分散ファイルシステムの
メタデータ管理に関する研究

２０２１年　３月

平賀　弘平

分散ファイルシステムの
メタデータ管理に関する研究

平賀 弘平

システム情報工学研究科
筑波大学

2021 年 3 月

概要

ファイルシステムのメタデータ操作性能は，数百万のクライアントが多数の小さなファイルにアクセスするハイパフォーマンスコンピューティングにおいて非常に重要となっている．本論文では並列ファイルシステムのためのスケーラブルな分散メタデータサーバ，*PPMDS* を提案する．PPMDS はファイルシステムの階層的ネームスペースを分散 key-value ストアサーバ上に効率的に構築する．Dynamic software transactional memory のアルゴリズムに基づくノンブロッキング分散トランザクションにより，複数のエントリをアトミックに更新する．さらに，メタデータ操作性能のより一層の改善のため，サーバサイドトランザクション実行，マルチリーダ化の最適化を行い，リモートプロシージャコール回数の削減と，不必要なブロッキングの回避を行った．性能評価では，128 クライアントから単一ディレクトリへ並列にファイル作成を行うワークロードにおいて，サーバ 5 台の利用により 142,000 ops/s の性能が得られ，サーバ 1 台の場合と比べて約 3.55 倍の性能向上が得られた．

目次

図目次

表目次

第 1 章

序論

High-performance computing（HPC）分野において，計算性能は CPU コア数と計算ノード数の増加によって年々高性能化している．ストレージの性能も同時に向上させなければ CPU とストレージの性能差がどんどん広がっていく．この性能ギャップを埋めるため，ストレージ側には主に 2 つの課題，I/O バンド幅とメタデータ操作性能がある．I/O バンド幅は原則として，ハイパフォーマンスなストレージ，例えば NVMe や SSD や不揮発性メモリを多数並べることで解決できる．しかし，それだけではメタデータ操作性能は向上しない．

メタデータは，分散ファイルシステムを管理するための内部データで，例えば階層的ネームスペース（ファイルパス），ファイルやブロックのロケーション，タイムスタンプやパーミッション情報などがある．メタデータ操作性能とは，例えば，ファイル作成性能，1 秒間にいくつのファイルを作成できるかである．つまり，スケーラブルなメタデータ管理は，階層的ネームスペースを並列で効率的に行う必要がある．

問題解決の 1 つの方法に Data partitioning がある．Subtree partitioning [1, 2] は，階層的ネームスペースをサブツリーに分割して，複数のサーバに割り当て，並列ファイルシステムのメタデータサーバのスケーラビリティを向上させる．これは，別のサブツリーへのアクセスが起こる時はメタデータ操作性能の向上が得られるが，同じサブツリーにアクセスする場合は性能向上が得られない．HPC アプリケーションは同一ディレクトリ内へ数百万のファイルを作成したり，オープンすることがよくある．この場合，Subtree partitioning はメタデータ操作性能の改善に役立たない．

Hash partitioning [3] は，単一ディレクトリ内のエントリを複数のサーバにハッシュ関数を用いて分割，割当する手法である．この手法は，エントリ操作をハッシュに対応するサーバに対してのみ行うため，同一ディレクトリへの数百万ファイル作成のようなメタデータ操作の性能を改善できるが，いくつかのメタデータ操作タイプ，例えばファイルの移動やディレクトリの削除等では，複数サーバにまたがったデータの一貫性制御が必要となる．これらには明らかに分散一貫性制御プロトコルが必要で，これには追加のコストがかかる．Two-phase commit [4] は，この目的のために広く使われているプロトコルで，何らかのブロッキング現象が起こらない限りは効率的に働く．しかし，トラン

ザクションプロセスはトランザクションの参加者の 1 つが失敗したり，コーディネータノードとの通信がネットワークパーティショニング等で失敗するとブロックしてしまう．この欠点については文献 [5] で取り組まれている．

本論文は，並列ファイルシステムのためのスケーラブルな分散メタデータサーバ，PPMDS を提案する．一般的なファイルシステムのセマンティクスを損なうこと無く，メタデータ管理パフォーマンスとスケーラビリティを改善する，ノンブロッキングトランザクションを用いた手法を提案する．論理的な木構造を key-value pair で表現し，階層的ネームスペースを複数のサーバで効率的に管理する．Key フィールドは親 inode 番号とエントリ名のペアで，Value フィールドはファイルのメタデータとなる．エントリはハッシュ関数で分割され，並列ファイル作成，削除，参照ができる．

複数の inode エントリを効率的にアトミックに更新するため，いくつかの問題のある two-phase commit を使わずに，Dynamic-sized software transactional memory [6] に基づいたノンブロッキング分散トランザクションを使う，これにより，一貫性を保ったディレクトリエントリ一覧の取得やファイルの移動等の操作が可能となる．さらに，サーバーサイドトランザクションプロセッシングと，open-for-read スキームのマルチリーダ化の最適化により，リモート関数呼び出しの回数を削減し，不必要なブロッキングを削減した．性能評価では，128 クライアントから単一ディレクトリへ並列にファイル作成を行うワークロードにおいて，サーバ 5 台の利用により 142,000 operations per second の性能が得られ，サーバ 1 台の場合と比べて約 3.55 倍の性能向上が得られた．

1.1 本論文の貢献

本論文の貢献を以下に示す．

- コンシステンシの毀損のないスケーラブルなメタデータサーバの設計と実装を示した
- トランザクション処理を改善するための，ノンブロッキング分散トランザクションの機能強化
- プロトタイプ実装により，単一ディレクトリへのファイルオペレーションのスケーラブルな性能を示した
- 提案メタデータサーバとオブジェクトストレージサーバを組み合わせた分散ファイルシステムのプロトタイプ実装において，メタデータサーバ本来の性能の 97.2 % の性能を引き出せることを示した

1.2 本論文の構成

本論文の構成を以下に示す．

第 2 章では，従来のファイルシステムと既存研究が，どのようなアプローチでメタデー

タを管理しているのかについて述べる．

第 3 章では，コンシステンシの毀損のないスケーラブルなメタデータサーバである PPMDS の設計と実装について述べる．

第 4 章では，提案メタデータ管理手法について，実験により有効性を検証する．

第 5 章では，本論文のまとめについて述べる．

第 2 章

関連研究

2.1 ファイルシステムのメタデータ管理手法

本節では，ファイルシステムのメタデータ管理手法について説明する．

2.1.1 Linux Local File Systems

Virtual File System

Linux では様々な種類のローカルファイルシステムが利用できる．Virtual File System（VFS）層は，Linux カーネルにおけるファイルシステムの抽象化層で，ファイルシステムの共通の処理を行い，必要に応じて具体的なそれぞれのファイルシステムに処理を分配する役割を持っている．

VFS はファイルを inode 構造体で管理する．inode はファイルと一対一で対応し，ファイルに関するメタデータ，例えば所有者，所有グループ，タイムスタンプ等が含まれる．また，ファイルシステム種類毎の固有のメタデータを保持するように拡張もできる．

ファイルシステムの階層的ネームスペースは dentry 構造体によって実現される．dentry は，エントリの名前と inode へのポインタを保持し，親ディレクトリの dentry へのポインタと，同階層の dentry のリンクドリストのポインタと，自身がディレクトリの場合は子階層のリンクドリストへの先頭ポインタを保持し，これらを辿ってパスのルックアップが行われる．

これらはメモリ上のデータ構造であり，物理的なストレージデバイス上でどのようにデータを管理し I/O を行うかは，個々のファイルシステムに委ねられる．

ext3 File System

一般的な Linux のファイルシステムの一つである Third Extended File System（ext3）[7] のメタデータ管理手法について説明する．ext3 はブロックデバイスを固定長（通常 1 KB から 4 KB 程度）の領域に区切る．それらには，ファイルシステム全体のメタデータを保持するスーパブロック，inode を格納する inode ブロック，実データを格納するデー

タブロック，空きブロックを管理するためのブロックが割り当てられる．inode は 128 byte の固定長で，例えば inode ブロックが 4 KB なら 32 個格納できる．inode にはエントリに紐づくメタデータと，実データが格納されているデータブロック番号の配列が含まれる．必要なデータブロックの数が多く，ブロック番号の配列の数が固定長で収まらない場合に備え，配列の後半は直接データブロックを示すのではなく，配列の続きを格納する間接ブロックを示す．間接ブロックを２段，３段と使用することで大きなサイズのファイルを扱う．

ディレクトリエントリは，基本的にはデータブロックに inode 番号とファイル名が並んでいるだけで，エントリの検索の際は先頭から順に読み込んでいく．オプションでインデックスを作ることもできる．

単一ディレクトリに大量のファイルを作成すると，パスのルックアップのために読み込むデータブロックが増大するだけでなく，inode のデータブロック番号配列が大きくなり，間接化のオーバヘッドが大きくなる問題がある．

2.1.2 Single Metadata Server

本節では，分散ファイルシステムにおいて最も基本的なメタデータ管理手法である Single Metadata Server による管理を行っている既存の分散ファイルシステムについて述べる．概要図を図 2.1 に示す．

Google File System

Google File System（GFS）[8] は，Google LLC が自社サービスのために開発した分散ファイルシステムである．GFS は本来バッチシステムの効率を上げる為にデザインされたシステムで，巨大なデータを複数のチャンクサーバに分散配置し，ストレージ I/O バンド幅の向上に向いたアーキテクチャとなっている．一方で，メタデータは１台の Single master metadata server が管理する．GFS のファイルを管理する最小単位であるチャンクサイズは，64MB と一般的なファイルシステムと比べて大きく設定されているが，それでもファイル数の増加によるメタデータ量の増大は早い段階で問題になった [9].

GFS が管理するデータが数 TB の頃は上手く動いていたが，それがペタバイトから数十ペタバイトへ増えていくと，メタデータも比例して増えていき問題となった．GFS のクライアントは，例えばファイルをオープンする時などにメタデータサーバと通信して操作要求を行うので，メタデータサーバの性能はクライアントのボトルネックとなった．数千ものクライアントが同時にメタデータサーバにリクエストをした場合，メタデータサーバは数千 ops/sec 程度のリクエスト処理性能に留まった．

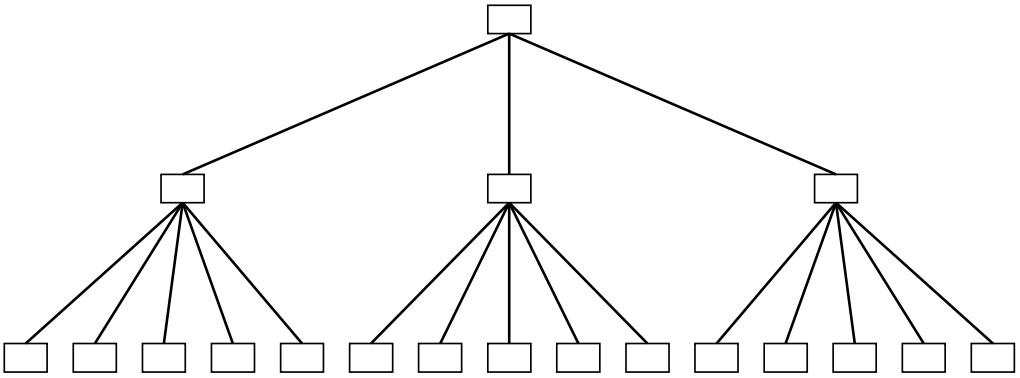


図 2.1　Single Metadata Server — 1 台のメタデータサーバがファイルシステムの全メタデータを集中管理

HDFS

Hadoop Distributed File System（HDFS）[10] は GFS のオープンソースクローンとして開発された分散ファイルシステムで，Hadoop MapReduce のストレージ基盤として使われている．初期では GFS と同じくメタデータサーバである Name Node が 1 台であった．後に Name Node の hot standby による Active/Passive 構成が可能になったが，あくまでも Active サーバが停止した際のバックアップであって，メタデータのサブセットを複数のサーバで分散管理する機能は無い．したがって，GFS と同様のスケーラビリティ問題がある．

Gfarm

Gfarm [11] は HPC 分野の Distributed data-intensive computing をサポートするために開発された分散ファイルシステムである．Gfarm はクライアントがアクセスする実データファイルを計算ノードのローカルストレージデバイスに分散配置し，実際のストレージアクセスをネットワーク越しではなく，クライアントローカルのディスクアクセスになるため，データローカリティを考慮した高スループット I/O アクセスを実現する．

一方でメタデータサーバは Active/Standby 構成による冗長化は行われているが，単一のマスターメタデータサーバが実データファイルの実際の格納場所を管理するため，GFS と同様のメタデータ操作性能のスケーラビリティ問題がある．

2.1.3 Subtree Partitioning

ファイルシステムの階層的ネームスペースを分散する代表的な手法である Subtree partitioning について説明する．概要図を図 2.2 に示す．

Subtree partitioning は，階層的ネームスペースのサブツリーを，個別のメタデータサーバに割り当ててメタデータ操作の負荷分散をする．この手法は階層的ネームスペースを水平分割できるならば，メタデータサーバを増やすことでリニアなメタデータ操作性能を得られる．また，メタデータサーバを追加した台数だけ，オンメモリで管理できる inode キャッシュの数も増加する．

PanFS

PanFS [12] は Panasas が開発したクラスタファイルシステムで, Subtree partitioning による階層的ネームスペースの分割を行っている．Volume と呼ばれる Subtree を各メタデータサーバに割り当てることで，粗粒度な階層的ネームスペースの分散を行っている．

