博士論文

階層ストレージにおけるアプリケーション透過なファイル最適配置手法に関する研究

A Study on Application-Transparent Optimal File Placement in Hierarchical Storage

2020年8月

東京大学大学院 情報理工学系研究科
システム情報学専攻

松澤 敬一

論文要旨

IT システムが扱うデータ量は年々増加傾向にある。そのため、ストレージにはより大容量・高性能が求められている。従来主要な記憶メディアとしては Hard Disk Drive (HDD) が用いられてきたが、2000 年代後半には NAND Flash を用いた Solid State Drive (SSD) が市場に流通し始め、性能の高さと HDD との互換性の高さからその後急速に普及した。2010 年代後半には、高速なインターフェース NVM Express (NVMe) が登場し、メディアのアクセス性能はさらに向上した。2019 年には不揮発性メモリが広く入手可能となり、今後の普及が見込まれる。このように現在は記憶メディアとして、性能・容量・コストが異なる複数の選択肢がある。そこで、データをアクセス特性に応じて適正な記憶メディアに配置し、IT システム全体でデータ格納における性能・容量・コストのバランスの適正化を図る階層ストレージの必要性が高まっている。

一方、ソフトウェアにおいては、アプリケーションからストレージを利用する方法として、ファイルを基本とするインターフェースが長らく利用されており、今後も継続して利用されるとみられている。そのため、階層ストレージを多様なアプリケーションに適用するには、これらのインターフェースを維持することが必須である。

そこで本研究では、高性能な記憶メディアを活用したアプリケーションの性能向上を目的とし、実用的な階層ストレージの実現に取り組んだ。そのような階層ストレージの実現においては、既存のアプリケーションの変更を要せず透過的に適用可能な方式であることと、データを最適なメディアに最適な順で配置することが解決すべき課題となる。本研究では、既存のアプリケーションで広く利用されているファイル及び POSIX のインターフェースに着目することで、これらの課題を解決した。POSIX インターフェースを維持することで、アプリケーションを変更することなく適用可能であり、またアプリケーションによるファイル単位のアクセスパターンに着目することでデータの配置先やその順番を最適化した。

本論文では、三つの主題について議論する。一つ目の主題は、古い機器を新しい機器に移行するために、データを機器間で複製することである。このような機器間のデータ複製は、今後新たな記憶デバイスを備えた機器が登場する際にも、既存のデータやそのデータを利用するアプリケーションを継続的に利用するために必要な技術である。本項では、新旧ファイルサーバの移行を対象とし、既存データを格納する移行元サーバから、新規に設置する移行先サーバにデータを複製する手法について論ずる。その際、稼働中の移行元サーバや、そのファイルを参照する外部のクライアントに対し、変更や長時間の停止時間を要さず、格納されたデータを複製する手法を提案する。提案手法では、ファイル単位の Post-Copy 方式で複製を行うことで複製に伴う待ち時間を最小化する。これは、データの複製開始前にクライアントが接続先のファイルサーバを切り替えておき、クライアントが新ファイルサーバ上で未複製のファイルにアクセスしようとすると、その時点で移行元サーバからアクセス要求への応答に必要なデータだけ複製するものである。一般的なデータセットにおいては、この複製による待ち時間は数秒程度で収まる。一般的なファイルシステムにおけるタイムアウト時間の 30〜120 秒未満であることから、広範囲のデータセットのファイルサーバに適用できることを確認した。また、Post-Copy 方式を実現する機構は全て新ファイルサーバで備えることで、既存の移行

元サーバやアプリケーションに変更を要しない手法であることを実現した。

二つ目の主題は、計算機内にある複数の記憶メディア間でデータをアクセス状況に応じて再配置することで、ストレージのアクセス応答時間を短縮する階層ストレージである。階層ストレージにおいて平均入出力性能を高める一般的な手法として、高速な記憶メディアを低速な記憶メディアに対するキャッシュとみなし、今後のアクセスが予測されるデータをプリフェッチする手法がある。しかし、近年のクラウドコンピューティング環境で用いられる Infrastructure-as-a-Service (IaaS) のように、仮想マシンが多数動作し、それぞれが異なるパターンでデータアクセスを行う実行環境においては、ホスト OS と仮想マシン間で多層に重なったストレージのソフトウェア処理によって、仮想マシン内のアプリケーションが生じさせるアクセスの局所性をホスト OS において観測できなくなる。その結果、アクセスの予測精度が低下し、プリフェッチの効果が減少する。提案手法では、これら仮想マシン内のアプリケーションは、仮想マシン内のファイルに対してアクセスの空間的な局所性を持つことに着目し、仮想マシン内のファイルと物理ディスク上の配置を対応付けるレイアウト情報を仮想マシンからホスト OS に送信する。これにより、ホスト OS 内でプリフェッチ対象のデータ領域を選択する際、物理ディスク上で不連続であっても、仮想マシン内のファイル上で連続する領域を、同時にプリフェッチすることで、アクセスの局所性に基づくプリフェッチの効果改善を図る。提案手法では、TPCx-V ベンチマークにおいて、レイアウトを認識しない場合にくらべ、17.1% のトランザクション処理性能の改善が見られた。

三つ目の主題は、近年登場した不揮発性メモリにデータを格納する際の、性能向上のためのソフトウェアインターフェースの検討である。現在のストレージに対するソフトウェアのインターフェースでは、OS が介在することで I/O スケジューリングなど記憶メディアへの I/O を軽減するためのソフトウェア処理と、複数プロセス間でデータやディレクトリ構造の一貫性を保つための排他処理に伴うソフトウェアの性能オーバヘッドが生じる。前者は従来の低速な記憶メディアに対しては有効であるが、不揮発性メモリのような高性能メディアに対しては、むしろ CPU の実行時間を要しオーバヘッドとなる。また、後者は実用アプリケーションでは稼働中にデータを他アプリケーションと共有することは珍しく、やはり排他処理が実質的に不要なオーバヘッド要因となっている。提案手法では、標準 C ライブラリと連携してユーザー空間のライブラリとして動作し、アプリケーションにプロセス固有のディレクトリを提供するユーザー空間ファイルシステム方式を提案する。本ファイルシステムは、プロセスのメモリ空間上にマップした不揮発性メモリ上にファイルデータを格納し、CPU の Load/Store 命令で読み書きするため、ファイルアクセスにおける OS の関与を不要とする。また、本手法では、標準 C ライブラリの備えるファイルアクセスインターフェースを維持することで、アプリケーションの変更なく適用可能となる。ファイルアクセスを行う filebench ベンチマークにおいて、提案手法は XFS ファイルシステムに対し 6.48 倍、先行研究である不揮発性メモリ向けファイルシステムの 1.47 倍の性能を出せることを確認し、提案手法の性能優位性を確認した。

本研究の結果、既存のアプリケーションの変更を要せず適用可能であることと、データの配置先を最適化できるという二つの課題を解決し、アプリケーションの性能を向上させる実用的な階層ストレージを実現できる見込みを得た。

Abstract

In recent years, amount of global data is growing tremendously. Various storage media are developed and produced to keep up with the storage demands. In early 2000s, hard disk drives (HDDs) were used as major storage media. After that, storage media using flash-memory were produced and solid-state drives (SSDs) were rapidly positioned as main storage media due to its high performance. In 2019, non-volatile memory (NVM) modules that applications can access via memory interface were produced. These various media have unique characteristics. Hence, hierarchical storage management (HSM) that moves data among media based on data access patterns to optimize its total performance, capacity, cost, and etc. was studied.

On the other hand, from perspective of software interface of storage, even recent applications for smartphone, enterprise system, and cloud environment continue to use file and POSIX interfaces for storage accesses. Even though the more sophisticated interfaces such as key-value store and object based storage are introduced, the middleware implementing them uses POSIX interfaces to access underneath media. That is, the file and POSIX interfaces will keep used now and in the future, and it is expected for HSM to support the current software interface.

The goal of my research is to establish a practical HSM to improve applications' performance. I addressed two challenges to achieve a practical HSM in this paper. First, the HSM should be transparently applicable for existing applications. The term "transparent" means that the technique is adopted to existing applications without any modification or rebuild of application codes and does not cause any erroneous behaviors such as time out error or severe performance degradation. Second, data should be placed optimally on storage media to provide better performance.

I focused the file and POSIX interface and my proposal leverages that interface and determines optimal data placement based on access patterns of files to solve two challenges.

This paper covers three topics. First, to use newly developed media, existing data in an old computer must be migrated to a newly deployed computer with such media. In this topic, I focused a file server migration scheme without long downtime and response time. Client computers and applications running there should keep running during data replication between servers. Existing solution based on pre-copying data requires long suspend period for clients before the completion of replication to confirm all files in both servers are synchronized. My proposal utilizes post-copy based file replication and it replicates file data on demand of clients accesses to the new file server. It achieves little downtime (¡30s) for switching server and does not cause long downtime for on demand replication.

Second, multiple virtual machines and user-developed applications run on a single computer in cloud era. In this topic, I propose a data prefetching scheme for HSM between SSD and HDD. Though HSM works effectively when it can predict future data access, various applications and a deep storage software

stack weaken temporal and spatial access locality and make I/O pattern prediction less accurate. I notice that access locality in consecutive file blocks in a guest OS is maintained even they are stored in scattered locations. Hence in my proposal, an agent program installed on the virtual machine obtains mapping between file data block and its location on a virtual volumes. It transfers this mapping and other information to a host OS. The host OS prefetches consecutive file blocks from HDDs at once and expects the future I/O requests will be issues to SSDs.

Third, NVMs for memory interfaces provide different interfaces and performance characteristics from conventional storages. To utilize its performance, applications should be modified to adopt its interface, and current file semantics is too strict. In my proposal, a user-space file system library is attached to a target process and it provides isolated directory for each process. Accesses to the directory are handled by the library, and the library reads and writes NVM without any context switch and system calls. Hence, its performance is better than conventional file system. Furthermore, its interfaces inherit POSIX and do not require existing application source code modification or re-build process.

Using these research's contributions, an HSM that migrates and place data among computers and storage media without modifying existing applications. Hence, the HSM with new storage media will provide optimal data access to improve better application performance.

目次

第 1 章

序論

1.1 背景

本節では、本論文の研究対象であるストレージを取り巻く近年の動向について整理する。

1.1.1 記憶メディアの動向

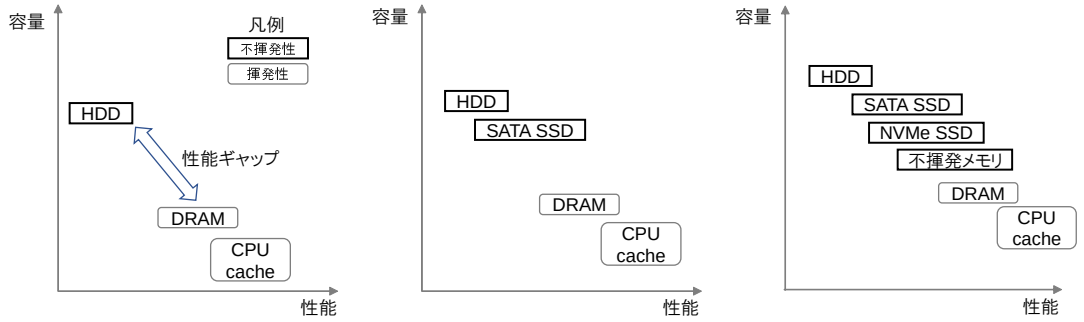
IT システムが扱うデータ量は年々増加している [1]。その大容量データを格納し処理するために、より高い性能と大きな容量を持つ記憶メディアの研究開発が行われてきた。図 1.1 は、主要な記憶メディアをアクセス性能と容量で両対数グラフにプロットした図を年代別に並べたものである [2, 3, 4]。

普及する記憶メディアは、性能と容量のバランスから、グラフ上で直線上に並ぶ傾向がある [2]。技術開発により既存メディアの性能・容量は向上するため、その直線は時間経過とともに右上に徐々にスライドしていく。

2000 年頃 (図 1.1(a)) は、CPU の動作周波数の増加に伴い、揮発性の記憶メディアである CPU のキャッシュメモリや DRAM の性能が急速に向上していた。その間、長らく不揮発性の記憶メディアとして磁気ディスクを用いた Hard Disk Drive (HDD) が用いられてきた。HDD は大容量である反面、入出力時にディスクを回転させる機械的な動作を伴うためアクセス性能、特に応答時間の短縮が難しく、揮発性と不揮発性の記憶メディアにおける性能ギャップは徐々に拡大していた。

2000 年代後半から 2010 年代 (図 1.1(b)) にかけては、CPU の動作周波数の上昇が緩やかになり、CPU のキャッシュメモリや DRAM の性能向上も鈍化した。一方でこの間の不揮発記憶メディアの進展は速く、それまで基礎研究が行われていた NAND 型フラッシュメモリを用いた Solid State Drive (SSD) の技術が成熟し、信頼性・価格・性能が実用段階に入った。SSD は機械的な駆動部を持たないために、HDD に比べて入出力の応答時間がはるかに短く、高い入出力性能が出るようになった。登場当初の SSD は HDD に比べ記憶容量が小さく高価なため、その利用は限定的であったが、HDD と互換性のある Serial ATA(SATA) 接続インターフェースを利用したことから徐々に HDD との置き換えが進み、2020 年には HDD と SSD の出荷台数が入れ替わるとみられている [5]。ただし HDD は記憶容量当たり価格では依然 SSD より優れるため、SSD によって完全に置き換えられるわけではなく、アーカイブ用途で継続使用されると見られている。その結果、図に示す通りこの年代には HDD と DRAM の間に、SATA 接続の SSD が主要記憶メディアとして配置された状況にある。

2010 年代後半から 2020 年 (図 1.1(c)) にかけて、SSD は技術開発によりさらに性能・容量を改善させていく。その結果、既存の HDD 向けのインターフェースがボトルネックとなって SSD の高性能を発揮できないため、より軽量なインターフェースである NVM Express (NVMe) が開発され、対応する SSD と合わ



(a) 2000 年頃　(b) 2010 年頃　(c) 2020 年

図 1.1: 主要記憶メディアのアクセス性能と容量の関係

せて市場を拡大中である [6]。この NVMe 接続の SSD は SATA 接続の SSD に比べるとさらに短い応答時間を達成しているが、まだ CPU キャッシュや DRAM と比べ入出力の性能は 3 桁以上の差がある。そこで、さらなる高性能な不揮発性の記憶メディアとして、不揮発性メモリ (Non-Volatile Memory, NVM) の研究・実用化が進んだ。不揮発性メモリは、様々な方式の研究開発が進行しているものの [7]、長らく研究サンプルでしか利用できず性能・容量・価格も実用的なものではなかった。しかし 2019 年に Intel 社が Optane DCPMM [8] を製品化し市場投入したことで、市場で容易に入手・利用可能となった。Optane DCPMM は DRAM と SSD の中間にあたる性能・容量・価格となっており、従来の揮発性と不揮発性の間を埋める記憶メディアとして注目されている。

図 1.1 において時期ごとの状況を比較することで、二つの傾向が読み取れる。一つは、記憶メディアの選択肢が増えていることである。今後も記憶メディアの種類は継続して増加していくと考えられる。例えば HDD は垂直磁気記憶や Shingled Magnetic Recording [9] といった大容量化・低価格化を高める技術の研究開発が進められており、SSD との差別化を図ろうとしている。市場においても、高性能高価格モデルの HDD は縮小する一方で大容量低価格のモデルは出荷数が増加傾向にあり、SSD と HDD は住み分けながら共存していく傾向が見られる [10]。一方不揮発性メモリは、先行して商用化された Intel Optane DCPMM 以外にも性能・容量特性の異なる様々な方式が研究されており [2]、今後も既存の記憶メディアの隙間を埋める新たなメディアが登場する可能性は高い。

もう一つの傾向は、揮発性記憶メディアと不揮発性記憶メディアの性能ギャップが縮んでいることである。揮発性記憶メディアは依然不揮発性記憶メディアより高速であるが、不揮発性記憶メディアの高速化によりその差が小さくなっている。2020 年現在において DRAM の読み書きの応答時間は 80 ナノ秒程度であり、HDD の応答時間である数ミリ秒程度に比べると 10000 倍以上のギャップがあった。しかし NVMe SSD の応答時間は 100 マイクロ秒程度、DCPMM の応答時間は 350 ナノ秒程度であり、急速に DRAM との差は減少している。

1.1.2　ストレージアクセスのソフトウェアインターフェースの動向

近年様々な記憶メディアが登場した一方で、ストレージ関するソフトウェアのインターフェースについては大きな変化は起きていない。図 1.2 に、近年のストレージアクセスにおけるソフトウェアインターフェースを示す。広く普及しているインターフェースに、Portable Operating System Interface (POSIX) [11] が定めたインターフェースがある。POSIX インターフェースは元々 C 言語に対するファイルアクセスを規定

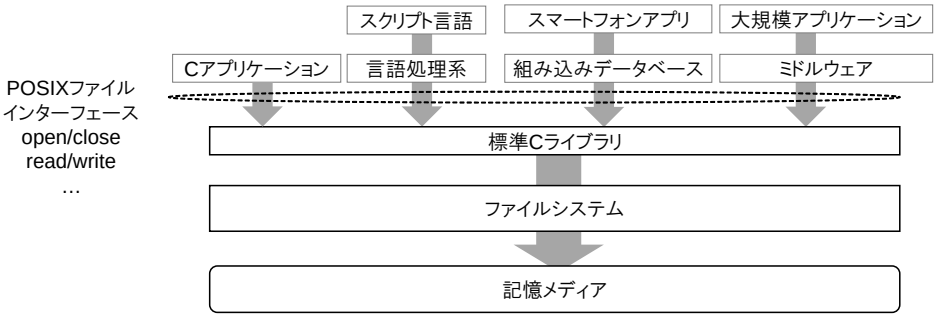


図 1.2: ストレージアクセスにおけるソフトウェアインターフェース

したものであり、OS が備える標準 C ライブラリによって提供される。しかし、現在 C 言語以外においても Python [12], Ruby [13], PHP [14] 等多くのプログラミング言語が POSIX インターフェースと互換性のあるインターフェースを用いている。また、POSIX インターフェースを言語として備えていなくても、Java における OpenJDK [15] のように実行処理系が内部で C 言語の標準ライブラリを呼び出すことで、間接的に POSIX インターフェースを用いているものもある。このように、事実上 POSIX インターフェースは言語を問わずストレージに対する共通のソフトウェアインターフェースとなっている。

アプリケーションによる使用状況については、先行研究 [16, 17] がデスクトップ・サーバ・スマートフォンのアプリケーションがいずれも POSIX インターフェースを用いていることを明らかにしている。同研究では、アプリケーションは必ずしもソースコード中で直接 POSIX インターフェースを用いているとは限らず、より抽象度の高いインターフェースを用いていると述べている。例えばスマートフォン用アプリケーションでは SQLite [18] などの組み込み型データベースが使用され、サーバ用の大規模アプリケーションでは PostgreSQL [19] などのリレーショナルデータベースなどが使用される。しかし、SQLite や PostgreSQL を含め、現在主流のデータベース [20] の大半は内部のデータ格納にファイル及び POSIX インターフェースを使用している。これらのデータベースはいずれも開発開始後 5 年以上利用されており、今後も継続的に使われることが予想される。加えて、近年利用の広がるクラウドコンピューティング環境でも、主要ベンダは NFS を用いた POSIX インターフェース互換のファイルストレージ機能を提供しており、その需要の高さが伺える [21, 22]。

ここで示した動向は、現在動作中のアプリケーションの多くは直接または間接的に POSIX インターフェースを介しファイルアクセスを用いていることを示している。また、データベースやクラウドサービスが POSIX インターフェースを利用していることから、この傾向は今後も続くと考えられる。

1.1.3 階層ストレージ

IT システムが扱うデータが増えるため、ストレージに対する格納データ容量や入出力性能、低コスト化の要件はより厳しくなる。単一の記憶メディアだけでストレージに対するすべての要件を満たすことができない場合、複数の記憶メディアを併用することが考えられる。複数の記憶メディアを使い分ける手法として、階層ストレージ管理 [23] の研究が古くから行われている。階層ストレージでは、データを用途やアクセスパターンに応じて適切な記憶メディアに配置・移動させ、容量・性能・コストのバランスに優れたストレージを提供する。

階層ストレージは、その実現手法によってインターフェースや効果、アプリケーションへの影響はさまざまである。新たなインターフェースを用いて、アプリケーションと密連携することで効果を高める手法もあれば、既存インターフェースとの互換性を重視し、アプリケーションを変更することなく適用できる手法も

ある。階層ストレージの適用には、得られる効果と適用環境の条件に応じた手法の選択が要求される。

1.2　研究目的と課題

本研究の目的は、高い性能を持つ新たな記憶メディアやインターフェースを活用して実用的でアクセス性能の高い階層ストレージを構築し、アプリケーションの性能向上に寄与することである。

本研究では、実用的な階層ストレージの実現において二つの解決すべき課題の解決に取り組んだ。一つはアプリケーションに透過的に適用できることである。階層ストレージの性能を高めるには、新たなハードウェアの利用が不可欠な一方で、アプリケーションからは従来のインターフェースを維持できることが期待される。従来のインターフェースを維持できない場合、メンテナンスされていない古いアプリケーションや商用アプリケーションなど、ソースコード変更が容易でないソフトウェアに適用できず、適用範囲が限定されてしまう。一方、従来インターフェースを継続して提供する階層ストレージであれば、既存アプリケーションの変更することなく適用でき、より実用的に利用できる。もう一つの課題は、性能向上のためにデータをアプリケーションの実行中に最適なメディアに最適な順で配置することである。新たな記憶メディアの高性能を活かすためには、アクセス頻度の高いデータを優先的にそのようなメディアに配置できることが必要である。

1.3　アプローチ

本研究では、ファイル及び POSIX インターフェースを利用したデータの配置を行う階層ストレージを提案する。

これらのインターフェースは図 1.2 で示した通り、アプリケーションと標準 C ライブラリとの境界に位置する。そのため、この境界を保った形で階層ストレージを実現できればアプリケーションの変更を要さず透過的に適用可能な階層ストレージを構築できる。

加えて、アプリケーションはファイル単位でデータにアクセスすることから、他の処理階層、例えばブロック I/O 層等でアクセスパターンを分析するよりも、ファイルに着目してアクセスパターンを分析する方が、より精度が高くアクセスの局所性の高いデータを把握できる。例えば、ディレクトリ階層を認識してファイル単位でアクセスを先読みしたり、ファイル内のブロック単位でのアクセスの局所性を認識したりをブロック I/O 層で行えない。

このように、ファイルおよび POSIX インターフェースを用いたデータ配置を行う階層ストレージは、アプリケーションへの透過的な適用においても、データの最適な配置においても有効な手段である。

1.4　本論文の構成

本論文は、以下のように構成される。第 1 章では、ストレージに関する近年の動向として、記憶メディアとストレージインターフェースの状況を俯瞰し、ストレージへの要求と、本研究の目的を述べた。以下各章にて、ストレージに関する三つの課題と階層ストレージによる解決のための提案手法について論ずる。図 1.3 に本研究の対象とする計算機システムの構成を示す。また、表 1.1 では各章の研究内容の対比を行う。

第 2 章 Practical Quick File Server Migration

新しい記憶メディアを備えた計算機を使用するためには、まずは古い計算機に格納されたデータを新しい計算機に複製し、機器の移行をしなければならない。第 2 章では、計算機としてファイルサーバを対象

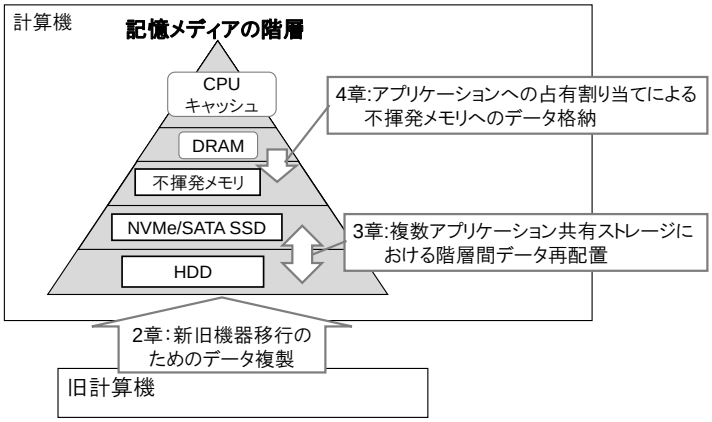
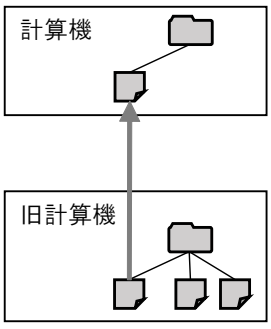
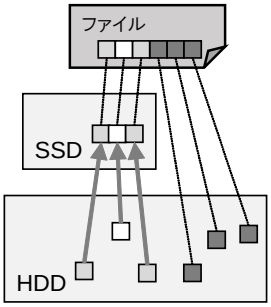
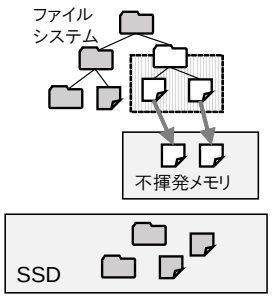


図 1.3: 本研究の対象とする計算機システムの構成

表 1.1: 各章の内容の対比

項目	第 2 章	第 3 章	第 4 章
トピック	新旧計算機間の移行	複数アプリケーションの共有ストレージ	不揮発性メモリへの高速アクセス
課題	データ複製中の平均・最悪 I/O 応答時間の短縮	平均 I/O 応答時間の短縮	平均 I/O 応答時間の短縮
着眼点	ディレクトリ構成とファイルのアクセスパス	ファイル内ブロックのディスク上での配置	アプリケーションデータのファイル配置
施策	以後の応答時間の短縮効果が高い順にファイルを複製	ファイル中の連続ブロックを合わせて先読み	プロセス内に閉じた固有のディレクトリを付与
動作イメージ			

とし、サーバ間の機器移行とそのためのデータ複製について論ずる。移行に際しては、ファイルサーバ上のデータを参照する外部計算機のアプリケーションは、データの複製中であってもファイルサーバを利用し続けられるようにし、アプリケーションを変更せずに移行を完了できることが要求される。提案手法では、アプリケーションからの要求に応じて、新ファイルサーバがオンデマンドで旧ファイルサーバのデータを段階的に複製することで、事前に長時間の停止時間を要さず移行を実現する。また、本手法ではオンデマンドでのデータ複製において祖先のディレクトリの情報が必要となる点に着目し、ルートディレクトリに近いディレクトリとそのデータを優先して複製することで、複製の進行が進むにつれ、オンデマンドでのデータ複製に伴う応答時間を徐々に短縮する。これらの機能は新ファイルサーバ側が全て備えることで、旧ファイル

サーバおよび外部計算機のアプリケーションの変更を要せず移行を行うことができる。

第 3 章 VM-aware Adaptive Storage Cache Prefetching

性能特性の異なる複数の記憶メディアを備えた計算機においては、既存データをアクセス特性に応じて適切な記憶メディアに再配置することで、アプリケーションの性能を向上させることができる。その際、アプリケーションのアクセスの局所性を認識し、今後のアクセス可能性の高いデータを積極的に高性能な記憶メディアに配置することで、その効果が高まることが期待できる。第 3 章では、近年のクラウドコンピューティング環境におけるアプリケーション動作を想定し、複数の仮想マシン上でユーザーが開発したアプリケーションを動作する構成において、SSD/HDD 間でデータの再配置を行い、HDD のデータを SSD に先読みすることでアプリケーション全体の性能を向上する手法を提案する。本章では、仮想マシン構成によってファイルアクセスの局所性がホスト OS 側で認識できず再配置の効果が上がりにくいことと、HDD への先読みとアプリケーションのためのアクセスが競合する問題について、対処法を検討する。提案手法では、仮想マシン内において、ディスク上のファイルの配置情報を取得しホスト OS に通知することで、ファイルアクセスの局所性に基づく先読み対象を選択可能とし、性能向上効果を高める。また、SSD キャッシュヒット率とアプリケーション性能の変化を踏まえて先読みの実行ペースを調整し、HDD への先読みとアプリケーションのためのアクセスの競合を押さえる。提案手法では、仮想マシンにおいてゲスト OS やアプリケーションが公開するインターフェースから得られる情報を用いて再配置を行うことで、アプリケーションの変更を要さず、複数アプリケーション全体の性能を向上する。

第 4 章 User-space Library File System for Low Latency Storage Service Leveraging Non-Volatile Memory

近年普及が始まった不揮発性メモリは既存の不揮発性メディアを超えたデータアクセス性能を備えている。一方、現行のストレージのソフトウェアインターフェースは旧来の不揮発性メディアの性能特性を前提とした構成になっており、その構成を不揮発性メモリに利用してもその性能を活かすことができない。一方、不揮発性メモリ向けの高性能を活かすためにアプリケーションを変更することは、既存のアプリケーションへの適用が難しくなり、実用面で好ましくない。第 4 章では、特に高速なデータの入出力性能が要求されるアプリケーションに対するストレージのインターフェースを提案する。本インターフェースは、従来の POSIX インターフェースと互換性を保つことで既存アプリケーションの変更を要せず透過的に利用可能である。また、本インターフェースではアプリケーションが動作するプロセスのみ参照できるプロセス固有ディレクトリを提供する。固有ディレクトリのアクセスは、ユーザー空間内で直接不揮発性メモリを参照することで完結するため、従来の長大なストレージスタックの実行を迂回し、応答性能を高めることができる。

第 5 章 結論

最後に本章にて一連の研究成果のまとめを行い、今後の課題について述べる。

第 2 章

Practical Quick File Server Migration

IT システムにおいてアプリケーションが長期動作していた場合、既存機器に大量のデータが格納されていることが考えられる。アプリケーションを新しい記憶メディアを備えた計算機で利用するためには、機器の移行とそのための既存データの複製が必要である。

本章では、新旧計算機間における機器の移行とデータの複製について述べる。アプリケーションの動作を継続させながら並行してデータ複製を行う事例として、新旧計算機がファイルサーバとして共有ディレクトリを提供し、外部の計算機上で動作するアプリケーションがそれを参照する構成が考えられる。そこで、本章においてはこのような新旧ファイルサーバ間の移行および移行に必要なデータ複製を、アプリケーションに透過的に行う手法について述べる。

2.1 はじめに

一度構築した IT システムは時間経過とともにソフトウェア・ハードウェアともに急激に陳腐化する。そのため、より高い性能、少ない電力消費、耐故障性のために定期的な IT システムの刷新・移行は必要不可欠である [24, 25]。しかし、IT システム移行の実に約 20% のケースにおいて、移行に伴うシステムのダウンタイムを理由にシステム移行を予定通りに完了できていない [26]。その結果、IT 管理者にとって、IT システムの移行は大きな関心事となっており、それはデータを格納するファイルサーバにおいても当てはまる。ファイルサーバの刷新の大きな理由の一つにファイルサーバのアップグレードが挙げられる [27]。この場合、既存のアプリケーションが動作していることが想定されるため、IT 管理者はファイル共有サービスを長時間停止することなく迅速に移行元サーバから移行先サーバへのデータ複製および移行を完了することが求められる [28]。

本研究では、このようなデータ複製を伴うファイルサーバの移行方式の検討に取り組んだ。研究の推進にあたり、実用的な移行の実現のために三つの達成目標を掲げた。一つめは、上述の通り移行中のファイル共有サービスのダウンタイムを極力削減することである。例えば、エンタープライズ用途のファイルサーバは平均 400 TiB のデータを格納している。このデータをファイルサーバ間で複製するには、数時間から数日を要する [27]。この間ファイル共有サービスを停止させてしまうと、ビジネスにおける大幅な機会損失を招く。よって、ファイル共有サービス移行におけるダウンタイム削減は必須の要求項目となる。ファイルサーバの移行にはプロトコルの仕様上、クライアントがアクセス先を切り替えるのに要する若干のダウンタイムは避けられない。しかしそのダウンタイムは極小であることが望ましい。加えて、ファイルサーバ移行に伴う処理が、クライアントからの通常ファイルアクセスに及ぼす影響も抑え、タイムアウト等アプリケーション動作を妨げることのないようにしなければならない。

