性能を示した．また，オブジェクトに対する書き込み性能の評価において，ブロックサイズの増加に応じて高い性能を示した一方で，並列書き込み性能の評価において性能向上の余地があることが分かった．本稿ではこれらの設計とその実装，評価，考察について述べ，今後の性能向上への課題について検討する．

1. はじめに

近年，計算機の性能向上や価格低下，また多数の計算機を並べ効率的に演算を行う技術が確立されてきたことにより，安価に大規模なデータセンターを構築し大量のデータを解析する技術が確立されてきた．このような中，これらの並列計算機で用いられるストレージデバイスがハードディスクからフラッシュを利用したデバイスなど読み書きが高速なストレージデバイスに移行しつつある．フラッシュを利用したデバイスには書き込みをアトミックに行ったりデータを保持する空間を物理的な容量を遙かに超えて利用できたりするなど，過去のハードディスクなどに比べより柔軟に利用できるデバイスが登場し，これらの機能を使うことでより高い性能が示されることが期待されている．

一方で，天文学や生命科学などの科学技術分野におけるデータインテンシブコンピューティングや e-サイエンスの分野においては，実験装置の発展やシミュレーション規模の拡大などによって扱うデータ量が年々増加の一途をたどっており，この大量のデータを書き込み速度の高速化の要求が高まっている．このような大規模な演算を行う並列計算機においてはデータを保存するために Gfarm [1] など分散ファイルシステムを利用することが注目されているが，これまでの分散ファイルシステムはストレージノードが利用するストレージデバイスとして従来のハードディスクなどのデバイスを用いることを想定した設計と成っており，新しいデバイスに最適化されていないため，デバイスの最大性能を引き出すことができていない．また，分散ファイルシステムではストレージノードの参加・脱退を行う際にデータのマイグレーションが行われるが，マイグレーション中もデータにアクセスすることができるようにするためにストレージノード側でバージョン管理を行うことが求められている．

本研究では，高速な分散ファイルシステムにおけるストレージノードの実現に向けて高速なストレージデバイスに適したバージョン管理を行う Object Storage を設計し，プロトタイプを実装し評価実験を行った．

本稿は 5 章で構成される．第 2 章では，関連研究について述べる．第 3 章では，本研究で行った Object Storage の設計とプロトタイプ実装について詳述する．第 4 章では，評価実験の詳細及び評価結果を示す．第 5 章で，結論と今後の課題と展望について示す．

2. 関連研究

本稿で提案する Object Storage としては提案手法以外にも OBFS [2] や BlobSeer [3] などがある．

OBFS は Object-based Storage Device(OSD) 向けに設計されたファイルシステムである．OBFS はストレージの空間を Region と呼ばれる単位で区切り，Region にオブジェクトを格納する．オブジェクトの ID の一部を Region の ID として利用することで間接参照を減らし高い性能を示すことができる設計となっている．一方でバージョン管理を行っておらず，また仮想アドレス空間を利用できる新しいデバイスに対応していない．これらの点において本稿で提案する Object Storage と異なる．

BlobSeer はフランス INRIA が開発している分散データ

[1] 筑波大学大学院システム情報工学研究科コンピュータサイエンス専攻
[2] 筑波大学システム情報系
[3] 独立行政法人科学技術振興機構 CREST
[a)] takatsu@hpcs.cs.tsukuba.ac.jp

ストレージである．データをオブジェクトとして保存することができ，またバージョン管理を行うことができる．BlobSeer は複数のサーバソフトウェアとクライアントライブラリで構成されるが, 同一ノード上にすべてのサーバアプリを実行することも可能である．しかしながら，BlobSeer はブロックデバイスに対して直接読み書きを行うわけではない．この点において本稿で提案する Object Storage と異なる．プログラミングインターフェースとして API ライブラリが提供されているため，クライアントアプリケーションを実装する場合はライブラリを利用する．