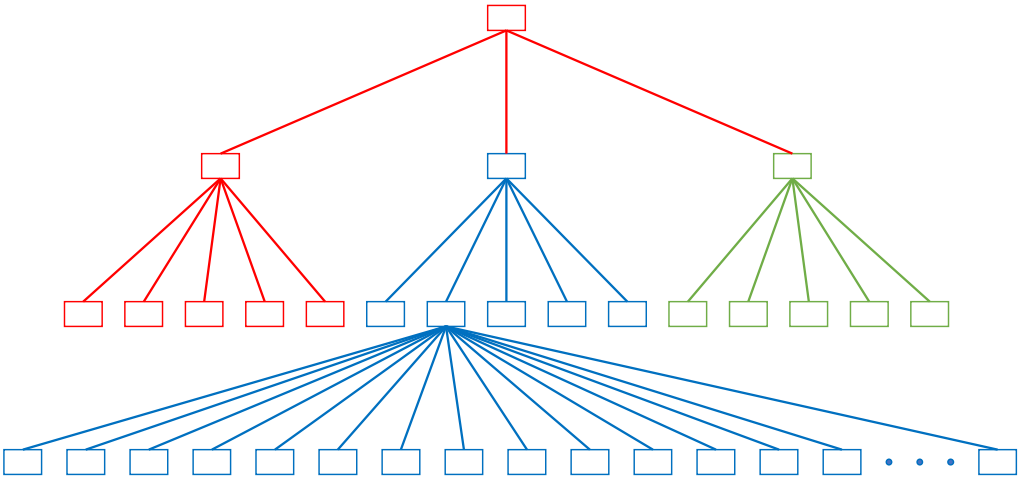


図 2.2　Subree Partitioning — 階層的ネームスペースをサブツリーに分割し，それぞれ別のメタデータサーバに割り当てる

PVFS

Parallel Virtual File System（PVFS） [13] は，クラスターコンピューティングのような並列アプリケーションによる同時アクセスを想定して作られた分散ファイルシステムである．PVFS Version 2 は，より細粒度な階層的ネームスペースの分散機能を備えている．これは Subtree 内のディレクトリを更に別のメタデータサーバに割り当てて分散ができる．

2.1.4 Dynamic Subtree Partitioning

Subtree Partitioning は，例えばあるサブツリー以下ばかりがよく使われていると，そのサブツリーに割り当てられているメタデータサーバにファイルシステム操作リクエストが偏ってしまう欠点がある．つまりネームスペース階層の垂直方向ではメタデータが分散されないため性能がスケールしない．また，ビジー状態のメタデータサーバとそうでないメタデータサーバが発生し，サーバリソースが無駄となる場合がある．

Dynamic subtree partitioning [14] は，メタデータ操作のロードバランスの改善のため，より動的なメタデータ管理を行う手法である．概要図を図 2.3 に示す．Dynamic subtree partitioning の基本的なアイディアは，ビジー状態のメタデータサーバのホットスポットとなっているディレクトリの一部を，ビジーでないメタデータサーバへ動的に移動して負荷分散を行う．

CephFS

Ceph File System（CephFS） [15] は RADOS という分散オブジェクトストア上に実装された POSIX 準拠な分散ファイルシステムである．Ceph はメタデータサーバの負荷状況に基づいて Dynamic subtree partitioning を行う．

HopsFS

HopsFS [16] は HDFS のシングルノードの shared-nothing なインメモリ NewSQL を利用した分散ファイルシステムで，Music streaming サービス提供のために，読み込みオペレーションに最適化されている．

単一ディレクトリ内のすべての inode は親 inode 番号によってパーティショニングされ，各メタデータサーバに割り当てられる．

2.1.5 Hash Partitioning

Dynamic subtree partitioning はディレクトリ単位のメタデータの分散を行うが，同一ディレクトリ内のディレクトリエントリは分散されず，単一のメタデータサーバが管理する．しかし，分散ファイルシステムを利用するいくつかのアプリケーションには，同一

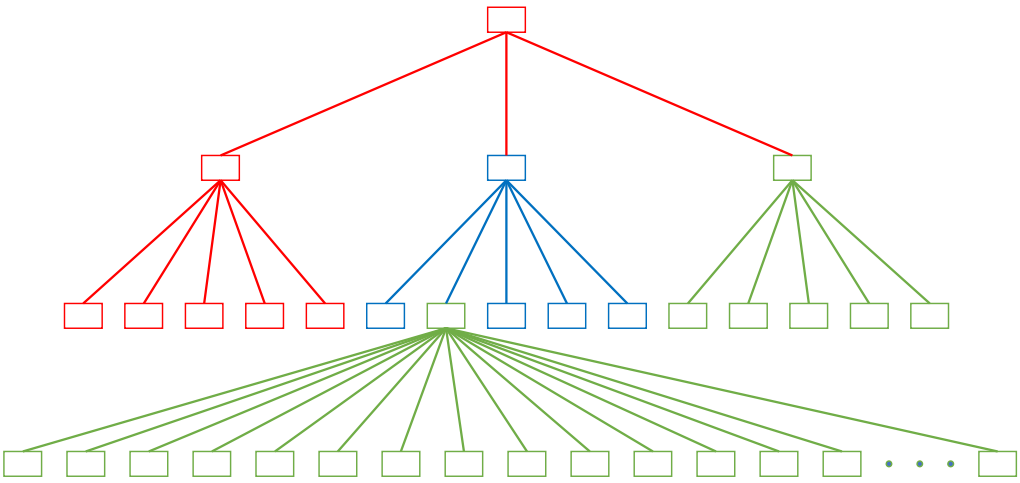


図 2.3　Dynamic Subtree Partitioning — より柔軟で細粒度な Subtree Partitioning

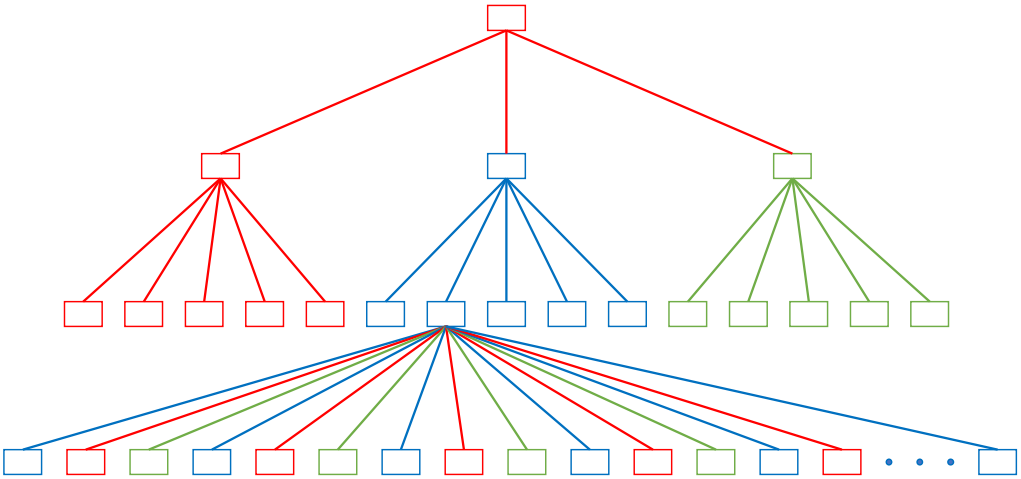


図 2.4　Hash Partitioning — 同一ディレクトリ内の dentry をハッシュで分散し，各メタデータサーバに割り当てる手法

ディレクトリ内に数百万以上の小さなファイルを作成するケースがある [17, 18]．Hash partitioning は，単一ディレクトリ内のエントリを複数のサーバにハッシュ関数を用いて分割，割当する手法である．概要図を図 2.4 に示す．いくつかの既存の分散ファイルシステムは，このようなラージディレクトリに対処するために Hash partitioning を行っている．

Lustre

Lustre [1, 2] は，クラスターコンピューティングで広く使用されている分散ファイルシステムである．Lustre は通常は 1 台のメタデータサーバによってすべてのメタデータ管理を行うが，Distributed Name Space（DNE）phase 1（DNE1）機能により，Subtree partitioning を実施できる．また，DNE phase 2（DNE2) は，ディレクトリエントリをストライプ分割し，別のメタデータサーバに割り当てて分散を行う．どのディレクトリをどのようなレイアウトで分散するかは，ユーザが指定する．inode は，inode に対応するディレクトリエントリを持つメタデータサーバに作られ管理される．

GPFS

（GPFS） [19] は，IBM によって開発された分散ファイルシステムである．GPFS はシンメトリックな client-as-server な分散ファイルシステムで，メタデータは共有ディスクストレージ上で管理する．GPFS はディレクトリエントリの分散のために，Fagin らの Extendible hashing [20] に基づいた Hash partitioning を行っている．ディレクトリが大きくなると，GPFS は新たなストレージブロックをアロケートし，Extendible hashing の overflowing bucket のデータを新しいブロックに移動し，inode からのポインタを更新して対応する．

GPFS はメタデータの一貫性制御のために分散ロックを使用している．メタデータの読み込みアクセスは分散ロックの共有リードロックによって同時にアクセスできるが，コンカレントなメタデータ更新操作では，共有ディスクストレージ上のディレクトリブロックを更新する前に，排他書き込みロックを取得しなければならない．したがって単一ディレクトリへの同時ファイル作成はロック競合によってスケールしない問題がある [3]．

2.1.6 Advanced Hash Partitioning

GIGA+

GIGA+ [3] は，単一ディレクトリ内のファイルを複数のサーバに分散するインデックスを提供するシステムで，既存のファイルシステムの補完を目的としている．GIGA+ はインクリメンタルなディレクトリ分割を行う．ディレクトリ内のファイル数がしきい値を超えると，それらをハッシュ関数で 2 つのグループに分割し，半分を別のサーバに移動させる．グループの分割方法は，ハッシュ値の空間を中央で前半，後半に分割して決定する．その後ファイル数が増えて再び分割しきい値を超えた場合，ハッシュ空間をさらに半

分に分割し，後半を別のサーバに移動する．このように，ハッシュ空間を再帰的に分割していき，インクリメンタルにディレクトリエントリを分散する．ハッシュ空間の分割は，各サーバが個々に判断できるため同期や直列化を必要としない．また，ハッシュ空間をどのように分割したかという情報と，現在のサーバ一覧があれば，どのハッシュ空間がどのサーバに割り当てられているかを決定できる．クライアントは，ハッシュ空間がどのように分割されたのかという情報をサーバから取得し，更新があった場合はその情報をキャッシュするので，基本的には 1 ホップで目的のファイルを格納しているサーバに問い合わせられる．GIGA+ は，少なくとも数百万のファイルを POSIX-compliant な単一ディレクトリに格納し，高いスケーラビリティと並列性を実現する．しかし，GIGA+ の提供する分散インデックススキームは，ディレクトリ分割中のファイルの移動は単純なコピーであるため，移動中の一貫性の問題や，失敗した際のアトミック性の保証は GIGA+ を使うシステム側で保証しなければならない問題がある．

OrangeFS

OrangeFS [21] は PVFS の商用サポート版の分散ファイルシステムである．OrangeFS はラージディレクトリのエントリを分散するために，GIGA+ を単純化したアルゴリズムを使用している．

SkyFS

SkyFS [22] は Extendible hashing に基づく Hash Partitioning を行い，GIGA+ のディレクトリパーティショニング中のパフォーマンス低下改善を行っている．

IndexFS

IndexFS [23] は，クラスタファイルシステムのための汎用的なメタデータサービスの提供を目的とするシステムである．IndexFS は複数の IndexFS Server を持ち，Dynamic namespace patitioning ポリシーにより，ディレクトリとディレクトリエントリの両方をマルチメタデータサーバに動的に分散する．

IndexFS はまず最初に階層的ネームスペースの各ディレクトリを 1 つの IndexFS Server に割り当てる．割り当てには，“power of two choices” ロードバランシング [24] を使っている．多くのクラスタファイルシステムの実際のユースケースでは，およそ 90% のディレクトリのエントリ数は 128 個以下と小さい [25] ため，ディレクトリエントリまで分散する必要はなく，ディレクトリを複数のメタデータサーバに分散割り当てすれば十分である．あるディレクトリのエントリを 1 台のサーバでまとめて保持すると，分散して保持する場合と比べて *readdir* 等のエントリスキャンオペレーションのオーバヘッドを削減できるメリットがある．

ディレクトリエントリ数が巨大に成長するディレクトリに関しては，第 2.1.6 章で述べた GIGA+ のパーティショニングにより，エントリー数がしきい値を超えるとインクリメンタルにエントリを分割し，別サーバに分散する．

IndexFS では，いくつかのメタデータ操作，例えばディレクトリ分割や rename 操作で，分散トランザクションプロトコルを必要としており，現在の実装では two-phase commit を使っている．トランザクションの障害時復旧に備えるため，Write-ahead logging（WAL）を src と dest の両方に作る必要があり，追加のオーバヘッドが掛かる．また two-phase commit はトランザクションの参加者の 1 つが失敗したり，コーディネータノードとの通信がネットワークパーティショニング等で失敗するとブロックしてしまう問題が知られている [5].