二つめは、移行による性能影響を抑えることである。移行のためのデータ複製処理自体が計算機のリソー

スを消費するため一時的な性能低下は避けられないが、その影響は極力少ないことが望ましい。その一方で、性能低下を押さえようとデータ複製の速度を抑えると、移行完了までの期間が長引いてしまう。この複製時のリソース消費と移行完了までの時間の短縮という相反する要求を同時に満たすことは難しい。さらに性能の観点では、移行に伴いファイルサーバに複雑な機構を組み込むことで、移行完了後も性能オーバヘッドが残るような手法は取るべきではない。これはファイルサーバ移行の主目的の一つに、機器の刷新による性能向上があるためである。

三つめに、異なる機種間でのサーバ移行に適用できることである。エンタープライズ用途のファイルサーバベンダは、自社の機種間にのみ適用可能なサーバ移行ツールを提供しているケースがある。しかし現実にはこのようなツールが適用できない様々なケースも存在する。例えば、移行元サーバの提供ベンダがファイルサーバ製品を開発終了してしまった場合や、小規模なファイルサーバから、大規模なエンタープライズ向けのファイルサーバに移行する場合などが考えられる。異なる機種間での移行を前提とすると、ファイルシステムなど内部構成が異なるファイルサーバ間での移行が必要となる。しかし、エンタープライズ用のファイルサーバ製品では一般に内部情報へのアクセスが制限されており、例えば移行元サーバの上で自製のプログラムを動作させたり挙動を変えさせたりすることは現実的ではない。つまり、異なる機種間での移行を前提とするならば、移行方法は移行元サーバの内部構造や機能に依存しない方式が求められる。

ファイルサーバの移行には、移行元サーバの既存データを移行先サーバへ複製する必要がある。サーバ間のデータ複製には、*Pre-copy* と *Post-copy* の二つの主要なアプローチがある。Pre-copy は、クライアントが接続先サーバを切り替える前にデータを全て移行先に複製するアプローチである。こちらは単純な方式であるため、従来からファイルサーバの移行のために広く使用されてきた。本方式の良い点は、サーバを切り替えた瞬間から、移行に伴う性能オーバヘッドがなくなることである。一方、短所として、データの複製に時間がかかることが挙げられる。データの複製中、IT システムを停止できる時間があれば良いが、そうでない場合、データを複製しながらクライアントによるファイルアクセスを継続することになる。Pre-copy の場合、複製中に更新されたファイルについて更新差分を繰り返し再送することで、移行元サーバと移行先サーバのデータの差分を埋めていく。また、最後の更新差分の再送処理の間は、クライアントアクセスを停止しておき、更新が行われない状況で差分を埋める。この手法は、クライアントアクセスが頻繁である環境においては再送を頻繁に行うために移行完了までの時間が長時間化したり、最後の再送処理によってダウンタイムが長時間化したりする問題が残る。対する Post-copy はクライアントの接続先サーバの切り替えを先行させ、その後データの複製を行うアプローチである。こちらは接続先サーバの切り替えを秒オーダーと短時間で行えるため、アプリケーションのダウンタイムを非常に短くできる。また、データの複製中にクライアントの更新アクセスが高頻度で行われても、更新は移行先サーバに直接反映されるため、サーバ間で再送を繰り返す懸念もない。

以後本章では、実用的なファイルサーバの移行手法について述べる。提案方式においては、ファイルの移行に伴うダウンタイムを削減するため、Post-copy 方式を応用した既存データの複製を行う。そのため、移行元サーバにあるファイルデータは、クライアントから移行先サーバに初めてアクセスがあった時点で複製する。移行先サーバにおいてこの動作を実現するため、**オンデマンド複製**を行う。ただしこのオンデマンド複製は、クライアントの要求に合わせて複製処理を行うためクライアントへの応答時間が伸びてしまう。そこで、この応答時間の短縮のため、**バックグラウンド複製**を並行して行う。バックグラウンド複製では、クライアントアクセスとは別に移行先ファイルサーバ内でプログラムが共有ディレクトリ内をクローリングし、未複製ファイルを順次複製していく。これにより、いざクライアントからファイルアクセスが来た時の応答時間の短縮を図る他、全ファイルの複製完了への時間も短縮する。バックグラウンド複製は未複製ファイルの探索や複製のため計算機のリソースを消費することから、クライアントアクセスが少ない時間 (深夜や週末) であることが望ましいが、性能低下を厭わないのであれば、クライアントアクセスと並行して動作

させ、計算機システムを停止させずに行うこともできる。

オンデマンド複製を実現させるため、各ファイルにおいて複製状況の状態管理を行う機構を備えなければならない。本研究では異機種間の移行を行うため、移行元サーバ側の内部構成に依存した手法は取れない。また、移行先サーバであっても、移行完了後も長期的に性能オーバヘッドが残るような手法は好ましくない。そこで、本手法では移行先サーバにてスタブを用いたファイル管理を行う。スタブは、ファイルシステムにおいて複製中ファイルの中間状態を管理するための情報を保持した仮想的なファイル (ないしディレクトリ) である。この管理は移行先サーバ内で行い、加えて複製には標準のファイル共有プロトコルを用いるため、本方式は移行元サーバの機種や機能に依存することなく適用できる。また、複製完了後は移行先サーバからスタブに関する情報は除去するため、性能オーバヘッドも残らない。

本稿では、実用的でかつ効率的なデータ複製を実現するため、バックグラウンド複製に際し、断続的なクローリング手法と、クローリングにおける探索順に関しても論ずる。断続的なクローリングは、バックグラウンド複製を一回の実行で終了できないほどデータ量の多いファイルサーバの移行に必要である。提案方式では、性能より実用における堅牢性を優先して、クローリング再開時に毎回ルートディレクトリからの再探索を行う。クローリングの探索順は、サーバ移行後の各ファイルの初回アクセスにおいて、平均および最悪応答時間に大きく影響する。本稿では探索順の候補を示し、ファイルサーバのアクセスパターンに応じた推奨手順を示す。また、性能評価を行いその推奨手順の妥当性を検証する。

さらに本移行方式では、実運用環境において欠かすことのできない、ファイル共有プロトコルの仕様によるいくつかの隘路への対処を示す。これらの隘路事項として、アクセス権管理や ID 管理、NFS 固有の命令である `truncate`, `hardlink`、タイムアウト処理、また Windows のエクスプローラー特有のファイル参照動作への対応がある。

本手法は、いわゆるオフィス向けデータ共有のためのファイルサーバを想定している。用途としては、部署内で使用する業務文書や実験データの共有ディレクトリや、メールサーバ・Web サーバ等のサーバアプリケーションのバックエンドストレージがある。典型的なワークロードは、ファイルサーバ向けのベンチマークである SPEC SFS [29] や Filebench [30] で代表されるとおりである。想定する装置やデータの規模は、数台〜数十台のディスクを備え、1 GbE〜10 GbE 程度のネットワーク帯域と 10 GiB〜10 TiB 程度のファイルシステムを複数備える程度とし、数十名〜数百のユーザーを想定している。大規模構成においては、ファイルサーバの内蔵ディスクだけではなく、Storage Area Network を介し外付けの大型ディスクアレイ装置にデータを格納するケースも想定する。この場合、100 TiB 級のファイルシステムと数千名規模のユーザーにも対応する。ファイルサーバの移行においては、複数の小規模ファイルサーバの集約も主要な移行ユースケースの一つであることから、複数の移行元サーバを一台の移行先サーバに移行するケースも想定する。また、移行に伴うオーバヘッドとして、ミリ秒〜数秒程度の応答遅延が許容されるケースを想定している。ここで述べた以外のユースケース、例えば超高性能・大容量を要するビッグデータ解析や High Performance Computing など、ミリ秒オーダーの僅かな応答遅延も許容されない高いサービスレベルを要求されるケースは対象としていない。

本研究では、提案するファイル移行方式について、Linux カーネルとユーザー空間プログラムの組み合わせとして実装を行った。本実装では、主要なファイルアクセスプロトコルとして、Network File System version 3 (NFSv3) および Server Message Block 3.0 (SMB3.0) の両方でのファイル移行をサポートしている。また、実際に企業で運用されている数百 TB クラスのファイルサーバを対象とした移行を成功させている。

本手法の評価実験において、サーバ移行時のクライアントにおけるダウンタイムを測定したところ、NFS、SMB ともに最大でも 3 秒以下であることを確認した。また移行後の初回ファイルアクセスにおいて、実用的なデータ配置においてほとんどのケースで 10 秒以下の応答時間を達成した。極端に巨大なファイルシス

テムとして深さ 16 段のディレクトリがそれぞれ 1000 個ずつ下位ディレクトリを持つファイルシステム (計 10^{48} ファイルを想定) を模したデータセットにおいても、最大応答時間は NFS で 84.55 秒、SMB で 23.25 秒であった。これらは、アプリケーションがタイムアウトで停止することなくアクセスを継続できる応答時間に収まっている。

移行にかかる時間は、実際のオフィス内共有ディレクトリを模した 677GiB のデータセットを複製した場合で、NFS で 7.43 時間、SMB で 12.17 時間となった。これは同じハードウェア構成で Pre-copy ベースの移行を行った場合に比べ 3 倍近い時間となった。しかしこの測定は複製中にクライアントのデータ更新を伴わない構成の値であり、実用的にクライアントアクセスによって、特に Pre-copy は複製時間の長時間化を引き起こす可能性がある。提案方式はクライアントアクセスが生じても大幅な複製時間の長時間化を行わない方式であり、この性能差は必ずしも問題とはならない。

評価実験において、提案手法は一旦移行が完了したデータに対し、オーバヘッド無しでクライアントアクセスが行えることも確認した。

本研究における貢献は下記のとおりである。

- ファイルサーバの移行に伴う三つの課題を整理し、それらを解消する実用的な移行方式を提案した。
- 実環境における移行を成功させるため、ファイル共有プロトコル依存の課題と対応方法を明らかにした。
- 実用に耐えうる実装方法を示した。
- 実際のオフィスで運用されていたファイルサーバのデータセットを用いた評価実験を行い、提案手法による移行の可否とその効率を明らかにした。
- データの複製順に対する移行中アクセス性能の特性を示し、ファイルサーバの使用法に対応して取るべき複製順を明らかにした。

本章の構成は以下のとおりである。2.2 節ではストレージ機器の移行とデータ複製に関する既存研究を整理し、提案方式の特徴を明らかにする。2.3 節で提案する移行方式を述べ、2.4 節では実環境への適用で重要となるプロトコル依存の課題とその対応方法について述べる。2.5 節では実装について説明する。2.6 節では提案方式を用いたファイルサーバ移行を行い、性能やファイルアクセスを行うアプリケーションへの影響を評価する。最後に 2.7 節で本章に関する結論を述べる。

2.2　関連研究

2.2.1　Pre-Copy 方式

Pre-copy 方式は、データの複製完了後にクライアントの接続先を移行先サーバに切り替える方式である。Pre-copy 方式は、ファイルサーバ移行におけるデータ複製方法としては最も一般的に利用されている。前述のとおり、Pre-copy 方式では更新差分を繰り返し移行元サーバから移行先サーバに複製し、更新差分が十分に小さいと判断できた時点で、移行元サーバでのクライアントアクセスの受付を停止し、最後の更新差分を複製した後に移行先サーバの利用を開始する。この方式は商用ファイルサーバ製品においても一般的に利用されており、rsync [31] や Robocopy [32] のようなディレクトリツリーの同期ツールが用いられる。本方式は、移行が完了してしまえば以後性能オーバヘッドが生じない点にある。しかしながら、この方式は二つの欠点が残る。

一つは長いダウンタイムである。この方式は最後の更新差分複製の手順においては移行元・移行先両方のファイル共有サービスを停止しなければならない。クライアントアクセス負荷の高いファイルサーバを移行

する場合、複製すべき更新差分の量が大きく、かかる時間も長くなる。加えて、ファイルサーバの外部から更新されたファイルを特定・列挙するのは、ファイル数に比例して時間がかかる。例えば、ファイルサーバの平均容量である 400 TiB 分のファイルを持つファイルサーバにおいて、ディレクトリツリーを辿りタイムスタンプを列挙しようとすると概算で 28 時間かかる。このような長時間ファイル共有サービスを停止することは、常時稼働を要請するシステムでは許容できない。

もう一つは移行にかかる時間の長時間化である。更新差分複製を繰り返す過程において、頻繁にデータの更新が行われる場合、同じファイルを複数回複製する可能性が高まる。これは移行にかかる合計時間を延ばすことになり、また更新差分検出・複製のためのオーバヘッドにより長期的なファイルアクセスの性能低下にもつながる。更新差分検出・複製に用いる CPU や I/O のリソースを絞ることで性能オーバヘッドは削減できるが、これは一方で複製完了までの時間長期化とのトレードオフになる。Pre-copy 方式を用いると、この相反する要件の解消が困難である。

2.2.2 Post-Copy 方式

Post-copy 方式は、クライアントの接続先サーバの切り替えをデータ複製前に行い、クライアントが移行先サーバのデータを初めて参照したとき、移行元サーバから移行先サーバにデータを複製する方式である。本方式ではサーバの切り替えは即座に完了するため、ダウンタイムは最小化される。

VNX File System Migration [33] は Post-copy ベースのファイルサーバ移行を行う商用ツールである。本研究が提案方式とこのツールの動作は類似しているが、このツールでは未アクセスファイルを先行的に複製する処理を行わない。そのため、初めて参照されるファイルへのアクセス時には長い応答時間が生じる。加えて、本手法では移行の最後に未アクセスファイルを移行元サーバから複製する手順を必要とし、その間ファイル共有サービスは停止しなければならない。

ファイルサーバ移行を意図したものではないが、Post-copy 方式によるデータ複製を行う関連研究に Panache [34] がある。Panache は広域分散ファイルシステムにおけるローカルキャッシュサーバ構築に Post-copy 方式を用いている。Panache ではローカルキャッシュサーバ上の名前空間上に、実際のファイルを構成するデータを保持しない orphan i ノードという i ノードを保持する。クライアントが orphan i ノードをアクセスすると、その時点でファイルのデータが遠隔サーバから複製される。この仕組みはその後ファイルサーバの移行に応用されている。ファイルサーバ構築用の商用ソフトウェアである IBM Spectrum Scale [35] は、Panache を元にしたファイルキャッシュの機構に、外部のファイルサーバからのデータ取り込みを実現する Active File Migration [36] という機構を取り込んだ。こちらは NFS を介したファイルサーバ移行を実現している [37]。ただし、本機能は基本的に専用プロトコルでの使用を想定して作られており、NFS や SMB と言った一般的なファイル共有プロトコルにおいてデータの複製は可能である者の、プロトコル固有の動作・仕様により生じる問題点への対応はなされていない。

Post-copy 方式の応用として、Pre-copy 方式と連携させることも考えられる。一部のファイルシステムでは、指定時刻におけるスナップショットを取る仕組みがある [38]。そこで、スナップショットのみ Pre-copy 方式で送信しておき、その後その送信期間中の更新差分のみ Post-copy 方式で送ることが考えられる。しかし、こちらは 2.2.1 項で述べた課題は依然残っており、データの更新頻度が高い状況においては、ダウンタイムや複製時間が増加してしまう。

2.2.3　ファイルアクセスの転送と複製

Global Name Space (GNS) [39] は、複数のファイルサーバの共有ディレクトリを統合した単一の名前空間を構築する手法である。実現には、各ファイルサーバの手前に、専用のアプライアンスやソフトウェアで構成された仮想化層を導入する。この仮想化層では、統合名前空間内のファイルと個々のファイルサーバ内のファイルの対応関係を管理する。そしてクライアントから統合名前空間内へのファイルアクセス要求を受けると、対応するファイルサーバにそのアクセス要求を転送する。そのため、クライアントはファイルが実際どのファイルサーバに格納されているかを認識することなく透過的に複数のファイルサーバを単一の名前空間からアクセスできる。

X-NAS [40] および NAS Switch [41] は、GNS を用いてファイルのサーバ間での複製を行う。類似の手順をとる商用のファイルサーバ移行製品も複数存在している [42, 43, 44, 45, 46]。これら製品は移行元のファイルサーバに対する仮想化層としてふるまい、バックグラウンドでサーバ間のファイル複製を行いつつ、ファイルの所在に応じてクライアントからのファイルアクセス要求を適切なサーバに転送する。一部製品では、仮想化層でファイルのキャッシュ機構も備えることで、アクセス要求の転送オーバヘッドを吸収している。

これらの方式は、移行完了後も仮想化層を継続的に使用し続けなければならない点が問題となる。まず、仮想化層によるアクセス要求の転送オーバヘッドは恒久的に残る。仮想化層でキャッシュ機構を備えることでオーバヘッドを軽減する方式もあるが、完全にオーバーヘッドが取り除かれるわけではない。また、GNS は複数のファイルサーバの名前空間を集約する機構なので、1 台のファイルサーバの移行への適用はコスト面で無駄が多く実用面では好ましくない。

ファイルアクセス転送のオーバヘッドが仮想化層に集中することを防ぐため、プロトコル固有の機能を応用する先行研究も存在する。Parallel NFS (pNFS) [47] は NFS プロトコルの拡張であり、単一サーバがボトルネックとならないようなデータ転送を実現する。pNFS では、ファイルのデータを格納するファイルサーバとは別に、ファイルの名前空間およびメタデータを格納するメタデータサーバがファイルのデータの所在を管理する。pNFS のクライアントはメタデータからファイルデータの配置情報を受信すると、以後のファイルデータのアクセスはこのメタデータサーバを介さず個々のファイルサーバとの間で行われる。よって、pNFS ではデータ転送を仲介する層が存在しないため、データ転送に伴うオーバヘッドが解消される。一方、メタデータアクセスに関してはメタデータサーバに負荷が集中するという問題が残る。pNFS を用いたデータの複製方式もすでに提案されており、こちらは先の GNS を用いた複製方式と同様、データを複製しながらメタデータサーバ内のファイルの所在情報を更新することで、クライアントに透過的にデータの複製を完了させる [48]。FSMT [49] は、Microsoft distributed file system (MS-DFS) の環境下においてファイルサーバ間のデータ複製を行うツールである。MS-DFS においても GNS の仮想化層に相当する処理を行うサーバがいるため、そのサーバがバックグラウンドでデータの複製を行う。しかし、これらのプロトコル固有の複製方式は、一般的に使われる NFS や SMB におけるファイルアクセスに対応できない。

ファイルサーバ内のデータやメタデータを単一のアーカイブファイルとしてまとめてエクスポートし、そのアーカイブファイルを別サーバでインポートすることでデータの複製を行う研究もある。DAB [50] や Cumulus [51] は、拠点間のデータの遠隔バックアップ用途でそのような手順をとっている。Windows Server Migration Tool [52] は Windows Server の備える機能であり、ファイルサーバにとどまらず Windows Server の全サーバ機能に対する移行をサポートする。ただしこれらの手法はいずれもアーカイブファイルの形式を共有したサーバ間でしか適用できず、異機種間の移行には利用できない。

2.2.4 ボリュームベース複製

ファイルシステムの複製を、ファイルではなくボリュームベースの複製で行う先行研究も存在し、商用ファイルサーバ製品で実用的に使用されている。Dell FluidFS NAS [53] や NetApp Volume SnapMirror [54] は、運用中の移行元サーバにおいてファイルシステムを構成するボリュームスナップショットを作成し、それを移行先サーバに送る。その間にも移行元サーバでクライアントアクセスは継続して受け付け、更新されたデータに関しても、同様にボリュームの更新差分がある領域のみデータを再送する。ボリュームベースの複製は、特に初回のスナップショット作成においてディスクに対しシーケンシャルアクセスを行えるためデータ転送の効率が良いという利点がある。
しかし、ボリュームベース複製の問題として、移行元・移行先の両サーバが同一のボリュームをマウントできなければならず、異機種間では利用できない。実際、上述の手法も自社製品間でのみ複製可能である。

2.3 設計

提案方式は、オンデマンド複製、バックグラウンド複製、スタブによるファイル管理、の三つの手法を組み合わせることでファイルサーバ移行を実現する。以下、それぞれ順に説明する。加えて、本節では実用において重要な、断続的な複製およびバックグラウンド複製における探索順について論ずる。オンデマンド複製とバックグラウンド複製はスタブによるファイル管理を用いているため、スタブ機能に関して一部先行して述べる。

2.3.1 オンデマンド複製

オンデマンド複製は、未複製ファイルに対するクライアントからのファイルアクセス要求に応じてファイルを段階的に複製する手法である。本複製機能により、移行先サーバは、移行元サーバが保持するデータが複製されていない状態でクライアントのファイルアクセス要求を受け付けることができる。このオンデマンド複製を行うためには、スタブの機構を応用する。スタブは元々階層ストレージ管理において使用された機構で、スタブファイルとは、あるファイル (またはディレクトリ) において、データおよびメタデータの一部がサーバ内部にはなく、外部に存在する複製途中状態を保持・管理できる特殊なファイルである。そのため、スタブファイルのデータの格納先を移行元サーバとすると、データ移行に応用できることになる。

スタブファイルの状態管理はサーバ内で行われ、クライアントには認識されない。クライアントがスタブファイルのデータ・メタデータを参照すると、サーバはその時点で対応する外部のデータ・メタデータを複製しクライアントに応答する。ディレクトリについても同様に、スタブディレクトリは初期状態ではディレクトリエントリをサーバ内に持たないが、クライアントに参照された時点でエントリを複製する。

クライアントが深いディレクトリ内のファイルにアクセスする場合、そのパス上に現れる祖先にあるディレクトリ群が順次複製され、そして最終的に対象のファイルにアクセスできる。この動作に関しては、2.3.3 項で詳細に説明する。

2.3.2 バックグラウンド複製

オンデマンド複製では、クライアントにアクセスされたファイルのみが複製される。そのため、オンデマンド複製だけではアクセスされないファイルは複製されず、いざアクセスにより複製されたとしても、初回アクセスの応答は上述の祖先のディレクトリの複製を伴うため時間が長くかかる。これらの問題に対応する

ため、本移行方式ではオンデマンド複製に加えてバックグラウンド複製を備える。バックグラウンド複製では、移行先サーバにおいて共有ディレクトリのツリー全体を探索するクローラが動作する。クローラは移行先サーバ内にあるスタブファイル・スタブディレクトリを順次アクセスし、オンデマンド複製を引き起こすことで共有ディレクトリ全体の複製を行う。

バックグラウンド複製はファイル共有サービスを動作させた状態で、クライアントアクセスと並行して実行可能である。ただしバックグラウンド複製はデータ複製のためにサーバのマシンリソースを消費するため、本設計においては、ストレージ管理者がクライアントアクセスの多い時間帯はバックグラウンド複製の一時中断を希望する場合も想定においている。

バックグラウンド複製においてはクローラがディレクトリツリーを探索するため、アクセス時刻のタイムスタンプのように探索自体がファイルのメタデータに対する副作用を与えかねない。そのため、ファイルシステムがメタデータを変更せずにファイルがスタブかどうかの状態を取得できるインターフェースを備えることで、副作用を抑止している。

バックグラウンド複製を実現するうえで考慮すべき二つの課題として、断続的なクローリングと、探索順戦略がある。断続的なクローリングは、複製対象のディレクトリの格納データ量が非常に多く、クローリングをユーザーが許容する時間内に完了しきれない場合に、複数回に分けて複製を行うために必要となる。例えば、一度の夜間や週末に複製が完了しないケースが想定される。探索順戦略は、複製対象のディレクトリツリーにおけるクローラの探索及びデータ複製順序を定めるものである。これらはどのファイルが先に複製完了するかに影響するため、複製処理中のクライアントアクセスにおける平均および最大応答時間に関与する。以下、断続的なクローリングと探索順戦略について説明する。

断続的なクローリング

バックグラウンド複製を断続的に行う場合、クローラが停止している間にクライアントが変更したファイル・ディレクトリの影響を考慮して、複製漏れファイルやデータの不整合が起きないようにしなければならない。一つの対策案としては、クローラが中断する際に直前の進捗状況を保存しておき、再開時は保存した状態から処理を継続することが考えられる。しかし、クローラの停止中にクライアントがディレクトリの削除やリネーム・移動と行ったディレクトリ構成の大幅な変更を行う場合、進捗状況の整合性を保つのは非常に困難である。これは設計・実装の複雑さに加え、その複雑な状態管理のためにクライアントのファイルアクセス性能を損なう懸念があり、よい対策案ではない。他の対策案としては、クローラ停止中におけるクライアントによるディレクトリツリー構造の変化を逐一保存して、クローラが探索済みの領域を追跡し、再開後は同じ部分木を繰り返し探索することをさけることが考えられる。しかしこちらも同様に状態変化の保存・追跡による性能低下を避けられない。

整合性を保つための複雑な手続きを避け、ファイルアクセスのオーバヘッドを生じないようにするため、本手法では単純に再開時は毎回共有ディレクトリのルートから探索をし直すこととした。ただし、全ファイル複製完了を判断する手段がなければクローラの動作を終了できないため、効率的な判断手法として、移行先サーバ内のファイルシステムにおけるスタブファイルの残数をカウンタとしてファイルシステム中に保持することとした。ディレクトリの探索に伴い、その配下に新規にスタブファイルを生成するとカウンタの値がインクリメントされ、ファイルの複製が完了するとデクリメントされる。クローラは探索を一巡するたびにカウンタの値を確認し、0 であれば複製完了として処理を終え、そうでない場合再度探索する。このカウンタは、複製未完了のファイル数とは一致しない。未複製のディレクトリにおいて、配下のファイル数を移行先サーバが認識しないためである。しかし、その場合も未複製のディレクトリの存在によってカウンタは正に留まるため、カウンタ値が 0 となることをもって複製完了として差し支えない。データセットに強く依存するが、一般にディレクトリ全探索は時間のかかる処理である。例えば 1 ファイルのメタデータ取得に 1

ミリ秒かかるとし、10 TiB のファイルシステムに 1 MiB のファイルが一千万個格納されているケースで概算すると、メタデータの確認だけで 3 時間かかる。本研究の前提として数十億ファイル規模のファイルシステムを想定していないため、この時間は、深夜や週末に実行することで実用環境において許容範囲と考える。カウンタとルートディレクトリからの再探索を伴うこのアプローチは単純であるものの、同じファイルを複数回参照することによるサーバリソースの過剰消費という短所を含む。あいにく、この効率とファイルの複製漏れを起こさない頑健性はトレードオフの関係となる。本研究では、効率性よりはエンタープライズ用途のファイルサーバに要求される高い頑健性を優先し本アプローチを採用した。もう一つの本方式の短所として、クライアントのアクセス頻度が高いファイルサーバの場合、クライアントアクセスの影響でクローラの動作が遅くなり、一回の実行でディレクトリツリー全体を探索できなくなる場合がある。その対策としては、QoS 制御やプロセスの優先度付けにより、クライアントアクセスがクローラに与える影響を軽減させることが考えられる。現時点ではこの対応は未着手であり、今後の課題としている。

時間経過やストレージ管理者の指示でクローラの動作を中断させる場合、巨大なファイルを複製中の場合であっても、複製完了を待たず即座に中断できることが望ましい。そのため、本クローラはファイルのデータを複製する際、一定サイズ毎にストレージ管理者からの中断通知を監視しており、巨大ファイル複製の途中でも複製作業を中断できる。本方式ではスタブの機能として部分的なデータ複製状態をサポートしており、またその状態もスタブ内で管理できるため、クローラの再開時は未複製のデータ領域から複製を再開できる。

現時点では、クローラの構成を単純でロバストに保つため、1 ファイルシステムあたり 1 プロセス・1 スレッドで動作させることにしている。並列クローリングは、クライアントアクセスによりディレクトリ構成が頻繁に書き換わる状況においてクローラ間の動作の同期が難しく [55]、探索漏れや複数回のデータ複製により、複製時のデータ不整合を及ぼしかねない。そのため、並列クローリングは今後の課題としている。

データ複製順の戦略

本稿における探索順の戦略とは、バックグラウンド複製におけるデータの探索・複製順を意味する。サーバ接続先変更後初回のファイルアクセスにおいては、祖先のディレクトリ及びファイルメタデータを先に複製したうえで、対象のファイルにアクセスすることになる。そのため、探索順の戦略は複製途中のクライアントファイルアクセスにおける応答時間に影響する。ここでは探索順の戦略を二つの観点、**ディレクトリ探索順**と**ファイル構成情報複製順**に分けてそれぞれ検討する。以下、両観点における選択肢を説明したうえで、2.6.5 項にて実際のデータセットを用いた性能評価を行う。

■ディレクトリ探索順　ディレクトリ階層は木構造を成すため、木構造の探索アルゴリズムとして主要な深さ優先探索 (Depth-First Traversal, DFT) と幅優先探索 (Breadth-First Traversal, BFT) の二つの手法を適用することが考えられる。これらはいずれも木構造全体を探索するため、探索及び複製にかかる時間の点では大きな差を生じないが、複製の順序が異なるため、複製中のクライアントアクセス応答時間への影響が異なる。

移行中のサーバにおけるクライアントアクセス時間は、祖先のディレクトリの複製状況に依存することから、バックグラウンド複製が進めばそれだけ応答が速くなる。同様に、最長応答時間はディレクトリ階層の深い位置にあるファイルによって決まるが、その時間も祖先のディレクトリの複製が済めば短縮される。ただし、もしクライアントアクセスが一部のディレクトリに偏っている場合、他の深い位置にあるファイルの祖先のディレクトリは複製が進まないため、DFT の場合その領域に探索が及ばないと探索の終了直前まで最長応答時間が短縮されない。逆に DFT で最初に探索された領域については、応答時間が速やかに改善するため、平均応答時間自体は DFT についても短縮される。このように、DFT の効果はクライアントのファ

イルアクセス順に強く依存し、DFT において複製順が早い領域とクライアントアクセスの集中する領域が近ければ大きな応答時間改善効果が出る。一方、BFT はディレクトリを浅い順に均等に複製するため、平均応答時間・最長応答時間はともにクライアントのアクセスパターンの影響を受けにくい。

一般に、ファイルシステムへのファイルアクセスには偏りが生じることが知られている。しかしながら本研究では対象としてオフィスにおける共有ディレクトリを想定していることから、先行研究 [56] が示す通り、多数のクライアントがそれぞれ異なるパターンでファイルアクセスを行うことで、万遍なくアクセスが生じ全体として偏りは小さいと仮定している。この仮定に基づけば、一般に BFT は DFT に比べ、バックグラウンド複製の進捗に対し平均および最長応答時間を効率よく短縮できることが期待できる。

一方で DFT における利点に、使用メモリ容量が少ない点がある。DFT においては、探索中に祖先のディレクトリの情報を保持するため、一般にスタック形式のデータ構造を用いる。この時、ファイルのパス名はプロトコルの仕様上上限があることからディレクトリの深さは高々数千となり、メモリ消費量も僅か (KiB ～MiB オーダー) となる。一方 BFT においては、ある深さに属するファイル名の一覧をすべてメモリ中に保持することとなり、一般にキュー形式のデータ構造を用いる。巨大なファイルシステムでは数億個規模のファイル数を扱うことから、キューに保持するファイル情報も数千万規模に到達することが考えられる。例えば 1 ファイル 1 KiB の情報を持ち、ある深さに 10 万個のファイルが存在するのであれば、合計で 100 MiB のメモリ消費を要することとなる。この消費容量は現在の計算機において問題となる大きさではないが、さらに巨大ファイルシステムを用いる構成や、搭載メモリの少ない小規模構成の機器においては問題となりうる。

このように、BFT と DFT はそれぞれ長短があるため、実装に際してはユーザーが探索順を選択できるようにした。しかしながら、一般には、メモリ消費量が現代の計算機で許容範囲であることと、応答時間の短縮が速やかに進むことから BFT を推奨する。DFT は計算機のリソース制限が強い場合や、将来的なアクセスパターンが高精度で予測可能な場合に有効である。