分散ファイルシステムにおいてストレージノードがデータをストアするために ext3 [4]/ext4 [5] や XFS，ZFS，Btrfs [6] などといった POSIX 準拠のファイルシステムが利用される．具体的には Ceph [7] では Btrfs ベースの OSD がが利用されており，Lustre [8] では ext4 や ZFS などのファイルシステムをベースとした OSD を利用されている．

これら POSIX 準拠のファイルシステム中でも NILFS2 [9] は Log Structured File System として設計されているファイルシステムであり，Log Structured File System の特徴を利用して連続スナップショット機能を提供している．同期書き込みなどを行ったタイミングで NILFS2 はチェックポイントの生成を行い，ユーザーは任意のチェックポイントをスナップショットとして変更することができる．各スナップショットはファイルシステムがマウントされたまま，オンラインでリードオンリーのファイルシステムとしてマウントできるため，オンラインバックアップに利用できるなどの特徴を備えている．このスナップショットの数に制約はなく，スナップショットはストレージの容量が許す限り生成できる．寿命に達したチェックポイントは回収デーモンによって回収されるが，そのチェックポイントが利用していたブロックについては，利用できるようにするためにさまざまな処理が必要となる．本稿で提案する Object Storage ではこの回収の処理を SDK に含まれる機能で処理を行うことができるため，この回収の方法が NILFS と異なる．NILFS は POSIX 準拠のファイルシステムのため，クライアントアプリケーションを実装する場合は POSIX 標準の低入出力関数を利用する．

DFS(Direct File System) [10] も POSIX 準拠のファイルシステムである．DFS は高速なストレージデバイスにおいてデバイスの物理的なアドレスではなく Virtual Storage Layer (VSL) によって提供される非常に大きな仮想的なアドレス空間を使うことで各ファイルが使う仮想的なブロックが衝突せず，効率的にストレージを使っている．また，DFS はファイルの inode 番号には VSL におけるアドレスを利用するため，データを書き込む際には inode 番号とファイル中のオフセットを使うことで高速に書き込むことができる．一方で，バージョン管理が行われていないことにより，同じ inode 番号のファイルの同じオフセットの箇所に対してデータを書き換える操作を行った場合，上書きされるため元のデータは残らない．この点において本稿で提案する Object Storage と異なる．DFS も POSIX 準拠のファイルシステムのため，クライアントアプリケーションを実装する場合は POSIX 標準の低入出力関数を利用する．DFS の設計をベースとしたファイルシステムとして directFS がある．

これらのような POSIX 準拠のファイルシステムにおいては，データを保存する以外にもデータの名前のデータの属性などのメタデータを提供するためにさまざまな処理を行っているため，分散ファイルシステムとして利用する場合において性能の低下を招いている．分散ファイルシステムにおいては，これらメタデータはメタデータサーバによって管理されるため，ストレージサーバ側においても管理を行う必要はない．本稿で提案する Object Storage では，これらメタデータの管理を行わない設計のためストレージノード側で二重に管理されることはない．

3. 不揮発性デバイス向けの Object Storage

本稿では，不揮発性デバイスとして Fusion-io 社の ioDrive をターゲットとする．ioDrive では SDK を利用することにより，仮想アドレス空間を利用することやアトミックなデータの更新を行うことができる．

SDK によって提供される仮想アドレス空間は 16EB 以上の空間となっており，実際の物理的な容量以上のアドレスが提供されている．この仮想的なアドレスからデバイス上の物理的なアドレスへのマッピングは ioDrive のドライバが行っており，ユーザーアプリケーションはデバイス上の物理的なアドレスを意識することなくデータを読み書きすることができる．

さらに，ioDrive は複数のスレッドからの書き込みリクエストを処理することができ，単一スレッドに比べ高い性能を示す特性がある．

また，本稿が提案する Object Storage は分散ファイルシステムにおいて利用されることを目的とする．分散ファイルシステムにおいては，Object Storage でバージョン管理が行われている場合において効率的にストレージノードのマイグレーションを行うことができる．

これらのことから，本稿で提案する Object Storage は以下の機能を含むことを目指す．

(1) VSL による仮想空間の効率的な利用
(2) 複数スレッド対応
(3) バージョン管理

3.1 設計

前述した機能を目指し，本稿で提案する Object Storage を次のように設計した．まず，VSL によって提供される膨大な空間を図 1 に示されるように固定長で分割し，この