第 3 章

PPMDS: 分散メタデータサーバの設計と実装

3.1 PPMDS: 分散メタデータサーバの設計

PPMDS の設計は，ファイルシステムの機能を損なうことなく，ファイル操作を複数メタデータサーバに分散して，スケーラブルなパフォーマンスを得ることを目的とする．この目的のために特に解決すべき問題は，どのように階層的ネームスペースを効率よく分散で管理するかである．従来のローカルファイルシステムはディレクトリ毎に，ディレクトリエントリを用いて階層的ネームスペースを管理しており，これは，エントリー名と inode 番号のリスト（または木構造）となっている．このデータ構造は，必ずしも分散環境において最適とは言えない．

PPMDS は，ある 1 つのディレクトリ中のエントリを管理するのに，従来のひとまとまりのディレクトリエントリを用いない．代わりに，ディレクトリ中のエントリーを複数のサーバーに分散して管理する．追加で管理しなければならない情報は，エントリーをどのサーバに分散したかを記録するサーバーリストのみで，ディレクトリごとに保存される．

また，PPMDS は複数サーバ間でノンブロッキング分散トランザクションを使う．これはファイルシステム操作同士が衝突していない場合，不必要なブロッキングを行わない．ディレクトリ作成操作は，複数のエントリを複数のサーバにアトミックに作成するが，ノンブロッキング分散トランザクションはこれを効率的に実現する．この設計はグローバルロックやディレクトリレベルの分散ロックを必要とせず，プロセスの並列ファイルシステム操作は，オペレーションが衝突しない限り，同時並列に実行されるのを保証する．

3.1.1　Lookup 操作とデータ構造

第 2.1.1 章で述べたように，従来のファイルシステムは単一マシンのブロックデバイスに最適化されたもので，inode データ構造でメタデータを管理している．しかし，それらのデータ構造は多くの非効率な間接化が含まれており，分散環境においては最適とは言えない．

ファイルシステムのメタデータは大きく分けて 2 種類ある．

inode　ファイルやディレクトリと 1 対 1 に紐づくメタデータ．実データ格納ブロックのリスト，UID，GID，タイムスタンプ，パーミッション等

dentry　階層的ネームスペース．/usr/local/bin/filename のようなパスを表現するためのメタデータ．

従来のファイルシステムの dentry の管理方法では，分散環境での並列処理を行う上でいくつか問題がある．ある 1 つのディレクトリのディレクトリエントリが単一のサーバで管理されていると，単一ディレクトリに対して大量の同時ファイルシステム操作が発生した場合，いずれそこがボトルネックになってしまう．

PPMDS は，ディレクトリエントリを持たない．ファイルシステムのメタデータは key-value ペアで管理する．各 key-value ペアは基本的に，従来のファイルシステムのメタデータでの inode に対応する．ファイルのメタデータは key-value ペアの value として格納される．そして，key は親ディレクトリの inode 番号（親 inode 番号，pino）とエントリの名前のペアが格納される．ディレクトリ内のエントリリスト取得操作（readdir，ls）は，全メタデータサーバに対して，ディレクトリの inode 番号で key の範囲検索を実行して実現する．Inode エントリは，pino とエントリ名のペアで一意に識別される．

図 3.1 に “/dir-x/file-a” のルックアップ操作の例を示す．図 3.1 は，ファイルルックアップの例を示す．/dir/file1 というファイルをルックアップする手順は，

1. (inode=1, “/”) を key としたルート inode エントリを読み込む．ルートディレクトリの親は存在しないので，親 inode 番号は 0 とする．
2. ルートディレクトリのパーミッションをチェックし，ルックアップが許可されていることを確認する．そして，ルートディレクトリの inode 番号（= 1）を取得する．
3. (1，“dir”）を key とした，“/dir-x” の inode エントリを読み込む. パーミッションのチェックと，“/dir-x” の inode 番号 （= 2）を取得する．
4. (2，“file1”）を key とした，“/dir/file-a” の inode エントリを読み込む.

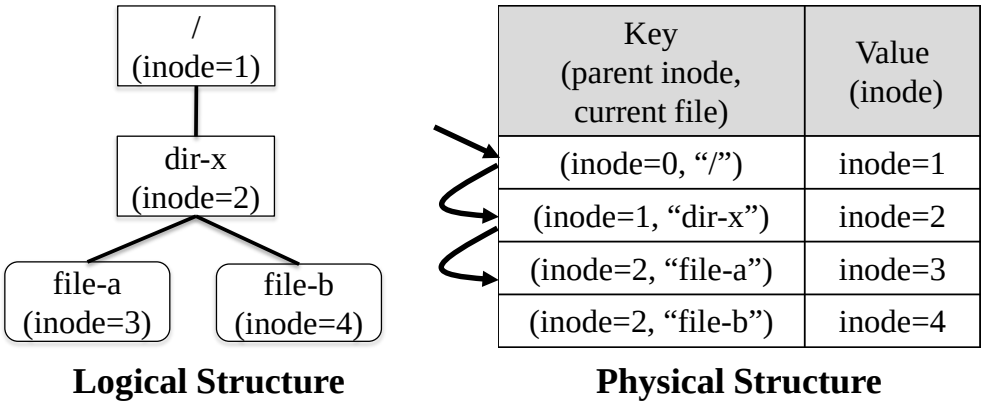


図 3.1 PPMDS の Lookup 操作例

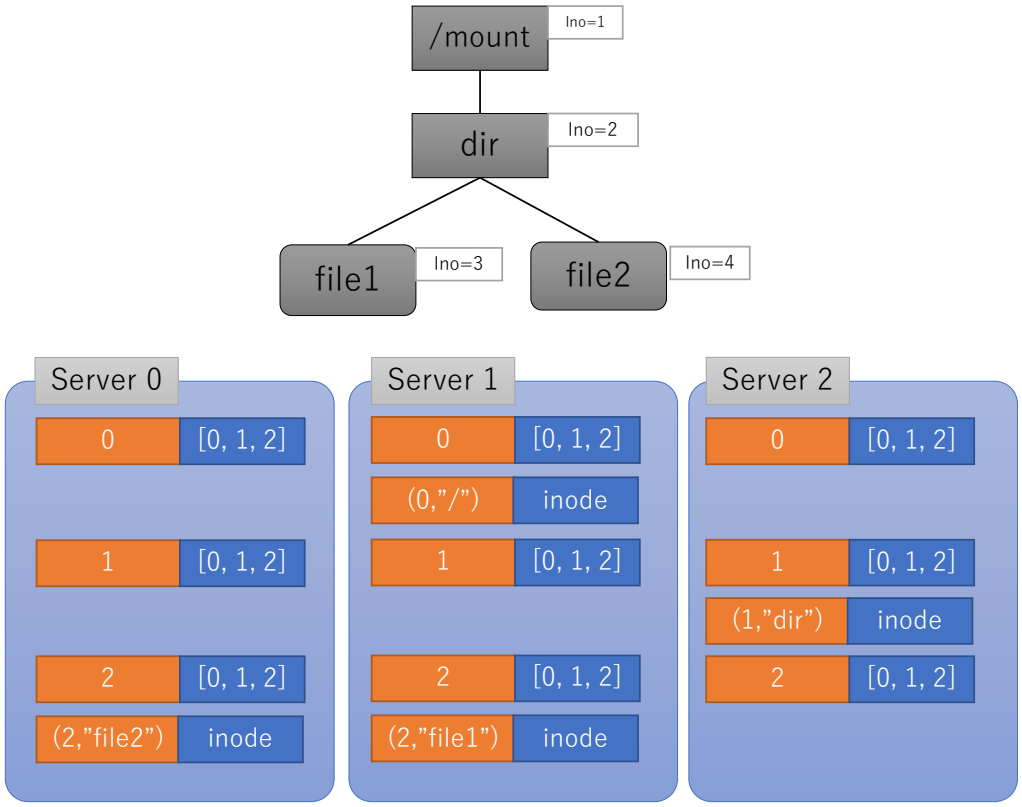


図 3.2　メタデータの分散データ構造—ディレクトリ毎に，ディレクトリの分散先サーバ情報を全サーバに持つ

3.1.2 メタデータの分散データ構造

PPMDS はメタデータを key-value ペアを使って管理する．これにより，自然に複数のメタデータサーバに分散できる．各メタデータサーバは独立した 1 つの key-value ストアを持つ．

readdir のようなディレクトリエントリのリスト取得のために，key-value ストアは範囲検索クエリ機能を持つ．ディレクトリが多くのエントリを持つ場合，それらのエントリを多くのサーバに分散するのは良い考えだが，ディレクトリのエントリ数が少ない時は，多くのサーバに少量のメタデータを分散し，全てのサーバに範囲検索リクエストを行うと，不必要なオーバヘッドが発生する．それを避けるため，ディレクトリ毎に，ディレクトリ分散サーバリストを管理する．

ディレクトリの分散サーバのリストを管理するために，ディレクトリ分散サーバリストを第 3.1.1 章で述べたメタデータのデータ構造とは別に，すべてのサーバに格納する．図 3.2 に概要図を示す．これはディレクトリ毎に作成され，メタデータのデータ構造の同じ key-value ストアに格納される．key はディレクトリの inode 番号，value はディレクトリ内の inode エントリを格納する分散サーバ識別子のリストである．ディレクトリ内の inode エントリは，リスト内のメタデータサーバ間でハッシュ関数を用いて分散される．すべてのメタデータサーバがサーバリストを持つ理由は，エントリを検索する際にトラフィックが集中するのを避けたいからである．サーバーリストが 1 つの場合，すべての検索プロセスが 1 つのサーバーリストにアクセスしてしまうため，パフォーマンスのボトルネックになってしまうため，エントリを検索する際のボトルネックを取り除くために，すべてのサーバに格納することにした．これはディレクトリを作成するための追加のオーバーヘッドを必要とするが，HPC アプリケーションではディレクトリの作成は稀なので，ディレクトリ作成のパフォーマンスよりもルックアップのパフォーマンスを優先させている．

3.1.3 ファイル作成トランザクション

新しいファイルの作成では，PPMDS はメタデータのコンシステンシを保つためにトランザクションが必要となる．トランザクションは他のトランザクションから分離していなければならない．トランザクション実行中に，ノードの障害等でオペレーションが予期せず失敗するかもしれない．そのような場合でも，分散システムは期待通りにサービスを継続しなければならない．しかし，もしシステムが期待通りにレスポンスを返さなかったら，コミット前のオペレーションはメタデータデータベースに反映されてはならない．これらの要求を満たすため，PPMDS は分散トランザクション処理機能を提供する．

以下に例として，新たなファイル “/dir-x/file-c” を作成する手順を説明する．

1. クライアントは親ディレクトリ “/dir-x” を lookup し，inode=2 を取得した後，(inode=2, “file-c”）のファイル作成要求を PPMDS サーバに送信する．
2. リクエスト受けたサーバ（α）はファイル作成のためのトランザクションを開始する．αはローカル key-value ストアから inode=2 を key としてディレクトリの分散サーバのリストを取得する．
3. αはそのリストの中から “file-c” を作成する対象サーバ（β）を決定し，(inode=2, “file-c”）のエントリを作成するようにβにリクエストする．βはエントリを作成し，αに応答を返す．
4. αはトランザクションをコミットする．

トランザクションは複数サーバ間にまたがる，いくつかの key-value ペアをアトミックに更新できる．ノンブロッキングトランザクションについて詳しくは第 3.2 章で述べる．

3.1.4　ディレクトリ作成トランザクション

新しいディレクトリを作成にも，トランザクションが必要となる．以下に，“/dir-y” を作成する場合の，手順例を述べる．

1. クライアントはルートディレクトリ “/” を lookup し，inode=1 を取得した後，(inode=1, “dir-y”) のディレクトリ作成要求を PPMDS サーバに送信する．
2. 選択された PPMDS サーバ（α）はディレクトリ作成のためのトランザクションを開始する．αはローカル Key-value ストアから inode=1 を key としてディレクトリの分散サーバのリストを取得する．
3. αはそのリストの中から dir-y を作成する対象サーバ（β）を決定し，(inode=1, ”dir-y”) のエントリを作成するようにβにリクエストする．
4. βは新しいディレクトリの分散サーバのリストを決定する．
5. βは一意の inode 番号を生成し，関連する全てのサーバにディレクトリの新規エントリを作成する．βはαに応答する．
6. αはトランザクションをコミットする．

ディレクトリを作成するには，関連するすべてのサーバに分散サーバのリストを作成するための追加のステップ 5 が必要となるが，ディレクトリ操作は，HPC アプリケーションにおけるファイル操作よりもはるかに稀であるため，このような設計上の選択を行った．

3.2　ノンブロッキングトランザクション

文献 [6] で，Herlihy らは，効率的なノンブロッキングトランザクションのスキームを提案している．これは効率的な分散トランザクション処理のテクニックで，Dynamic-sized software transactional memory（DSTM） [6] に基づいている．熊崎らは文献 [26, 27]

で DSTM によるノンブロッキングトランザクションのスキームを拡張し，key-value ストアを使って実現した．本研究はで熊崎らのノンブロッキングトランザクションスキームを利用している．

3.2.1 概要

KVS 上の複数の Key-Value ペアに対する一連の読み込み，書き込みを 1 つの処理単位としたものがトランザクションである．トランザクションは，複数のクライアントから並行に発行され，各サーバはそれぞれのトランザクションを並行に処理するが，システム全体ではトランザクションがある逐次的な順番で処理されたかのように振る舞う．また，トランザクションは Serializable 分離レベルを満たす．この特性により，コミットに到達するトランザクションは，他の処理中のトランザクションから分離される．

重要な特性として，本トランザクション実装は，obstruction-freedom [28] である．これは，トランザクションが十分に長い時間単独で走ったら進行するという進行保証を表す．より強い進行保証である lock-freedom [29] や wait-freedom [30] と同様に，obstruction-freedom はノンブロッキングである．ノンブロッキングトランザクションは，障害や遅延が発生したトランザクションが，他のトランザクションの進行を邪魔しないという特性を持つ．

3.2.2 データ構造

ノンブロッキングトランザクションをサポートするため，KVS は，Key-Value ペアを図 3.3 に示す構造で保持する．各 Key-Value ペアは，Key，バージョン番号，Old Value，New Value，*Transaction ID* から構成される．Transaction ID は，その Key-Value ペアの *Owner* を表す．Transaction ID を Key として KVS に問い合わせれば，Owner の現在の状態を取得できる．Owner の状態は，*Committed*, *Aborted*, *Active* の三種類がある．Owner が現在 Committed であれば，既にその Key-Value ペアを最後に処理したトランザクションは処理を終えているため，New Value が最新の値となる．Owner が現在 Aborted であれば，最後に処理したトランザクションは処理を終えているが，処理は途中で中断している．トランザクションは，Key-Value ペアの Owner になる際に，その時点での最新の値を Old Value に退避するため，Owner の状態が Aborted である場合は Old Value がコミット済みの最新の値となる．Owner が Active であれば，現在トランザクションがその Key-Value ペアを処理中であることが分かる．