■ファイル構成情報複製順　個々のファイルの複製においても、選択肢として**一括複製方式**と**メタデータ先行複製方式**が考えられる。一括複製方式では、クローラは一度のディレクトリツリー探索において各ファイルのデータとメタデータを一度に複製する。一方、メタデータ先行複製方式ではクローラがディレクトリツリー探索を二回に分けて実施し、一回目の探索でメタデータを複製して二回目でデータを複製する。

この順序の選択において注目すべき点は、ファイルアクセスの応答時間は祖先のディレクトリの複製状況の影響を受ける一方で、その他のファイルの複製状況の影響は受けない点にある。一括複製方式では、あるディレクトリの複製を終え、次のディレクトリの複製に進む際、ファイルのデータも複製するために時間がかかる。一方、メタデータ先行複製ではデータの複製を後回しにするため、ディレクトリの複製が速やかとなり、最長応答時間が速やかに削減される。メタデータ先行複製の課題として、2 回のディレクトリ探索を行うことでサーバリソースと時間を一括複製方式よりも多く消費し、結果として複製完了までの総時間が延び、複製作業中の複製完了済みファイルに対するアクセス性能も低下する。

これらの選択肢においても、ディレクトリ探索順同様ユーザーが選択可能としている。どちらが最適かを判断するのは、データセットの傾向や使用するサーバリソース、さらにはクライアントのアクセス状況に依存するため一概に決定できない。しかしながら、一般にアクセス応答時間はユーザー体験に大きく影響することから、最長応答時間を速やかに削減するメタデータ先行複製が多くの場合好ましいと考える。

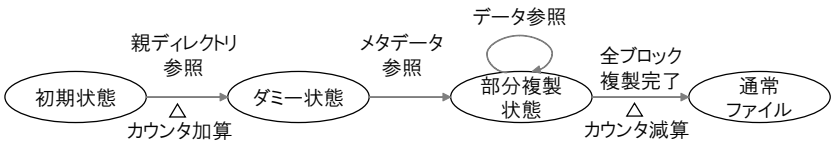


図 2.1: ファイル毎のスタブの状態遷移

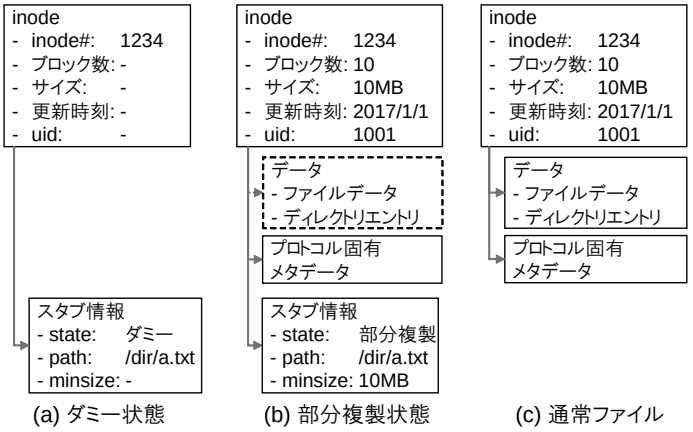


図 2.2: サーバ内部におけるディスク中のファイルの管理構造

2.3.3 スタブによるファイル管理

オンデマンド複製の実現手段として、提案手法ではスタブの考え方を応用した。上述の通り、スタブとは、ファイルシステム内部において、一部のデータ及びメタデータがサーバ内部に格納されていない状態のファイルまたはディレクトリを意味する。しかしながら、ファイルシステム外部からはこのスタブは通常のファイルまたはディレクトリとして見える。クライアント (もしくはクローラ) がスタブにアクセスすると、その時点でファイルサーバはサーバ内部に格納されていないデータまたはメタデータを未複製と判断し、その時点で移行元サーバから取得して移行先サーバ内部に複製したうえでクライアントに応答する。すでに移行先サーバに複製され内部に格納しているデータ及びメタデータはそれ以上複製を行わないので、同一ファイルへの 2 回目以降のアクセスは応答時間を短縮できる。以下の項目では、ファイルに対するスタブの構成例を示すが、同一の説明はディレクトリにも適用される。

スタブファイルは、内部的には、**ダミー状態**と**部分複製状態**の二つの状態のいずれかをとる。図 2.1 に状態遷移を示し、図 2.2 に各状態におけるサーバ内のディスク上の管理データ構造を示す。スタブファイルは、ファイルシステム内では通常ファイル同様に inode として管理される。ダミー状態のスタブは、inode におけるほとんどの属性は空であり、データブロックへのポインタも保持しない (図 (a))。部分複製状態のスタブは、inode の属性は埋められているが、ただしファイルデータ本体は、部分的にしか埋められていない (図 (b))。加えて、部分複製状態のスタブはプロトコル固有のメタデータを通常ファイル同等に格納する。プロトコル固有のメタデータの例としては、アクセス権管理情報、拡張属性、NTFS 代替ストリーム、各種タイムスタンプがある。データの複製状態の管理は、ファイルシステムが管理するファイルブロックの割り当て情報を用いる。通常ファイルシステムでは、スパースファイルに対応するために領域未割り当てのブロックを保持することが可能である。スタブにおいてはこの未割り当てのブロックを未複製のブロックとみなすことで、ファイルのブロック単位での複製状態を管理する。ダミー状態と部分複製状態の明確な違い

は、ダミー状態のスタブを構築する要素は、移行元のファイルに関する情報を保持しない、すなわち移行先サーバは移行元サーバの inode を参照せずにダミー状態のスタブを構築可能な点にある。対する部分複製状態のスタブは、移行元のファイルのメタデータを反映したものとなっている。この状態遷移の過程で、2.3.2 節に示すカウンタ加減算も実施される。なお、ディレクトリについてはダミー状態から通常ファイル状態に直接遷移し、部分複製状態に留まることはない。ディレクトリはファイルデータを保持しておらず、さらにディレクトリエントリは一括して取得するため部分的に取得した状態にならないためである。

オンデマンド複製の処理過程で作成されたファイルは、最初ダミー状態のスタブとして作成され、続くクライアントアクセスによりメタデータが参照された時点で部分複製状態に遷移する。このダミー状態は、ディレクトリを複製する際にディレクトリ内の各ファイルの情報取得を即座に行わず遅延せることでディレクトリ参照時の応答時間軽減の効果がある。もしディレクトリに数千ファイルが格納されていてそれらのファイルを復元しようとする場合、メタデータを複製するだけでも数千回のネットワーク通信とディスクアクセスが必要となり時間がかかってしまう。しかし、通常ディレクトリ参照時点で必要になるディレクトリ内のファイルに関する情報は、ディレクトリエントリに含まれるファイル名と inode 番号のみである。そこで、本方式ではダミー状態を設けることで、必要最小限のメタデータのみ持つファイルを許容し、メタデータの複製を個々のファイルアクセス時まで遅延させることができる。

複製中のファイルにおいて、複製状態の管理と、移行先と移行元ファイルの対応関係を保持するため、各スタブはファイルシステム中でスタブ情報 (図中 (a)(b)) を持つ。スタブ情報は、ファイルの複製状態 (“state”) と移行元サーバにおけるパス名 (“path”)、その他追加データを持つ。追加データには、プロトコル固有のメタデータを格納する。現在は NFS 固有の事項として取り扱う、単一のエントリ (“minsize”) だけが格納されている (本エントリの用途は 2.4.2 項参照)。スタブ情報は実装依存の方法で inode と関連づいており、現在の実装ではファイルの拡張属性に格納している。メタデータとデータの複製が完了すると、部分複製状態のファイルは通常ファイルとなり、不要となるスタブ情報は削除される (図中 (c))。なお、スタブとなるのは移行元サーバから複製するファイルのみで、クライアントが移行先サーバ上に作成したファイルは最初から通常ファイルとして格納されスタブにはならない。

ファイルは、上位のディレクトリの移動などに伴い、個々のファイルを直接対象とした操作を行わなくてもパスが変化することがある。このような場合においても、移行先サーバ上のスタブと移行元サーバ上の複製対象のファイルの関係を保持して以後の複製を可能とするため、スタブ情報中のパス名はフルパスで管理される。新規に作成されるスタブにおいて、このパス名は親ディレクトリのスタブ情報中のパス名を元に付与させる。例えば移行元サーバ上で “/A/B/” というディレクトリがあったとき、移行先サーバにおいてクライアントが親ディレクトリ “/A/” を “/X/” にリネームした場合を考える。この場合、ディレクトリ “/A/B/” は移行先サーバにおいては “/X/B” に配置される。しかし、このディレクトリ “/X/B/” のスタブ情報のパス名は、“/A/B/” を保持し続ける。ここで、このディレクトリ内にあるファイル “C” をスタブとして作成することを考える。このファイルは移行先サーバにおけるパス名は “/X/B/C” となるが、スタブ情報中のパス名は親ディレクトリのスタブ情報中のパス名に基づき “/A/B/C” が付与される。このパス名は確かに移行元サーバにおける対応するファイルの位置を指しており、以後クライアントが移行先サーバのファイル “/X/B/C” にアクセスする場合は、移行先サーバは正しく移行元サーバのファイル “/A/B/C” からデータを複製できる。

バックグラウンド複製では、前述の通りファイルシステムは未複製ファイル数のカウンタを保持する。複製開始時にはカウンタ値はルートディレクトリの分のみ “1” として設定される。ダミー状態のディレクトリがアクセスされると、移行元サーバの対応するディレクトリの情報を取得し、ディレクトリエントリに格納されたファイル数分だけダミー状態のファイルが作成される。その際、ファイル数分カウンタの値に加算される。反対に部分複製状態のファイルが通常状態になるとカウンタがデクリメントされる。カウンタの値が

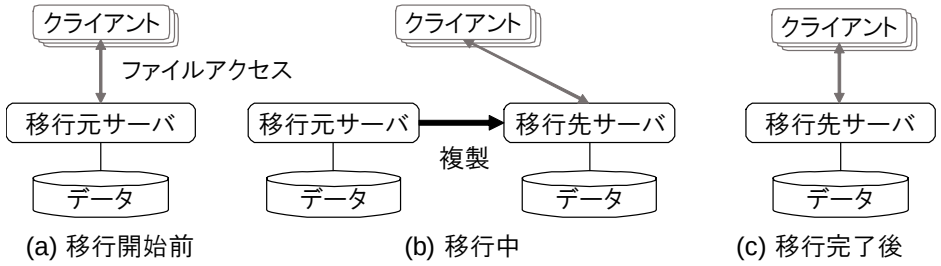


図 2.3: ファイルサーバの移行処理フロー

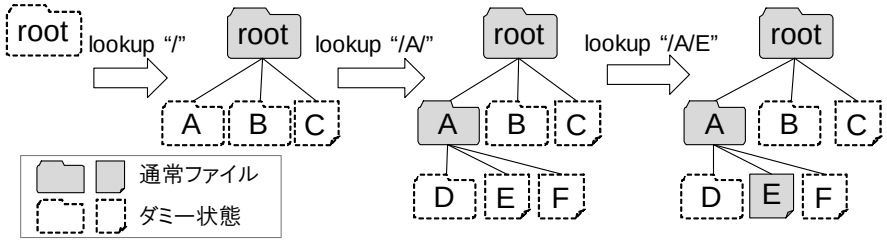


図 2.4: ファイルアクセス時のオンデマンド複製実行例

0 になった時点で、スタブ状態のファイルおよびディレクトリが存在しない、すなわち共有ディレクトリ全体の複製が完了したとみなせることになる。

移行中における不慮の電断やシステムのクラッシュに対応するため、このカウンタ値はファイルシステム中の不可視ファイルに記録され、更新時は毎回ディスクに書かれることになる。予期せぬリブートによりカウンタ値と実際のファイルの状態の不整合が疑われる場合、サービス起動時にディレクトリを探索してカウンタ値を再計算する。各ファイルに関するカウンタ値の変化は常に一方向であり、現状のファイルの状態から一意にカウンタのとるべき値が確定するため、再計算で整合性が取れた状態に復元できる。ディレクトリの探索は 2.3.2 節でも述べた通り時間がかかる処理である。しかしエンタープライズ用途のファイルサーバにおける予期せぬリブートはそもそも頻度の高い現象でないこと、またそもそも fsck や chkdsk のような既存のファイルシステムの整合性検証ツールは時間がかかるものであることから、この方針は許容可能なものと考える。

2.3.4 移行手順

図 2.3 にファイルサーバ移行の処理フローを示す。移行開始前は、クライアントは移行元サーバにアクセスしている状態である。この状態で、ストレージ管理者は移行先サーバの設置・初期設定を行う。次に、ストレージ管理者は移行元サーバについて全ファイルにアクセスできる権限をもつアカウントを作成する。この作業は移行元サーバに依存するが、商用のファイルサーバは通常バックアップ用に全ファイルを参照できるアカウントや権限を提供しており問題とはならない。続けて、ストレージ管理者は移行先サーバにおいてダミー状態のルートディレクトリを作成し、移行元サーバの複製対象の共有ディレクトリディレクトリを参照するようにする。この時点でオンデマンド複製が実行可能となるので、クライアントのアクセスを受け付ける準備が完了する。この作業は、共有ディレクトリ毎に個別に行うことができる。そのため、N 対 1 のサーバ移行も可能である。

接続先サーバの切り替えに際し、ストレージ管理者は移行元サーバの設定を変更してデータを参照のみ可

能とし、全クライアントは以後移行先サーバを remount するよう促す。よって、以後移行元サーバのデータ変更は許容されず、データの変更を行う場合は必ず移行先サーバに接続していなければならない状態となる。この時点ではまだ移行先サーバにはルートディレクトリに対応するダミー状態のディレクトリしか存在していないが、オンデマンド複製が有効なのでこの時点でクライアントは移行元サーバにあった任意のファイルパスを参照可能となる。図 2.4 に例を示す。クライアントがファイル “/A/E” にアクセスしようとすると、ルートディレクトリの lookup 処理が行われ、その際ルートディレクトリがダミー状態から通常状態に遷移するとともに、ルートディレクトリ下にダミー状態のディレクトリ “/A,” “/B,” “/C” が作成される。続けて、アクセス対象のパス解決のために “/A” が lookup され、“/A” が通常状態になると同時にその下に “/A/D,” “/A/E,” “/A/F” がダミー状態として作成される。最後に、クライアントは “/A/E” のデータ・メタデータを参照可能となる。

移行元サーバと移行先サーバの切り替えにおけるクライアントの remount 処理は不可避である。ファイル共有においてサーバはクライアントからの接続に対するセッション情報を内部保持しており、異機種である前提の下で、移行先サーバから移行元サーバのセッション情報を引き継げないためである。remount 処理の影響は、アプリケーションのファイルアクセス特性により異なる。単一ファイルを open し続けるアプリケーション、例えばオフィスファイルの編集ソフトや、データベースサーバにおいては、remount に伴いファイルの open/close を要する。アプリケーションによってはこの動作は実質再起動を必要とする。短時間ファイルを open してデータを読み込むケース、例えば画像ビューアや Web サーバは、remount 処理中のファイルアクセスが行えないだけで、影響はわずかである。この remount に伴う制限は不便であるが、セッション情報を引き継げない異機種間移行においては必要な処理である。また、remount が完了すればアプリケーションは速やかに再起動及び処理継続が可能となる。

サーバの切り替え後、クローラは移行先サーバの共有ディレクトリ全体の探索を行い、全ファイルのデータ・メタデータの取得を行う。その後、スタブ数カウンタが 0 になるとクローラは終了して複製手順も完了となる。この時点で移行は完了となり、ストレージ管理者は移行元サーバを停止可能となる。

2.4　プロトコル固有の対応項目

本節では、実環境においてファイルサーバ移行を行う上で必要な、プロトコル固有の動作とそれに対する対応について述べる。

2.4.1　アクセス権および ID 管理

ファイルアクセスプロトコルによって、アクセス権やユーザー ID の管理方法は異なる [57]。そのため、複数のプロトコルに対応するファイルサーバでは、これら複数の管理方法に対し内部的にアクセス権のマッピングを取り、双方向に変換を行って対処する。しかしこの内部のマッピングはサーバ内部の実装依存であって外部から取得することができず、標準的なマッピングルールもない [58, 59] ため、移行先サーバにおいて移行元サーバ内のマッピングの再現はできない。ユーザー ID についても同様で、ディレクトリサーバのない環境では、NFS における uid/gid と SMB における sid はファイルサーバ内で対応付けが管理され、外部からその対応を取り出せない。

幸い、多くのケースで実際に使用されるアクセス権の設定は限定的であり、細かなアクセス権のマッピングの差異が実用上で問題となることはまずない。そこで、本移行手法においては、NFS と SMB 両方をサポートするファイルサーバの移行では SMB を用いることを推奨する。これは SMB のアクセス権の設定が NFS より細粒度で行われるため、SMB のアクセス権情報を移行元サーバから移行先サーバに複製し、移行

先サーバ内で NFS のアクセス権のマッピングを取るとおおよそ対応付けられるためである。ID も同様で、sid の方が ID の取りうる値の範囲が広いため、SMB で sid を複製したうえで、uid/gid を内部で対応付けるのが良い。なお、エンタープライズ環境では通常ディレクトリサーバが運用されており、ID の対応はディレクトリサーバで行われるためファイルサーバ内での対応は多くのケースで不要である。

2.4.2 ファイルの truncate 処理

2.3.3 項で述べた通り本手法においては、スタブファイルにおいて複製済みデータ領域の管理は、ファイルのブロック領域未割り当てブロックの情報を用いて行う。ただし、NFS の truncate 処理を行う場合、ファイルサイズの拡大を指定すると末尾に新たな未割り当てブロックが生じる。このブロックは、新規に作成されたものか、複製未実施領域を示すものかこのままでは判断がつかない。そこで、本方式ではスタブ情報に埋め込んだ”minsize” フィールドの値を用いる。この値は、サーバ移行開始以降におけるファイルサイズの最小値を示し、移行開始後の truncate によりサイズが縮小した際に更新される。このサイズより後ろのオフセットにある未割り当てブロックは、移行開始後に生成されたものであることがわかるため、その領域へのアクセス時にデータ複製を要しないと判断する。

NFS のより新たなバージョン 4.2 では、ホール生成命令が追加された [60]。この命令では、ファイルの末尾以外の位置に未割り当てブロックを生成できるため、“minsize” フィールドを用いた未複製ブロックの判定は行えない。この対策としては、ブロックの extent 管理とは別に複製/未複製領域を管理するビットマップ等を保持することが考えられる。しかしこれは複製完了までの期間とはいえ、ファイルのデータアクセス時に毎回ビットマップの参照を要するため、性能低下の要因になる。別の対応方法としては、ファイルシステムのブロック管理の処理を改造し、未割り当てブロックに、未複製か複製後のホール生成のいずれか生成理由を格納するフラグを追加することである。こちらは元々備えるブロック管理の処理に追加処理を組み込むため性能低下はほとんど生じないが、ファイルシステムの互換性は維持されなくなる。現状今回実装対象としたファイルサーバにおいて NFS バージョン 4.2 は非対応のため、ホール生成命令への対応も未実施としている。

2.4.3 ハードリンク

2.3.3 項の項で述べた通り、スタブ管理部は移行先サーバ内のスタブと移行元サーバ内のファイルの対応関係をパス名によって保持する。ここで、移行先サーバ内において移行元サーバ内のハードリンク関係を認識しない場合、移行元サーバ内で互いにハードリンクの関係にある複数のパスを持つファイルが、それぞれ個別に移行先サーバに複製され、ハードリンクの関係も失われてしまう問題が生じる。この問題の対応のため、移行元サーバから NFS プロトコルでファイルを複製する際、ファイルの inode 番号とリンクカウントを確認する。リンクカウントが 2 以上の場合、そのファイルは別のパス名でハードリンクされたファイルを保持している可能性がある。しかし NFS プロトコルでは、直接ハードリンクされたファイル群を特定するインターフェースを持たない。そこでこのような場合、移行先サーバの隠しディレクトリ中に inode 番号から生成された名前を持つ一時ファイルを作成し、本来ファイルを複製するパスとその一時ファイル間でハードリンクを張る。もし同一の inode 番号を持つ他のパス名のファイルが複製対象となった場合、同様に inode 番号から一時ファイルのパスを生成することで、移行先サーバにおいてハードリンク関係を保つべきファイルが特定できる。これらの一時ファイルは、複製完了後に削除される。

2.4.4　アクセス遅延への対応

提案方式では、多数のファイルを保持するディレクトリに接続先切り替え後初めてアクセスするとき、多数のスタブファイルを生成するために応答時間が長くなる。NFS バージョン 3.0 はステートレスなプロトコルであるため、無応答の要求の状態を確認する手段もなく、応答時間が長すぎるとクライアントがファイルアクセスのタイムアウトエラーと判断し、アプリケーションの停止を引き起こしてしまう。この事態を回避するため、本手法では NFS で定義される NFS3ERR_JUKEBOX エラーコードを利用する。このエラーコードは元々テープ装置等データ取得に時間がかかる記憶メディアの利用を想定して定められたもので、このエラーコードを受けたクライアントは、一定時間後に再送を行う。Linux の標準的な NFS クライアントは 0.7 秒のタイムアウト猶予を持ち、NFS3ERR_JUKEBOX を受け取ると 5 秒後に再送を試みる。そこで、本構成におけるファイルサーバは 0.5 秒以内にスタブ作成が完了しない場合、一旦 NFS サーバ機能が NFS3ERR_JUKEBOX を返す。

NFS の場合、パスのルックアップ処理はクライアントが実行するため、深いディレクトリに格納されたファイルにアクセスする場合、1 ディレクトリ毎に NFS3ERR_JUKEBOX 応答が発生しファイルへのアクセス時間が長くなることが考えられる。しかしながら、NFS クライアントには無応答ではない状態を保持できるので、タイムアウトによるエラーは生じない。この時間については、2.6.2 項にて評価する。

2.4.5　画像サムネイル

Windows のエクスプローラーは、ディレクトリを開くと中にある画像ファイルや文書ファイルのサムネイル画像を作成しようとする。この動作は、ファイルデータの取得のためオンデマンド複製を引き起こす可能性がある。これにより、多数の画像ファイルを格納するディレクトリを開く際、クライアントへの応答時間が長時間化する要因となる。エクスプローラーでは、Windows のファイルシステムで定義されるオフライン属性が付与されているファイルに対し、サムネイル画像作成を抑止する。この属性は、対象ファイルがテープ等の即座に参照できない場所に格納されていることを示すための属性である。そこで、本実装においては SMB でスタブファイルのファイル属性を参照するとオフライン属性が付与されるようにし、サムネイル画像作成を抑止している。

2.5　実装

本節では提案方式の実装について述べる。図 2.5 に移行先サーバのソフトウェア構成を示す。本実装では、ユーザー空間のプログラムとしてファイル共有サービスプログラム、クローラ、ファイル取得プログラムの三つのプログラムが動作する。加えて、カーネル内のファイルシステム上でスタブ管理部が動作する。

2.5.1　ファイル共有サービスプログラム

ファイル共有サービスプログラムは、クライアントに対するファイル共有機能を提供する。本実装においては、既存の OSS のプログラムに、2.4 節で述べるプロトコル固有の処理への対応を追加して利用した。NFS による共有には、Linux カーネル 2.6.30 が持つ nfsd サブシステム、SMB による共有には Samba 4.1.0 を用いた。

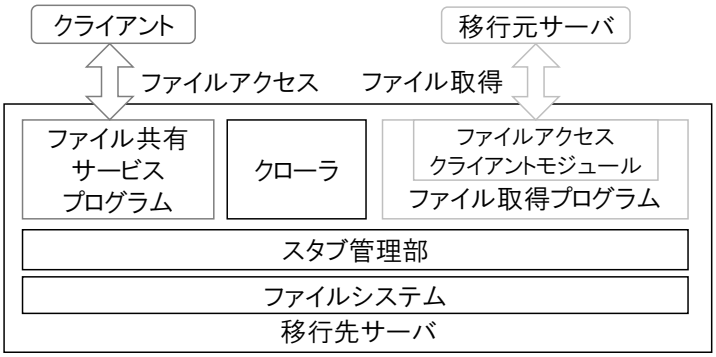


図 2.5: ソフトウェア構成

2.5.2 ファイル取得プログラム

ファイル取得プログラムは、スタブ管理部の要請に応じて移行元サーバからデータ・メタデータを取得する。ファイル取得プログラムは C 言語で記述されたユーザー空間で動作するプログラムである。カーネル内で動作するスタブ管理部とは、UNIX domain socket 上に構築した双方向キューを介して通信する。スタブ管理部は、データ・メタデータを移行元サーバから取得したい場合、このキューに対象ファイルのパスと取得したいメタデータ・データオフセットを含む要求パケットを追加する。ファイル取得プログラムはこのキューに要求パケットが追加されると、その内容に応じて移行元サーバのデータ・メタデータを取得してファイルのデータ・メタデータを更新したのち、スタブ管理部に応答する。

ファイル取得プログラムは、移行元サーバとの通信を行うために必然的にネットワーク通信の応答待ちが生じる。そのため、効率よくデータ・メタデータが取得できるようスレッドプールを備えて複数の取得要求を並列に実行できるようにしている。また、一度に取得するデータサイズは必ずしもクライアントの要求サイズとは一致しないようにしている。大きすぎるデータ取得サイズは、取得完了までに時間がかかり応答時間が伸びてしまう。反対に小さすぎるデータ取得サイズでは、同じファイルのデータを取得するのにかかる通信回数が増えてしまいやはり効率が良くない。一般に NFS や SMB における単一アクセスのデータ送受信サイズの上限が 1 MB であることから、本実装ではその数回分として 10 MiB 単位で取得を行う。この値における大容量データのシーケンシャルアクセス性能については 2.6.4 節にて評価を行う。

ファイル取得プログラムは移行元サーバへのアクセスのため、プロトコル別のファイルアクセスクライアントモジュールを用いる。NFS クライアントとしては、Linux カーネルの持つ NFS クライアントを利用し、SMB クライアントとしては、Samba が提供する libsmbclient ライブラリを用いる。移行が開始すると移行元サーバは読み込み専用となるため、移行元サーバのデータは更新されない。その前提のもとで、クライアントの各種パラメータは複製性能が最大となるように設定した。NFS クライアントにおいては、1 リクエストのデータ通信サイズはプロトコル上限の 1 MiB に設定し、またクライアントメタデータキャッシュは 1 週間としている。SMB においても、ファイルの転送効率のよい SMB3.0 を利用し、非同期並列転送機能を用いて転送を高速化している。

ファイル取得プログラムは、ファイル取得のプロトコルとして NFS バージョン 3 または SMB3.0 を用いる。これらはファイル共有プロトコルの業界標準であることから、市場にあるほぼすべてのファイルサーバから移行できることとなる。ストレージ管理者は NFS バージョン 3 と SMB3.0 いずれのプロトコルを用いるかは選択可能であるが、プロトコル固有のメタデータは選択したプロトコルの物しか複製できない。その

ため、移行元ファイルサーバで主として利用されたプロトコルを用いるのが好ましい。

2.5.3　クローラ

クローラはバックグラウンドプロセスとして動作するユーザー空間のプログラムである。クローラは移行先サーバ上にマウントされた移行元サーバのファイルシステムを探索し、内部のファイルを POSIX インターフェースを用いてアクセスする。唯一 POSIX に含まれないインターフェースとして、スタブ管理部にファイルがスタブかどうかを問い合わせる `ioctl` システムコールを利用する。

2.3.2 節で述べた通り、クローラはファイルのデータアクセスを固定長ブロック単位で行う。そのためデータの複製や中止の単位はこのブロックサイズとなる。実装にあたり、このブロックサイズはファイル取得プログラムの動作に合わせ 10 MiB とした。このサイズは 1 Gbps の通信路でデータを行う場合にも 1 秒以下で転送が完了することから、ストレージ管理者が中断を指示する場合にも妥当な応答時間で中断できる。

2.5.4　スタブ管理部

スタブ管理部は以下の三点の実現に向けた実装を行った。まず、エンタープライズ用ファイルサーバでは多数のクライアントによる負荷がかかるので、移行が完了した後に性能オーバヘッドが残るような仕組みは好ましくない。次に、スタブ管理部は NFS や SMB など複数のファイル共有プロトコルを介した利用だけではなく、ファイルサーバ内のプログラムによるローカルアクセスにも対応できなければならない。これはファイルサーバにおいてはクライアントアクセス以外にも、バックアップ、データ圧縮、ウイルスチェックなどの用途でローカルアクセスが行われるためである。さらに主要ファイルシステムと互換性の取れないディスク形式は、将来的に移行先からの再移行が困難となるため好ましくない。

実装にあたりいくつか選択肢があるが、上記三点を踏まえそれぞれ決定した。まず、スタブ管理部の実装にはユーザー空間とカーネル空間の選択肢がある。ユーザー空間の場合、機能の独立性が高く実装・保守が容易で、今後のカーネルバージョンアップへの追随も容易である。しかし、ユーザー空間実装による性能オーバヘッドは無視できない。例えばユーザー空間でファイルシステムを実現する手段としては Filesystem in Userspace (FUSE) が利用可能であり、単一ディスクを用いる構成では良い性能が出るという先行研究がある [61]。しかしこれはあくまで応答時間に関する評価であって、CPU の利用率は高くなる (単一ディスクでありながら 1CPU の 31% を消費する) ため、数十〜数百ディスクを備えるエンタープライズ環境では許容できない。そのため今回はカーネル空間でスタブ管理部を実装した。しかしながら、ユーザー空間は性能が十分であるならば有力な候補である。現在 Linux において高速なユーザー空間ファイルシステムを実現するフレームワークである Zero-copy User-mode File System (ZUFS) [62] が提案されている。このようなフレームワークが実用可能となった暁には再検討の余地がある。

次の選択は、スタブ管理部をカーネル内部を直接変更して組み込むか、カーネルモジュールとして独立して実装するかという点にある。理想的には、本機能はカーネルの付随機能なのでモジュールとして独立させた方がバージョンアップへの追随も考慮すると望ましい。しかし、本実装ではファイルのダミー状態を実現するにあたり、カーネル内部の変更が不可欠だった。図 2.2 で示した通り、スタブ管理部の実現には一部メタデータ項目が未充当の inode を保持しなければならない。ただし、Linux カーネルにおいてメモリ中に保持された inode のメタデータキャッシュは VFS 層のディレクトリ管理と密接に関係しており、カーネルの中心的な処理となっており外部モジュールの範囲の処理では実現できない。そのため、今回の選択ではカーネルの VFS 層を直接変更し、スタブ管理層を導入した。

三つめの選択は、VFS 層変更を要する機能以外を、どこに実装するかという点である。今回は VFS 層の

表 2.1: 使用機器一覧

	クライアント	移行元サーバ	移行先サーバ
CPU	Intel Xeon E5-2430	Intel Xeon E3-1231 v3	Intel Xeon E3-1231 v3
DRAM	32 GiB	32 GiB	32 GiB
データディスク	無し	SATA SSD × 6	NVMe SSD × 2
OS (NFS 測定時)	Ubuntu Server 18.04.1 LTS		改造済み Debian(Linux 2.6.30.1)
OS (SMB 測定時)	Windows Server 2016		on VMWare ESXi 6.5

中心部分を直接改造する他に、上述の FUSE や Stackable File System を加えるという手段もある。また、ファイル共有サービスプログラム側に改変を行うという手段も考えられる。例えば OSS の選択ファイル共有サービスプログラムである NFS-Ganesha [63] や Samba は、プラグインモジュールによる機能拡張が可能なつくりになっている。しかし、ファイルサーバは複数のプロトコルを備える必要があることと、ローカルファイルアクセスにも対応できなければならないことから、本実装では一プロトコルにしか対応できないファイル共有サービスプログラム側の改変は行わないこととした。

これらの選択の結果、スタブ管理層はカーネルの VFS 層内部において、XFS ファイルシステムの上に構築した。その際スタブ情報はファイルの拡張属性部分に格納することとした。拡張属性は現行の主要ファイルシステムがほぼ備えている機能であることから、提案手法は他のファイルシステムにも容易に導入可能であると考える。