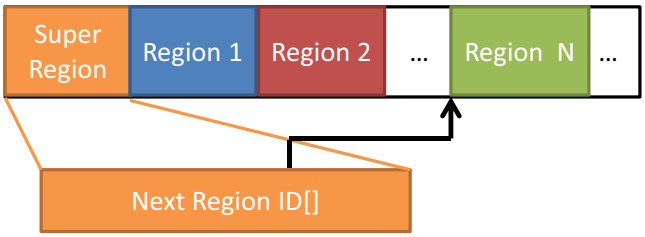

図 1 VSL による空間を Region に分割

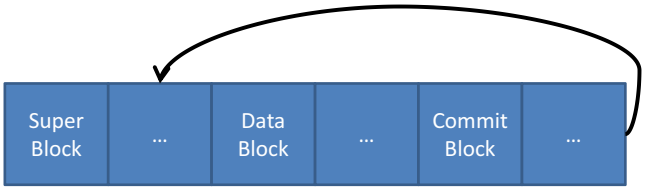

図 2 Region の詳細

分割された各領域のことを Region と定義する．この図 1 に示されるように膨大な空間を固定長で分割することで，各 Region にアクセスする場合においてアドレスの計算を行う必要をなくしオーバーヘッドを低く抑えたり，多数のオブジェクトの数を保証したりすることができる．先頭の Region を Super Region として定義し，Object Storage のさまざまな情報が保存される．二つ目以降の Region でユーザーからのデータを管理するが，各 Region がひとつのオブジェクトによって使われる領域となり，Region とオブジェクトが 1 対 1 対応をするため Region のセクタ番号をオブジェクトの ID として利用する．オブジェクトの ID は読み書きなどの操作においてキーとなる情報であり ID が Region のセクタ番号と一致させることで，ID から直接オブジェクトに対して読み書きを行うことができ，読み書きオペレーションの性能の向上を図る．

各 Region の詳細を図 2 に示す．図 2 に示されるように，各 Region は Super Block と，Commit Block と Data Block による循環ログによって成り立っている．このように循環ログとしてデータを管理することで書き込みを高速に行うことが可能である．Commit Block は毎回の書き込みによって生成されるオブジェクトのバージョンを管理する．書き込みごとに Commit Block を生成することで，すべてのバージョンを保存する．

このような構造の Region において要求された各オペレーションに対応する動作は以下のような設計となる．

(1) 作成
Object の ID は前述の通り Region の ID である．そのため作成オペレーションの場合は，まず新しい ID を計算によって求める．そしてこの新しい ID に対応する Region の Super Block を初期化と Super Region の更新を行う．この初期化と更新に伴う書き込みを SDK によるアトミック書き込み命令を使うことで，書き込みに失敗した場合においてもロールバックを行う必要がなく，同じ仮想アドレスのブロックを繰り返し使うことができるため失敗した場合における性能の向上や VSL による仮想的なアドレス空間のより効率的な利用を図る．

(2) 書き込み
書き込みを行う際には，Object の ID から対応する Region の循環ログにデータを追記することで書き込みを完了する．追記の際においては Commit Block なども書き込む必要があるが，これらもまとめて一つの書き込み命令で行うことで高速化を図る．この書き込みを SDK によるアトミック書き込み命令を使うことで，書き込みに失敗した場合にロールバックを行う必要がなくなり書き込みが失敗した場合における性能の向上を図る．

(3) 読み込み
読み込みを行う際には，Object の ID から対応する Region の循環ログの中から指定されたオフセットに対応するデータを読み込みユーザーに返す．

(4) クリーニング
クリーニングは各 Object について，タイムスタンプから古いデータに対応する Commit Block とその Data Block に対して TRIM コマンドを発行していくことでその Block に対応した物理的な Block を解放させることでクリーニングを行う．