トランザクションは，Key-Value ペアに対して読み込み，書き込みを行う前に，Key-Value ペアを *Open* して，Owner になる必要がある．もし，Key-Value ペアの Owner の状態が Active であれば，現在他のトランザクションがその Key-Value ペアの Owner であり処理中なので，処理が終わるのを待つか，Owner 権を奪うかを選択しなければならない．Owner 権を奪うには，他のトランザクションの状態を Aborted へ変更する．トラ

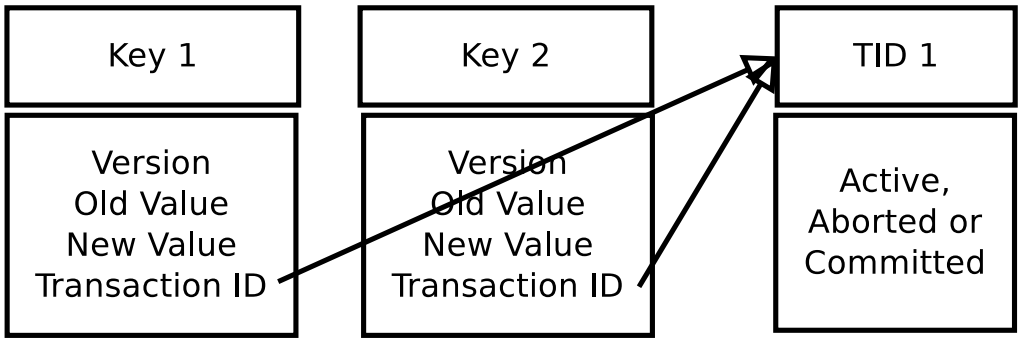


図 3.3　トランザクショナルキーバリューデータ構造

ンザクションはこの判断をコンテンションマネージャと呼ばれるコンポーネントに依頼する．この競合解決方法は，obstruction-freedom を満たす範囲内ならば，どのようなアルゴリズムでも良い．現在は文献 [26] と同様にバックオフ待機アルゴリズム [31] を用いている．これは，ある待ち時間上限の範囲内では，一定時間待機後に Owner の状態を再チェックし，Active なら前回の 2 倍の時間待機を繰り返し，待ち時間上限を超過したら Owner 権を奪うというアルゴリズムである．

トランザクションのコミット処理は，自身の Transaction ID に関連付けれている状態を，Active から Committed に変更することで成される．この変更は，Compare And Swap （CAS）によって，アトミックに行う．もし，この変更に失敗したら，トランザクションは他のトランザクションによって処理が中断されたと判断する．

3.2.3 拡張 1: サーバーサイドトランザクション実行

文献 [26] の提案は，標準的な memcached [32] 互換のキーバリューストアの利用を前提としている．これには，キーバリューストアサーバ側は手を加えずにクライアント側のみでトランザクションを実現することで，幅広いキーバリューサーバに対応できるという利点がある．我々はより高性能なメタデータサーバを目的としており，memcached との互換性は必要としていない．

そこで本研究では，オリジナルのノンブロッキング分散トランザクションの機能拡張として，トランザクション実行をサーバ側に移譲し，トランザクション実行中のキーバリューストアへの通信回数の削減を行った．概要図を図 3.4 に示す．

クライアントがトランザクションを開始すると，キーバリューペアにアクセスするために，操作ごとにサーバへのリモートプロシージャの呼び出しが発生する．さらに，トランザクションの開始時と終了時には，クライアントは，比較とスワップのアトミック操作によって，リモートサーバに格納されたトランザクションの状態を変更する必要がある．しかし，サーバ側で実行すると，この一連のリモートプロシージャの呼び出しを減らすことができる．キーと値のペアへのアクセスは，サーバでローカルに処理することができる．この場合，クライアントはサーバへのトランザクションの開始を要求するだけとなる．例えば，ファイルを作成するために，クライアントは，親 inode 番号とエントリ名を渡して PPMDS サーバにファイルの作成を依頼し，その結果を待つ．PPMDS サーバがトランザクションを開始すると，ローカルのキーバリューストアにトランザクションの状態を作成する．トランザクション中，ローカルのキーバリューストアに目当てのペアが格納されている場合，キーバリューペアへのアクセスはローカルで行われることがある．トランザクションをコミットする際には，常にローカルの key-value ストアに格納されているトランザクションの状態を変更する．サーバサイドトランザクション実行により，トランザクション内の多くのリモートプロシージャの呼び出しを排除することが可能である．

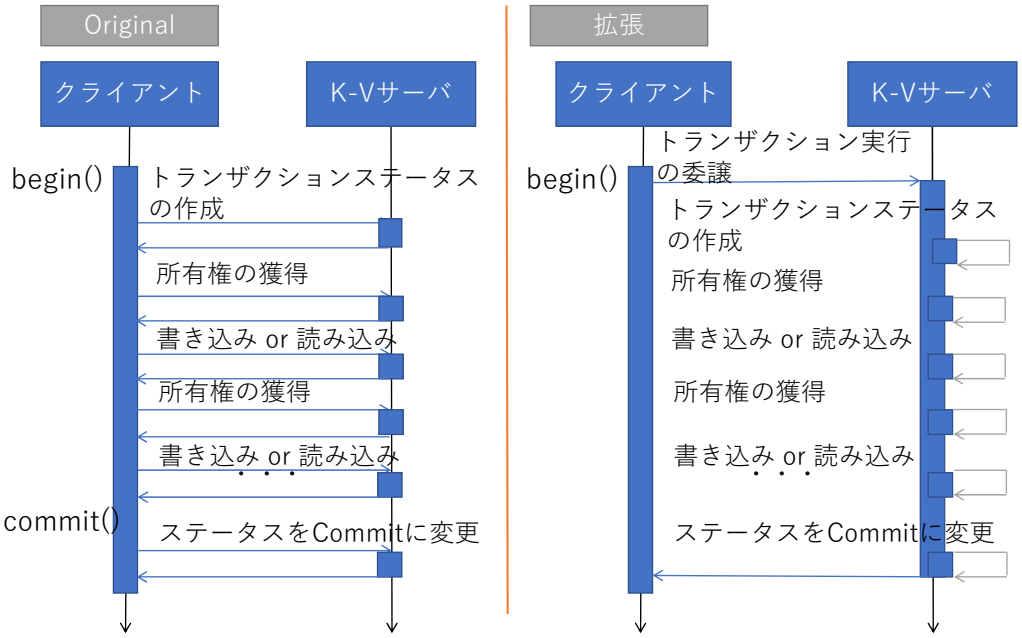


図 3.4　ノンブロッキング分散トランザクション拡張 1: サーバーサイドトランザクション実行

3.2.4 拡張 2: マルチリーダ化

トランザクショナルキーバリューストアでは，Key-Value ペアへアクセスする前に Key-Value ペアを Open する．この時，別のトランザクションが処理中であったらトランザクションは待たされるが，文献 [26] の Open は，書き込み，読み込みを区別しないため，読み込み Open の競合によって無駄な待機が発生する．

提案手法では，Open を読み込み Open と書き込み Open を区別するように変更し，読み込み同士の Open が衝突しないように機能拡張を行った．図 3.5 は読み込みトランザクションが commit に成功した場合のシーケンス図を示す．図 3.6 は読み書き衝突により，トランザクションが abort する場合のシーケンス図を示す．

読み込み Open は Key-Value ペアの Owner 権の移動を行わない．代わりに，トランザクションは当該 Key-Value ペアの Key とバージョンを一時領域に保持しておく．そして，コミット処理の際にもう一度 Key-Value ペアを読み込み，バージョンを比較する．バージョンが一致しない場合，他のトランザクションが該当 Key-Value ストアを書き換えたことが分かるため，コミット処理を中断する．

3.3 実装

図 3.7 に PPMDS アーキテクチャの概要を示す．PPMDS は，マルチマスター型サーバである．各 PPMDS はローカルに key-value ストアを持ち，クライアント接続を持つ．クライアントは FUSE [33] フレームワークを用いて実装した．

3.3.1 PPMDS Server

各サーバは Kyoto Cabinet [34] をローカル key-value ストアとして使う．Key の範囲検索機能が必要となるため，TreeDB を inode 構造のための key-value ストアとして使用した．トランザクションの state の保存には，パフォーマンスの良い StashDB を利用した．どちらの key-value ストアも，ロック粒度は行レベルである．また，key-value ペアに対する任意の read-modify-write 操作をサポートしている．この機能により，トランザクション実装に必要な CAS 等のアトミックオペレーションを実装した．

メッセージ通信には，MessagePack-RPC for C++ [35] を使用した．サーバのソフトウェアスタックを図 3.7 に示す．各サーバはローカル key-value ストアへのアクセスのための，KVS RPC サーバインタフェースを持つ．また，ファイルシステムのメタデータ操作のための FS RPC サーバインタフェースを持つ．

KVS RPC サーバインタフェースは，リモートクライアントや他のサーバからのサーバローカル key-value ストアへのアクセスインターフェスで，ノンブロッキングトランザクションの実装に必要な以下の atomic 操作を実装している．

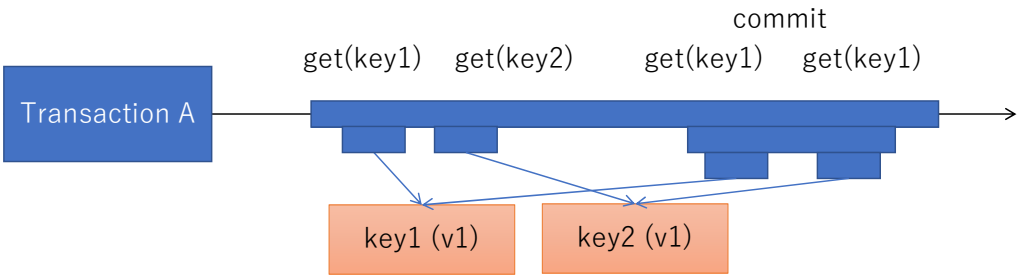


図 3.5　ノンブロッキング分散トランザクション拡張 2: マルチリーダ化 — commit 成功時のシーケンス図．commit 処理中に読み込み Open した KV-pair をもう一度 get し，バージョンが変わっていないことを確認する．

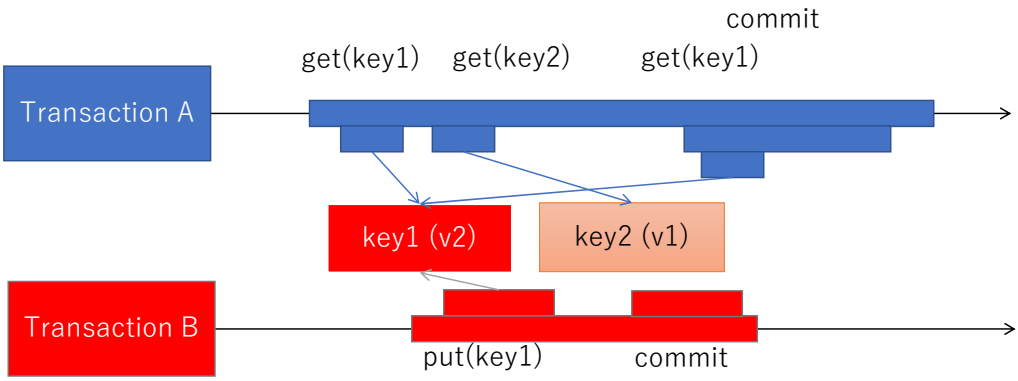


図 3.6 ノンブロッキング分散トランザクション拡張 2: マルチリーダ化 — 読み書き衝突によるトランザクション A が abort した場合のシーケンス図．トランザクション B によって key1 の KV-pair のバージョンが変わっており，読み書き衝突を検出する．トランザクション A は key1 のバージョン不一致を commit 処理中に検出し，自身のステータスを abort にする．

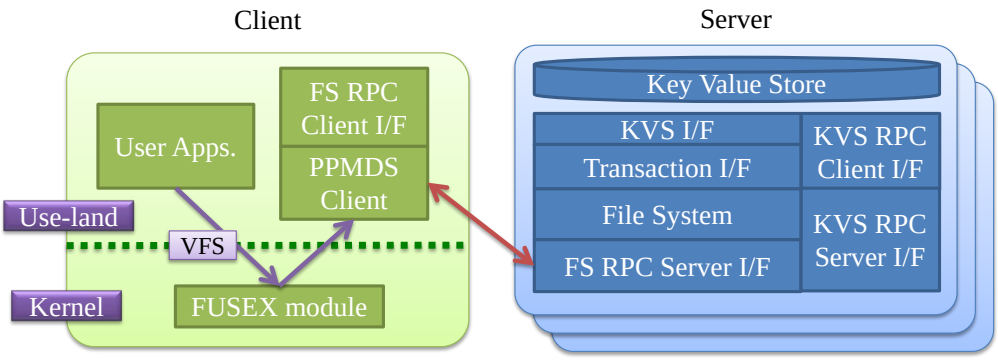


図 3.7　PPMDS アーキテクチャ

(version, value) = gets(key)
: value と現在のバージョンを返却する．

cas(key, value, version)
: 現在の key-value ペアのバージョンが渡した version と一致した場合のみ，key-value ペアを更新する．

add(key, value)
: key が存在しない場合のみ，key-value ペアを追加する．

replace(key, value)
: key が存在する場合のみ，value を更新する．

key = uadd(value)
: ユニークな key を生成し，その key と渡された value のペアで保存し，key を返す．

FS RPC サーバインタフェースは，以下のファイルシステム操作をサポートしている．

- ファイルシステム初期化
- file creation
- file stat
- file removal
- directory creation
- directory stat
- directory removal

read や write 等のファイルのデータの読み書き操作は今の所サポートしていない．

トランザクションインターフェスはノンブロッキング分散トランザクションをサポートしている．もし，ターゲット key-value ペアがローカル key-value store に存在する場合は，直接ローカルファイルシステムにアクセスする．そうでない場合は，リモートの PPMDS サーバに KVS RPC クライアントインターフェースを使ってアクセスする．