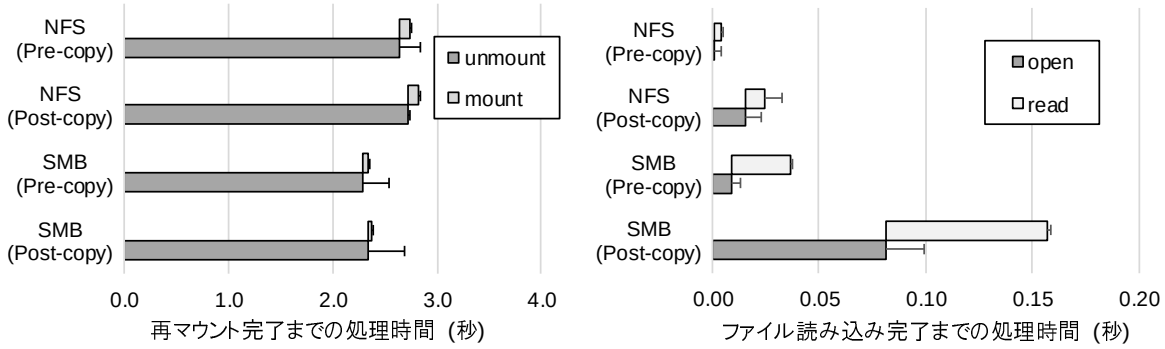
2.6 評価

本節では提案手法における性能評価を行う。評価対象は、移行時におけるダウンタイム、複製を含めた総複製時間、移行後のファイルアクセスにおけるオーバヘッド、移行中のファイルアクセスにおける応答時間である。

2.6.1 環境構築

表 2.1 は実験に使用したクライアントおよびサーバの機器構成を示す。測定にあたり、転送スループットの測定実験 (2.6.4 項) では等しい構成の 3 台のクライアントを用いた。その他、ダウンタイムの測定実験 (2.6.2 項) 及び応答時間の測定実験 (2.6.5 項) ではクライアントは 1 台のみ使用している。提案方式を実装した新サーバにおいては、改造した Linux カーネル 2.6.30.1 を含む専用の Debian ディストリビューションを用い、NFS と SMB 両方の実験において用いた。この Debian は、最新の NVMe SSD の対応ドライバを持たないため、仮想マシンとして動作させハイパーバイザによる SATA SSD エミュレーションを介して動作させている。クライアントおよび移行元サーバとしては、NFS の測定実験には Linux 環境 (Ubuntu Server 18.04.1 LTS)、SMB の測定実験には Windows 環境 (Windows Server 2016) を用いている。これらは一般的な範囲の性能チューニングは行っているが、コードの改造は行っていない。

測定対象のプロトコルには、クライアントアクセスとファイル複製双方で NFS バージョン 3 と SMB3.0 を用いた。NFS バージョン 3 は POSIX 互換のメタデータを持ち、ステートレスであるシンプルなプロトコルである。一方 SMB3.0 はステートフルなプロトコルであり、アクセス権管理情報や DOS アクセス時の属性、ファイルスタンプなど NFS に比べ多様なメタデータを持つ。そのため、一般的に NFS より SMB の方がファイルアクセス時に複雑な命令群が発行される [59]。



(a) ダウンタイム　　(b) 初回アクセスの応答時間

図 2.6: 最小構成 (深さ 1, 幅 10) におけるダウンタイムと初回アクセスの応答時間

各計算機は、ファイルアクセス用に 10 Gb Ethernet で接続されているほか、移行元サーバと移行先のサーバでは追加で 10 Gb Ethernet の直結接続がある。そのため、クライアントアクセスとデータ複製が帯域を共有・干渉することはない。

以降に示す測定結果はいずれも 3 回以上実施し、その平均値を取っている。

2.6.2　サーバ切り替え時の待ち時間

提案手法において、サーバの切り替えに伴い生じるクライアントの待ち時間は、サーバのダウンタイムと初回アクセス応答時間からなる。サーバのダウンタイムとはクライアントがサーバにアクセスできない時間とし、初回アクセス応答時間とはサーバ切り替え後に個々のファイルに初めてアクセスするときの応答時間とする。以下各項目について見ていく。比較対象としては Pre-copy による複製を取り上げる。

これらの待ち時間の測定にあたり、以下に示す実験を行った。まず、移行元サーバに巨大なファイルを複数作成しておく。次に、自製のテストプログラムをクライアントで動作させる。このテストプログラムは、移行元サーバにおけるファイルを open し、そのデータを先頭から順に read していく。その際クライアントのメモリ搭載量より多くのデータを読むことで、クライアント内のページキャッシュの影響を排除する。続けてクライアントにてサーバ切り替えを行う。テストプログラムはアクセスしているファイルを close し、続けてクライアントによる移行元サーバの unmount と、移行先サーバの mount を行う。最後に、テストプログラムは同じパスのファイルを開き、close 時の続きのデータを read する。unmount と mount 対象のディレクトリは同じであり、テストプログラムは同じパスをアクセスすることで、mount 後に移行先サーバを介して同一ファイルにアクセスすることができる。テストプログラムはシングルスレッドであり、mount 後初回 read の応答時間は他の影響を受けずに正確に取ることができる。

移行先サーバにおいて各ファイルに対する初回 open 要求に対しては、そのファイルパスを構成する祖先の各ディレクトリの内容をすべて取得し、ディレクトリエントリの内容をダミー状態のスタブファイルとして作成しなければならない。open 要求への応答時間にはその時間も含まれる。そのため、open 要求への応答時間はディレクトリツリーの構成やアクセス対象のファイルの位置の影響を受ける。これらを規定するパラメータとして、ファイルの深さ (ルートディレクトリから対象ファイルまでに辿る必要があるディレクトリ数) とディレクトリの幅 (個々のディレクトリの直下に保持されるファイル・ディレクトリ数) を定める。以下の測定では、ファイルの深さとして 1,2,4,8,12,16、ディレクトリの幅として 10,100,1000 と複数のパラメータで測定する。

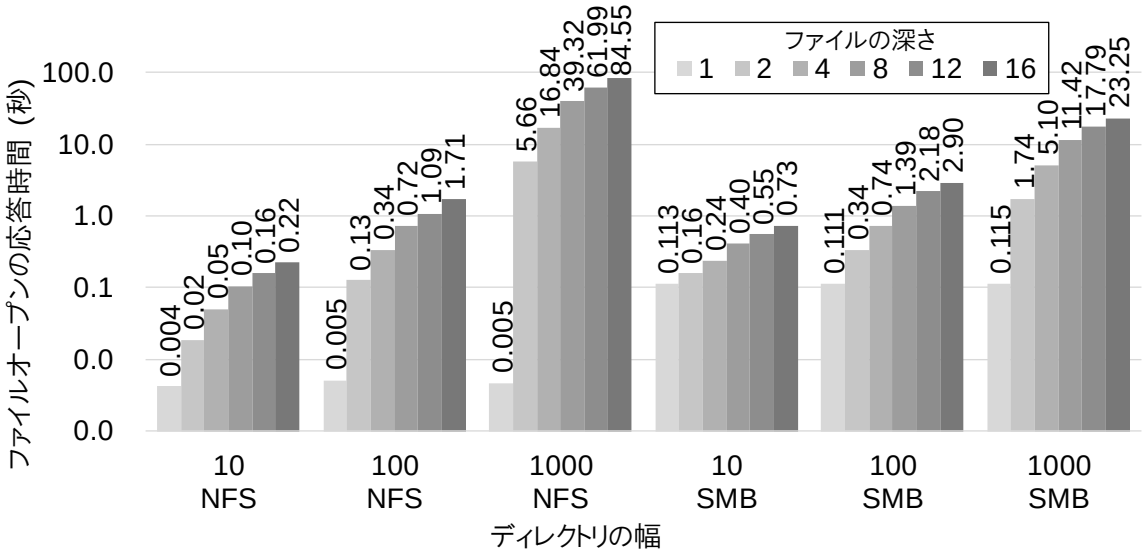


図 2.7: ディレクトリ構成変更時の初回ファイルアクセス応答時間

図 2.6(a) は最小構成 (ディレクトリの深さ 1、幅 10) におけるダウンタイム (クライアントの unmount 及び mount)、(b) は初回アクセスの応答時間 (open および read) で分類したものである。両棒グラフ、上半分は NFS、下半分は SMB の測定値である。いずれも上側が Pre-copy、下が Post-copy の測定値である。エラーバーは複数回測定における標準偏差を示している。

ダウンタイムについて、移行元サーバの unmount 時間は NFS で 2.3 秒、SMB で 2.7 秒であった。この時間はクライアント側キャッシュの解放によるものであるため、プロトコルや移行方式の影響はうけない。その後の新サーバの mount 時間は NFS で 0.1 ms、SMB で 0.04 ms と非常に短い。いずれもプロトコルによって若干の差があるものの全体の時間と比べれば無視できる。すなわち、ダウンタイムは概ね 3 秒以下と言える。Pre-copy の場合のダウンタイムは、データセットやワークロードによって大幅に異なるため一概に評価できないが、最後の増分コピーに依存するためより長くなる。

初回アクセスの応答時間に関しては、提案手法におけるファイル open の時間はオンデマンド複製によるメタデータ取得を要するため、Pre-copy よりも NFS で 15 ms、SMB で 72 ms 長くなった。SMB では Windows の標準機能である Windows Defender サービスの動作によってより長くなる傾向がみられた。Windows Defender はマルウェア検出のためにアプリケーションのファイル open 要求時にファイルの先頭と末尾を読み込む動作を挿入する。オンデマンド複製では、これによりベンチマークプログラムが読む対象ではないブロックの取得動作が生じ、応答時間を長くしていた。read 要求に関しても提案手法の方が長く、平均で NFS で 6 ms、SMB で 48 ms 長くなっていた。こちらもオンデマンド複製によるデータ取得動作による。しかしいずれも提案手法の方が時間を長くするとはいえ、合計時間は NFS で 102 ms、SMB で 164 ms である。これはダウンタイムに比べると 1 桁以上小さい値であり、相対的に無視できる範囲と言える。

提案手法の応答時間へのオーバヘッドの支配的な項目は深くて幅の広いディレクトリにあるファイルの open にかかる時間である。深さや幅の定量的な影響を計るため、深さや幅を変えながら測定した結果を図 2.7 に示す。この図では棒グラフを六つのグループに分けており、左側三つが NFS、右側三つが SMB に対応する。それぞれ、ディレクトリの幅が 10, 100, 1000 の場合を示す。グループ内の各値は、ファイルの深さを 1, 2, 4, 8, 12, 16 で変えた場合の値に対応する。縦軸が対数軸である点に留意すること。

ディレクトリの深さが 1 である、すなわちアクセス対象のファイルがルートディレクトリの直下に置かれている場合、ディレクトリの幅は open 時間に影響しない。これは mount 時点でルートディレクトリ内の

表 2.2: 移行時間評価の使用データセット

	総容量	ユーザー数	ディレクトリ数	ファイル数	ファイルサイズ	深さ	用途
(a)	677 GiB	200	109 K	802 K	844 KiB	13.1	研究開発データ
(b)	387 GiB	30	47 K	353 K	1,097 KiB	8.1	オフィス文書
(c)	211 GiB	30	743	8 K	26,324 KiB	6.1	メディア

ファイルのスタブが生成されるため、open 時間に計上されないためである。それ以上の深さである場合、open にかかる時間はおよそ (幅) × (深さ) と線形の関係となる。これはこの値が作成するスタブファイルの数に一致するため妥当な結果である。全体的に NFS の方が応答時間は短い。これは NFS の方が単純なプロトコルでありディレクトリエントリの一括読み込み命令 (READDIRPLUS) の転送量が小さいため、スタブ作成に必要な移行元サーバと移行先サーバの通信量が少なくて済むためである。

ただし、NFS におけるディレクトリ幅 1000 の測定値は、他の結果と異なり NFS の方が SMB に比べ長い応答時間を示す。これはディレクトリ参照時のオンデマンド複製により 1000 個のスタブを作るために時間がかかった結果、サーバ側が 0.5 秒経過した時点で NFS3ERR_JUKEBOX エラーコードをクライアントに返したためである。Linux の NFS クライアントはこのエラーを受け取ると 5 秒後に再送を試みるため、実際のスタブ作成にかかる時間が 5 秒以下であったとしても、ディレクトリ毎に 5.5 秒かかることになる。再送までの待ち時間をチューニングすることで最終的にかかる時間を削減することは可能であるが、クライアント側がディレクトリのオンデマンド複製にかかる時間を把握している必要があり現実的には行えない。むしろ、時間がかかっても再送動作が正しく動作することでタイムアウトによるファイルアクセスエラーが生じない点を考慮すべきである。加えて、通常ディレクトリに格納するファイルは数十個オーダーであり、全ディレクトリに 1000 個ファイルを格納するような構成は稀である [64]。また、そのような極端な構成であったとしても NFS で最長 84.55 秒、SMB で 23.25 秒であり、通常 OS やアプリケーションが設けるタイムアウト時間と同程度の数十秒オーダーの時間を達成している。

2.6.3　移行にかかる総時間

本節では、実際に企業内ファイルサーバのデータセットを用いて実利用を模した環境において、移行にかかる時間を評価する。用いたデータセットは、数百名規模の社員が約 10 年使用してきたファイルサーバを元にした。このファイルサーバはオフィス向けの共有ディレクトリおよびホームディレクトリ用途で用いられており、ユーザーの大半は Windows 搭載 PC から 1 GbE のネットワークを介し SMB で利用している。そのため、格納されたデータ及びディレクトリ構成は手作業で作られており、プログラムで機械的に生成されたものではない。このファイルサーバは 2400 以上の共有ディレクトリを持ち、80 TiB 以上のデータを備える。ファイルサイズやディレクトリの深さの分布は、先行研究における調査結果 [65, 64, 66] とおおむね一致しており、平均ファイルサイズは 724 KiB で、ファイルサイズ分布はパレート分布モデルに従う。ファイルの深さの平均値は 9.62 である。これらの特性から、このファイルサーバは一般的なファイルサーバを反映したものとみなすことができる。

本実験において、表 2.2 に示す代表的な三つの共有ディレクトリのデータを用いた。データセット (a) は、研究開発部門の共有ディレクトリである。この共有ディレクトリは研究開発で用いるデータを含むため、実験データのアーカイブである巨大ファイル (拡張子は.zip や.tgz) から、ソースファイルなどの小サイズファイル (拡張子は.c, .h, .bmp など) を万遍なく含む。ファイルサイズの平均も、ファイルサーバ全体の平均

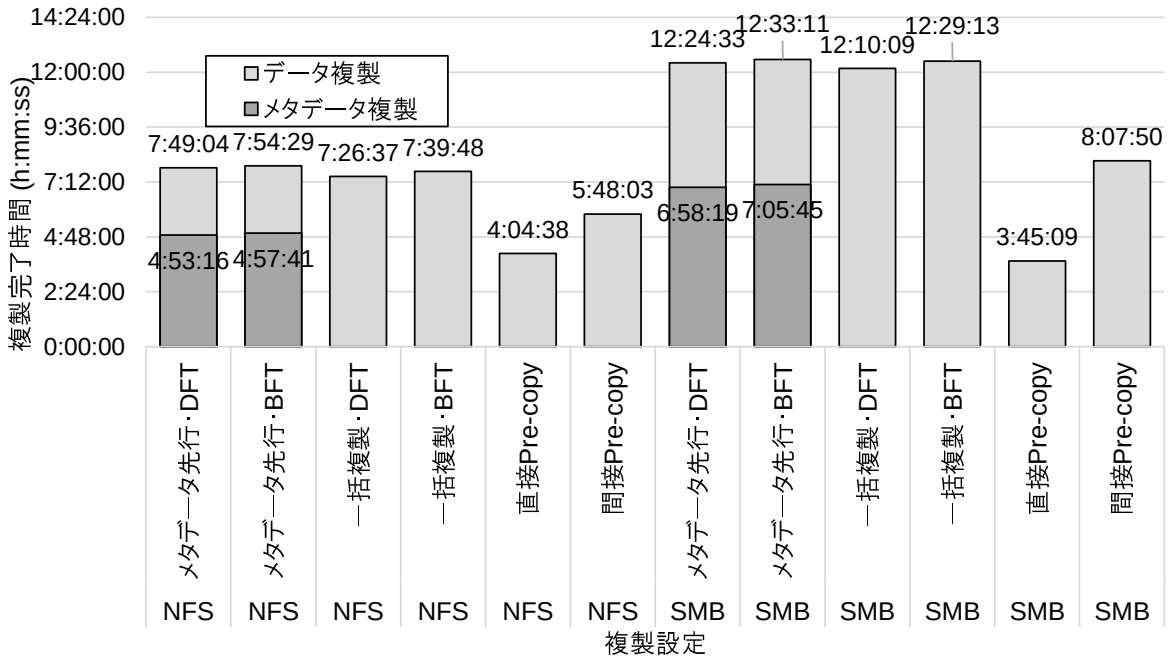


図 2.8: データセット (a) における総複製時間

と同程度である。ただし、このデータセットにおけるファイルの深さの平均値は、他のデータセットより大きい。これは、組織構成を反映しているためと考えられる。この共有ディレクトリはユーザー数が多く、内部では複数の研究グループがある。そのため、各研究グループがディレクトリを個別に作成し、深さを増す傾向がみられる。データセット (b) は総務部門の共有ディレクトリである。そのためこちらは、オフィス文書 (拡張子.xls, .ppt, .pdf など) が中心であることから、ファイルサイズのばらつきが小さい。データセット (c) は、社内イベントのデータを格納する共有ディレクトリである。この共有ディレクトリは写真や動画 (拡張子.mpg, mp4, .jpg など) を多く格納しており、平均ファイルサイズが他のデータセットに比べ極端に大きい反面、容量の割にファイル数は極めて少ない。

実験に際してはこのファイルサーバ上で 'ls -lR' コマンドを実行してディレクトリ構成とファイル名長・ファイルサイズを取得し、その構成を実験用の移行元サーバにて再現した。そのため厳密なディスク上のファイル配置やメタデータの設定は実稼働中のサーバとは異なる。

各データセットについて、6 種類の設定に対し複製時間を測定した。4 種類は、提案手法のデータ探索戦略に関してディレクトリ探索順 (DFT, BFT) およびデータの取得順 (メタデータ先行, 一括複製) の組み合わせを行った。残り 2 種類は Pre-copy で、NFS では rsync、SMB では robocopy を使いサーバ間でファイルを複製する。直接 Pre-copy は移行先サーバでこれらのコマンドを直接動作させて複製し、間接 Pre-copy は別途クライアントを両サーバに mount させ、そこで rsync, robocopy を動作させる。間接 Pre-copy は移行元サーバ・移行先サーバいずれも専用のプログラムを必要としないことから、ストレージベンダにおいても推奨する手法となっている [67, 68]。本実験では、提案手法においてはバックグラウンド複製のみを用いてデータ複製を行う。そのためクローラが共有ディレクトリの探索を一巡するまでの時間をもって複製時間とみなす。

図 2.8、図 2.9、図 2.10 はそれぞれデータセット (a)(b)(c) の複製時間を示している。棒グラフにおいて左半分の 6 項目と右半分の 6 項目がそれぞれ NFS と SMB の結果に対応する。6 項目の内訳は、左二つがメタデータ先行複製、中二つが一括複製時の結果でそれぞれ左からディレクトリ探索順が DFT と BFT の

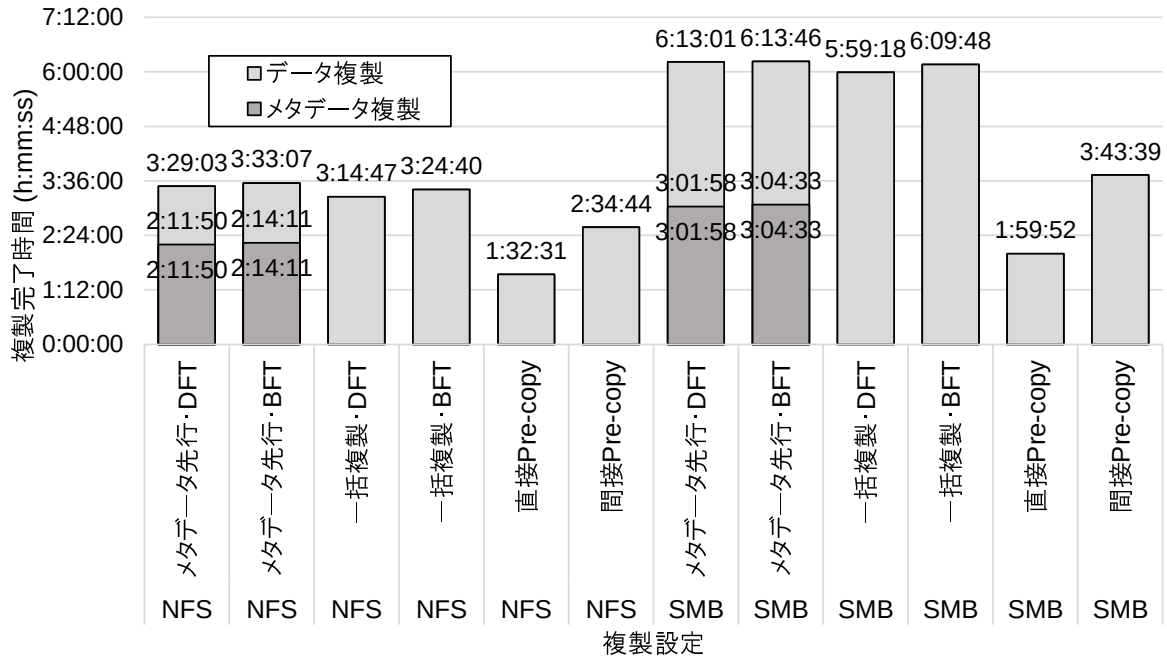


図 2.9: データセット (b) における総複製時間

場合を示している。右二つは直接 Pre-copy と間接 Pre-copy の結果である。メタデータ先行複製については全複製時間においてメタデータ複製分の占める時間も示している。

提案手法はほとんどのケースで Pre-copy よりも遅い。その増分は直接 Pre-copy と比較すると、最悪値において NFS で 2.30 倍、SMB で 3.39 倍である。間接 Pre-copy と比較すると NFS で 1.38 倍、SMB で 3.69 倍である。これは提案手法が多様なメタデータを複製しようとするため通信回数が増えることと、オンデマンド複製用途に向け小サイズのデータブロックを用いており、通信転送速度の点で Pre-copy に比べ効率が悪いことによる。ただし提案手法のダウンタイム軽減効果を考えると、この総複製時間の増分は許容範囲と考える。

測定結果を個別に見ていくと、総複製時間はデータセット、プロトコル、データ探索順戦略によって変化している。ほとんどのケースにおいて、SMB は NFS より時間がかかっており、データセット (a) で 1.75〜1.84 倍、データセット (b) で 1.59〜1.63 倍、データセット (c) で 2.77〜2.87 倍である。これは SMB のメタデータが NFS より豊富であることとステートフルで通信回数の多いプロトコル仕様による。SMB のメタデータは ACL 等サイズが大きいため、取得には NFS より多くの命令を発行する必要がある。加えてデータの取得においても、SMB の場合はデータ参照前後に open, close 命令を要する等通信回数が増加する。さらに、提案手法ではオンデマンド複製の実装にあたり要求毎に open/close を行うため、大サイズファイルの複製時に通信回数が増加する。一方 NFS ではメタデータとデータがそれぞれ 1 命令で取得できる。これらの特性が NFS に比べ SMB が遅い要因となっている。

提案手法において、データ探索戦略であるディレクトリ探索順やデータ取得順は、総複製時間に対し影響が少ない。メタデータ先行複製はディレクトリ探索を 2 回行うことで一括複製より時間が増える傾向にあるが、それでも最長で 7.3% の増加にとどまり、ほとんどは 5.0% 以下の増分である。BFT は DFT より総複製時間が長くなる傾向があるが、その差分も最大で 4.8% である。傾向として BFT と DFT の差は一括複製で大きくなる。これはデータ複製の過程で行うことでファイルサーバ内のメモリがページキャッシュとして利用されやすくなり、メタデータキャッシュの追い出し圧力が高まるためである。BFT においては、

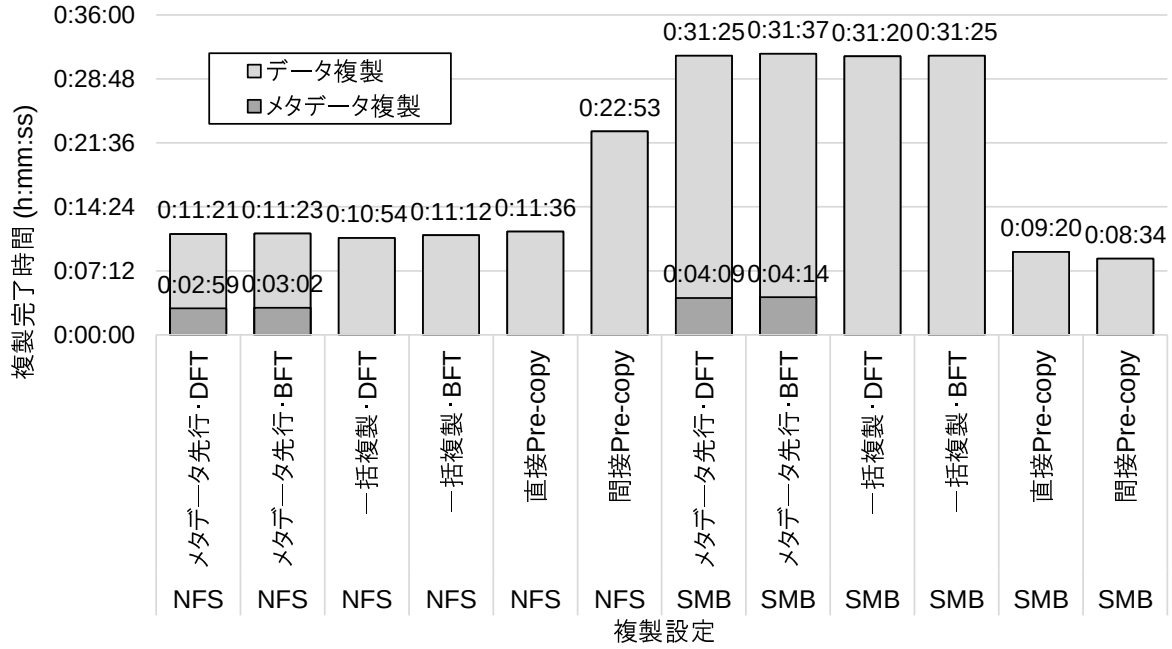


図 2.10: データセット (c) における総複製時間

ディレクトリエントリや inode のメタデータを多くメモリ中に保持しておきたいためこの動作は不利になるためである。

データセット (c) においては、総複製時間に占めるメタデータ先行複製にかかる時間の割合が短い。データセット (a)(b) では NFS で最小 62.4%、SMB で 48.5% の時間占めていたが、データセット (c) は NFS で 26.3%、SMB で 13.2% に留まる。これはデータセット (c) では平均ファイルサイズが大きく、反対にファイル数が少ないためである。この結果はデータ探索戦略による性能差の小ささにも表れている。

総じて、データ複製戦略における総複製時間への影響は小さい。そのため、ストレージ管理者がデータ複製戦略を定めるときは、他の要求事項、例えばファイルアクセスの応答時間を基準とすることができる。応答時間については 2.6.5 項にて評価を行う。

2.6.4 移行中のデータ転送性能

ここではサーバ移行作業中の転送速度への影響を見るため、二つの実験を行った。一つ目は大サイズファイルのデータをシーケンシャルに読み込み、そのスループットを測定した。これはファイルのデータアクセスに対する性能評価を目的としている。二つ目は複数の種類のファイルアクセス要求における混在ワークロードにおいて、時間当たりの処理回数を測定した。こちらはメタデータとデータ両方へのアクセスを含み、より実用的なワークロードと言える。両実験においては、下記の状況を再現して測定を行った。

(A) **移行中**: ファイルは移行元サーバで作成し、オンデマンド複製を有向にした状態でクライアントは移行先サーバにアクセスする。

(B) **移行完了後**: ファイルは移行元サーバで作成し、バックグラウンド複製を完了させたうえでクライアントは移行先サーバにアクセスする。

(C) **通常アクセス**: 移行は行わず、クライアントはファイル作成・ファイルアクセス共に移行先サーバに対して行う。

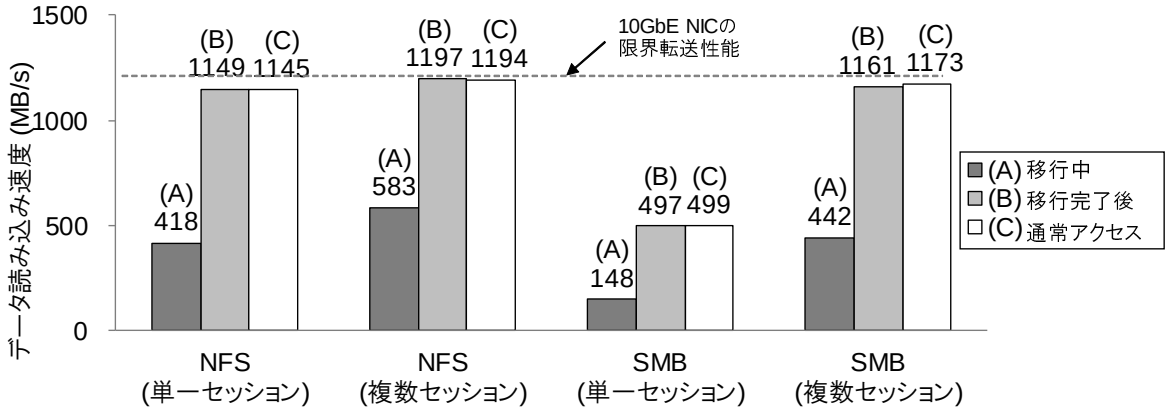


図 2.11: シーケンシャルリード時のスループット.