3.2 プロトタイプ実装

これらの設計に基づきオブジェクトの作成及びデータの書き込みを行うことのできるプロトタイプ実装を行った．プロトタイプ実装はライブラリの形として実装され，ユーザーがアプリケーションを開発する場合はこのライブラリに含まれる API を利用する．ユーザーアプリケーションが提案する Object Storage をロードする際には Super Region から各スレッドが作成する次の Object の ID の一覧をメモリに読み込むことで初期化を行い，設計に基づきオブジェクトの作成やデータの書き込みなどの処理を行う．

プロトタイプ実装における Region の詳細な実装を図 3 に示す．図 3 に示されるように，各 Region において Super Block は，最後の Commit Block のアドレスの他，Region を循環ログとして扱うために Region 内のデータがストアされている先頭と末尾のブロックのアドレスを保持している．Commit Block は，ひとつ前の Commit Block のアドレスと書き込みが行われたタイムスタンプ，書き込まれたデータが保持されている Data Block のアドレスが保持されている．Data Block は，書き込みが行われたデータが保持されている．書き込みリクエストごとに Commit Block を作成し，線形リストとして扱うことでバージョンを管理する．

オブジェクトを作成する際においては，新しく作成され

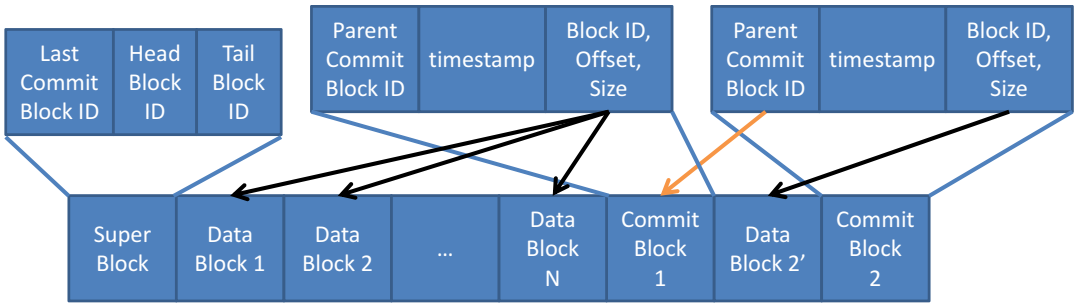

図 3 プロトタイプ実装における Region の詳細

る Object の ID は，作成を行うスレッドごとに異なり，この ID は Super Region に含まれメモリ上に保持される．そのため，オブジェクトを作成する際には，メモリ上から作成を行うスレッドに対応する ID を取得する実装となっている．このような実装にすることにより作成オペレーションを行う複数のスレッド間でロックを行わなくて済むため性能の向上につながる．

書き込みオペレーションにおいては，Object の ID から Super Block のアドレスが示されるため，Super Block のデータを読み込むことで，最後の Commit Block のアドレスを求める．このアドレスを Parent Commit Block ID とする新しい Commit Block となるデータを作成し，書き込みリクエストのデータ及び新しい Super Block のデータとともに書き込みユーザーに結果を返す実装である．

読み込みとクリーニングにおいては，SDK において VSL からデータを読み込む API が提供されていないため実装を行っていない．

3.3 最適化

この設計においては，作成オペレーションにおいて各オペレーションごとに毎回 Super Region の更新が行われるが，更新が行われるために作成オペレーションにおいてデバイスに転送されるデータ量が増えるため，作成性能が低下する．そのため，Super Region に含まれている情報はメモリ上に保持され，読み込みが行われるのは初期化の際のみであるため，Super Region を減らす最適化を行う．

まずメモリ上に保持してある次の Object の ID の一覧から作成を行うスレッドに対応した ID を取得し，メモリ上のデータを更新する．オブジェクトの ID が 1024 の倍数の場合は，Super Region における Object の ID についても 1024 進めた上で書き込む．この操作は Super Block を初期化する書き込みとともに行う．このように 1024 個ごとに Super Region の更新が行われるため，オブジェクトの作成の際に書き込まれるデータ量を減少させることができる．また，電源断などの障害時においても，失われるのは 1024 個分のアドレスのみであり，これらは仮想的なアドレスであるため物理的な容量としての損失はない．この最適化により，作成オペレーションにおいてデバイスに書き込むデータ量をおよそ 50%削減することができるため，性能向上が期待できる．