3.3.2 PPMDS Client

PPMDS client は FUSE low-level interface，fuse_lowlevel_ops を使って実装した．以下のコールバック関数を実装している．

- init
- destroy
- lookup
- getattr
- opendir
- readdir

- releasedir
- mkdir
- rmdir
- create
- unlink

PPMDS は親 inode 番号とエントリ名で inode エントリを管理する．つまり，実装の際にすべての操作で親 inode 番号とエントリ名が要求される．FUSE low-level interface は，getattr 以外の操作ではこの 2 つの情報を得られる．getattr は親 inode 番号が得られない．そこで，FUSE kernel module を拡張し，pino とエントリ名が得られるように，以下の関数を実装した．

fuse_req_get_idpair(req, &pino, &name)

req は FUSE low-level interface で使用される fuse_req_t 構造体．fuse_req_get_idpair コールが成功すると，親 inode 番号 pino とエントリ名 name が返却される．この拡張 FUSE を *FUSEX* と名付けた．

第 4 章

評価

本章では提案メタデータ管理手法の有効性を実験により検証するため，第 4.2 章単一ディレクトリへの並列メタデータ操作性能評価，第 4.3 章分散メタデータサーバのスケーラビリティ評価，第 4.4 章 Lookup キャッシュの効果，第 4.6 章既存研究との比較評価，第 4.7 章オブジェクトストレージと組み合わせた際の性能調査，を行った．

4.1 評価環境 1

第 4.2 章単一ディレクトリへの並列メタデータ操作性能評価と，第 4.3 章分散メタデータサーバのスケーラビリティ評価，に使用した評価環境について述べる．

評価環境を表 4.1 と表 4.2 に示す．

すべてのクラスタノードはデュアルソケットの 2.40 GHz quad core Xeon E5620 プロセッサと 24GB のメモリを持っている．11 ノードをクライアントに，14 ノードをメタデータサーバとして使用した．クラスタのネットワークは，クライアントノードは Gigabit Ethernet で接続，メタデータサーバは 10 Gigabit Ethernet で接続されている．クライアントノードクラスタとメタデータサーバクラスタ間のネットワークは 10 Gigabit Ethernet で L2 スイッチのカスケード接続がされている．

評価にはメタデータパフォーマンス測定に mdtest HPC benchmark [36] を使用した．mdtest は並列ファイルシステムのメタデータベンチマークで，MPI で実装されている．mdtest はファイルやディレクトリを指定のディレクトリに指定したツリー構造で作成する．その後，作成したファイルやディレクトリのメタデータを読み取り，最後に削除をする．これらの操作は MPI プロセスによって並列に実行される．今回の評価では，各 mdtest プロセスは単一ディレクトリ内に 12,000 エントリ作成操作を行った．最大 88 クライアントプロセスから，1,056,000 エントリの作成を行った．

表 4.1　評価環境 1: Server Nodes

CPU	Xeon E5620 2.40 GHz
Sockets/node	2
Cores/socket	4 + Hyper-Threading
Memory	24GB
Num of Nodes	14
Kernel	Linux 2.6.26-2-amd64 x86_64

表 4.2　評価環境 1: Client Nodes

CPU	Xeon E5620 2.40 GHz
Sockets/node	2
Cores/socket	4 + Hyper-Threading
Memory	24GB
Num of Nodes	11
Kernel	Linux 3.0.0-12-server x86_64

4.2　単一ディレクトリへの並列メタデータ操作

図 4.1 に単一ディレクトリへの並列ファイルシステムメタデータ操作を行った評価結果を示す．グラフの横軸は PPMDS サーバノードの数，縦軸は 1 秒あたりのファイルシステム操作数を表す．合計クライアントプロセス数は 88 プロセスで，11 クライアントノードを使用した．各クライアントノードは，8 クライアントプロセスを実行した．

ファイル関連操作，ファイル作成，ファイル stat，ファイル削除と，ディレクトリ関連操作のうち，ディレクトリ stat は，3 台のメタデータサーバまでスケーラブルなパフォーマンスを示し，約 62,000 ops/sec に達した．サーバが 1 ノードの場合と比べて，約 2.58 倍の性能向上が得られた．4 台以上の MDS メタデータサーバでスケールしなかった理由は，クライアントプロセスの数が十分でなく，62,000 ops/sec 以上のベンチマーク負荷が掛けられなかったと考える．これについては，4.3 節でクライアント数を変化させた際の PPMDS メタデータサーバのスケーラビリティ評価を示す．

ディレクトリ関連操作のうち，ディレクトリ作成とディレクトリ削除については，メタデータサーバの数を増やした場合のメタデータパフォマンスの改善は得られなかった．これは，PPMDS はディレクトリ操作より，ファイル操作を重視する設計になっている．PPMDS はディレクトリ作成時，分散トランザクションを使ってサーバリストを全サーバに作成している．これが，パフォーマンス低下の原因になっている．また，ディレクトリ削除は，分散トランザクションによってディレクトリエントリの存在チェックを全サーバに対して行ってから削除している．このため，これらの性能結果は期待通りである．

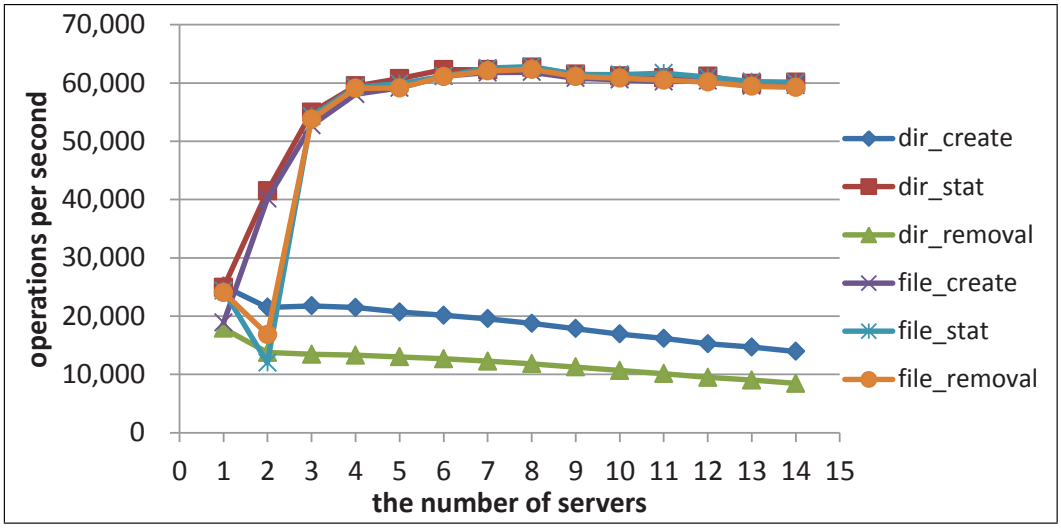


図 4.1　88 クライアントによる並列メタデータ操作性能

4.3 分散メタデータサーバのスケーラビリティ

クライアントプロセス数を増やした時の，単一の共有ディレクトリで並列にファイルを作成した場合の性能を図 4.2 に示す．横軸ははクライアントプロセスの数，縦軸は秒間のファイル作成数を示す．

1 台の PPMDS サーバを使用している場合，最大 33 のクライアントプロセス，25,000 のファイル作成まで性能が向上した．これは，1 つの PPMDS サーバの最大性能と考えられる．33 クライアントプロセスまでは，パフォーマンスはクライアントのリクエストレートによって制限される．

PPMDS サーバを 2 台使用した場合，最大で 44 のクライアントプロセスまでスケールし，40,000 ファイル作成毎秒に達した．

一方，PPMDS サーバ数が 4 台以上の場合は，似たようなスケーラブル性能を示した．これはクライアントリクエストレートの上限に達していると考えられる．4 台以上の PPMDS サーバの性能評価には，より多くのクライアントプロセスが必要となると考える．

4.4 Lookup キャッシュの効果

本研究の提案メタデータサーバ PPMDS の inode lookup 性能における，キャッシュの効果，影響について評価を行った．図 4.3 に結果のグラフを示す．横軸はメタデータサーバの数で，1 台から 15 台まで変化させた．縦軸は秒間のファイル作成数を示す．クライアントプロセス数は固定で，128 プロセス使用している．

ファイル作成操作の際，クライアントがどこにファイルを作成するかによって，*unique* と *shared* の 2 種類に分別した．unique は，プロセス毎に別々のディレクトリを作り，その中にファイルを作成する．shared は，すべてのプロセスが，同一のディレクトリの中に並列にファイルを作成する．一般的に，shared の方がファイル作成時に同一ディレクトリへのアクセス競合が発生するため，スケールし難いアクセスパターンとなる．

クライアントはファイル作成操作を行う際，与えられたパスの lookup 操作を行うが，その際，本提案手法では，経過時間による Expire 付きの LRU キャッシュによる lookup 結果のキャッシュを行っている．指定のキャッシュ期間を経過するか，キャッシュ容量を超過したら最も最後にアクセスされたキャッシュから削除を行う．本評価では，Expire を 1 秒に設定し，結果をキャッシュする場合と，全くキャッシュを行わない場合とで比較を行った．

評価結果について，unique と shared 両方で，lookup の結果をキャッシュした場合が，キャッシュを行わない場合よりも良い性能を示した．unique は，サーバが 15 台でキャッシュを行った場合にピーク性能，270,874 ファイル作成毎秒だった．shared は，サーバが 15 台でキャッシュを行った場合にピーク性能，157,326 ファイル作成毎秒だっ

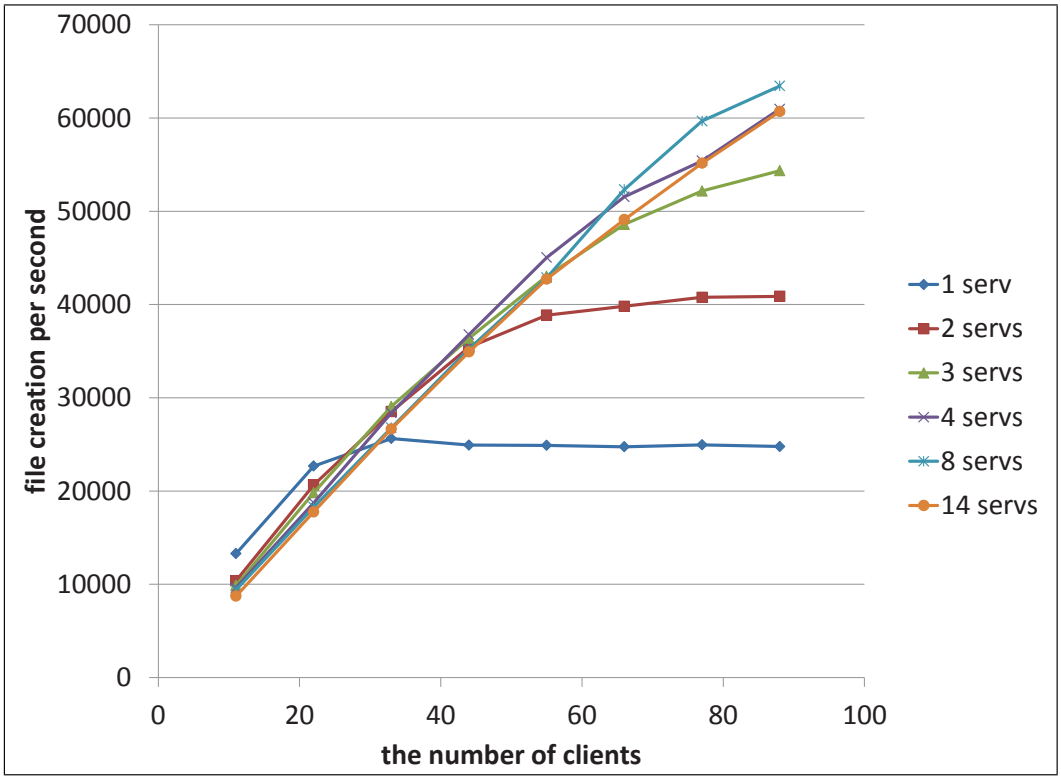


図 4.2 単一ディレクトリへの並列ファイル作成性能

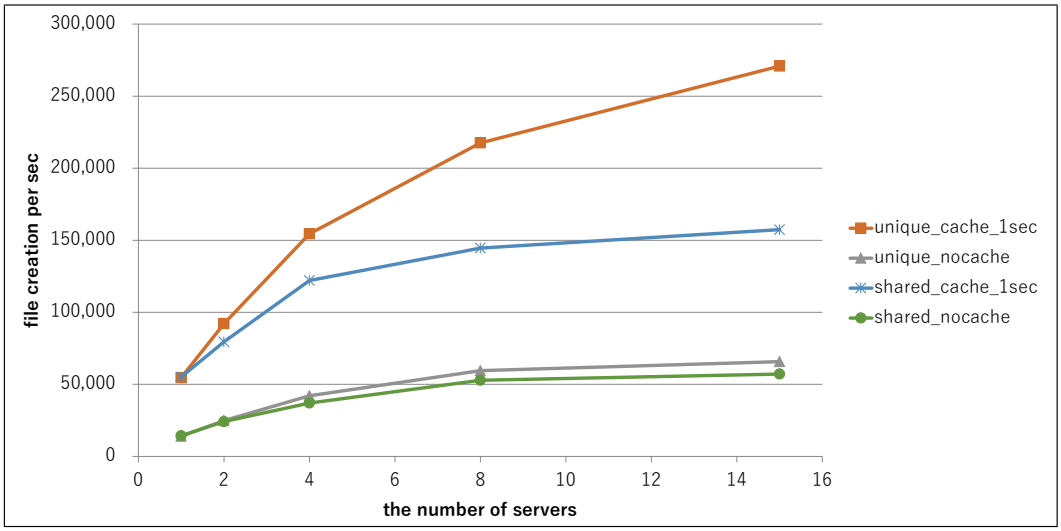


図 4.3　lookup キャッシュの効果

表 4.3 評価環境 2: メタデータサーバノード

CPU	Xeon E5-2695 v2 2.40 GHz
Sockets/node	2
Cores/socket	12
Num of Nodes	5
Memory	64GB
Network	Mellanox Technologies MT27500 4x FDR (56Gbps)