各実験はクライアントのセッション数を単一の場合と複数の場合で実施した。移行先サーバでは、Linux の NFS クライアントにおいて、同一 IP アドレスへの NFS アクセスが並列化されないことの対策として、同一の共有ディレクトリを複数の IP アドレスでアクセス可能に設定している。単一セッションでは、1 台のクライアントノードを使用し、一つの IP アドレス上の共有ディレクトリを mount して測定した。複数セッションでは、3 台のクライアントノードを使用し、それぞれから三つの IP アドレス上の共有ディレクトリを mount してノードあたり 3 並列、全体で 9 並列でファイルアクセスを行い測定した。

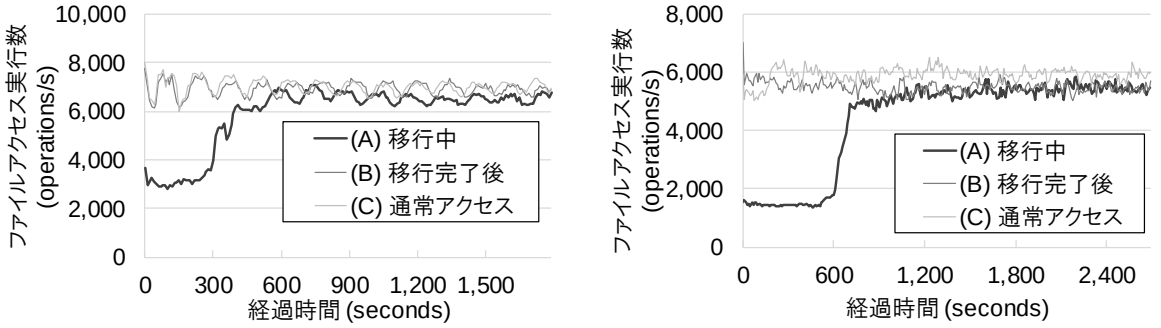
シーケンシャルアクセス性能のスループット

本測定ではセッション毎に 10 GiB のファイルを作成し、各セッションでシーケンシャルリードを行った。クライアント側のページキャッシュの影響を防ぐため、スループットの測定値は十分に時間がたち転送速度が安定した段階で取得した。

図 2.11 に測定結果を示す。縦軸は各測定条件における、全セッションのデータ読み込み速度の合計である。四つの棒グラフのグループはプロトコルとセッション数に対応しており、左から NFS・単一セッション、NFS・複数セッション、SMB・単一セッション、SMB・複数セッションに対応する。グループ内の三つのバーは、前述の (A)(B)(C) の各測定条件に相当する。(A) はオンデマンド複製を伴うので他の構成よりも性能が低下する。一方で (B) と (C) は同等の性能を出している。単一セッションの SMB 以外のケースでは、(B)(C) 共にネットワーク帯域 (約 1200 MB/s) に迫る性能を出しておりこれだけではネットワークがボトルネックとなり (B)(C) の性能差が検証できない。一方ネットワークがボトルネックに至らない単一セッションの SMB のケースでも (B)(C) で同等の読み込み速度が出ていることから、提案手法は移行中こそオンデマンド複製のために性能低下が生じるものの、移行が完了した後は読み込み性能への影響は無視できる程度ということがわかる。また、(A) の測定値においても最低値の単一セッション・SMB のケースにおいても 1 GbE のネットワーク帯域 (約 120 MB/s) は超えていることから、1 GbE の NIC を持つオフィス用途の PC からの利用には差し支えない性能が出ていると考えてよい。

混在ワークロードにおけるスループット

本実験では混在ワークロードの再現のために、Filebench [30] v1.5.0-alpha3 を用いた。Filebench は多用なワークロードを提供するが、ここでは共有ファイルサーバを想定した *fileserver-new* を利用する。本実験では、3 台のクライアントノードを用い、それぞれ 3 つの Filebench プロセス、全体で 9 プロセスを実行した。Filebench の各プロセスは、データセットとして 40,000 個で計 5 GiB のファイルを作成する。プロ



(a) NFS (b) SMB

図 2.12: 混在ワークロードにおける性能

セス毎に 40 個のスレッドが生成され、ファイルサーバに対し作成・読み込み・書き込み・追記・削除を繰り返す。データアクセスについては、位置やサイズは正規分布に従いランダムである。各プロセスは固有のディレクトリを作成してファイルアクセスを行うため、プロセス間で参照するファイルは衝突しない。一方プロセス内の各スレッドは同一のデータセットを用いるため、ファイルを深さ優先探索順で列挙して、互いに同じファイルに同時アクセスしないよう、スレッド間で排他を取りながら未処理のファイルを先頭から順次アクセスしていく。この排他処理はクライアント内で行われるものでファイルサーバは関与しない。

Filebench は Windows 環境での実行に対応しないため、本実験は NFS・SMB ともに Linux クライアントを用いている。クライアント側の設定として、サーバ性能を正確に測るため、クライアント側のキャッシュはデータ・メタデータ共に不使用としている。三つの測定条件については、2.6.4 節と同じ条件を用いた。

図 2.12 に、Filebench 起動後の経過時間に対応する、秒間アクセス処理数の推移を示す。本実験において、クライアントノードは CPU やメモリ消費が飽和しておらず、Filebench のプロセス数・スレッド数を増加しても性能は改善しない。そのため、クライアント性能がボトルネックにはなっていないと判断できる。

(A) 移行中の性能はオンデマンド複製を伴うため、測定開始直後の性能は (B)(C) に比べ低い。NFS で 300 秒程度の間 54% 減、SMB で 600 秒程度の間 74% 減の性能しか出ない。Filebench ではファイルのアクセス順が深さ優先探索順固定なので、全ファイル一回ずつ参照するまでは毎回メタデータのオンデマンド複製が生じ、その後は生じない。これが性能低下中の性能が安定して低く、その後に急速に向上する理由である。SMB の場合、メタデータの取得に時間がかかりオンデマンド複製の時間が NFS より長いため、この間の性能低下の度合いも NFS より大きく、また性能が低い時間も長い。

一方データに関してはランダムで参照されるため、徐々にオンデマンド複製が進むために全体としては全ファイル一回参照した後も、まだ移行は完了しておらず緩やかに性能が向上する。

ただし、十分な時間を経過しても、(A) と (B)(C) では完全に性能は一致せず、数 % 程度の性能低下が残る。これは移行先サーバにおいてディスク上でファイルの断片化が起きているためである。Filebench はランダムアクセスを行うため、オンデマンド複製によってアクセスされたブロックから順に移行先サーバのディスクにデータが格納される。これにより、断片化が生じやすくなっている。ただしこの問題は、実用上は問題になりにくい。近年のファイルシステムはマウント中のオンラインデフラグ機能を備えており [69]、移行完了後にファイル共有サービスを止めることなく解消可能なためである。

プロトコル間の測定結果を比較すると、(a) NFS では測定値が 150 秒程度の間隔で周期的に振動している。これはデータセット及び Filebench のアクセス順、加えて NFS のファイルアクセスの仕様によるもの

である。本データセットが作成するファイルは、一定の深さのディレクトリに固定されず、様々な深さに分散配置されている。そのため、深さ優先順にファイルを列挙して順にアクセスするという Filebench の仕様においてはアクセス対象のファイルの深さが周期的に変化する。加えて、NFS のファイルアクセスではファイルパスの構成要素毎に GETATTR 命令を実行しディレクトリの所在確認を行う。よって、Filebench におけるファイルの 1 アクセスにかかる NFS 命令数が、パス名長によって変化するため、このような振動が生じる。SMB ではフルパスをサーバに送ってファイルアクセスを行うため、このようなパス名長に応じた命令数の変化は生じず、長周期の振動が起きない。

2.6.5　移行中の応答時間

本項では、探索順戦略の効果を評価するため、バックグラウンド複製の進捗状況に応じたファイル応答時間を測定する。この実験では、表 2.2 のデータセット (b) を用いる。移行中におけるクライアントのファイルアクセスはオンデマンド複製による祖先のディレクトリの移行を引き起こすので、ファイルアクセスの順序が性能測定値に影響するため、適正な評価にはアクセスパターンに再現性が無ければならない・本項では、データセットにおいてファイルをランダムに 1000 個選択してアクセス順番を定め、全ベンチマークにおいて同じファイル選択・アクセス順を用いる。使用するクライアントは 1 台であり、用いるプログラムも Python で記述したシングルスレッドのプログラムであるため、全てのアクセスは直列に行われる。このプログラムでは 1000 個のファイル名一覧を参照して、ファイル順に open して 1 byte 読み込んで close するという動作を繰り返す。その際各処理にかかった時間を記録していく。移行状況に応じた探索順戦略の影響を確認するため、バックグラウンド複製を 0%(オンデマンド複製のみ)、25%、50%、75%、100%(移行完了済み) の状態で中断させ、このプログラムを実行した。2.6.5 節ではディレクトリ探索順、2.6.5 節ではデータ移行順における評価を行う。

ディレクトリ探索順

ここでは、データ移行順を一括複製に固定した場合における、ディレクトリ探索順による性能への影響を測定する。サーバ移行の進捗は、ファイル・ディレクトリ数で算出する。すなわち、移行対象のファイル・ディレクトリ数に対する移行完了済みのファイル・ディレクトリ数の比率を進捗とみなす。

BFT と DFT では、同一のパーセンテージはファイル・ディレクトリがともに同数移行されたことを意味しない。BFT はディレクトリの移行が先行しやすく、DFT はファイルの移行が先行しやすい。加えて、ファイルサイズやディレクトリ構成は均一ではないため、同一のパーセンテージに至るまでにかかる時間も異なっている。しかしながら、サーバ移行については 2.6.3 項でも確認した通りメタデータ移行が占める時間は大きいため、移行完了の進捗を移行済みのデータ容量で算出するよりもファイル・ディレクトリ数で算出するのは自然と考える。ファイルアクセスにおける応答時間を決める支配的な要因は、2.6.2 項で述べた通り祖先のディレクトリの深さと幅である。これを踏まえて測定結果を分析する。

図 2.13 にファイル open に対する平均アクセス応答時間の測定結果を示す。上段に八つの棒グラフは NFS、下段の八つの棒グラフは SMB の結果を示す。本グラフの応答時間は対数軸であることに留意すること。“未複製” はバックグラウンド複製を行っていない状態、“複製完了” は複製を終了した状態で、それぞれ複製の進捗は 0%、100% とみなす。他の項目は、DFT 及び BFT における複製の進捗に対応したファイル open の平均アクセス応答時間を示す。概ね BFT は同じ進捗の DFT に比べて短い平均アクセス応答時間を示す。唯一 NFS の 75% 複製時のみ BFT の方が時間がかかるが、これはこの理由については後述する。例えば複製進捗 50% において、BFT においては平均アクセス応答時間が NFS では 51% 減 (826 ms)、SMB で 69% 減 (326 ms) となった。この結果は、BFT が平均アクセス応答時間を効率よく下げる戦略で

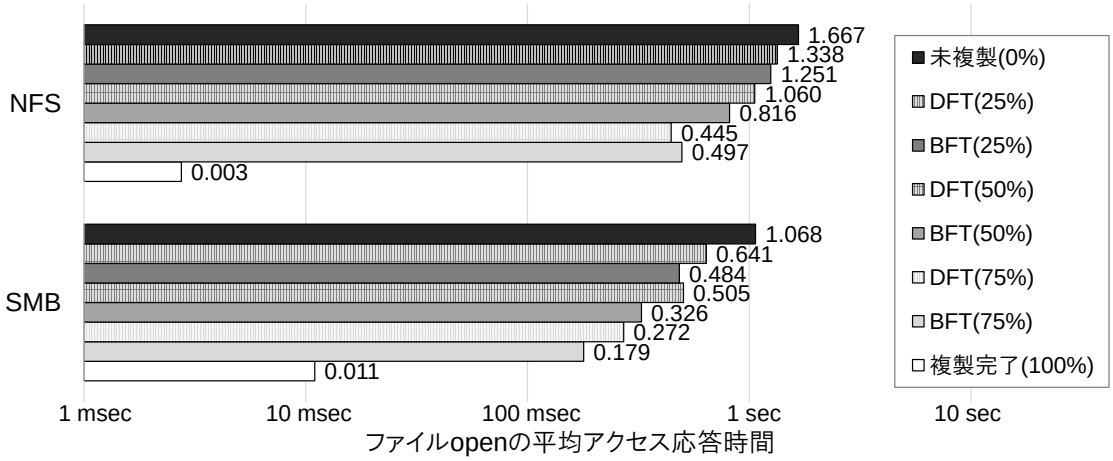


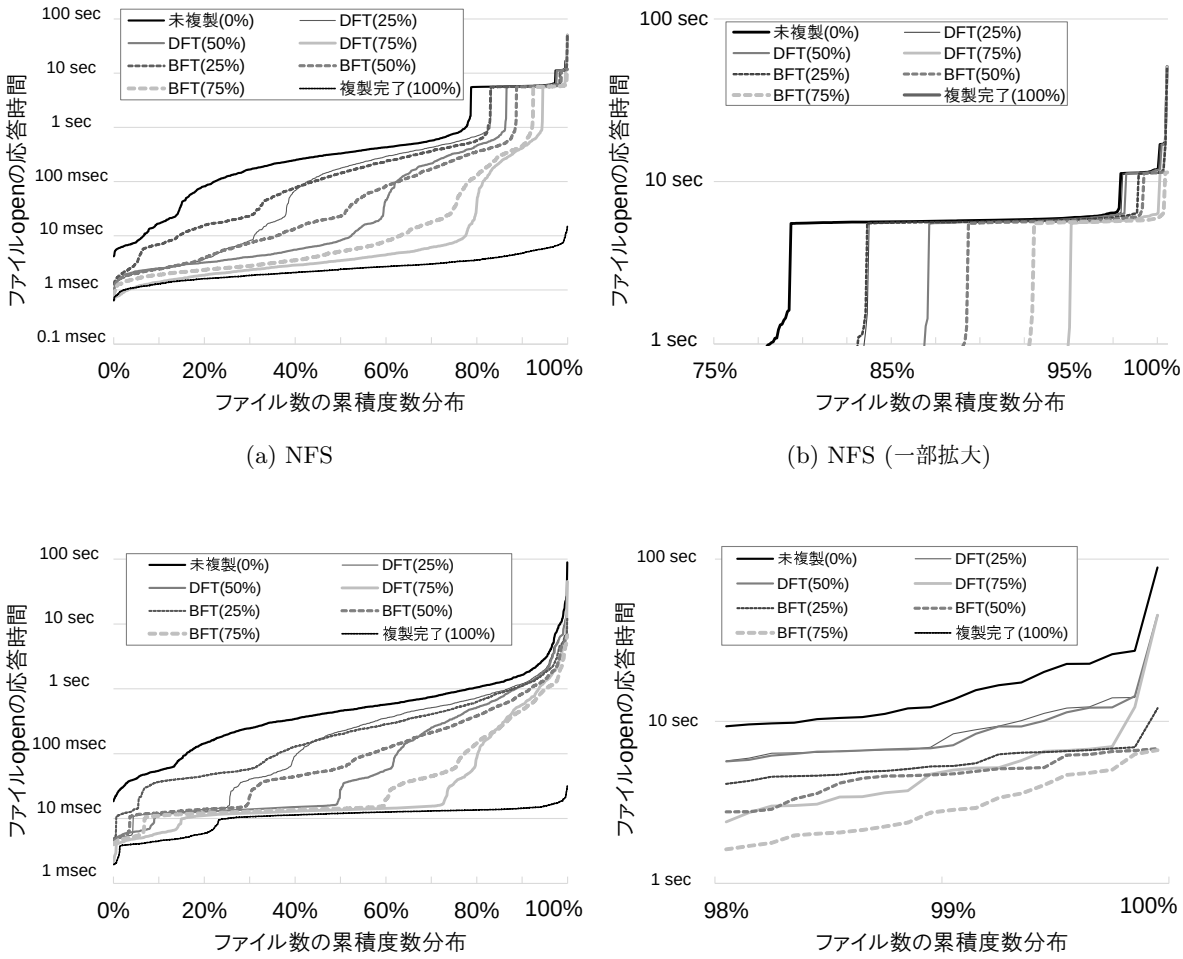
図 2.13: DFT と BFT における平均アクセス応答時間

あることを示している。

図 2.14 は、DFT(実線) と BFT(点線) におけるファイルアクセス応答時間の累積度数分布を示している。横軸は累積度数分布で、縦軸はファイル open 時の応答時間を対数軸で示したものである。複製開始前は、NFS アクセスの 78%、SMB アクセスの 79% が応答時間 1 秒以下だったが、複製が 25% 進んだ段階で、すでに NFS アクセスの 82%、SMB アクセスの 88% が応答時間 1 秒以下となっている。このように複製が進むほどアクセス応答時間のカーブは右下方向に移動する。進捗が 75% の段階では、90% 以上のファイルでアクセス応答時間が 1 秒以内となる。BFT と DFT を比較すると、それぞれの性能特性がより明確となる。BFT は一般に全ディレクトリにあるファイルに対し均等に応答時間を短縮する一方で、DFT は一部のディレクトリから急速に応答時間を短縮する。その結果、DFT は BFT に比べ累積度数の小さい間は応答時間が短いが、いずれその関係は逆転し、最悪応答時間では BFT が優れる。ただし NFS における進捗 75% の結果のみ例外である。

図 2.14(a) は NFS における結果を示している。総じて、プロトコルの単純さを反映して NFS は SMB より応答時間が短い。図 2.14(b) はその一部拡大図である。度数分布の 80%〜95% の領域に、段差が生じている。これは NFS3ERR_JUKEBOX エラーコードによるクライアントの 5 秒間の再送待ちによって、待ち時間に離散的な差が生じたことによる。11 秒、16 秒にも若干の段差が見られるのは、同様に複数回同エラーコードが発生したことによる。バックグラウンド複製が進むごとに、祖先のディレクトリのオンデマンド複製の回数は減り、NFS3ERR_JUKEBOX エラーコードの発生確率も下がるため、最悪応答時間は減少していく。複製進捗 50% の段階では、10 秒以上の応答時間は DFT では 2.4% だったが、BFT では 1.4% にとどまっている。

図 2.14(c) は SMB における測定結果を示している。全体的な傾向は NFS と同じだが、若干異なる点も見られた。まず、カーブにおいて NFS にあったような離散的な変化は見られなかった。次に、いずれの測定結果においても 3 ms〜10 ms の箇所に緩やかな段差が見られる。これは複製完了後も起きていることから、複製方式とは関連しない理由による。具体的には、クライアント側で実行される Windows Defender が、クライアントによるファイルオープン時にデータの先頭部分を読み込むために応じる。一般的にはファイルの先頭と末尾で 2 回読み込みを行うために (オンデマンド複製がなくても)10 ms 程度応答時間が増えるが、0 バイトファイルではこの読み込み回数が 1 回、32 KiB 以下の小サイズファイルではこの読み込み回数が 1 となるため、差異が生じる。実際、今回用いたデータセットではアクセス対象の 1% が 0 バイトファイルであり。25% が小サイズファイルに相当することからこのような現象が起きた。実際、複製完了後の



(a) NFS

(b) NFS (一部拡大)

(c) SMB

(d) SMB (一部拡大)

図 2.14: アクセス応答時間の累積度数分布

測定では、累積度数分布 25% の周辺から 10 ms 以上の応答時間となっている。次に、累積度数分布上での位置は測定によって 20%〜80% と開きがあるが、全体的に 10 ms 強の応答時間でやはり緩やかな段差がある。これはメタデータのオンデマンド複製が行われたことによる。1 ファイルに対してメタデータとデータのオンデマンド複製が生じると、約 10 ms かかる。SMB ではメタデータの取得に NFS より長い時間がかかるため、応答時間はファイルの複製が完了済みかどうかに強く影響を受ける。

一部拡大図 (図 2.14(d)) では、全体として BFT が DFT より最悪応答時間が短いことがわかる。複製進捗 50% において、99% のアクセスが DFT では 6.8 秒かかる一方、BFT では 4.7 秒である。

他と異なる結果である、NFS の進捗 75% の結果についてより詳細に分析していく。ここでは平均応答時間 (図 2.13) では BFT が DFT より短いが、最悪応答時間 (図 2.14(b)) では BFT は DFT より長い。これはデータセットの特性と、NFS の特性による。前述のとおり、NFS の応答時間は、パス解決の途中で生じる NFS3ERR_JUKEBOX エラーコードの発生回数に伴う待ち時間の影響を強く受ける。DFT において、深いディレクトリを偶然先行的に複製した場合、このエラーの発生頻度は急速に低下する。一方 BFT では深いディレクトリについても緩やかにしか発生頻度を低下させない。結果として、このケースのみ平均応答時間が DFT の方が BFT よりも短くなった。

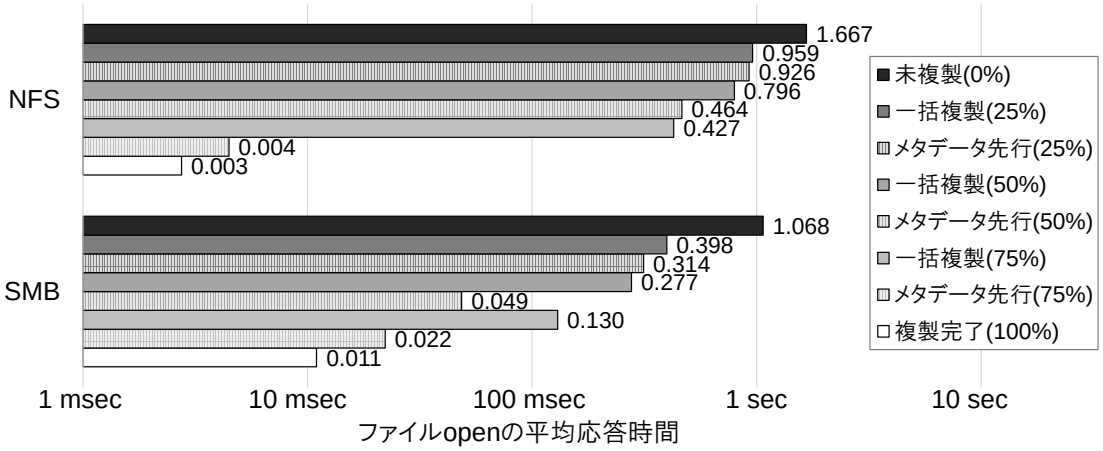
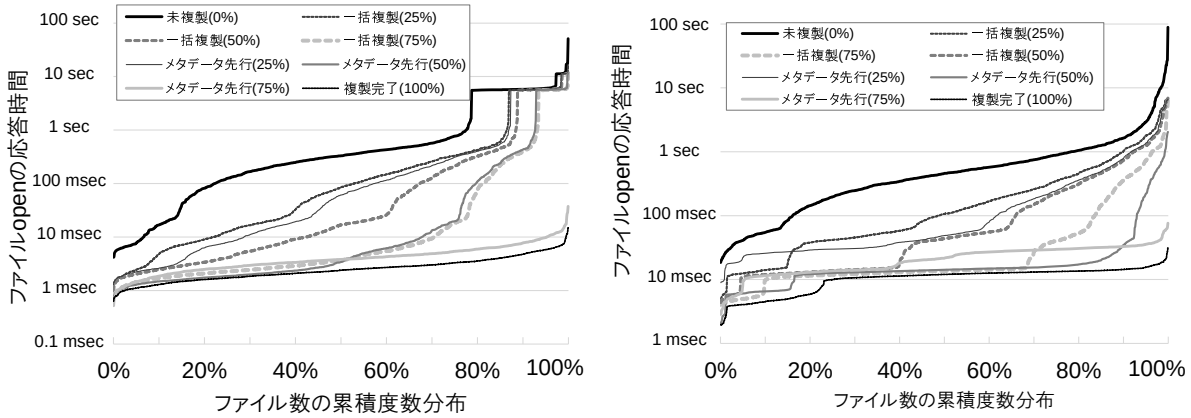


図 2.15: 一括複製とメタデータ先行複製における平均アクセス応答時間



(a) NFS

(b) SMB

図 2.16: 一括複製とメタデータ先行複製におけるアクセス応答時間の累積速度分布.

一般に、利用者の視点で考えるとストレージ管理者はタイムアウトが引き起こすビジネスの中断を重要視するため、平均応答時間よりも最悪応答時間を気にかける。そのため、多くのケースで BFT の方が好ましい。

データ取得順

ここでは、ディレクトリ探索順を BFT に固定し、データの一括複製とメタデータ先行複製の比較を行う。本節では、複製の進捗をファイル数ではなく複製時間で行う。これはメタデータ先行複製においては先に全ファイルメタデータの複製を行うため、複製完了ファイル数を用いた基準は正確でないためである。

図 2.15 はファイルの平均アクセス応答時間を示す。図の構成は図 2.13 と同じである。一見してグラフからわかる通り、いずれの構成においてもメタデータ先行複製の方が、一括複製よりも応答時間が短い。特に、進捗が進むほどその差は開いていく。進捗 50% においては、メタデータ先行複製では未複製時との比較で NFS で 42% 減 (796 ミリ秒)、SMB で 71% 減 (277 ミリ秒) である。進捗 75% においては、NFS で 74% 減 (427 ミリ秒)、SMB で 88% 減 (130 ミリ秒) である。このように、メタデータ先行複製は進捗に伴い平均アクセス応答時間がより効果的に現象していく。

図 2.16 はファイルのアクセス応答時間の累積分布を、一括複製 (点線) とメタデータ先行複製 (実線) でそれぞれプロットしたものである。一括複製は、複製進捗をファイル数でなく実行時間で行う点以外は図 2.14 と同じ構成であるため、グラフにおいてもほぼ同じ結果である。メタデータ先行複製では、進捗が進まないうちはメタデータしか複製しないため、累積度数の少ない範囲では、データのオンデマンド複製が生じる分応答時間が一括複製より長い。しかしほとんどのファイルに対しては一括複製より短い時間で応答しており、いずれのグラフにおいても、同一の進捗であれば累積度数が大きい領域では実線が点線の右下に来ている。進捗 50% においては、99 パーセンタイルにおけるアクセス応答時間は、一括複製では NFS で 11.25 秒、SMB で 4.63 秒もかかるのに対し、メタデータ先行複製では NFS で 5.83 秒、SMB で 0.82 秒である。進捗 75% においては、メタデータ先行複製では NFS・SMB いずれもメタデータの取得は完了している。そのためこの時のアクセス応答時間は複製完了時とほぼ同等であり、最悪応答時間を見ても NFS で 37 ミリ秒、SMB で 76 ミリ秒にとどまる。

このグラフにおいて、メタデータ先行複製 (50%) の線がメタデータ先行複製 (75%) と NFS では累積度数 50%、SMB では累積度数 90% のあたりで交差している。本来複製が進むほどアクセス応答時間は短くなるため線は交差しないはずであるが、今回の測定結果において異なる結果となった理由は Linux のキャッシュ管理にある。進捗 50% の場合、複製作業はまだ 1 周目のメタデータ取得であるため、移行先サーバのメモリは、メタデータキャッシュに利用される。複製が進みデータ複製が開始されると、メモリはデータを格納するためのページキャッシュに使われる割合が増加する。その結果、メモリ中に格納されたメタデータキャッシュは追い出され、ファイルアクセス時にメタデータ読み込みのためのディスクアクセスで 10 ミリ秒程度の時間がかかる。ただし、この時間はクライアントタイムアウトには程遠い値であり、メタデータ先行複製の最悪応答時間の軽減効果が大きいことから、実用的には一括複製よりメタデータ先行複製の方が好ましい。

2.7　まとめ

本章では、三つの技術を用いた Post-copy 型のデータ複製アプローチによる実用的なファイルサーバの移行方法について述べた。一つめは、移行に伴うダウンタイムを最小化するため、移行開始後即座にクライアントの接続先を移行先サーバに切り替え、クライアントアクセス時に適宜移行元サーバからファイルの複製を行うオンデマンド複製である。二つめは、移行後の性能低下の対策として導入した、先行的なバックグラウンド複製の併用である。三つめは、異機種間の移行を実現するために行った、移行先サーバにおけるスタブ管理機構の活用である。これにより移行先サーバ側の対応のみで複製中ファイルの中間状態の管理が可能になり、移行元サーバの構成に依存しない移行が可能となった。これらの移行に関する基本的な検討に加え、本稿では大容量データを抱える実用的な移行シナリオで必要となる断続的なバックグラウンド複製や、その際のディレクトリ探索順戦略についても述べた。加えて、業界標準のファイル共有プロトコルにおけるプロトコル固有の問題についても対策法を論じた。これらの検討内容は、実際に稼働しているファイルサーバの移行事例にも利用された。

本稿では提案方式の妥当性を論じるための性能評価についても述べた。実験結果では、本方式は移行元サーバから移行先サーバへの切り替えに伴うダウンタイムは 3 秒以下であり、主要な OS のタイムアウト時間より短いダウンタイムを実現した。また、移行のためのデータ複製作業中にも、並行してクライアントアクセスを受け付けることが可能であることを示した。これにより、クライアントに対し、サーバ移行に伴うタイムアウトなどの不具合を起こさず移行を行うことができることを示した。さらに、複製中のクライアントアクセス応答時間の短縮のため、バックグラウンド複製におけるデータ探索順で深さ優先探索とメタデータ先行複製方式を併用することで、複製の進捗に伴い平均応答時間および最長応答時間が速やかに減少する

ことを確認した。

本章における提案手法は、移行元サーバや外部計算機上のアプリケーションの改造を要さず、また移行作業においてアプリケーションに対し長時間の停止・応答時間を生じさせないことから、本研究の目的とする、アプリケーションに透過的な移行を実現できていると考える。

2.7.1 提案方式の制限事項

提案手法では、異機種間の移行を実現するため、データやメタデータの取得に標準のファイルプロトコルのみを用いている。そのため、標準のファイルプロトコルでサポートされない機能や構成情報は、本手法では複製できない点が制限事項となる。

一例として、一部の商用ファイルサーバはデータ量削減機能として圧縮や重複排除を提供している。これらはプロトコルで規定された機能ではないため、NFS や SMB は、データ削減に関する情報を取得することができず、提案手法ではデータ削減された状態を保ったまま複製することはできない。ただし、移行後の後処理として移行先サーバで別途圧縮や重複排除を行うことはできる。この場合、アルゴリズムやパラメータの違いにより、移行前後で完全に同じデータ削減効果を得ることはできないが、実用上は問題ない。

別の例として、容量クォータ機能がある。容量クォータ機能は一般にファイル共有プロトコルで共通仕様が規定されておらず各機種がそれぞれプロトコルの外部で独自に実装しているため、ファイル共有プロトコルを介した制御が行えない。そのため、容量クォータの設定に関しては、ストレージ管理者が手作業で複製することになる。ただし一般にクォータの設定はそれほど多数付与されないため、実用上移行の障壁となることはない。

2.7.2 今後の課題

2.3.2 節でも言及した通り、バックグラウンド複製の性能はマルチスレッドによる探索並列化によって向上し、その結果総複製時間の短縮につながる可能性がある。しかし、提案方式ではクライアントアクセスを維持したままバックグラウンド複製を行う前提を取っている。クライアントアクセスのディレクトリ移動などにより、大規模なディレクトリ構成の変更があり得る構成において、ディレクトリを漏れなくかつスレッド間で重複なく探索することは難しい。そのためマルチスレッドによる探索並列化は未実施であり今後の課題である。

2.4.2 項で述べた、NFS バージョン 4.2 におけるホール生成命令の対応はスタブ管理のデータ構造の改造を必要とする。また、そのための性能低下も懸念される。現状 NFS バージョン 4.2 はそれほど普及しておらず、本命令の使用頻度は低いことから、性能低下の防止を優先し未対応である。

実装の選択肢については、2.3.3 項で述べた通り再考の余地があるが、現状ユーザー空間プログラムの実装や Stackable File System の採用は性能面と現状の VFS 層の構成を見ると困難である。エンタープライズ向けのファイルサーバとして用いるには、ユーザー空間プログラムの実装には性能改善を要するし、カーネル本体の改変を抑止するには、VFS 層が拡張可能な設計を取る必要がある。

移行完了後の性能については、2.6.4 節でも述べた通りクライアントがランダムアクセスを行うケースにおいてオンデマンド複製はディスク上でファイルの断片化を引き起こしやすい。バックグラウンド複製は通常シーケンシャルアクセスを行うためこの問題を引き起こさない。両者の性能への影響は、データセットやアクセスパターンに依存するところが大きく、さらなる評価を要する。

第 3 章

VM-aware Adaptive Storage Cache Prefetching

本章では、SSD と HDD を備えた階層ストレージにおいて、データをアクセス特性に応じて最適な記憶メディアに再配置する手法について述べる。一般に階層ストレージは、データアクセスの局所性などの指標に基づき以後のアクセスパターンを予測して高性能な記憶メディアに配置するデータを選択する。しかし、クラウドコンピューティングに代表される、単一の計算機において複数の仮想マシンやアプリケーションを動作させる現代の計算機環境においては、今後のアクセスパターンの予測に基づくデータの選択が難しくなっている。提案手法では、仮想マシンを動作させる計算機環境のホスト OS において SSD を HDD のキャッシュとして用いる階層ストレージを構成し、仮想マシンにキャッシュ機構付きボリュームを提供する。本階層ストレージでは、仮想マシン内のゲスト OS が管理するファイルブロックの論理的な連続性に着目し、アクセス頻度が高いブロックの近傍のブロックを一括して SSD にプリフェッチする。これにより、仮想マシン内のファイルアクセスにおいて、SSD キャッシュに対するキャッシュヒット率を向上させ、アクセス応答時間を短縮する。

3.1 はじめに

第 1 章でも述べた通り、階層ストレージは性能や容量、コスト等の特性の異なる複数の記憶用メディアを使い分け、対コストにおける性能や容量を向上させる。2020 年時点において、SSD は HDD より遙かに高い I/O 性能を持つが、容量あたりの価格は未だ HDD よりも高い [3, 4]。そのため、SSD と HDD を組み合わせて構築する階層ストレージは、より高い対コスト性能を実現することが期待できる [70, 71, 72]。