プロトタイプ実装においては，この最適化を適用させた場合と適用させない場合の双方の関数を用意した．

表 1 評価を行ったノードの性能

CPU	Intel(R) Xeon(TM) E5620 CPU 2.40 GHz (4 コア 8 スレッド) x2
メモリ	24GB
OS (Kernel)	CentOS 6.4 (3.4.4)
ストレージ	Fusion-io ioDrive 160GB

4. 評価

本研究で設計した Object Storage のプロトタイプ実装について複数の手法で評価を行った．

4.1 評価環境

提案する Object Storage と既存の各ファイルシステムを評価を行ったノードの性能を表 1 に示す．

4.2 比較対象

本稿の提案手法との比較対象には関連研究にも挙げた ext3, ext4, XFS, Btrfs, directFS を取り上げる．これらは通常の POSIX ファイルシステムであるが，さまざまな分散ファイルシステムのストレージノードのバックエンドシステムとして POSIX ファイルシステムを使うためこれらを比較対象として選んだ．しかしながら，これらは通常の POSIX ファイルシステムであるため，今回の提案手法とは階層モデルが異なる．これらファイルシステムと提案手法の階層モデルを図 4 を示す．図 4 に示されるように比較対象として取り上げた各ファイルシステムはカーネル空間で動作するファイルシステムとなっている．directFS 以外のファイルシステムは ioDrive を通常のブロックデバイスとして利用し，directFS は SDK が提供するインターフェースを利用するファイルシステムとなっている．

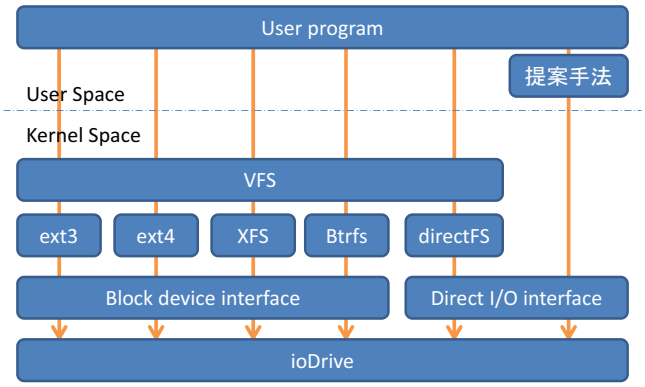

図 4 提案手法と各ファイルシステムの階層モデル

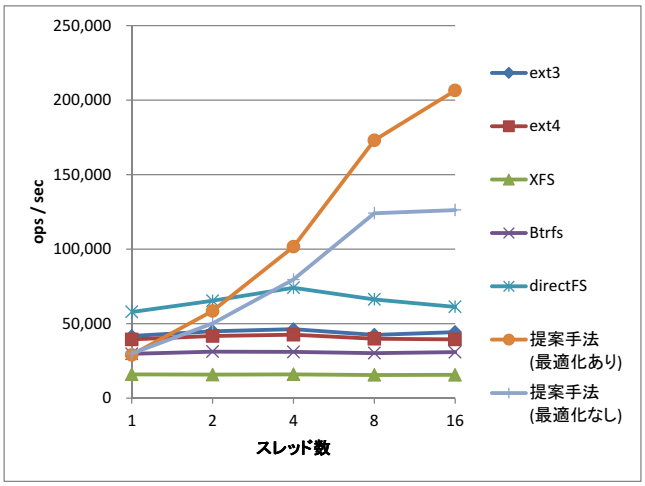

図 5 単位時間あたりのオブジェクト作成性能の評価結果

また，これらファイルシステムはユーザーアプリケーションからは VFS を経由してアクセスすることになる．一方で提案手法においては，ユーザー空間で動作するライブラリとして実装してあるため，ユーザーアプリケーションからは，直接 API を呼び出す設計となっている．