た．キャッシュを行わない場合と比較すると，サーバ 15 台の場合，unique は，4.12 倍の性能向上，shared は 2.75 倍の性能向上を得られた．

ファイルシステムのメタデータ操作において，lookup 結果のキャッシュによる性能への影響は大きく，1 秒程度の短時間の LRU キャッシュで unique，shared 両方の操作パターンでメタデータ操作の性能向上が得られることが示された．

4.5 評価環境 2

第 4.6 章既存研究との比較評価，第 4.7 章オブジェクトストレージと組み合わせた際の性能調査，に使用した評価環境について述べる．

表 4.3 にメタデータサーバを実行した計算機のスペック，表 4.4 に FUSE とクライアントベンチマークを実行した計算機のスペック，表 4.5 にオブジェクトストレージサーバを実行した計算機のスペックについてそれぞれ示す．オブジェクトストレージサーバについての詳細は第 4.7 章で述べる．また，これらの計算機を含む評価環境全体の概略図を図 4.4 に示す．

図 4.4 に示されるように，メタデータサーバノードは 1 ノードから最大 5 ノードまでノード数を変化させて評価した．クライアントノードは最大で 8 ノードを用いて，プロセス数を 1 から 128 まで変化させて評価した．オブジェクトストレージサーバノードは 10 ノードに固定した．これらの計算機はすべて Infiniband ネットワークで繋がっているが，通信は IP over Infiniband 上で行った．

メタデータパフォーマンス測定には mdtest を用いた．mdtest の 1 プロセス毎に 5,000 個のファイルとディレクトリの作成を行った．クライアントプロセスの数は最大で 128 プロセスなので，最大で合計 640,000 個のファイル，ディレクトリがそれぞれ作成される．

4.6 既存研究 IndexFS との比較評価

提案メタデータ管理手法の実装である PPMDS と, 従来研究の 1 つである IndexFS [23] について，同じ評価環境とベンチマークをを用いて評価を行った．IndexFS は分散ファ

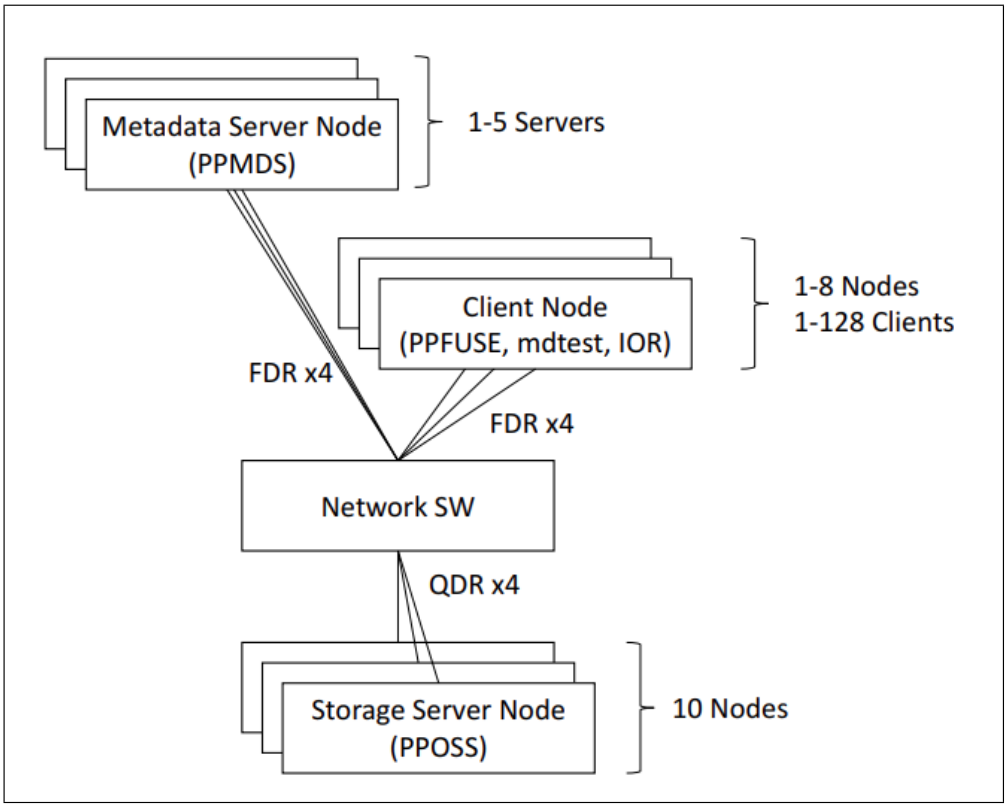
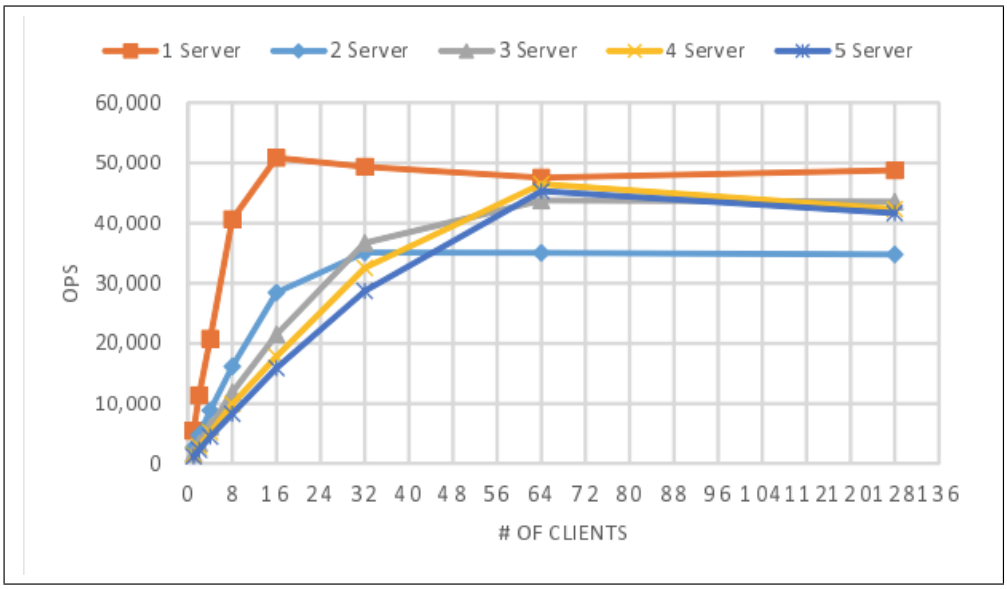
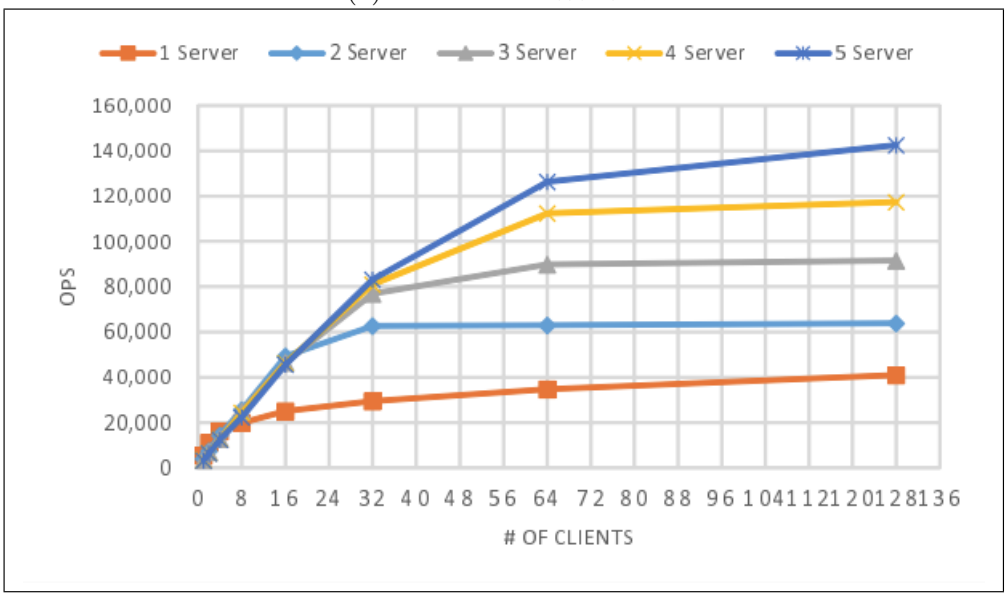


図 4.4　評価環境 2: 評価環境全体図

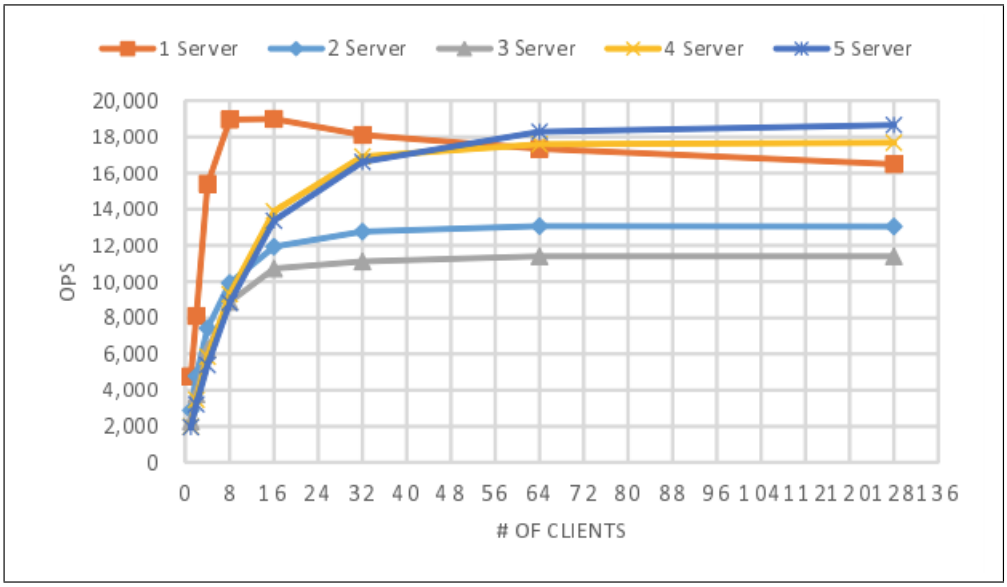


(a) ディレクトリ作成性能

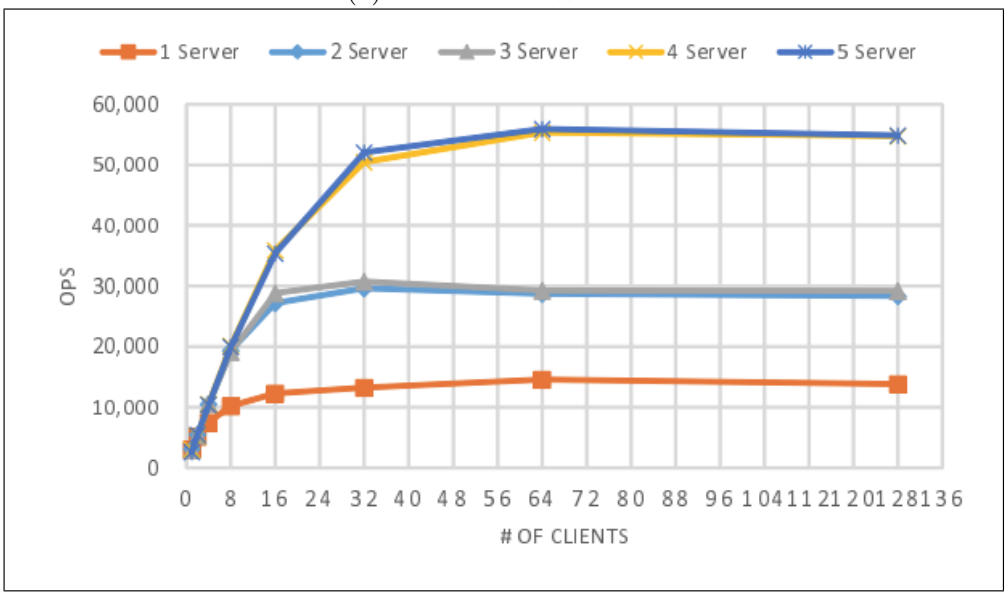


(b) ファイル作成性能

図 4.5　PPMDS の単一ディレクトリに対するメタデータ操作性能



(a) ディレクトリ作成性能



(b) ファイル作成性能

図 4.6　IndexFS の単一ディレクトリに対するメタデータ操作性能

表 4.4 評価環境 2: クライアントノード

CPU	Xeon E5-2665 2.40 GHz
Sockets/node	2
Cores/socket	8
Num of Nodes	8
Memory	64GB
Network	Mellanox Technologies MT27500 4x FDR (56Gbps)

表 4.5 評価環境 2: オブジェクトストレージサーバノード

CPU	Xeon E5620 2.40 GHz
Sockets/node	2
Cores/socket	4 + Hyper-Threading
Num of Nodes	10
Memory	24GB
Network	Mellanox Technologies MT26428 4x QDR (32Gbps)

イルシステムに汎用的なスケーラブルメタデータ管理機能を提供するのを目的に開発されたシステムで，GIGA+ [3] という Hash Partitioning アルゴリズムの一種を実装して，ラージディレクトリのエントリを複数メタデータサーバに分散管理する機能を持っている．IndexFS はメタデータを Key Value store の一種である LevelDB [37] 上で管理している．これは，log-structured merge-tree (LSM-tree) [38] の実装の一つである．Key はファイルの親ディレクトリの inode 番号とファイル名のハッシュ値のペア，Value は実ファイル名と，inode が格納されている．文献 [23] によると IndexFS は，メタデータサーバの台数が 128 ノードまでは性能がリニアにスケールし，メタデータアクセスが重要なワークロードに最適化されたシステムで PPMDS と共通の点が多い．これらの理由から IndexFS と性能を比較する．