階層ストレージの構築法の一つに、低速な記憶メディアに対し、少量の高速な記憶メディアをキャッシュとして用いる手法がある。この手法についてこれまで様々な研究がなされている [73, 74, 75, 76]。これらキャッシュによる性能向上効果はキャッシュヒット率に依るため、アプリケーションからデータに対する時間的及び空間的なアクセスの局所性に強く依存する。もし記憶メディアへのアクセスが、限られた領域に限定されており、さらにその領域が時間を経ても変化しないのであれば、その領域のデータを高速な記憶メディアにキャッシュすることで、以後の同領域に対するアプリケーションのデータアクセス要求に高速に応答可能となり、アプリケーション全体の性能を改善することができる。反対にアクセスされる領域が時間変化を伴う場合、高速な記憶メディアへキャッシュすべき領域も変化する。そこで、今後のアクセスされるであろう領域を予測して、低速なメディアのデータをキャッシュにプリフェッチしておくことは、アプリケーションによる同データへのアクセスのキャッシュヒット率を向上させ、アクセス時の応答時間を短縮させるために有効である。

しかし、IaaS(Infrastructure as a Service) に代表される、仮想マシンの動作を伴う近年のクラウドコンピューティング環境においては、ホスト OS において観測できるアクセスの空間的・時間的な局所性がそれぞれ失われる。空間的局所性については、仮想マシン内でゲスト OS が格納するファイルのデータは、記憶メディアに格納されるまでに複数のストレージ処理のソフトウェアスタックを経由するうえで再配置されることで失われる。例えば仮想マシンを動作させる典型的なストレージ構成として、ゲスト OS 及びホスト OS で二重にファイルシステムを持つほか、それぞれで Logical Volume Management (LVM) のようなボリュームの仮想化の仕組みや、ホスト OS による仮想ディスク管理の機能がある。これらは上位のストレージ階層のデータを再配置して下位の階層に格納するため、本来仮想マシン内のアプリケーションから見てアクセスの局所性が高いファイル内の一連の領域を、物理ディスク上で不連続な領域に対応付ける場合がある。時間的局所性を失わせる要因は、クラウドコンピューティング環境においては、複数の仮想マシンが単一のホスト OS 上でマルチテナントで動作することである。個々の仮想マシンは異なるワークロードのデータアクセスを行い、加えて仮想マシンは頻繁に作成・削除・移行されうる。そのため、ホスト OS 及びホストのストレージシステムにとって、ゲスト OS の将来のアクセスパターンは時間的にも空間的にも予測しにくいものとなっている。

本章では、仮想マシンを用いるクラウドコンピューティング環境における階層ストレージの適用を想定し、仮想マシン内に保持する情報を活用した記憶メディア間のキャッシュプリフェッチ手法を提案する。本手法では、ホスト OS から仮想マシン内の情報を利用するため、仮想マシン内でエージェントプログラムを動作させる。このエージェントプログラムとホスト OS が協力し、ゲスト OS やそのアプリケーションが生成する空間的時間的なアクセス局所性に関する情報を、ホスト OS において活用する。空間的アクセス局所性をホスト OS で認識するため、本手法ではエージェントプログラムが仮想マシン内におけるファイルシステムのレイアウト情報とそのアクセス情報を取得し、ホスト OS に伝える。これらの情報に基づき、ホスト OS は物理ディスクではなく仮想ファイル中で連続するデータブロックを一括してプリフェッチすることで、キャッシュプリフェッチの効果を高める。加えて、本手法ではアクセス局所性の時系列変化を捉えるため、仮想マシン内のアプリケーションの性能変動を監視する。これにより、アプリケーションによるアクセスパターンの変化を素早く捉えて柔軟なキャッシュプリフェッチを行い、キャッシュヒット率が低下する期間を削減してアプリケーションの性能を回復する。しかし、キャッシュプリフェッチはキャッシュヒット率を向上する一方、過剰なプリフェッチはそれ自体が記憶メディアの入出力帯域を消費し、却ってアプリケーション全体の性能を損なう懸念がある。この問題に対応するため、本方式ではアプリケーションが示す性能統計情報に基づき、プリフェッチの実行時間を動的に調整する。

本研究のプロトタイプでは、著名な RDBMS である PostgreSQL [19] を対象アプリケーションとし、KVM を用いた仮想マシンで実行する構成を対象としてキャッシュ機構付き仮想ボリュームを実装した。本プロトタイプでは、PostgreSQL が持つテーブルやインデックスの格納されたファイルの情報と、ゲスト OS である Linux カーネルのファイルシステムが持つ仮想ボリューム上のファイルレイアウト情報を取得し、ホスト OS にてキャッシュプリフェッチを行う。多くのアプリケーションは、ファイルシステムやその上に構築されたデータベースシステムを利用している。また多くの実用データベースシステムはデータの格納のためにファイルシステム上に大サイズファイルを作成することから、ファイルのレイアウトを認識してプリフェッチを行う本手法は広範囲のアプリケーションに対し有効であると考える。

本プロトタイプを SSD と HDD からなる階層ストレージに適用して TPCx-V ベンチマークプログラムを用いて性能評価した。その結果、本手法が提案するアプリケーションやゲスト OS の持つ情報の利用によって、これらの情報を用いないプリフェッチ手法と比較し、同容量の記憶メディアを用いた場合で 17.1% の性能向上を達成することを確認できた。

本章の以降の構成は下記のとおりである。3.2 節にて関連研究の分類と、本研究との比較を行う。3.3 節で

は本研究の適用対象のソフトウェア構成を整理し、階層ストレージの機構をどのように組み込むべきか考察する。3.4 節では考察結果を踏まえて本手法のキャッシュプリフェッチ戦略について述べる。3.5 節では、階層ストレージを実現するソフトウェアの構成を示す。3.6 節でプロトタイプ実装について述べ、3.7 節ではプロトタイプを用いた性能測定実験とその結果を示す。最後に、3.8 節にて本手法についてまとめる。

3.2 関連研究

本節では、複数の記憶メディアで構成される階層ストレージのうち、低速メディアに対する高速メディアを用いたキャッシュ機構の構成法について述べる。

3.2.1 ヒント情報を用いないキャッシュ

本項では、先行研究である低速な HDD に対する高速な SSD を用いたキャッシュ機構のうち、外部からデータアクセスに関するヒント情報を与えない方式について、キャッシュのみ行う手法と、プリフェッチを併用する二つのアプローチに分類して整理する。前者は、時間的なアクセスの局所性が強いワークロードに対し効果が高く、後者は以後のアクセスパターンの予測がつくワークロードに対し効果が高い。

キャッシュのみ行う手法

Solid-State Hybrid HDD [77] は、デバイス内で閉じたキャッシュ機構を備えた HDD である。HDD のコントローラ上に少量のフラッシュメモリを組み込んでおき、コントローラ内でキャッシュ管理を行う。Linux Bcache [78] や FlashCache [76] は、OS のブロックサブシステム管理層に実装されたキャッシュ機構であり、キャッシュ機能付ボリュームを提供する。これらの方式は、アプリケーションが階層の存在を意識せず透過的にキャッシュ機構を利用することが出来る一方、アプリケーション固有の振る舞いに応じた性能最適化は行えない。

SSD Bufferpool Extension [73] 及び FaCE [74] は、特定のアプリケーションに特化し、アプリケーションで SSD を占有してデータをキャッシュする手法を示している。これらの手法は、データ構造などアプリケーション固有の知識を用いて最適なデータ配置を行える一方、アプリケーションが限定され適用範囲が狭く、また SSD を占有する前提のため複数のアプリケーションが同時に動作する構成は考慮されていない。

vCacheShare [79] は、複数の仮想マシンが動作している構成において、各仮想マシンの I/O パターンをホスト OS にて監視し、仮想マシンに割り当てる SSD のキャッシュ容量を調整する手法である。また、KNOWAC [80] はアプリケーションからデータアクセス時にヒント情報を付与し、OS がそのヒント情報を用いてアクセスパターンを認識する手法である。これらの手法はアクセスパターンが周期的な場合に有効であるが、時間経過により変化するワークロードには適さない。

Counter Stacks [81] は一連の I/O 要求において、同一データの時間的な呼び出し間隔を監視し、そこからキャッシュサイズとキャッシュヒット率の関係を推測する。この手法は、対コスト性能で最適なキャッシュサイズを求めることができるが、キャッシュに残すべきデータ領域の選択は支援しない。

キャッシュプリフェッチ

QuickMine [82] はコンテキストを考慮したキャッシュプリフェッチを行う。この手法では、アプリケーションがヒント情報を付与した I/O 要求を行う。OS のストレージ処理層はこのヒント情報を元に I/O 要求をグルーピングする。そしてグループ毎の I/O 要求の履歴を用いて、近い将来に高確率でアクセスされるであろうディスク領域を推測し、SSD にプリフェッチする。LAM [83] は周期的に I/O パターンを監視し、

やはり今後アクセス頻度が上昇するであろう領域を SSD にプリフェッチする。

これらのプリフェッチ方式は、I/O パターンを見てプリフェッチの速度を変化させる。しかしながら、これらの従来手法はあくまで個々の I/O 要求のみを監視して行っており、アプリケーションの挙動やメディア上のデータ配置を考慮していない。

3.2.2　ヒント情報に基づくキャッシュ

キャッシュ管理に際し、アプリケーションからヒント情報を OS のストレージ処理層に通知することでキャッシュヒット率を情報させる先行研究も多数提案されている。前述の QuickMine に加え、CLIC [84]、hStorage-DB [85]、DHIS [86] はアプリケーションと OS のストレージ処理層が互いに連携してキャッシュ管理を行っている。これらの手法は、いずれもアプリケーションが発行する I/O 要求に、優先度や用途と言った情報をヒント情報として加えることで、ストレージ処理層にキャッシュ戦略の判断材料を与えるものである。

Moirai [87] は仮想マシン機構を備えた複数のサーバと、ネットワーク接続された共有ストレージ機器からなる構成におけるキャッシュ管理の手法を提示している。Moirai では、サーバ上のゲスト OS が共有ストレージ機器に I/O 要求を行うと、ホスト OS が介在してワークロードの特性情報を付与する。共有ストレージ機器は SSD と HDD で構成されており、この特性情報を用いてデータのキャッシュを行う。

ヒント情報に基づくキャッシュは、ヒント情報を用いないケースに比べ効率的なキャッシュ割り当てが可能である。しかしながら、ヒント情報を扱うために OS のストレージ処理層からアプリケーションまでを通じて、広範囲にソフトウェアスタックの更新を要する。そのため、クラウドコンピューティング環境のようにシステムソフトウェアとアプリケーションの所有者が異なっている場合や、多様なアプリケーションを動作させる場合には事実上適用困難である。

3.2.3　仮想マシン内におけるアプリケーション動作情報の取得

ホスト OS から仮想マシン内のアプリケーションの動作を監視する手法は、以下の通り分類されるアプローチがある [88]。

■CPU 及び I/O の監視　仮想マシンモニタは CPU 及びストレージ機器を仮想化し、ゲスト OS に提供する役割をもつことから、仮想マシンモニタを動作させるホスト OS において、CPU 及び I/O を性能オーバヘッドを伴わず利用状況を監視することができる。vCacheShare [79] は SSD キャッシュの管理にあたり、このアプローチで仮想マシン毎に I/O の履歴をとり、キャッシュの割り当て量を定めている。しかし複数の I/O 要求間の関係を解析するわけではなく、以降にアクセスされるデータを予測することはできない。この手法は限られた情報のみを用いて性能を向上させることを狙ったものであり、パブリッククラウドのような、所有者の異なる複数の仮想マシン及びアプリケーションが動作する構成に適している。

■ファイルシステム構造の認識　仮想マシン上で動作するゲスト OS のファイルシステムのディスク上の構造をホスト OS から認識する手法として、VM introspection がある。これは仮想マシンの動作を外部から観測し、内部情報を特定する手法である。

VMDI introspection [89] では、ホスト OS から仮想ディスク上のデータ構造を解析し、ファイルシステムにおける個々のデータ部とメタデータ部を推測することて、両者に対しホスト OS のバッファリング戦略を変えることで性能改善を図る。IDS [90] は性能向上を目的としたものではないが、同じく仮想ディスク上のファイルシステムの構造を分析することでマルウェアの混入を探索するものである。

表 3.1: キャッシュ方式の選択肢

分類	選択肢
実装対象 OS	仮想マシン内・ゲスト OS ／ 仮想マシン外・ホスト OS
実装対象階層	アプリケーション ／ 仮想ディスク管理 ／ ファイルシステム ／ ボリューム管理
キャッシュ先デバイス	DRAM ／ SSD
キャッシュ方式	Read キャッシュ ／ Write-back キャッシュ ／ プリフェッチ・先読み

ディスクに対する VM introspection では、一般にファイルシステムの構造に基づきディスク上のデータとファイルの構成情報の対応付けを行う。そのため、実現にはファイルシステムの内部構造に関する知識が必要な上、構造を特定するためにディスクへのアクセスを要し、性能オーバヘッドが生じる。よって、本手法はファイルシステムの内部構造が明らかにされないプロプライエタリな OS であったり、ファイルシステムや OS のバージョンによって内部構造に変更が行われたりした場合正しく動作しない。そのため、このアプローチはゲスト OS が均一的に設定されることが期待できる状況のみで有効であり、適用範囲が限定的である。

■システムコール呼び出しへの介入 このアプローチは、アプリケーションによるゲスト OS に対するシステムコール呼び出しをホスト OS にて割り込んで受け取り、呼び出し時の情報を取得するものである。Sky [91] はアプリケーションのシステムコール呼び出しを監視することに加え、ホスト OS からゲスト OS にシステムコールを発行する仕組みを提供する。これにより、ホスト OS は仮想マシン内のアプリケーションのファイルアクセスを監視できるほか、仮想マシン内のファイル上のデータと仮想ディスク上の格納セクタ位置の対応関係も取得できる。これにより、ホスト OS にてページキャッシュ管理の効率化や、複数仮想マシン間での重複排除を行う。

この手法は、ゲスト OS の種類を特定可能で、かつそこで行われるシステムコールの仕様が明らかである構成でなければ利用できない。また、システムコール呼び出しに逐一割り込むため、CPU の処理オーバヘッドが大きい点が問題となる。

■エージェント方式 このアプローチは、仮想マシン内に情報取得のためのエージェントプログラムを追加し、そのプログラムがホスト OS と連携することで仮想マシン内の情報をホスト OS に通知する。Lares [92] や SYRINGE [93] は、ストレージに関する手法ではないが、仮想マシン内のマルウェア検知のためにゲスト OS のカーネルを改造し、アプリケーションの動作を監視する機構を加える手法を提案している。

エージェントプログラムの利用は、マルウェア監視以外にも、性能モニタリング、データ管理、障害検知等の目的で商用システムにおいても利用されている [94, 95]。このアプローチは、システム管理者が個々の仮想マシンへのアクセス権限を持つような実行環境に適している。

3.3 階層ストレージ及びキャッシュ機構の実施方法の考察

クラウドコンピューティング環境で用いられるような、計算機上で複数の仮想マシン上を介しアプリケーションを動作させる構成においては、仮想マシン上のファイルから物理メディアに至るまで、複数のストレージ管理のソフトウェア階層が存在する。このうち、クラウドコンピューティング環境で広く使用されている構成として、OS に Linux、仮想マシンモニタに KVM を用いた場合、記憶階層においてキャッシュ機構を組み込む場合の選択肢は、表 3.1 に示す通り複数考えられる。

このうち一部の機構は Linux 及び KVM で既に組み込まれている。ホスト OS 及びゲスト OS 中のファイルシステムはページキャッシュを備えており、ファイルのデータを DRAM 中に格納して Read キャッシュおよび Write-back キャッシュとして扱うことができる。また、同じくファイルシステムでは、今後読み込みが期待されるファイルデータをページキャッシュに先読みする動作を行う。仮想マシンモニタでは、仮想ディスクイメージに対する Write-back キャッシュ機能を備える。この機能はホスト OS 内のページキャッシュを制御する形で実装している。

この前提の元で、階層ストレージの構成を考える。まず DRAM をキャッシュとして用いる機構は、すでにゲスト OS・ホスト OS がそれぞれページキャッシュおよび先読み機構を備えており、追加導入する必要はないと考える。ただし、DRAM の容量をゲスト OS とホスト OS にどのように配分するかは検討の余地がある。本研究の前提として、クラウドコンピューティング環境を想定していることから複数の仮想マシン間で負荷が不均等かつ時間変化する可能性がある。この場合、キャッシュに用いる DRAM 容量は仮想マシン毎に固定で割り当てるよりも、動的に調整できる方がよい。これは、一旦仮想マシンに割り当てた DRAM 容量を後で回収・再配置することは性能オーバヘッドが大きいためである。一方プリフェッチについては、仮想マシン内のファイルシステムの構成を認識しているゲスト OS で行う方が、より適切な領域をプリフェッチ対象として選択できる。一般にファイルデータの先読み量はそれほど大きくない (Linux の場合、1 ファイルシステムで 1 MB 以下) ことから、そのため、ゲスト OS にはアプリケーション動作に必要な最小限の DRAM 容量のみ付与し、残りはホスト OS でページキャッシュとして利用することで間接的に全仮想マシンで共有するのが良いと考える。同様に、SSD と HDD の階層ストレージの管理を行い、SSD を HDD に対するキャッシュ先デバイスとして扱う場合、ホスト OS で行った方が仮想マシン間で動的に限られた SSD の容量を融通することができ、仮想マシン間の負荷変動に対し有効であると考える。KVM において複数仮想マシンを動作させる場合、それぞれはホスト OS 上で異なるプロセスとして動作し、標準では互いに協調動作などは行わない。そのため、同様の理由で仮想マシンモニタにおいて SSD と HDD の階層管理を行うことは好ましくない。

よって、仮想マシン間で動的に SSD の容量を調整するには、ホスト OS で一括して SSD と HDD の階層ストレージの管理を行うのが最善と言える。ホスト OS 内においても、仮想ディスク管理・ファイルシステム・ボリューム管理部と階層ストレージの機能を実現する複数の処理階層がありうる。このうち、仮想ディスク管理やファイルシステムは、スナップショット機能や容量の Thin-Provisioning を行う上では有効な機能であり、仮想マシンの運用管理の点では有効である。しかしながら、これらの機能は複雑な管理情報を保持するために仮想マシン内のデータを物理ディスク上で断片化させるため、I/O に対し高い処理性能を要求する場面においてはオーバヘッドとなる。そこで、本研究では仮想ディスク管理及びファイルシステム自体を用いず、OS のボリューム管理部が階層ストレージ機構を実現し、その SSD キャッシュ機能を備えたボリュームを直接仮想マシンの仮想ディスクとして割り当てるものとする。

キャッシュ方式については、Read キャッシュ・Write-back キャッシュはもちろん有効であるが、加えて I/O 応答時間の短縮にはプリフェッチが最も有効と考える。HDD は小サイズ I/O の性能が低いため、プリフェッチを行わないと、HDD に対する小サイズ I/O が発生した際アプリケーションの性能が低下してしまう。プリフェッチの場合、アクセスの局所性を考慮して周辺の領域を含め大サイズの I/O を発行することで、小サイズ I/O を避け HDD の性能をより活かす形でデータの読み込みを行うことができ、アプリケーションが認識する平均 I/O 応答時間の短縮効果を高められる。

3.4　キャッシュプリフェッチ戦略

本節では、提案するキャッシュの管理方式およびプリフェッチ戦略について述べる。

3.4.1 キャッシュ管理方式

提案方式の基本的なキャッシュプリフェッチ戦略は、ディスク上において、頻繁なアクセスが見込める領域を含む、論理的に連続したブロックを SSD にプリフェッチすることである。本章における論理的に連続とは、仮想マシン内のファイルブロックが連続していることを意味する。これらのブロックは物理ディスク上では必ずしも連続しない。プリフェッチは、セクタ単位ではなく、連続したセクタを複数まとめたブロック単位で行う。それには二つの理由がある。第一に、アクセスの時間的・空間的局所性により、ディスク上でアクセスされたセクタの近傍セクタをキャッシュに配置することでキャッシュヒット率の向上が見込めるため、セクタより大きな単位で階層間の移動を行うことは有効である。もう一つは、一般に記憶メディアに対して同容量のデータの読み書きを行おうとした場合、大サイズの I/O 要求を行う方が、小サイズの I/O 要求を繰り返すのに比べ高いスループットを出せることである。特に HDD は、内部に機械の駆動を伴うためランダムアクセスの応答時間が伸び、この傾向が顕著に表れる。よって、HDD と SSD の間の階層移動およびプリフェッチをある程度大きな単位で行うことは、性能向上に有効である。

本研究は仮想マシンを伴うクラウドでの利用を前提とするため、キャッシュ機構を組み込む場合はゲスト OS に透過的に利用できる技術であり、かつゲスト OS やアプリケーションの改変を要しない構成である必要がある。そのため、提案方式ではキャッシュの管理はホスト OS 内で行い、ホスト OS はゲスト OS 及びアプリケーションが外部に公開しているインターフェースのみで得られる内部情報を用いる。加えて、過剰なプリフェッチが HDD の帯域を消費することによるアプリケーションの性能低下を防ぐため、本手法ではアプリケーションの性能統計情報を観測し、プリフェッチの実行速度を動的に変更する。変更方法としては、アプリケーションの実行時間を複数の固定長ウインドウに分割し、個々のウインドウ内でプリフェッチを行う時間の割合を統計情報に基づき増減させることで実現する。

3.4.2 ボリューム管理

本手法では、高速な記憶メディアを低速な記憶メディアに対する Write-back キャッシュとして用いる。以下、記憶メディアによって構成されるボリュームを物理ボリュームと呼び、ホスト OS が仮想マシンに提供するキャッシュ機能を備えたボリュームを論理ボリュームと呼ぶ。ホスト OS では、キャッシュ用の高速なメディアによる物理ボリュームとその後ろにある低速なメディアによる物理ボリュームを組み合わせて論理ボリュームを構築し仮想マシンの仮想ディスクとして割り当てる。そのため、ホスト OS が提供するファイルシステムおよび仮想ディスクイメージのストレージ階層は用いない。各メディアの管理を単純化するため、個々の記憶メディアと論理ボリュームの記憶領域は、固定長チャンクの配列として扱う。空間的な局所性をより効率よく扱うため、ここでは論理的に連続したいくつかのチャンクを、キャッシュ単位として扱う。プリフェッチは、このキャッシュ単位毎に行う。

図 3.1 は論理ボリュームにおけるキャッシュ単位とチャンクの対応関係の一例を示している。連続したチャンクを識別するため、ここではゲスト OS において物理ボリュームや論理ボリュームの LBA(Logical Block Address) ではなく、ゲスト OS におけるファイルブロックのオフセット値を参照する。ゲスト OS において、ファイルブロックのオフセットと、論理ボリューム上の LBA の対応付けを行うことで、LBA からファイルを特定する逆方向のマッピングテーブルを構築する。具体的には、以下の手順でキャッシュ単位を構築する。$B(file, index)$ をファイル $file$ における $(index)$ 番目のチャンクに対応する論理ボリューム上のチャンク番号とする。ファイル $file$ における k 番目のキャッシュ単位は、キャッシュ単位を構成するチャンク数を N とするとき $U(file, k) = \{B(file, k*N), B(file, k*N+1), \ldots, B(file, k*N+N-1)\}$

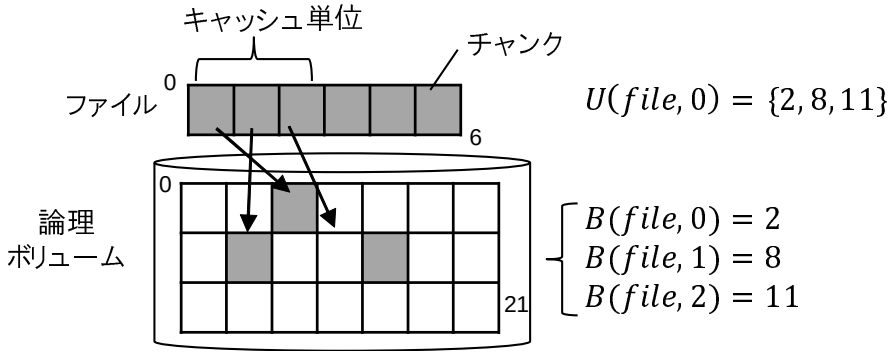


図 3.1: キャッシュ単位とチャンクの対応関係 ($N = 3$).

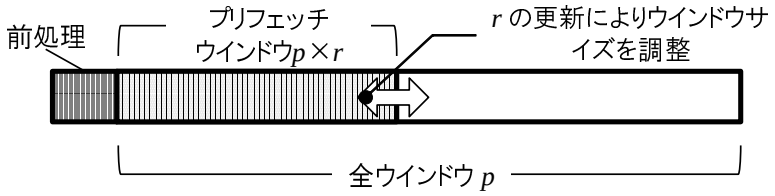


図 3.2: プリフェッチのタイムウインドウ

と表せる。

ゲスト OS のファイルシステムは、チャンクやキャッシュ単位の境界を認識しないため、ファイルを構成するブロックは必ずしもキャッシュ単位とアラインメントが合うとは限らない。しかしながら、実用的な多くのファイルシステムではフラグメントを防ぐようブロック割り当てを行うためにフラグメントは少ないことが期待でき、殆どのケースでは単一のキャッシュ単位内には 1 ファイルのブロックが格納されることを前提とする。実際、Linux の実装において ext4 は 8 MB の連続領域をファイルブロックに割り当てる傾向にあり、XFS も 1 MB 以上の連続領域を割り当てる傾向にある。

ホスト OS では、チャンク単位で I/O の統計情報を取得してキャッシュ単位毎に集計する。そして集計した統計情報に基づきキャッシュ単位をアクセス頻度順に並べ、順次低速なメディアからプリフェッチを行う。つまり、同一のキャッシュ単位に属するチャンク群は、うち一部のチャンクにアクセスが集中した場合、連動してプリフェッチされるチャンクとなる。よって、適切なキャッシュ単位のサイズはアプリケーションによるアクセスの空間的な局所性に依存する。

3.4.3　動的プリフェッチ速度調整

過剰なプリフェッチによるアプリケーションの性能低下を防ぐため、本方式では性能統計情報に応じてプリフェッチの速度を動的に調整する。プリフェッチは、キャッシュヒット率の改善効果を高めることができるが、過剰に行うとプリフェッチ自体が低速なメディアの I/O 帯域を占有していまい、本来優先すべきアプリケーションの発行する I/O の応答を送らせてしまう。そのため、プリフェッチ速度の制御はアプリケーション全体の性能を向上させるために有用である。

低速なメディアとして HDD を用いる場合、HDD は機械的な駆動部を持つために並列な I/O 処理に対し大きな性能低下を起こしてしまう。そのため、たとえプリフェッチの I/O 量が少ないとしても、常時プリフェッチを行うこと自体がアプリケーションの I/O を妨害してしまう。

そこで本手法では、プリフェッチの実行をウインドウ内のタイムスライスで制御する。図 3.2 は 1 周分のウインドウを示す。p を単一のウインドウの長さを秒単位で示すものとし、r $(0 \leq r \leq 1)$ をウインドウ 1 周においてプリフェッチを行う時間の割合とする。すなわち、プリフェッチはウインドウ毎に $p \times r$ 秒間実行することになり、本手法では r を更新することでプリフェッチの速度を調整する。なお、プリフェッチを行っている間もアプリケーションによる通常の I/O 要求は実行可能とする。ただし I/O 帯域の多くをプリフェッチに使用するため、その性能は低いとみなす。

r を正しく設定するため、本手法ではホスト OS においてウインドウ毎に各種の統計情報を集計する。時間経過に伴う I/O のアクセスパターン変化に対応するため、統計情報は時間を重みとした重み付平均値を用いる。この重み付平均値と直近の測定値に大きな乖離があった場合、アプリケーションの動作が変わったと判断し、r を変化させる。すなわち、$s(item, t)$ を測定項目 $item$ における時刻 t における測定値とすると、ウインドウ i 回分前の測定値に対する重みを (w_i) としたとき、重み付平均値は:

$$s'(item, t) = \left(\sum_i w_i \times s(item, t-i) \right) \Big/ \left(\sum_i w_i \right)$$

とする。

本方式では、重みとして時間経過に伴い指数関数的に減少する値として $w_i = 2^{-i}$ を用いることにした。これは、直近のアクセスパターンの変化に素早く追従することを優先するためである。以下直近のタイムウインドウにおける統計情報 $C(t)$ を $s(item, t)$ の集合、過去の統計情報の重み付け平均 $C'(t)$ を $s'(item, t)$ の集合とする。本手法では、割合 r の変化量 Δr を、両統計情報を Easing 関数 E をもとに $\Delta r = E(C(t), C'(t))$ として計算する。

関数 E はアプリケーションの動作に依存するため、アプリケーションの動作および関連する記憶メディアの統計情報 (例: キャッシュヒット率、I/O の応答時間、I/O の入出力量) をもとに定める。試作の実験時における E の詳細は、3.6 節にて述べる。

3.5 システム構成

図 3.3 は提案手法のシステム構成を示す。本システムは、三つの構成要素、**ボリューム管理部**、**キャッシュ割り当て管理部**、**エージェントプログラム**からなる。ボリューム管理部はホスト OS 内で動作し、物理ボリュームから論理ボリュームの作成・管理を行う。キャッシュ割り当て管理部もホスト OS 内で動作し、収集した各種性能統計情報に基づき、階層間移動を行うチャンクを定め、ボリューム管理部にプリフェッチを指示する。仮想マシン内で動作するエージェントプログラムは、VM 内で動作し、ホスト OS 上で動作するキャッシュ割り当て管理部に仮想マシン内部の性能統計情報を通知する。

3.5.1 ボリューム管理部

ボリューム管理部は、ホスト OS 上で動作し、高速な記憶メディアからなるキャッシュデータを格納するための物理ボリューム (キャッシュ用ボリューム) と低速な記憶メディアからなる物理ボリューム (バッキングボリューム) で構成されるキャッシュ機能を備えた論理ボリュームを作成する。各論理ボリュームは、仮想ディスクやホスト OS のファイルシステムを介さず、直接仮想マシンに割り当てられ、ゲスト OS 内でファイルシステム等を介して利用される。その後、仮想マシン内のアプリケーションのデータは、そのファイルシステム上にファイルとして格納される。