4.3 単位時間あたりのオブジェクト作成性能の評価

この評価では，単位時間に作成することのできるオブジェクトの数を各ファイルシステムと提案手法について性能を評価を行う．分散ファイルシステムにおいては，複数のクライアントから同時にファイルを作成するリクエストが届く．そのため，この評価においては実際にオブジェクトを生成するワーカースレッドの数を変化させて評価を行う．

本評価では各ファイルシステムと提案手法についてスレッド別に 10 回ずつ性能を評価し，平均を算出することで評価を行う．

評価結果を図 5 に示す．この図 5 においては，横軸がワーカースレッドの数であり，縦軸が単位時間あたりに作成したオブジェクトの数を表している．そのため，縦軸の数値が高い方が高い性能を示している．

この評価では，既存のファイルシステムがスレッド数に

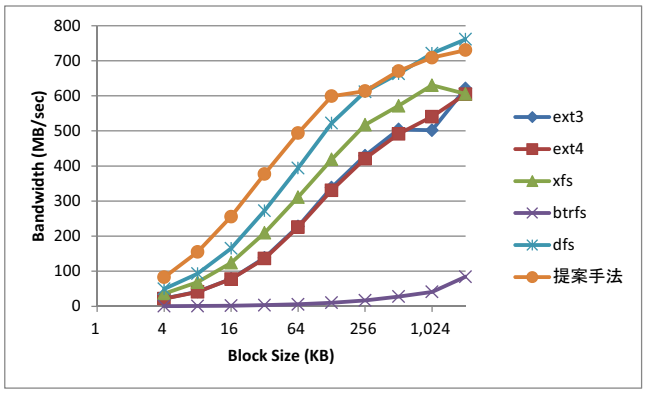

図 6 ブロックサイズ別の単位時間の書き込み性能の評価結果

かかわらずほぼ一定の性能を示すのに比べ，提案手法が最適化の有無にかかわらずスレッド数の増加に合わせて単位時間あたりに作成されるオブジェクトの数が増加することを示している．特に 16 スレッドの場合においては最適化を行っている場合の提案手法が既存のファイルシステムの中で最も高い性能を示している directFS に比べると 3.34 倍と高い性能を示している．これは，提案手法が既存のファイルシステムとは違い，新しく作成されるオブジェクトの ID やブロックの位置を他のスレッドと競合しないような設計になっているためと考えられる．

また，この評価結果からは 16 スレッドの場合において最適化を行うことで最適化を行わない場合に比べると 164%の性能が出ていることが示されている．これは最適化を行うことにより，作成オペレーションのたびに書き込まれるデータ量が減少したためと考えられる．

4.4 ブロックサイズ別の単位時間あたりの書き込み性能の評価

この評価では，単位時間あたりに書き込むことのできるデータサイズを書き込むブロックサイズ別に評価を行う．評価の際には，提案手法及び既存のファイルシステムにおいてあらかじめ作成してあるオブジェクトやファイルに対して書き込みを行う．既存のファイルシステムにおいては，書き込みごとに同期を行い確実にストレージデバイス側にデータを反映させる．本評価では各ファイルシステムと提案手法についてブロックサイズ別に 10 回ずつ性能を評価し，平均を算出することで評価を行う．

評価結果を図 6 に示す．この図 6 においては，横軸がブロックサイズであり，縦軸が単位時間あたりに書き込むことのできたデータサイズを示している．そのため，縦軸の数値が高い方が高い性能を示している．

この評価結果からは，提案手法とすべてのファイルシステムにおいて書き込み性能が書き込まれるデータのブロックサイズの増加にあわせて転送速度が高くなっていることが示されている．ブロックサイズが 512 バイト以下の場合では提案手法が他のファイルシステムに比べ高い性