PPMDS の評価結果を図 4.5，IndexFS の評価結果を図 4.6 にそれぞれ示す．

図 4.5 のうち PPMDS のディレクトリ作成性能を示す図 4.5 (a) について，メタデータサーバのノード数が 1 ノードで，クライアント数が 16 プロセスの場合に 50,840.7 ops/s と最も高い性能を示している．メタデータサーバが 2 ノード以上の場合は，クライアント数が少ない時はメタデータサーバが少ない方が高い性能を示し，クライアント数が多い時はメタデータサーバが多い方が高い性能を示した．1 ノードの場合に最も高い性能を示したのは，ディレクトリの作成時はメタデータサーバがディレクトリのエントリを分散させるサーバのリストを，関係するすべてのサーバに対して作成するためである．

一方で，PPMDS のファイル作成性能を示す図 4.5（b）においては，ノード数の増加に伴って性能がスケールしており，メタデータサーバのノード数が 5 ノードでクライアントが 128 プロセスだった場合が最も性能が高く，142,564.4 ops/s となった．PPMDS

は，ディレクトリ作成性能に比べてファイル作成性能が高いが，これは上記で述べたディレクトリ作成時のサーバリスト作成をファイル作成時は行う必要が無いためである．

図 4.6 のうち，IndexFS のディレクトリ作成性能をすめす図 4.6 について，こちらも PPMDS と同様に，メタデータサーバのノード数が 1 ノードで，クライアント数が 16 プロセスの際に 18,987.9 ops/s と最も高い性能を示している．これはメタデータサーバのノード数が 2 ノード以上になった場合に，メタデータサーバ感の通信が発生するためと考えられる．

IndexFS のファイル作成性能を示す図 4.6（b）について，メタデータサーバのノード数が 2 ノードと 3 ノード，4 ノードと 5 ノードの場合において性能差はほぼ無い．これは，IndexFS が使用している GIGA+ [3] Hash partitioning アルゴリズムが，2 のべき乗単位でディレクトリ分割を行っている影響と考えられる．

PPMDS と IndexFS のファイル作成性能を比較すると，メタデータサーバのノード数が 5 ノードでクライアントが 128 プロセスの場合に，PPMDS は IndexFS の 2.60 倍の性能となっている．PPMDS においてファイル作成時には，親ディレクトリの分散先サーバリストの読み込みと格納先サーバに対してメタデータの格納に伴う書き込みが行われるが，これらの読み込みと書き込みは多数のファイル作成リクエストが発行されても衝突しないため，処理に伴う同期待ちが発生しないため，IndexFS に比べて高い性能を示したと考えられる．

4.7　オブジェクトストレージと組み合わせた際のメタデータ操作性能

PPMDS はファイルシステムのメタデータのうち，ファイルシステムの階層的ネームスペースと inode メタデータを扱うが，PPMDS 単体ではファイルの読み込み，書き込み機能はサポートしていない．実際に分散ファイルシステムとしての全機能を提供するには，ファイルの実データを保存するためにオブジェクトストレージシステム等と組み合わせることを想定している．鷹津らは文献 [39] で，分散オブジェクトストレージサーバ，*PPOSS* を提案している．PPOSS は，バックエンドに OpenNVM [40] を用いた Non-volatile Memory ストレージデバイスに最適化されたローカルオブジェクトストレージを用いている．PPMDS と PPOSS を組み合わせて実際にファイルの読み書きが可能な分散ファイルシステムを構成した場合，メタデータ操作パフォーマンスがどうなるのか評価を行った．

PPMDS と PPOSS を組み合わせる際に，以下に示す最適化を行っている．評価では最適化によるパフォーマンスの変化を分析するため，最適化の段階に分けて評価した．

PPMDS（オブジェクトストレージ無し）
: ベースラインとして，PPMDS のみの評価．

+PPOSS

PPMDS と PPOSS を組み合わせた場合の評価．ファイル作成時，PPMDS は PPOSS にファイルの実データを格納するためのオブジェクト作成リクエストを発行する．クライアントは，PPMDS と PPOSS 両方のファイル作成処理が完了するまで待つ．

+Bulk Creation(N=64)

並列ファイル作成リクエストに最適化するため，PPMDS は PPOSS に同時に N 個のオブジェクトを作成するようリクエストし，作成したオブジェクトをプールする．本評価では同時に 64 個のオブジェクトを作成する．PPMDS はクライアントからファイル作成リクエストがあると，オブジェクトプールから未使用のオブジェクトを割り当てることで，PPOSS との通信なしにファイル作成ができる．Bulk Creation 最適化で，PPMDS から PPOSS に発行されるオブジェクト作成リクエストは N 回に 1 回に削減される．

+Object Prefetching

Bulk Creation は PPMDS から PPOSS へのオブジェクト作成リクエストを 1/N 回に削減するが，オブジェクトプールが空の場合，クライアントのファイル作成リクエストは新たな N オブジェクト作成のために待機してしまう．この問題に対処するため，Object Prefetching 最適化は，PPMDS から PPOSS へのオブジェクトの Bulk Creation リクエストを別スレッドで実行し，これらの通信の影響を隠す．

評価結果を図 4.7，図 4.8，図 4.9，図 4.10 に示す．各グラフの横軸はクライアントプロセス数，縦軸はファイル作成操作の K ops/s を示す．

PPMDS のみの図 4.7 と PPMDS と PPOSS を組み合わせた図 4.8 を比較すると，ピークパフォーマンスが得られたメタデータサーバが 5 ノードでクライアント数が 128 プロセスの場合に，PPOSS を組み込むと性能は PPMDS 単体性能の 62.8 % となった．クライアント数が少ない 16 プロセスの場合では，11.5 % に低下した．これは，ファイル作成毎に PPOSS にオブジェクト作成リクエストを発行しているため，その間クライアントが待機されているためと考えられる．

次に，Bulk Creation 最適化を実施した場合の性能を図 4.9 に示す．この評価では，PPMDS が PPOSS にオブジェクト作成リクエストを発行する回数は 64 回に 1 回となり，一度に 64 個のオブジェクトを作成する．メタデータサーバが 5 ノードでクライアント数が 128 プロセスの場合にピークパフォーマンスが得られ，PPMDS 単体の場合に比べて 83.7 % の性能となった．クライアント数が少ない 16 プロセスの場合は 65.2 % となり，最適化無しと比べて前者は 1.34 倍，後者は 5.69 倍となった．Bulk Creation によって通信回数が削減され，メタデータ操作パフォーマンスが向上したと考えられる．

最後に，Object Prefetching 最適化を行った場合のファイル作成性能を図 4.10 に示す．メタデータサーバが 5 ノードで，クライアント数が 128 プロセスの場合に 138,577.7 ops/s のピークパフォーマンスが得られ，これは PPMDS 単体の場合に比べて 97.2 % の

性能となった.

本評価により，提案メタデータ管理手法を用いて実際に分散ファイルシステムを構成する場合，オブジェクトストレージと組み合わせる際に十分な最適化を実施すれば，十分な性能が得られることが示された.

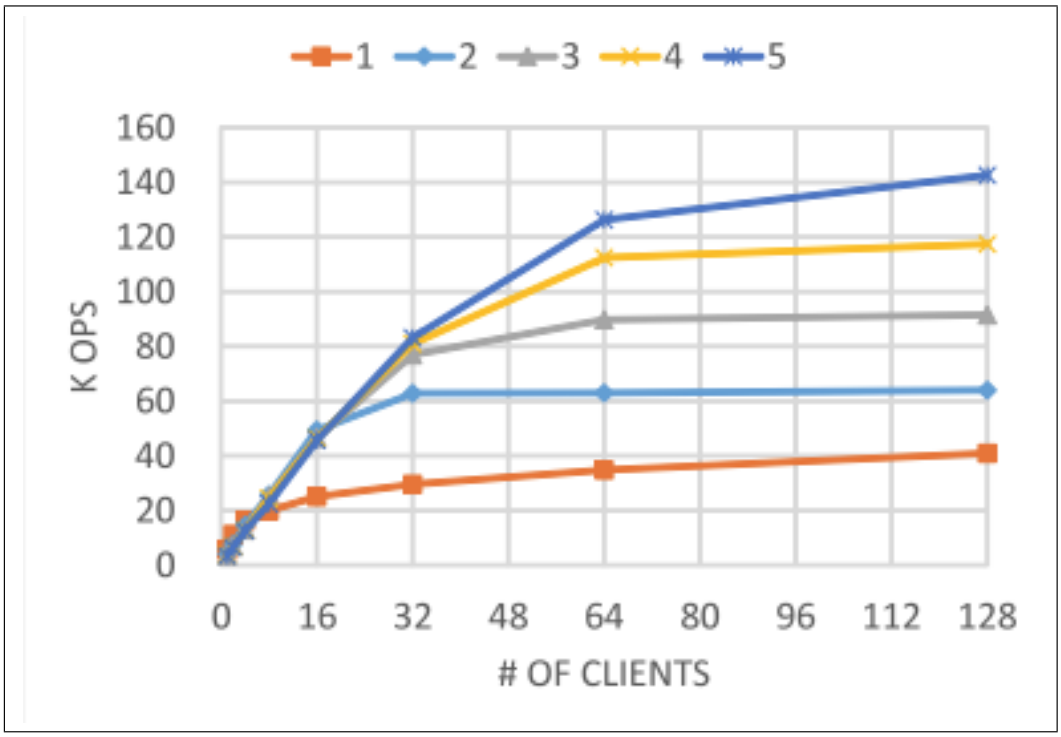


図 4.7　ファイル作成性能: PPMDS（オブジェクトストレージ無し）

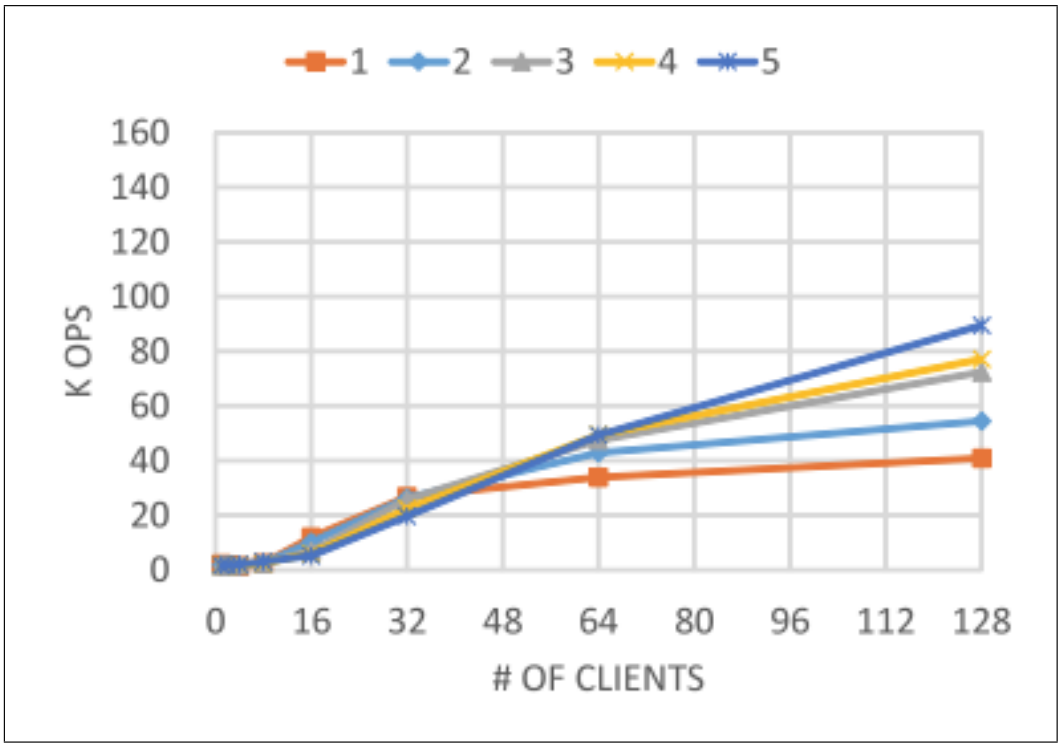


図 4.8　ファイル作成性能: PPMDS + PPOSS

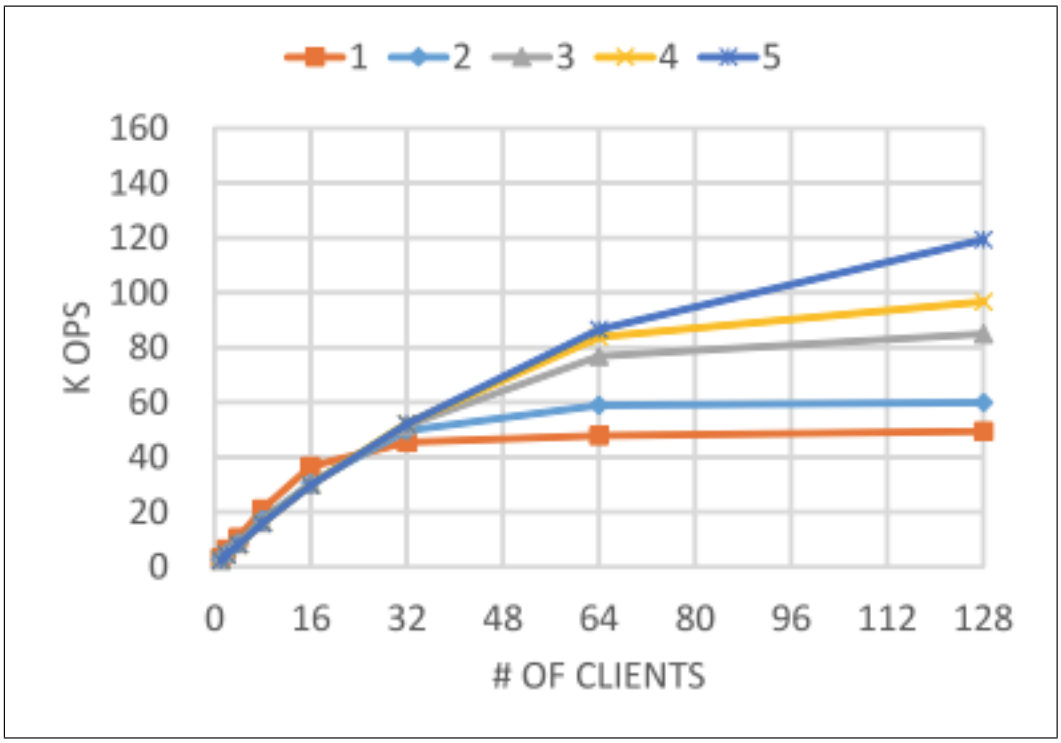


図 4.9　ファイル作成性能: + Bulk Creation (N=64)