図 3.4 がボリューム管理の概略図である。ボリューム管理部は、論理ボリュームと物理ボリュームをそれ

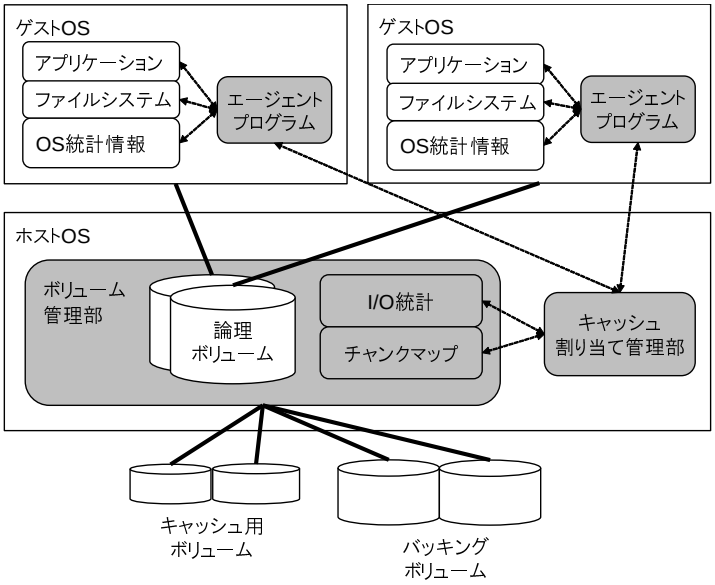


図 3.3: システム構成

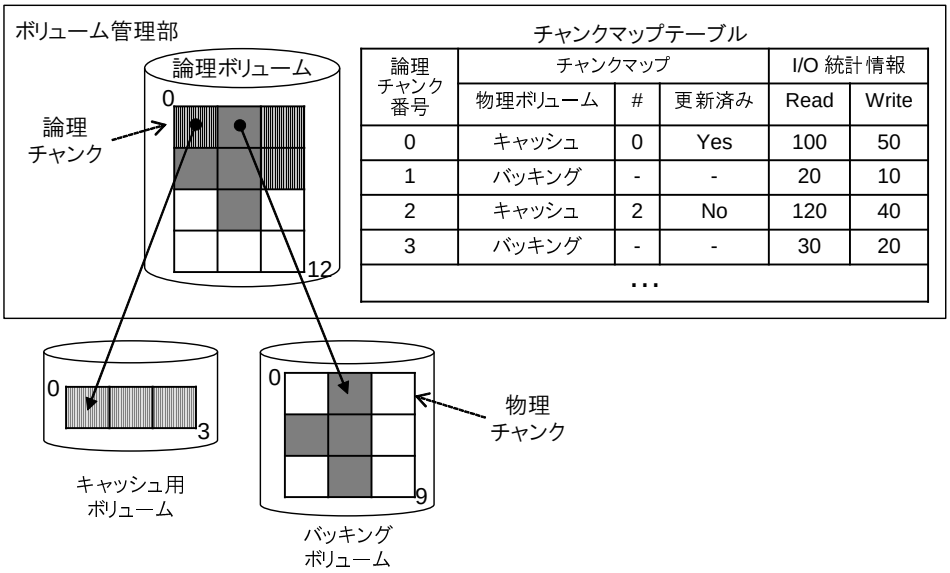


図 3.4: ボリューム管理概略

ぞれ固定長のチャンク列として扱う。論理ボリューム上の各チャンク（論理チャンク）は、物理ボリューム上のチャンク（物理チャンク）と対応付けられる。この対応は、ボリューム管理部がチャンクマップテーブルとして保持する。論理ボリューム上のチャンクと、バッキングボリュームのチャンクのアドレスは、線形でマッピングされる。すなわち、バッキングボリューム上で連続するチャンクは、論理ボリューム上でも連続するようにする。これは、論理ボリュームへのシーケンシャルアクセスが、バッキングボリュームに対してもシーケンシャルアクセスを維持できるようにする設計である。論理チャンクのうち、プリフェッチ対象として選択されたチャンクは、チャンクマップテーブルの記載に従いキャッシュ用ボリューム上のチャンクに再マップされる。反対にキャッシュ用ボリューム上にマップされた論理チャンクがキャッシュから追い出される場合、再びマップ先は元のバッキングボリュームの物理チャンクに戻る。チャンクマップテーブルで

は、キャッシュ用ボリューム上にマップされた各チャンクに対してプリフェッチ以降のデータ更新の有無をフラグとして保持しており、それにより論理チャンクがキャッシュ用ボリュームから追い出される場合、バッキングボリュームに対する Writeback 処理の要否を判定する。

ボリューム管理部は、それ自体が直接プリフェッチ対象のチャンクを指定するわけではなく、外部から階層移動を指示するインターフェースを設けている。キャッシュ割り当て管理部が論理チャンクのプリフェッチをこのインターフェースを介して指示すると、ボリューム管理部は下記の手順によりキャッシュ用ボリュームとバッキングボリュームの間におけるチャンクの階層移動を行う。まず、アプリケーションによる対象論理チャンクへの I/O 要求を一時的にブロックする。この間の I/O 要求は、階層移動が完了するまで待たされる。そのため応答時間が長くなるが、I/O はエラーとはならないので、アプリケーションの動作に不具合を生じることはない。次に、キャッシュ用ボリューム上の空き物理チャンクを選択する。もしすでに全物理チャンクが使用されている場合、Least Recently Used (LRU) 順で代わりにキャッシュから論理チャンクを 1 つ追い出す。この追い出し対象となる論理チャンクに対しても、同様に一時的に I/O 要求をブロックしたのち、チャンクマップテーブル中で更新フラグがセットされている場合はデータをキャッシュ用ボリュームからバッキングボリューム Writeback 処理を行う。そしてチャンクマップテーブルのエントリを更新し、追い出し対象の論理チャンクへの I/O ブロックを解除する。その後、キャッシュ用ボリューム上の物理チャンクを確保したのち、バッキングボリュームの物理チャンクのデータをコピーし、同様にチャンクマップテーブルのエントリを更新し、プリフェッチ対象の論理チャンクへの I/O ブロックを解除する。これらの処理は、複数のプリフェッチ対象の論理チャンクに対し並列に処理を行うこともできる。

ボリューム管理部はこれらの処理に加え、論理チャンク毎に実行される I/O 回数・I/O 量の統計情報を Read/Write 別に集計する。この情報はやはり外部から参照可能となっており、キャッシュ割り当て管理部がプリフェッチ対象とする論理チャンクを選択するのに用いる。

3.5.2 キャッシュ割り当て管理部

キャッシュ割り当て管理部はキャッシュ用ボリュームに配置すべき論理チャンクの判定およびプリフェッチのタイムウインドウの再計算を行う。キャッシュ割り当て管理部は各仮想マシン内で動作するエージェントプログラムと通信し、仮想マシンの構成情報や統計情報を取得する。この情報の例には、ゲスト OS におけるファイルブロックと論理ボリューム上の LBA の対応関係や、アプリケーションによるファイルの利用状況等がある。また、キャッシュ割り当て管理部はボリューム管理部とも情報のやり取りを行い、論理チャンク毎の I/O 統計情報を取得する。3.4 節に記述するキャッシュ戦略に従い、キャッシュ割り当て管理部はキャッシュ単位毎にプリフェッチを行う論理チャンクを定め、ボリューム管理部にプリフェッチを指示する。

3.5.3 エージェントプログラム

エージェントプログラムは、ファイルシステムのレイアウト情報やアプリケーションによるファイルの利用状況、アプリケーションや OS の性能統計情報を収集し、ホスト OS 中のキャッシュ割り当て管理部に通知する。ファイルシステムのレイアウト情報の取得には、ゲスト OS が提供するシステムコールを陥る。アプリケーションに関する情報の取得方法はアプリケーション依存である。一般に、アプリケーションの設定ファイルやログファイルを参照するほか、アプリケーション自身が情報を返すインターフェースを備えるなら、そのインターフェースを通じて問い合わせることで取得できる。例えば Linux を用いたゲスト OS の性能統計情報は、例えば procfs [96] を利用して取得できる。

3.6　実装

本節では、ホスト・ゲスト OS として Linux、アプリケーションとして PostgreSQL を対象とし、提案手法の実装について説明する。

3.6.1　ボリューム管理部

ボリューム管理部は、Linux カーネルにおける Device Mapper [97] カーネルモジュールとして実装した。このモジュールは、キャッシュ用ボリュームとバッキングボリュームを設定するとそこから容量を指定して複数の論理ボリュームを作成することができる。キャッシュ機能を備えた論理ボリュームを作成する場合、Device Mapper の管理コマンドである `dmsetup` コマンドにて SSD と HDD 分をそれぞれキャッシュ用ボリュームとバッキングボリュームとして設定する。本モジュールは、`ioctl` システムコールによりアプリケーションに対するインターフェースを提供しており、外部からチャンクマップテーブルの変更や性能統計情報の取得が可能である。

本実装において、標準チャンクサイズは MB とした。小さいチャンクサイズは、より細粒度でのキャッシュ管理を行えるという利点があるが、チャンクマップテーブルや性能統計情報を格納するのにより多くのメモリ容量を要する。例えば、本実装ではチャンク毎に 24 バイトの管理情報を持つ。仮にチャンクサイズを 4 KB とすると、ディスク容量の 0.6% の管理情報をメモリに確保しなければならない。1 TB の HDD に対し、6 GB のメモリを消費することになるが、現状のサーバのディスクとメモリの容量を踏まえるとこの比率は大きすぎる。もう一つ小さいチャンクサイズの問題点として、バッキングボリュームである HDD は小サイズの I/O への性能が低いため、階層間のチャンクデータの移動の効率が下がる点がある。これらの理由を踏まえ、プリフェッチを効率よく行えるよう、1 MB のチャンクサイズを標準とした。

3.6.2　キャッシュ割り当て管理部

キャッシュ割り当て管理部はホスト OS 上でバックエンド動作するプログラムで、Python で実装した。キャッシュ割り当て管理部は定期的にホスト OS の性能情報を取得し、また ssh を介してエージェントプログラム経由でゲスト OS およびアプリケーションの性能情報を取得する。加えて、前述の `ioctl` システムコールを介し、ボリューム管理部にデータのプリフェッチを指示する。

キャッシュ割り当て管理部は 3.4 節で述べた、ウインドウ内におけるプリフェッチの実行割合 (r) 調整も行う。本実装おいては、Δr を定めるためのメトリック ($item$) として、秒間トランザクション実行数 (以下 TPS) およびキャッシュヒット率 (以下 CHR) を用いた。TPS はアプリケーション依存のメトリックであり、ゲスト OS 内で取得できる、一方の CHR はボリューム管理層で集計する。その際、小サイズの Read アクセスの性能向上を図るため、I/O の転送バイト数ではなく、I/O の実行回数を計測対象とする。これは、PostgreSQL のようなデータベースシステムの用途としてよくあるトランザクション処理や Decision Support System のような処理においては、同期的小サイズ Read 要求の実行回数が多く、かつそれらの応答性能がアプリケーション全体の性能に大きく寄与するためである [98, 99]。PostgreSQL を含めた近年の RDBMS は小サイズの Write 要求も実行するが、通常これらの RDBMS は WAL (Write Ahead Logging) 等の手法を用い先に同期的大サイズ Write を発行後、遅れて非同期に実行されるものであり、本来 RDBMS が対象とするクエリ処理を遅延させるものではない。また、HDD は大サイズ I/O を効率よく実行できるため、大サイズ Read を削減するためにキャッシュプリフェッチを行っても、限られたキャッシュ用ボリュームの容量を多く必要とする割に、性能改善効果が小さい。よって、データベースを用いるアプリケーション

全体の性能を考慮すると、小サイズ Read アクセス性能の改善が最大の要因となるため、Read I/O 実行回数を計測する。

プリフェッチウインドウの割合 r を計算するため、本実装においては以下のモデルを用いた。まず変数として以下を定める。$IOPS$ は各記憶メディアにおける最大 I/O 性能 (秒間 I/O 実行数) である。TPS は前述の PostgreSQL における秒間トランザクション実行数である。CHR は論理ボリュームに対するキャッシュヒット率である。p は 3.4 節で述べたウインドウの長さである。

$IOPS_{SSD}$ は $IOPS_{HDD}$ よりも十分に大きな値をとるため、以下では $IOPS_{SSD}$ を使い切る I/O 要求は発行されず、この値は無視できると考える。CHR は論理ボリュームに対する I/O 要求のうち、SSD に対し実行された I/O 回数の割合を用いる。反対に、HDD に対し実行された I/O 回数の割合は $1 - CHR$ となる。

ウインドウ中、プリフェッチ実行中はアプリケーション発行の I/O は 0 であると近似すると、ウインドウ内で平均してプリフェッチのために使用される HDD の帯域の割合は r であり、アプリケーションのために使用される HDD の帯域の割合は $1 - r$ となる。よって、近似的に以下の比例関係が成り立つと想定する。

$$TPS \approx IOPS_{HDD} \times \frac{1-r}{1-CHR}$$

$IOPS_{HDD}$ はメディアの特性で定まる値なので、定数とみなすことができる。その場合、上記式はさらに簡潔に $TPS \times (1 - CHR) \approx 1 - r$ と記述することができる。よって本実装では、TPS と CHR を測定し、それに基づき Δr および r を調整するようにした。Easing 関数 E は下記のように設定した。

- CHR が上昇中は、まだプリフェッチにより性能改善の余地があると考え、E は正の値を返す。すなわちプリフェッチの実行割合を増加させる。
- $TPS \times (1 - CHR)$ が上昇中である場合、同様にプリフェッチによる通常アプリケーション I/O 性能への影響より、プリフェッチによるアプリケーション性能改善効果のほうが大きいとみなし、やはり E は正の値を返す。
- CHR 及び $TPS \times (1 - CHR)$ が変化しなくなった場合、プリフェッチによる通常アプリケーション I/O 性能への影響を考慮し、r を徐々に減少させるよう E は負の値を返す。
- CHR が急激に変化した場合は、ワークロードが変化した場合とみなすことができる。その場合、E を大きく増加させ、一時的にプリフェッチを加速させる。

3.6.3 エージェントプログラム

エージェントプログラムは Python で実装した。本プログラムは、ゲスト OS 上で PostgreSQL にクエリを発行し、システムカタログ情報を取得する。PostgreSQL のシステムカタログ情報は、PostgreSQL が格納するテーブルやインデックスのデータを保持するファイル名の情報を備える。よって、ファイルシステムに対しにおいて `ioctl(FIEMAP)` を発行するとファイルブロックと仮想ボリューム上の LBA の対応関係が取得できるので、システムカタログ情報を合わせてテーブル・インデックスの格納された LBA を取得することができる。エージェントプログラムは、加えて Procfs からゲスト OS の統計情報を取得し、さらにベンチマークプログラムの実行ログを監視してベンチマークプログラムにおける実効性能を取得する。これらの情報は、キャッシュ割り当て管理部の指示に従い ssh 経由でホスト OS に送られる。

表 3.2: キャッシュ方式

		プリフェッチ	レイアウト認識	動的速度調整
(a)	FlashCache	N	N	N
(b)	単純プリフェッチ	Y	N	N
(c)	プリフェッチ (レイアウト認識あり)	Y	Y	N
(d)	プリフェッチ (レイアウト認識・速度調整)	Y	Y	Y

3.7　評価実験

本節では、性能測定実験の結果を示す。実験にあたり、表 3.2 に示す 4 通りのキャッシュ動作を評価した。“(a) FlashCache” は SSD をキャッシュとして用いる従来研究の一つで、プリフェッチを伴わず過去の I/O データをキャッシュに置く方式である [76]。また、ボリュームに対し適用される方式で、ファイルシステムやアプリケーションの動作は考慮しない。“(b) 単純プリフェッチ” は従来のプリフェッチ手法で、ファイルシステムのレイアウト認識や、動的速度調整は行わない。“(c) プリフェッチ (レイアウト認識あり)” はファイルシステムのレイアウトを認識し、前述のキャッシュ単位やチャンクの連続性を意識したプリフェッチを行う。“(c) プリフェッチ (レイアウト認識あり・速度調整)” は、加えて動的プリフェッチ速度調整も行う。

3.7.1　実験環境

本実験では、CPU として Intel Xeon E5-2620 を 2 基備え、メモリを 40 GB 持つサーバを用いた。メモリ容量のうち 95% は仮想マシンの使用メモリとして割り当てている。そのため、ホスト OS がページキャッシュとして利用できるメモリ領域はデータセット容量に対して 2 桁小さく、ページキャッシュの影響は無視できる。

ゲスト OS およびホスト OS は、いずれも CentOS 7.3 を用いており、カーネルバージョンは 3.10.0 である。記憶メディアとして、キャッシュ用に NVMe PCI SSD を用いた。バッキングボリューム用としては、SATA 7200 RPM の HDD を 6 基備え、RAID カードを用いて RAID0+1 構成にて単一のボリュームを構成した。

ゲスト OS 上のファイルシステムは ext4 を用いた。データベースとしては、PostgreSQL 9.3.16 を利用した。ホスト OS における物理・仮想ボリュームのチャンクサイズは、断りのない場合 1 MB を標準とする。PostgreSQL はテーブルやインデックス格納に 1 GB のファイルを用いる。また、ext4 はおよそボリューム上において 8 MB 単位で大ファイルのファイルデータをアラインさせる。そのため、1 MB のチャンクサイズは、このアラインメントの約数となっておりチャンク内に複数のファイルデータが混在する可能性は低い。

本実験にあたり、ベンチマークプログラムとして TPCx-V [100] を用いた。このベンチマークは、クラウド上で複数の仮想マシンが動作する様子を模したものとなっており、各仮想マシン内でデータベースを動作させる。図 3.5 に TPCx-V の構成を示す。

TPCx-V では、3 種類の仮想マシン計 13 個を用いる。コントローラ VM は、ベンチマーク全体の挙動を制御する仮想マシンで一つだけ存在し、Tier-A 仮想マシン群に Remote Method Invocation (RMI) 経由でフェーズの変更などを指示する。そのためコントローラ VM 自体の性能は、全体の測定結果に影響しない。

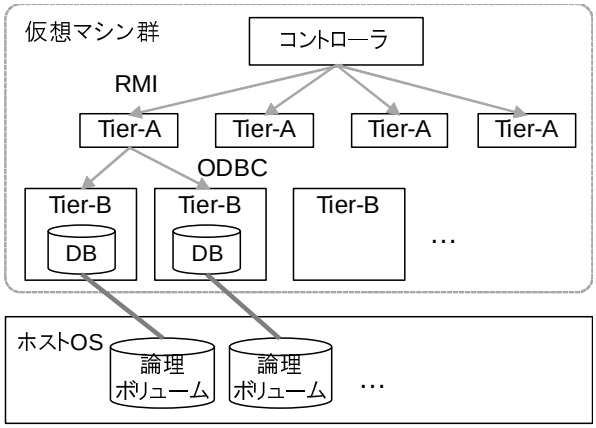


図 3.5: TPCx-V の構成

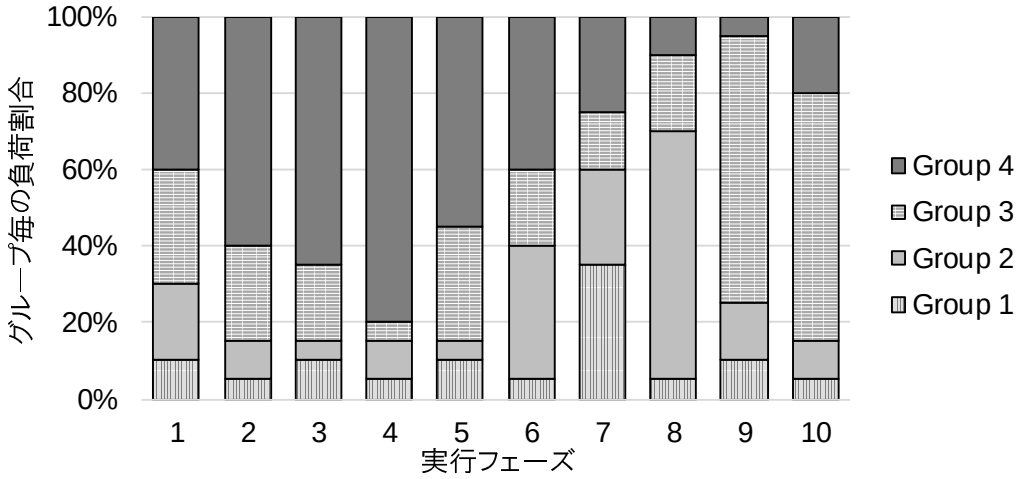


図 3.6: TPCx-V の負荷変動

Tier-A 仮想マシン群は四つの仮想マシンからなり、ビジネスロジックを模した動作を行う。Tier-A 仮想マシン群はデータベースを保持しておらず、Tier-B 仮想マシン群に Open Database Connectivity (ODBC) を用いたクエリ実行を行う。Tier-B 仮想マシン群は 8 個の仮想マシンからなり、それぞれ RDBMS を備える。これらの RDBMS がデータを置くファイルシステムが、本研究で提案するキャッシュ機構を備えた論理ボリュームとなる。

Tier-A の仮想マシン一つと、Tier-B の仮想マシン二つは合わせて一つのグループを構成し、Tier-A の仮想マシンは、同一のグループにある Tier-B 仮想マシンにのみクエリ実行を要求する。そのため、この実行環境では四つのグループが同時に動作することになる。

Tier-A の仮想マシンではトランザクションの比率が高いクエリを発行するスレッドと、分析処理を模したクエリを多数発行するスレッドが並列で動作する。その結果、Tier-B の仮想マシンが論理ボリュームに発行する I/O は、Write に比べ Read が圧倒的に多く、全体の約 80-85% が Read である。

Tier-B の仮想マシンが保持するデータ量はグループによって異なるが、全仮想マシン合計では 750 GB である。そのため、本実験にあたり SSD のキャッシュ容量はデータ量より小さい 100 GB を上限とした。

TPCx-V は 10 実行フェーズで構成されるテストセットを 2 回繰り返す。各実行フェーズにおいては、四

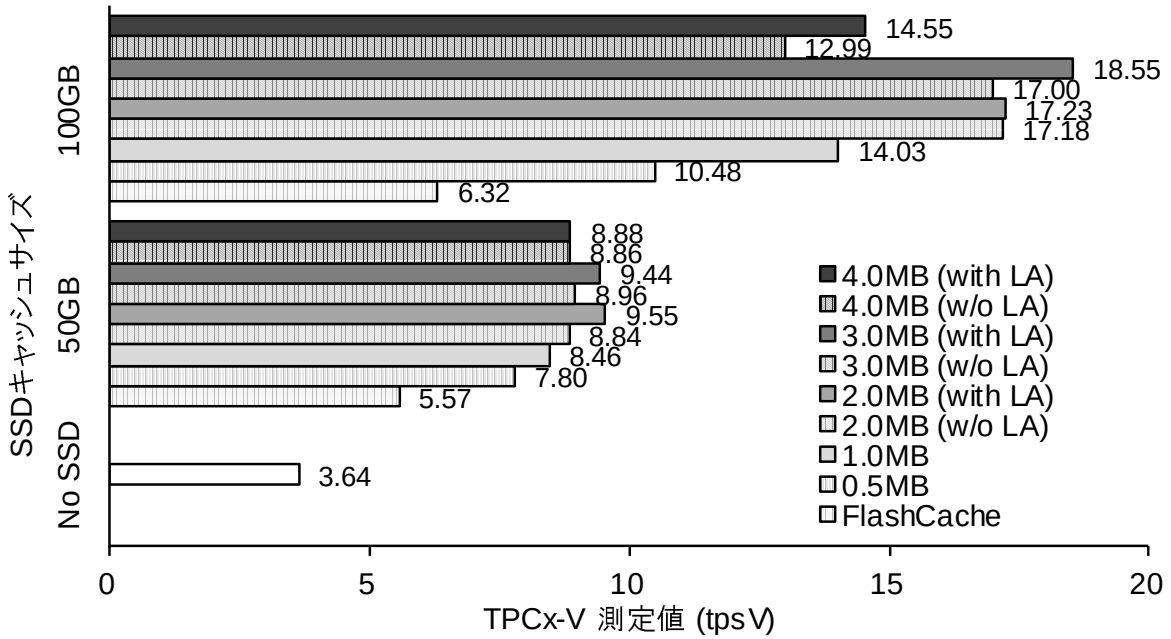


図 3.7: レイアウト認識の有無による TPCx-V 実行結果への影響

つのグループ毎に実行負荷の比率が変化する (図 3.6)。RDBMS の性能は一番遅いクエリの実行結果に引きずられるため、グループ間の実行負荷の変化に素早く追従し、キャッシュヒット率を上昇させることがベンチマーク性能すなわちアプリケーション性能の改善につながる。テストセットの実行時は、事前にデータセット生成を行う初期化フェーズと、100 秒間のウォームアップフェーズが実行される。本実験においては、この初期化及びウォームアップフェーズにおいてもプリフェッチ動作を行うものとする。

3.7.2　キャッシュ単位とレイアウト認識の効果

キャッシュ単位とレイアウト認識の効果を評価するため、ここではキャッシュの構成として (a) FlashCache, (b) 単純プリフェッチ (“w/o LA”), (c) プリフェッチ (レイアウト認識あり, “with LA”) について性能測定を行った。(b)(c) の評価においては、キャッシュ単位 N を 1 (1.0 MB) から 4 (4.0 MB) まで変化させた。加えて、標準のチャンクサイズより小さなチャンクサイズ・キャッシュ単位として 0.5 MB の構成も比較した。キャッシュ単位が 0.5 MB および 1.0 MB の場合、このサイズはチャンクサイズと一致する。すなわち、これらは論理的に隣接するチャンクを考慮したプリフェッチを行わないことに相当するので、レイアウト認識を行わないと同義である。SSD のキャッシュサイズは、100 GB、50 GB、0 (キャッシュなし) で測定した。本実験ではプリフェッチの実行時間の割合は $r = 0.4$ で固定させている。

図 3.7 に実行結果を示す。横軸の “tpsV” は、TPCx-V 固有のメトリックスで、秒間あたりに完了したトランザクションから計算される値である。よって、この値が大きいほど、RDBMS の処理が高速であり、アプリケーションの動作が速くなることが期待できる。この実験中、CPU・メモリ・ネットワークボードおよび他のコンポーネントが性能ボトルネックの要因となることはなく、ストレージへの I/O がボトルネックとなっていることは確認済みである。CPU やメモリがボトルネックでないことから、提案手法による統計情報の分析に伴う CPU の処理負荷は無視できる。実行結果より、一般的に SSD キャッシュの容量が増えると、キャッシュの手法に関わらず性能が上昇することがわかる。ただし同一のキャッシュ容量においては、キャッシュ手法により性能差が表れる。提案プリフェッチ手法は、従来手法の FlashCache を大きく上回っ

ており、キャッシュサイズ 100 GB の構成においては最大で 2.93 倍 (18.55 ÷ 6.32) の性能改善を達成した。

キャッシュサイズが 50 GB および 100 GB の場合、キャッシュ単位が中サイズ (2.0 MB, 3.0 MB) の構成においてプリフェッチが良い結果を示している。一般に、HDD の性能特性により、大きいキャッシュ単位は階層間のデータ移行の効率を向上させる。実際、キャッシュ単位が 0.5 MB の場合は移行の速度は 33 MB/s にとどまるが、3.0 MB の場合は 54 MB/s となっていた。(階層間の移行速度は、各メディアの入出力性能より大幅に小さい。これは SSD 上のチャンクと HDD 上のチャンクを交換する際、SSD と HDD に Read と Write を 1 回ずつ、計 4 回の I/O を要するためである) 一方、キャッシュ単位を多くしすぎると本来アクセス局所性の高い範囲を超えて多くのデータを階層間移行させるため、キャッシュヒット率が低下し、アプリケーションの性能を悪化させてしまう。

加えて、ウォームアップフェーズではキャッシュ単位が 4.0 MB の場合の階層移行の速度が 3.0 MB の場合よりも低下した。後者は 79 MB/s に対し、前者は 63 MB/s にとどまる。これは HDD からのデータ読み込みにおける Tail latency の影響である。測定中、一部のセクタへの読み込みが数十 ms かかるケースに遭遇した。これは、RAID カードや HDD コントローラ内部のキャッシュにないデータを参照する場合に発生する時間であり、キャッシュ単位が大きすぎるとそのような読み込みが増加するため 4.0 MB の場合の階層移行の速度が低下した。

提案方式であるファイルシステムのレイアウト認識を用いたプリフェッチでは、レイアウト認識を用いない場合に比べ性能向上効果が見られた。キャッシュサイズ 100 GB、キャッシュ単位 3.0 MB の構成でレイアウト認識を伴うことで最良の性能値 18.55 tpsV を達成しているほか、キャッシュ単位 4.0 MB の構成ではレイアウト認識の有無で 12% の性能差が生じている。総じて、キャッシュ単位を大きくするほどレイアウト認識の効果が増加する。ゲスト OS 上におけるファイルデータが仮想ボリューム内で断片化している場合、レイアウトを認識しないプリフェッチではアクセス頻度の高いファイルと関係ない仮想ボリューム上の領域をプリフェッチしてしましい、キャッシュヒット率の向上効果を下げてしまう。またこの影響はキャッシュ単位が大きいほど顕著となる。そのため、性能改善効果は、実験中最大のキャッシュ単位である 4.0 MB で最大となったといえる。

図 3.8 は、キャッシュサイズ 100 GB の構成においてキャッシュ単位のサイズとレイアウト認識の有無を切り替えて測定した場合における TPCx-V の実行フェーズの時系列における推移と秒間トランザクション実行回数 (TPS) の変化を示す。横軸が時間軸であり、縦軸が秒間トランザクション実行回数を示す。tpsV は、全実行フェーズ間における TPS の累積値を元に算出する値のため、tpsV の値は瞬間的な TPS の値とは一致・比例せず、折れ線グラフと横軸に囲まれた面積に比例した値となる。ベンチマークの動作開始後、先頭の第 8 フェーズまでの間は、まだ SSD キャッシュに十分な階層移動が行われていない状態のため、どの構成においても TPS 値は増加していく。この間、キャッシュ単位が中程度 (2.0 MB および 3.0 MB) の構成は増加ペースが速く、その結果が高い tpsV 値に反映されている。第 8 フェーズ以降は、SSD のキャッシュが一度全領域埋まった状態になり、以後は安定した処理性能を示す。

しばしば実行フェーズの境界において TPS 値が非連続的に増減する様子が見られる。これは実行フェーズ間において負荷の高まるグループが変化した結果、キャッシュヒット率に変化が生じるためである。しかし、その後は変化した I/O パターンに応じてプリフェッチ・キャッシュ管理を行うため、徐々に性能は回復する。最良ケースであるレイアウト認識を含むキャッシュ単位 3 MB 構成では、他の構成より全フェーズにおいて総じて TPS 値が高く (例:第 10 フェーズ) また他の構成よりもフェーズ境界後の性能値の回復が速い (例:第 16 フェーズ)。これらの結果により、tpsV 値においても高い値を得ることができている。

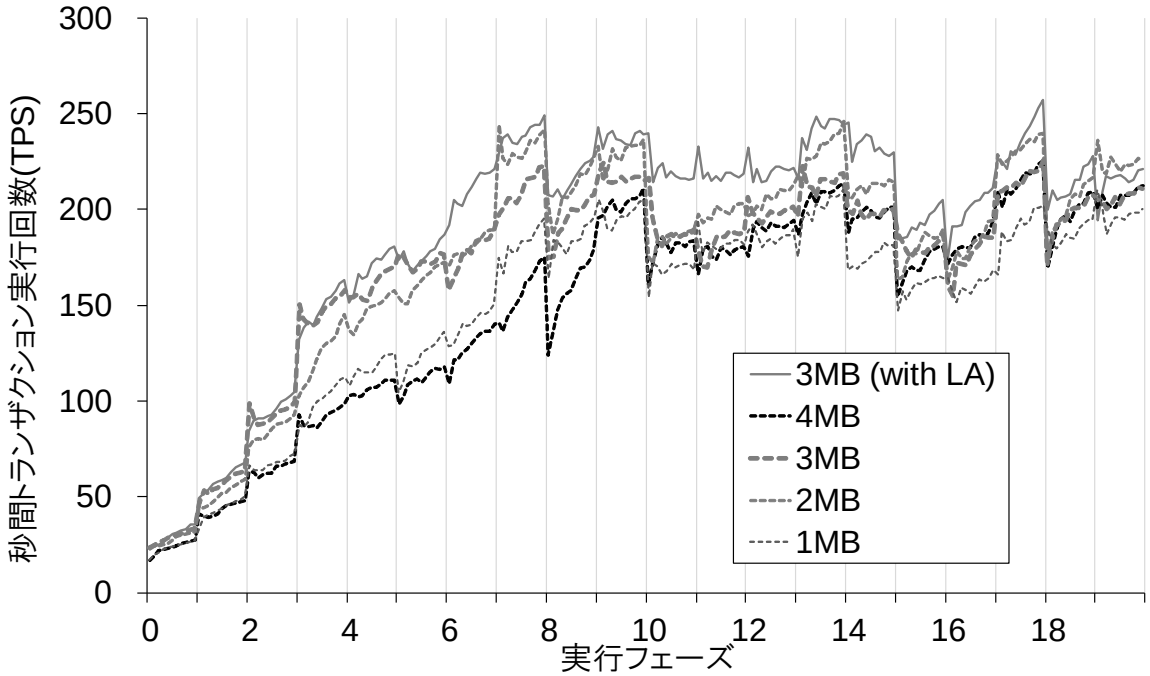


図 3.8: 実行フェーズの推移とトランザクション処理性能

3.7.3　キャッシュヒット率と性能メトリクス値

本項では、プリフェッチ速度の調整に用いる性能メトリクス値について、TPS 値が安定した状態でメトリクス値も安定することを確認する。図 3.9 は、ベンチマーク実行中におけるキャッシュヒット率及び性能メトリクス値 ($TPC \times (1 - CHR)$) を時系列でグラフとしたものである。測定時のキャッシュ管理は、レイアウト認識を用いたキャッシュ単位 3 MB の構成であり、キャッシュサイズ 50 GB と 100 GB の 2 通りを用いた。いずれも、ベンチマーク開始後第 8 フェーズ程度まではキャッシュヒット率は上昇する。また、当然ながらキャッシュ容量が多い構成のほうがキャッシュヒット率は高い。その後キャッシュヒット率は凡そ安定するが、特にキャッシュ容量が少ない 50 GB の構成では多少の増減が発生し、実行フェーズの境界において急減することもある。この挙動は、図 3.8 における TPS 値の増減の傾向と一致している。性能メトリクス値については、キャッシュヒット率の安定以降は、キャッシュ容量によらず凡そ近い値をとる。これは、本性能メトリクスの算出方法は、キャッシュ容量を問わず利用できることを示す。