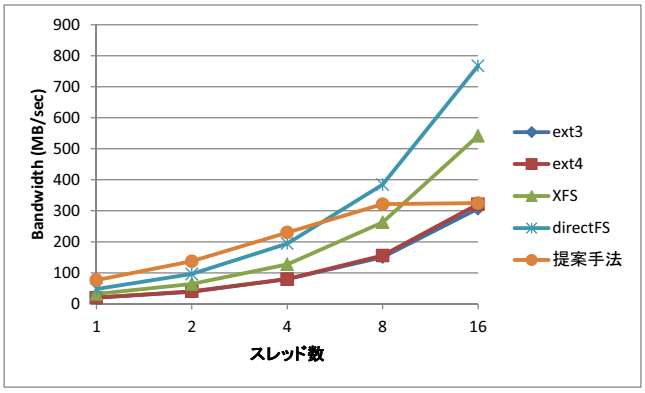

図 7 並列書き込み性能の評価結果

能を示しているが，ブロックサイズが 1MB 以上の場合では提案手法よりも directFS の方が高い性能を示している．特にブロックサイズが 2MB の場合においては，提案手法は directFS の 96.0%の性能となっている．これは，提案手法において書き込まれるデータサイズが書き込みリクエストに含まれるデータに加え，Commit Block のデータと Region Super Block のデータが書き込みリクエストごとに書き込まれるためであると考えられる．

4.5 並列書き込み性能の評価結果

この評価では，4KB のデータを単位時間あたりに書き込むことのできるデータサイズの評価を行う．分散ファイルシステムにおいては，複数のクライアントから同時にデータを書き込むリクエストが届く．そのため，この評価では実際にデータを書き込むワーカースレッドの数を変化させて評価を行う．この評価では，提案手法及び既存のファイルシステムにおいてあらかじめ作成してあるオブジェクトやファイルに対して 4KB のデータの書き込みを行う．既存のファイルシステムにおいては，書き込みごとに同期を行い確実にストレージデバイス側にデータを反映させる．本評価では各ファイルシステムと提案手法についてスレッド別に 10 回ずつ性能を評価し，平均を算出することで評価を行う．

評価結果を図 7 に示す．この図 7 においては，横軸が書き込みを行うワーカースレッドであり，縦軸が単位時間あたりに書き込むことのできたデータサイズを表している．そのため，縦軸の数値が高い方が高い性能を示している．また，この評価において，Btrfs は 1 回の試行に 1 時間程度の所要時間がかかるため比較対象から外した．

この評価結果からは，提案手法とすべてのファイルシステムにおいてスレッド数の増加に合わせて転送速度が高くなってることが示されている．スレッド数が 4 以下の場合は，提案手法が最も高い性能を示しているが，スレッド数が 8 の場合は directFS が提案手法よりも高い性能を示すようになり，スレッド数が 16 の場合は directFS と XFS が提案手法よりも高い性能を示すようになった．特にスレッド数が 16 の場合において，提案手法は XFS に比べ 60.0%の性能を示し，directFS に比べ 42.3%の性能を示している．これは，提案手法においてデバイスに書き込みを行うリクエスト数がが書き込みリクエストにデータの書き込みリクエストに加え，Commit Block のデータと Region Super Block の書き込みリクエストがあるためと考えられる．また，directFS は複数のリクエストを束ねて一度の書き込みとして扱う機能がある．それによってリクエスト数が大幅に減少したため性能が向上したと考えられる．

5. まとめと今後の課題

本研究では，高速な分散ファイルシステムの実現に向けて，高速で高機能なストレージデバイスにおいて効率的にデータを保存することのできる Object Storage の設計とプロトタイプ実装を行い，その評価をさまざまなアクセスパターンで行った．

オブジェクト作成性能の評価では，オブジェクトを 1 秒間に作成することのできる性能をさまざまなスレッド数で評価を行った．この評価では，提案手法がスレッド数の増加に応じて高い性能を示した．特に 16 スレッドの場合においては，既存のファイルシステムの中で最も高い性能を示している directFS に比べると 3.34 倍と高い性能を示した．