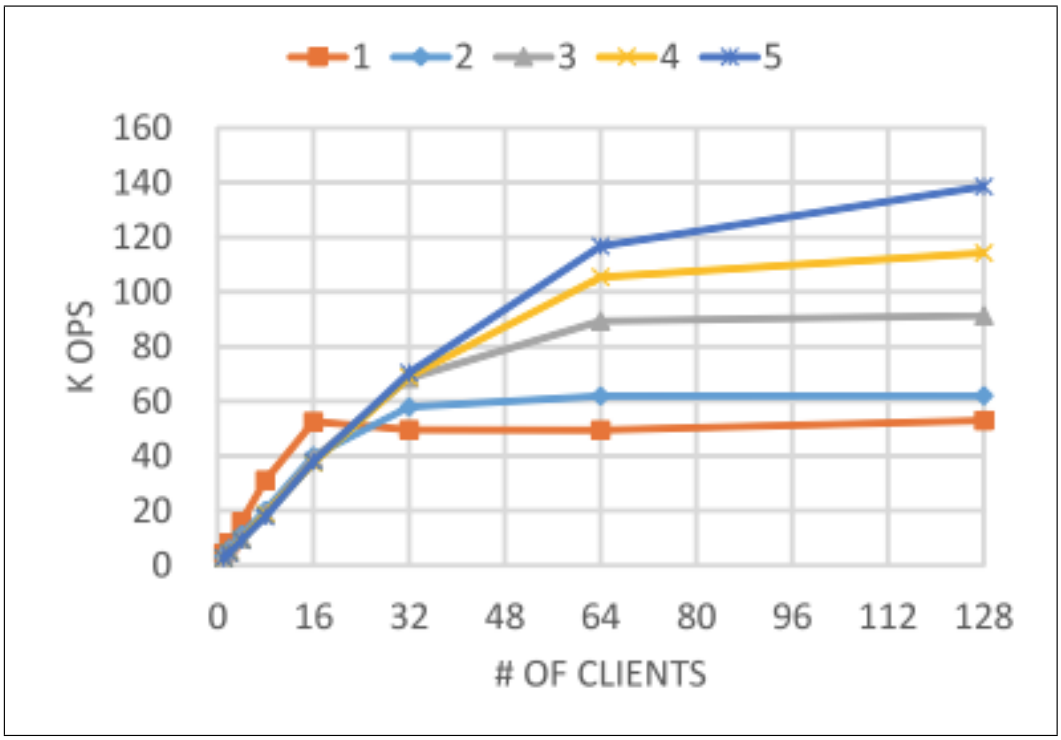


図 4.10　ファイル作成性能: + Object Prefetching

第 5 章

結論

5.1 本論文のまとめ

近年，産業と科学両方の分野において分散ファイルシステムで扱うファイルの数が増大し，ファイル作成やファイルオープンのようなメタデータアクセスが，深刻な性能ボトルネックとなっている．この問題に対処するために，本論文はスケーラブルなメタデータサーバシステム，PPMDS を提案した．

スケーラブルな分散メタデータデザインには，主要な 2 つの問題がある．1 つは，どのようにして階層的ネームスペースを並列に効率的に管理するかで，もう 1 つはどのようにして複数サーバに跨ってファイルシステム操作をサポートするかである．階層的ネームスペースのために，PPMDS は inode エントリーを親 inode 番号とエントリ名のペアを key として，複数サーバ間で扱った．全てのファイルシステム操作をサポートするため，PPMDS は key-value ストア向けノンブロッキング分散トランザクションを利用した．

PPMDS はメタデータ操作性能をサーバサイドトランザクション処理と open-for-read スキームのマルチリーダ化により改善した．これらの手法により，リモート関数呼び出しの数と不必要なブロッキング時間の削減を行った．

性能評価により，PPMDS の実装は，単一ディレクトリへの 142,000 ops/sec のファイル作成性能を 5 メタデータサーバで達成した．サーバ 1 台の場合と比べて約 3.55 倍の性能向上が得られた．

5.2 今後の課題

今後は，代表的な HPC アプリケーションを用いて，より大規模な環境での性能評価を行いたいと考えている．

謝辞

本論文の執筆にあたり，ご指導いただきました筑波大学計算科学研究センター教授　建部修見先生に深く感謝の意を表します．

また，お忙しい中本論文の審査をお引き受けいただきました筑波大学計算科学研究センター教授　天笠俊之先生，同教授　額田彰先生，筑波大学システム情報系准教授　阿部洋丈先生，慶應義塾大学環境情報学部准教授　川島英之先生に深く感謝いたします．

また，川島英之先生，筑波大学学術情報メディアセンター准教授　大山恵弘先生には本研究にあたり多くの助言とご指導をいただきました．

最後に，筑波大学 HPCS 研究室の先生方，学生の皆様には，長年に渡り様々な面でご支援をいただきました．特に鷹津冬将博士には，共同研究者として様々な御協力をいただきました．深く感謝の意を表します．

参考文献

[1] Peter J. Braam. Lustre. http://www.lustre.org/.

[2] Philip Schwan. Lustre : Building a File System for 1,000-node Clusters. In *Proceedings of the 2003 Ottawa Linux Symposium*, pp. 380–386, Ottawa, Ontario, Canada, July 2003.

[3] Swapnil Patil and Garth Gibson. Scale and Concurrency of GIGA+: File System Directories with Millions of Files. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, p. 13, USA, February 2011. USENIX Association.

[4] Jim Gray. Notes on Data Base Operating Systems. In *Operating Systems, An Advanced Course*, pp. 393–481, 1978.

[5] George Samaras, Kathryn Britton, Andrew Citron, and C. Mohan. Two-Phase Commit Optimizations and Tradeoffs in the Commercial Environment. In *ICDE*, pp. 520–529, 1993.

[6] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer. Software Transactional Memory for Dynamic-Sized Data Structures. In *Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing*, pp. 92–101, New York, NY, USA, 2003. Association for Computing Machinery.

[7] Ext3 Filesystem. https://www.kernel.org/doc/html/latest/filesystems/ext3.html.

[8] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pp. 29–43, New York, NY, USA, October 2003. Association for Computing Machinery.

[9] Marshall Kirk McKusick and Sean Quinlan. Case Study: GFS: Evolution on Fast-forward. *ACM Queue: Tomorrow's Computing Today*, Vol. 7, No. 7, p. 10, August 2009.

[10] The Apache Software Foundation. Apache Hadoop. https://hadoop.apache.org/.

[11] Osamu Tatebe, Kohei Hiraga, and Noriyuki Soda. Gfarm Grid File System. *New Generation Computing*, Vol. 28, No. 3, pp. 257–275, 2010.

[12] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason

Small, Jim Zelenka, and Bin Zhou. Scalable performance of the Panasas parallel file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, USA, February 2008. USENIX Association.

[13] Philip H. Carns, Walter B Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A Parallel File System for Linux Clusters. In *Proc. of the Extreme Linux Track: 4th Annual Linux Showcase and Conference*, pp. 391–430. MIT Press, October 2000.

[14] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller. Dynamic Metadata Management for Petabyte-Scale File Systems. In *SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, November 2004.

[15] Sage Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI '06)*, pp. 307–320, November 2006.

[16] Salman Niazi, Mahmoud Ismail, Seif Haridi, Jim Dowling, Steffen Grohsschmiedt, and Mikael Ronström. HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pp. 89–104, 2017.

[17] Henry Newman. What is HPCS and How Does Impact I/O. Presentation at Lustre Scalability Workshop, May 2009.

[18] Andrew Fikes. Storage Architecture and Challenges. Presentation at the 2010 Google Faculty Summit, July 2010.

[19] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the Conference on File and Storage Technologies (FAST'02)*, pp. 231–244, January 2002.

[20] Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. Extendible hashing—a fast access method for dynamic files. *ACM Transactions on Database Systems*, Vol. 4, No. 3, pp. 315–344, September 1979.

[21] Shuangyang Yang, Walter B Ligon III, and Elaine C Quarles. Scalable Distributed Directory Implementation on Orange File System. In *SNAPI '11 Proceedings of the 2011 International Workshop on Storage Network Architecture and Parallel I/Os*, 2011.

[22] Jing Xing, Jin Xiong, Ninghui Sun, and Jie Ma. Adaptive and scalable metadata management to support a trillion files. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, Vol. 1, pp. 1–11, Portland, OR, USA, November 2009.

[23] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion. In *SC*

'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 237–248, November 2014.

[24] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 12, No. 10, pp. 1094–1104, October 2001.

[25] Brent Welch and Geoffrey Noer. Optimizing a hybrid SSD/HDD HPC storage system based on file size distributions. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–12, May 2013.

[26] 熊崎宏樹, 津邑公暁, 齋藤彰一, 松尾啓志. 分散キーバリューストアを対象としたオブストラクションフリートランザクションの実装. 情報処理学会研究報告, Vol. 2011-OS-118, No. 16, pp. 1–7, July 2011.

[27] 熊崎宏樹. 分散キーバリューストア上でのトランザクションの実装. Master Thesis, 2011.

[28] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *23rd International Conference on Distributed Computing Systems, 2003. Proceedings.*, pp. 522–529, May 2003.

[29] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pp. 289–300, New York, NY, USA, May 1993. Association for Computing Machinery.

[30] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, Vol. 13, No. 1, pp. 124–149, January 1991.

[31] Byung-Jae Kwak, Nah-Oak Song, and L. E. Miller. Performance analysis of exponential backoff. *IEEE/ACM Transactions on Networking*, Vol. 13, No. 2, pp. 343–355, April 2005.

[32] Memcached — a distributed memory object caching system. https://memcached.org/.

[33] Nikolaus Rath. The reference implementation of the Linux FUSE (Filesystem in Userspace) interface. https://github.com/libfuse/libfuse.

[34] Mikio Hirabayashi. Kyoto Cabinet: A straightforward implementation of DBM. https://dbmx.net/kyotocabinet/index.html.

[35] Sadayuki Furuhashi. MessagePack. https://msgpack.org/.

[36] HPC IO Benchmark Repository. https://github.com/hpc/ior.

[37] LevelDB is a fast key-value storage library written at Google that provides an ordered mapping from string keys to string values. https://github.com/google/leveldb.

[38] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, Vol. 33, No. 4, pp. 351–385,

June 1996.

[39] Fuyumasa Takatsu, Kohei Hiraga, and Osamu Tatebe. PPFS: A Scale-out Distributed File System for Post-petascale Systems. *Journal of Information Processing*, Vol. 25, pp. 438–447, June 2017.

[40] NVM user-space Primitives API library repository. https://github.com/opennvm/nvm-primitives, 2014.

付録 A

業績一覧

本論文の研究内容の一部は，以下の論文で発表済みである．

査読付き論文誌

1. Kohei Hiraga, Osamu Tatebe, and Hideyuki Kawashima. Scalable Distributed Metadata Server Based on Nonblocking Transactions. *Journal of Universal Computer Science*, Vol. 26, No. 1, pp. 89–106, 2020
2. Fuyumasa Takatsu, Kohei Hiraga, and Osamu Tatebe. PPFS: A Scale-out Distributed File System for Post-petascale Systems. *Journal of Information Processing*, Vol. 25, pp. 438–447, June 2017.
3. Fuyumasa Takatsu, Kohei Hiraga, and Osamu Tatebe. Design of Object Storage Using OpenNVM for High-performance Distributed File System. *Journal of Information Processing*, Vol. 24, No. 5, pp. 824–833, 2016
4. Osamu Tatebe, Kohei Hiraga, and Noriyuki Soda. Gfarm Grid File System. *New Generation Computing*, Vol. 28, No. 3, pp. 257–275, 2010.

査読付き国際会議（口頭発表）

1. Kohei Hiraga, Osamu Tatebe, and Hideyuki Kawashima. PPMDS: A Distributed Metadata Server based on Nonblocking Transactions. In *Proceedings of the Second International Workshop on Data Science Engineering and Its Applications*, pp. 202–208, Valencia, Spain, October 2018. IEEE **(Best Paper)**

査読付き国際会議（ポスター発表）

1. Kohei Hiraga and Osamu Tatebe. PPMDS: A Distributed Metadata Management System for Parallel File Systems based on Software Transactional Memory. PRAGMA 22 Workshop, Poster, April 2012

国内シンポジウム・研究会（口頭発表）

1. 鷹津冬将, 平賀弘平, 建部修見. 高性能分散ファイルシステムのための分散メタデータサーバ PPMDS の評価. 情報処理学会第 155 回 HPC 研究会報告 (HPC155), Vol. 2016-HPC-155, No. 1, pp. 1–11, August 2016
2. 平賀弘平, 建部修見. ノンブロッキングトランザクションに基づく分散ファイルシステムのための分散メタデータサーバの設計と実装. 情報処理学会第 135 回 HPC 研究会報告 (HPC135), Vol. 2012-HPC-135, No. 28, pp. 1–9, July 2012