書き込み性能の評価では，あらかじめ作成されてあるオブジェクトに対して，1 秒間に書き込むことのできる性能をさまざまなブロックサイズで評価を行った．この評価では，提案手法がブロックサイズの増加に応じて高い性能を示した．ブロックサイズが 512 バイト以下の場合では提案手法が他のファイルシステムに比べ高い性能を示した一方で，ブロックサイズが 1MB 以上の場合では提案手法よりも directFS の方が高い性能を示した．特にブロックサイズが 2MB の場合においては，提案手法は directFS の 96.0%の性能を示した．

並列書き込み性能の評価では，複数の小さな書き込みリクエストを同時に処理できる性能をさまざまなスレッド数で評価を行った．この評価では，提案手法がスレッド数の増加に応じて高い性能を示した．スレッド数が 4 以下の場合において提案手法が最も高い性能を示した．一方で，スレッド数が 8 の場合は directFS が提案手法よりも高い性能を示し，スレッド数が 16 の場合は directFS と XFS が提案手法よりも高い性能を示した．特にスレッド数が 16 の場合において，提案手法は XFS に比べ 60.0%の性能を示し，directFS に比べ 42.3%の性能を示した．

今後の課題として，以下のことが挙げられる．

(1) 書き込みリクエストのアグリゲーション
本稿による提案手法では，Object Storage に対する書き込みリクエストごとにデバイスに対して書き込みリクエストを発行した．しかしながら，並列書き込み性

能の評価においてこの設計は性能向上につながらないことが明らかになった．そのため，複数の書き込みリクエストを束ねて一つの書き込みリクエストに行う機構が必要であると考えられる．

(2) 読み込み機能及びクリーナーの実装と評価
本稿によるプロトタイプ実装では，利用したライブラリ上の問題で読み込み機能及びクリーナーを実装していない．分散ファイルシステムとして利用するためには必要な機能と考えられるため実装する必要がある．

謝辞 本研究の一部は，JST CREST「ポストペタスケールデータインテンシブサイエンスのためのシステムソフトウェア」による．

参考文献

[1] Tatebe, O., Hiraga, K. and Soda, N.: Gfarm Grid File System, *New Generation Comput.*, Vol. 28, No. 3, pp. 257–275 (2010).
[2] Wang, F., Brandt, S. A., Miller, E. L. and Long, D. D. E.: OBFS: A File System for Object-based Storage Devices, *IN PROCEEDINGS OF THE 21ST IEEE / 12TH NASA GODDARD CONFERENCE ON MASS STORAGE SYSTEMS AND TECHNOLOGIES, COLLEGE PARK, MD*, pp. 283–300 (2004).
[3] Nicolae, B., Antoniu, G., Bougé, L., Moise, D. and Carpen-Amarie, A.: BlobSeer: Next Generation Data Management for Large Scale Infrastructures, *Journal of Parallel and Distributed Computing*, Vol. 71, No. 2, pp. 168–184 (2011).
[4] Ts'o, T. Y. and Tweedie, S.: Planned Extensions to the Linux Ext2/Ext3 Filesystem, *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*, Berkeley, CA, USA, USENIX Association, pp. 235–243 (2002).
[5] Mathur, A., Cao, M., Bhattacharya, S., Dilger, A., Tomas, A. and Vivier, L.: The New ext4 Filesystem: Current Status and Future Plans, *Proceedings of the 2007 Linux Symposium*, pp. 21–33 (2007).
[6] Btrfs, https://btrfs.wiki.kernel.org/.
[7] Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. E. and Maltzahn, C.: Ceph: a scalable, high-performance distributed file system, *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, Berkeley, CA, USA, USENIX Association, pp. 307–320 (2006).
[8] Koutoupis, P.: The lustre distributed filesystem, *Linux J.*, Vol. 2011, No. 210 (2011).
[9] 佐藤孝治，小西隆介，木原誠司，天海良治， 盛合敏：ログ構造化ファイルシステム NILFS の設計と実装，情報処理学会論文誌. コンピューティングシステム， Vol. 2, No. 1, pp. 110–122 (2009).
[10] Josephson, W. K., Bongo, L. A., Li, K. and Flynn, D.: Dfs: A file system for virtualized flash storage, In FAST '10: Proc. of the Eighth USENIX Conf. on File and Storage Technologies (2010), USENIX Association.