3.7.4　動的プリフェッチ速度調整

本節では、TPCx-V における動的なプリフェッチ速度調整の効果を評価する。以下、キャッシュの構成としてキャッシュ単位 3.0 MB、レイアウト認識有効を固定する。キャッシュサイズは 50 GB と 100 GB の 2 通りを試す。測定において、動的なプリフェッチ速度調整 (性能メトリクス値を用いた、プリフェッチ実行比率 r の変更) を有効にした場合と、r に固定値として 0.2 から 0.8 の値を持たせた場合を比較する。

図 3.10 に測定結果を示す。r を固定値とした場合、キャッシュサイズ 100 GB では $r = 0.6$、50 GB では $r = 0.2$ が最良の結果となった。一方、動的速度調整が有効なケースでは、いずれのキャッシュサイズにおいても固定値よりも高い結果を示した。最良の結果 19.91 tpsV は、レイアウト認識も動的速度調整も行わ

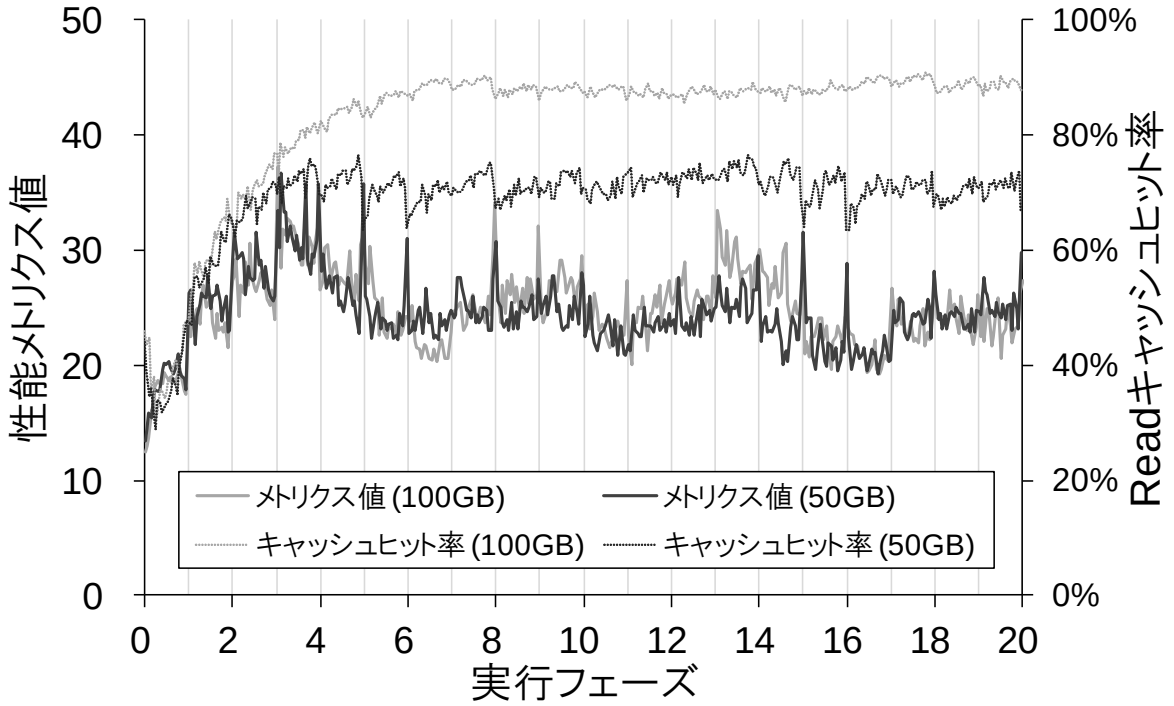


図 3.9: キャッシュヒット率と性能メトリクス値の推移

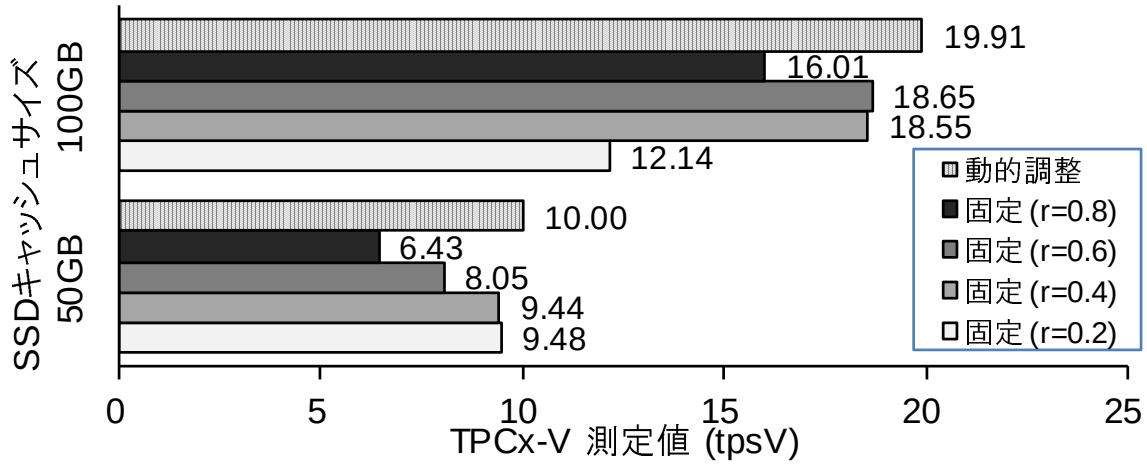


図 3.10: プリフェッチ速度調整を伴う TPCx-V の実行結果

ない場合である、図 3.7 における 17.00 tpsV と比較すると、同一のキャッシュサイズであるにも関わらず 17.1% の性能改善効果を得たことになる。

図 3.11 は、キャッシュサイズ 100 GB の構成におけるトランザクション実行性能の推移を示す。r が大きい場合 (0.8)、初期の実行フェーズでは性能上昇が速い。しかし、性能が安定状態に入ると以後は性能が上がらず、およそ 130 TPS 程度にとどまり他の構成よりも低い性能となる。これは、キャッシュに有効なデータを移行する初期の実行フェーズではプリフェッチを積極的に行い、その後はプリフェッチが HDD の帯域を消費し、アプリケーション本来の I/O 要求を妨げないようにプリフェッチの実行を抑えることが有効な戦略であることを意味する。提案する動的なプリフェッチ速度調整は正にこの戦略に対応づく動作を行う。

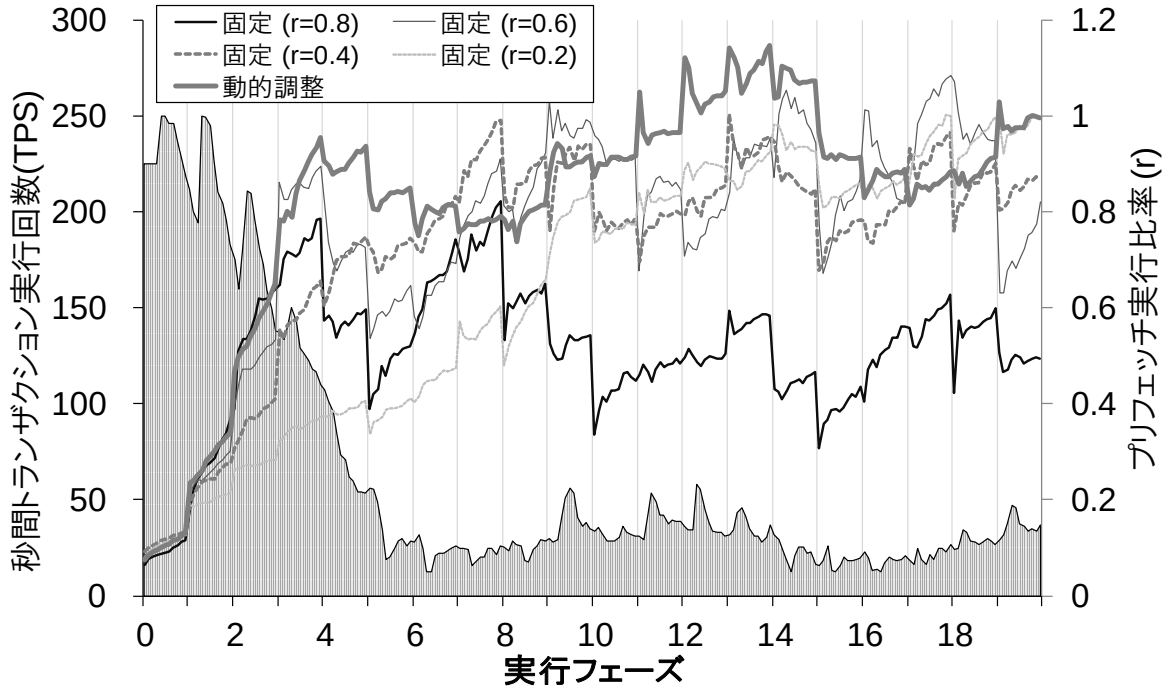


図 3.11: 動的速度調整とトランザクション実行性能の対応

図中緑色に塗られた領域は、動的速度調整における r の推移を示す。この結果は、まさに上記有効な戦略にのっとった動作を行っている。最初の数フェーズでは r が 1 に近く、急速にプリフェッチを行う。その後、徐々に r は減少していき、安定状態では r は 0.1 から 0.2 程度の低い値にとどまる。r はしばしば実行フェーズの強化をまたぐ際に一時的に増加しているが、これはキャッシュヒット率の急低下をワークロードの変化とみなし、急速にヒット率を回復させるための措置である。これらの動作の結果、提案手法は他の手法より全体として高いトランザクション実行回数を保持し、r に固定値を用いる場合に比べ、高い tpsV 値を得ることができた。

3.8　まとめ

階層ストレージの効果を向上するにはキャッシュ戦略が重要である。クラウドコンピューティング環境においては、ホスト OS 上で複数の仮想マシン及びアプリケーションが並列に動作することから、ホスト OS でディスクアクセスの局所性を観測してキャッシュヒット率を高めることは難しい。

提案手法では、ゲスト OS におけるファイルシステムの仮想ボリューム上のレイアウト情報を用いてアクセスの空間的な局所性をホスト OS にて復元する。そして物理ボリュームではなくゲスト OS におけるファイルブロックの論理的な連続性に基づき、HDD のデータを SSD にプリフェッチする。加えて、提案手法では過剰なプリフェッチで HDD の帯域を消費することによるアプリケーション性能の低下を防ぐため、動的にプリフェッチの実行速度を調整する。そのため、アプリケーションおよびゲスト OS・ホスト OS について性能特性情報を取得し、直近の情報と、過去の重み付き統計情報を比較して、プリフェッチにより性能向上が見込める場合にプリフェッチの速度を増加させる。

本研究では、提案手法のプロトタイプとして、Linux および PostgreSQL を対象に階層ストレージ管理機能を実装し、クラウドコンピューティング環境を想定した TPCx-V ベンチマークにおいて性能評価を

行った。提案するファイルシステムのレイアウト認識と動的速度調整を伴うプリフェッチを行うことで、それらを行わない単純なプリフェッチと比較して同容量の SSD キャッシュを用いた場合で最大で 17.1% の性能向上効果を確認できた。また、Linux 上で利用可能なストレージに対するキャッシュ機構の先行研究 FlashCache と比較し、3.15 倍の性能を達成した。

提案手法では、通常ホスト OS から観測できない仮想マシン内のアクセスの局所性やアプリケーションの動作を、エージェントを仮想マシン内に動作させるという形でホスト OS が観測できるようにしている。一方で、仮想マシン内のアプリケーション自身は改造することなく提案手法が適用可能であり、その点でアプリケーションの透過性は保たれている。そもそもデータの階層間移動は仮想マシンの外部で行われているため、アプリケーションは自身が階層ストレージを利用していることすら認識しない。また、データの階層間移動もチャンク単位で行うため個々の移動中チャンクに対する応答待ち時間は僅かであり、その点でもアプリケーション動作に不具合を与える危険性は無い。このように、提案手法ではファイルに着目した性能向上を、アプリケーションに悪影響を及ぼすことなく実現した。

第 4 章

User-space Library File System for Low Latency Storage Service Leveraging Non-Volatile Memory

本章では、近年実用化され普及が始まった不揮発性メモリに適した、高いアクセス性能を提供するストレージアクセスのためのソフトウェアインターフェースについて論ずる。

4.1 はじめに

従来の計算機において、I/O 要求への応答時間を占める支配的な要因は HDD アクセスの応答待ち時間であった。そのため、ソフトウェアでは一般に OS が HDD に発行する I/O のスケジューリングやページキャッシュ・バッファキャッシュ管理を行うことで HDD アクセス回数を削減し、応答時間の短縮を図ってきた。しかし、メモリインターフェースでアクセスできる不揮発性メモリ (Non-Volatile Memory, NVM) の技術開発が進み、メディアの応答時間が短くなると、相対的にストレージアクセスに関するソフトウェアスタックの処理オーバヘッドがボトルネックとして顕在化するようになった [101]。

データベースやメールサーバ、ソフトウェア開発などのアプリケーションは、多数のファイルを参照・変更し、大量の小サイズ I/O を発行する [29, 30]。そのためアプリケーションの処理性能改善にはこれらファイルアクセスの応答時間短縮が不可欠である。NVM の短い応答時間を活かしファイルアクセスの応答時間を短縮するには、ストレージアクセスに関するソフトウェアスタックの処理オーバヘッド削減が必要である。加えて、NVM を実用的にかつ広範囲なアプリケーションで活用するためには、アプリケーションの改変を要せず適用可能なソフトウェアスタックでなければならない。

NVM を扱うソフトウェアスタックについては、様々な先行研究がある。NVM を仮想的なブロックデバイスとみなしてファイルシステムを構築する手法 [102] は、アプリケーションに対し従来ファイルシステムと同等のインターフェースを提供できるため、アプリケーションの改変を要せず NVM を利用することができる。しかし、従来のファイルシステムは HDD や SSD を前提としてセクタ単位のアクセスを行うよう設計されており、バイト単位でアクセスできる NVM の特性を活かせずデータ入出量が増加し処理効率が悪い。また、ファイルアクセス時に逐一システムコールを呼ぶことで、カーネルモードへのコンテキストスイッチや、ディスク応答待ちのためのプロセス切り替えが生じ、これらが処理オーバヘッドとなる。異なるアプローチとして、NVM の性能特性に適したファイルシステムが複数提案されている [103, 104, 105]。これらはバイト単位でメディアにアクセスする前提でデータ構造を再設計し、データ入出力量を削減して処理効率を改善している。一方で、従来のファイルシステム同様にこれらは OS 内で実装されるために、アク

セス時の OS へのコンテキストスイッチは軽減されない。アクセス時のコンテキストスイッチを削減する手法として、NVM 上のファイルデータブロックをプロセスのアドレス空間にマップして、ユーザーモードのプログラム上からファイル内のデータをメモリとして直接参照する Direct Access (DAX) [106] がある。DAX はファイルのメタデータのアクセスには性能改善効果がなく、また従来と異なるインターフェースでアクセスするため、アプリケーション側に改変を要する。さらに応答性能を改善する手法として、ファイルアクセスのインターフェースを用いず、DAX でマップした領域に、固有のデータ構造を構築するライブラリ [107, 108] やプログラミングのフレームワーク [109] も研究されている。これらは高い性能改善効果が期待できるものの、アプリケーションの改変を要し、かつその作業は一般的なアプリケーションプログラマには難しいといわれている [109]。

本章では、バイト単位アクセスが可能な NVM の性能特性を活かし、かつ既存アプリケーションの改変を要せず適用可能なストレージのソフトウェアインターフェースとそのソフトウェアスタックの構成を提案する。本研究では、多くのアプリケーションがアプリケーション固有のデータを一部のディレクトリに排他的に格納していることに着目し、アプリケーション毎にプロセス固有のディレクトリを提供する方式をとる。本ディレクトリに対応するファイルシステムを構成するデータは、NVM 上のプロセスが占有する領域に対し、CPU のメモリ Load/Store 命令を用いて格納する。この処理は OS へのコンテキストスイッチ不要であるため、性能向上効果を高める。本方式では、ファイルシステムの機能をユーザーモードライブラリとして提供する。本ライブラリは従来の POSIX インターフェース [11] の関数呼び出しの延長で動作するため、アプリケーションの改変や再コンパイルを要せず適用可能である。

本研究では、提案方式を Linux 及び glibc に適用し、先行研究に対する性能評価実験を実施した。filebench ベンチマークを用いた実験において、提案手法は一般的な HDD 向けファイルシステム XFS や NVM 向けの NVM 向けのファイルシステムの先行研究 NOVA に対し、少ない NVM のアクセス回数やタスクスイッチ回数でのファイルアクセスを実現した。その結果 XFS に対して最大で 6.48 倍、対し最大で 1.47 倍の性能改善効果を得ることができ、提案手法の有用性を示した。

4.2　関連研究

NVM の活用に関するストレージのインターフェースについては、ハードウェアや記憶メディアに関する研究からソフトウェアに関する研究まで様々な先行研究がある。本節では、ハードウェアのインターフェースとしては DRAM 同様にメモリインターフェースを持つものに範囲を限定し、その中で NVM の活用に関するソフトウェアの従来研究を分類する。

表 4.1 に方式の分類と、それぞれが満たす特性を示す。表中の各特性は下記の通りである。“✓” は各方式が特性を満たしており、“✗” は満たしてないことを意味する。特性 (a)〜(d) はアクセス性能に関するもので、それぞれの意味は下記の通りである。

(a) **効率的データ構造**:　従来ファイルシステムは、ディスクをセクタ単位で読み書きする前提で設計されており、より小サイズの I/O 要求に対して実際のディスク読み書きのデータ量が増え、効率が低い。NVM はバイト単位でアクセスできるため、その特性を活かしたデータ構造を擁する方式はメディアの性能面で優れていると分類する。

(b) **コピー回数**:　アプリケーションがファイルのデータを読み書きする際、通常のファイルシステムにおいては、NVM・ページキャッシュ・標準ライブラリのバッファ・アプリケーション内のバッファの四つのメモリ領域の間で、 計三回のデータコピーが生じる。方式によってはより少ないコピー回数で読み書きが行え、効率がよく性能面で優れているといえる。

表 4.1: NVM 活用方式の分類

方式	アクセス性能				使いやすさ	
	(a) 効率的データ構造	(b) コピー回数	(c) コンテキストスイッチレス	(d) 特化チューニング	(e) プロセス間共有	(f) アプリケーション変更不要
仮想ブロックデバイス方式	✗	3	✗	✗	✓	✓
NVM 向けファイルシステム	✓	2〜3	✗	✗	✓	✓
DAX 対応ファイルシステム	✗	1〜2	✓	✗	✓	✗
固有データ構造構築	✓	1	✓	✓	✗	✗
部分ライブラリファイルシステム	✓	1	✗	✗	✗	✓
ライブラリファイルシステム	✓	1	✓	✓	✗	✓

(c) **コンテキストスイッチレス**: アプリケーションからファイルのデータを読み書きする際、通常のファイルシステムではシステムコール呼び出しによるカーネルモードへのコンテキストスイッチが生じたり、他プロセスと排他処理・状態の一貫性維持のためのスリープによるコンテキストスイッチが生じる。方式によってはこれらのコンテキストスイッチを省略でき、性能面で優れているといえる。

(d) **特化チューニング**: アプリケーションのアクセスパターンに特化したパラメータや構成のチューニングを行うことができる場合、優れていると判断する。

表中の特性のうち、(e)・(f) はアプリケーションからの利用のしやすさに関するもので、それぞれの意味は以下の通りである。

(e) **プロセス間共有**: データがプロセス間で共有でき、同時にアクセスできる場合、優れていると判断する。

(f) **アプリケーション変更不要**: 各活用法において、アプリケーションのソースコード変更や再構成が不要で適用できる場合、優れているとみなす。

以下、既存研究を各活用方式で分類し、上記特性の有無を整理する。

4.2.1 仮想ブロックデバイス方式

本方式は、NVM を仮想的にブロックデバイスとして扱い、XFS など従来のディスク向けのファイルシステムを介して利用する。これは従来から DRAM を対象に RAM Disk として利用されてきた方式を、NVM に適用したものであり、NVM の利用法として最も基本的な使用法とされている [102]。

本方式の利点として、主要な OS では NVM 向けの仮想ブロックデバイスドライバが標準で提供されていること、また、アプリケーションからは従来のファイルシステムのインターフェースを用いてアクセスするため、利用に際し OS・アプリケーションともに改変不要であることがある。また、プロセス間共有も可能であることから、使いやすさにの特性 (e)(f) を満たす。一方性能に関しては、特性 (a)(b)(c)(d) をいずれも満たすものではなく、他の方式と比較して性能は低い。

4.2.2 NVM 向けファイルシステム方式

本方式は、NVM 向けの専用ファイルシステムを構築する方式である。これらの方式は、NVM がバイト単位でアクセス可能という特性を踏まえてデータ構造やデータ処理順を効率化することで、セクタ単位ア

クセスを前提する従来ファイルシステムよりも効率的に NVM へのアクセスを行うことができる。そのため、特性 (a) を満たす方式である。また、実装によってはページキャッシュを要しないため、特性 (b) に上げたコピー回数は従来ファイルシステムよりも少ない 2 回で済むものがある。本方式に該当する先行研究には、BPFS [103]、SCMFS [104]、NOVA [105] などが挙げられる。本方式ではファイルアクセスのインターフェースは従来と同等であることから、特性 (c)(d)(e)(f) も仮想ブロックデバイス方式と同じとある。

4.2.3　DAX 対応ファイルシステム

Direct Access (DAX) は、メモリ上にファイルシステムを構築することを想定したファイルシステムにおける OS 内のメモリ管理方式で、以下の 2 点の特徴がある [106]。

- ファイルデータにアクセスする場合、ページキャッシュを介さない。
- `mmap` 関数実行時、ファイルのデータを構成するメモリ領域が直接アプリケーションのアドレス空間にマッピングする。

前者の特徴により、ページキャッシュに対するデータのコピーが削減され、後者の特徴によりデータアクセスが CPU の Load/Store 命令で完了するためカーネルモードへのコンテキストが不要となり特性 (b)(c) を改善する。DAX は近年の Linux および Windows 及びその主要ファイルシステムで標準サポートされている。また、上述の NVM 向けのファイルシステムでありながら、DAX 対応も行う先行研究に UMFS [110] がある。

DAX の性能面での利点を最大限に生かすためには、`mmap` の使用が前提となる。`mmap` を用いることで、標準ライブラリのバッファを介する必要がなくなり、特性 (b) に示すデータのコピー回数は最小の 1 となる。そのためにはアプリケーションが `mmap` 未対応の場合、性能最大化のトレードオフとして変更が必要となる。DAX は、データアクセスはコンテキストスイッチを生じないが、ディレクトリツリーの名前空間やファイルのメタデータ操作は従来通りの動作を行いカーネルモードへのコンテキストスイッチが生じるため、特性 (c) の改善効果は部分的な物に留まる。

4.2.4　固有データ構造構築方式

固有データ構造構築方式は、NVM 上で固有のデータ構造を構築し、ファイルシステムとは異なるインターフェースにて NVM を利用する方式である。多くは DAX と併用され、`mmap` によりアドレス空間にマッピングしたメモリ領域を用いる。構築するデータ構造によって様々な先行研究がなされている。例えば不揮発ヒープを提供する方式には pVM [107] や HEAPO [108] があり、Key Value Store の構築例として pmemkv [111] がある。また、NVM を前提としたデータ構造を構築するためのプログラミングのデザインパターンも提案されている [109]。これらはアプリケーションに閉じたデータ構造を構築するため特に性能面で優れており、特性 (a)(b)(c)(d) の全てにおいて優れている。

一方で、本方式はそれぞれ固有のインターフェースに対応するために大幅なアプリケーションの改変を要し、かつその実装難度や対応コストは高いと言われている [109, 112]。また、プロセス間のデータ共有も行えない。そのため特性 (e)(f) の点で劣る。

4.2.5　ライブラリファイルシステム方式

ライブラリファイルシステム方式は、ファイルシステムの機能をユーザーモードのライブラリとして実装するものである。先行研究として、ユーザーモードのライブラリが、ファイルシステムの機能を部分的に担

当する方式 (以下、部分ライブラリファイルシステム方式) と、全機能を担当する方式 (以下、ライブラリファイルシステム方式) がある。

部分ライブラリファイルシステム方式の先行研究には SplitFS [113] や ZoFS [114] がある。これらはデータアクセスはユーザーモード内で直接 NVM を参照して処理を完結させ、メタデータに関する処理のみカーネルモードで実行している。そのため、プロセス間でデータ共有を行いたい場合、ディレクトリの名前空間やアクセス管理についてカーネルが提供する標準のファイルシステムと同程度の機能を提供することができる。そのため、特性 (a)(b)(e)(f) で優れている。一方で、メタデータアクセスが多いアプリケーションに対しては依然としてコンテキストスイッチが生じるため、性能向上効果は限定的である。

ライブラリファイルシステム方式では、ファイルアクセスはプロセス内で完結することから、部分ライブラリファイルシステム方式に加えて特性 (c)(d) についても改善できる。本研究の提案手法も本方式に分類される。本方式の先行研究として、ユーザーモードでのストレージアクセス機能を提供するライブラリ SPDK [115] はファイルシステムのサブセット BlobFS を提供している。ただし BlobFS は専用インターフェースで提供されるためアプリケーションの改変を要する。pmemfile [116] は、POSIX インターフェースでアクセス可能なファイルシステムを提供するユーザーモード動作のライブラリである。pmemfile では、標準 C ライブラリのシステムコール呼び出し部に動的にバイナリパッチを充てることで、アプリケーション側を無改造で利用可能としている。pmemfile の設計方針や制限は、提案方式と共通部分が多い。ただし、pmemfile は継続的なメンテナンスがされておらず小規模なマイクロベンチマークが行われたのみで、実際のアプリケーションへの適用事例がなく、その定量的な比較・評価を行うことができない。[*1]。

4.3 アプリケーションのデータ配置の分析

データベースやメールサーバ等の一部のアプリケーションでは、ディレクトリツリーの一部に、同一の所有者・アクセス権で構成されたファイル群としてアプリケーション固有のデータを配置することが知られている [114]。これは、これらのアプリケーションが実質的にそのツリー下を排他的に使用することを意味している。

実際に、Ubuntu Linux 環境において広く使用されている 2 つのデータベースプログラムである MongoDB [117] と MariaDB [118] を対象データ配置について調査した。いずれも、アプリケーションの設定ファイルは`/etc` ディレクトリ、ログファイルは`/var/log` ディレクトリ以下に格納している。これらのディレクトリは、設定ファイル及びログファイルの格納先として多数のアプリケーションで共有されている。

一方、データベースに格納されるデータは、それぞれ固有のディレクトリに格納している。リスト 4.1に MongoDB のデータが格納される`/var/lib/mongodb` ディレクトリ、リスト 4.2に MariaDB のデータが格納される`/var/lib/mysql` ディレクトリにおいて `ls` コマンドを実行してファイル一覧を出力した結果を示す。いずれもファイルの所有者はほとんど mongodb 及び mysql となっている。また、ファイルのパーミッションもその他のユーザーには付与されていない。即ち、これらのファイルは mongodb 及び mysql ユーザーのみが参照可能である。加えて、MongoDB のデータには mongod.lock や WiredTiger.lock のように排他処理を想定したファイルが格納されている。これらの事実から、MongoDB や MariaDB は他のプログラム、他のユーザーとディレクトリ内のデータを共有することを想定しないことが推測できる。

リスト 4.1: MongoDB のファイル配置

```
/var/lib/mongodb$ ls -l
total 184
```

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

```
-rw------- 1 mongodb mongodb 16384 Jun  1 22:52 collection-0-7431864268696936249.wt
-rw------- 1 mongodb mongodb 16384 Jun  1 22:52 collection-2-7431864268696936249.wt
-rw------- 1 mongodb mongodb  4096 Jun  1 22:52 collection-4-7431864268696936249.wt
drwx------ 2 mongodb mongodb    48 Jun  1 22:52 diagnostic.data
-rw------- 1 mongodb mongodb 16384 Jun  1 22:52 index-1-7431864268696936249.wt
-rw------- 1 mongodb mongodb 16384 Jun  1 22:52 index-3-7431864268696936249.wt
-rw------- 1 mongodb mongodb  4096 Jun  1 22:52 index-5-7431864268696936249.wt
-rw------- 1 mongodb mongodb  4096 Jun  1 22:52 index-6-7431864268696936249.wt
drwx------ 2 mongodb mongodb   110 Jun  1 22:50 journal
-rw------- 1 mongodb mongodb 16384 Jun  1 22:52 _mdb_catalog.wt
-rw------- 1 mongodb mongodb     0 Jun  1 22:52 mongod.lock
-rw------- 1 mongodb mongodb 16384 Jun  1 22:52 sizeStorer.wt
-rw------- 1 mongodb mongodb   114 Jun  1 22:50 storage.bson
-rw------- 1 mongodb mongodb    46 Jun  1 22:50 WiredTiger
-rw------- 1 mongodb mongodb  4096 Jun  1 22:52 WiredTigerLAS.wt
-rw------- 1 mongodb mongodb    21 Jun  1 22:50 WiredTiger.lock
-rw------- 1 mongodb mongodb  1080 Jun  1 22:52 WiredTiger.turtle
-rw------- 1 mongodb mongodb 57344 Jun  1 22:52 WiredTiger.wt
```

リスト 4.2: MariaDB のファイル配置

```
/var/lib/mysql$ ls -l
total 122936
-rw-rw---- 1 mysql mysql    16384 Aug 23 18:06 aria_log.00000001
-rw-rw---- 1 mysql mysql       52 Aug 23 18:06 aria_log_control
-rw-r--r-- 1 root  root         0 Aug 23 18:06 debian-10.3.flag
-rw-rw---- 1 mysql mysql      972 Aug 23 18:06 ib_buffer_pool
-rw-rw---- 1 mysql mysql 12582912 Aug 23 18:06 ibdata1
-rw-rw---- 1 mysql mysql 50331648 Aug 23 18:06 ib_logfile0
-rw-rw---- 1 mysql mysql 50331648 Aug 23 18:06 ib_logfile1
-rw-rw---- 1 mysql mysql 12582912 Aug 23 18:06 ibtmp1
-rw-rw---- 1 mysql mysql        0 Aug 23 18:06 multi-master.info
drwx------ 2 mysql mysql     4096 Aug 23 18:06 mysql
-rw-rw---- 1 root  root        16 Aug 23 18:06 mysql_upgrade_info
drwx------ 2 mysql mysql       20 Aug 23 18:06 performance_schema
-rw-rw---- 1 mysql mysql    24576 Aug 23 18:06 tc.log
```

4.4　設計

本研究では、4.3 節の分析結果を踏まえて、プロセス固有のディレクトリを NVM 上に構築・アクセスする手法を提案する。この固有ディレクトリは OS や他プロセスから参照できない代わりに、アクセス毎にカーネルや他プロセスとの排他処理を不要とすることで、アクセスのオーバヘッド削減を図る。

本手法は、性能オーバヘッドの削減とアプリケーションの改変が不要であることを両立するため、ライブラリファイルシステム方式を採用する。図 4.1 にソフトウェア構成を示す。(a) は、NVM 上にファイルシステムを構築した通常の構成で、仮想ブロックデバイス方式に相当する。アプリケーションが標準ライブラリに実装された POSIX インターフェースの関数を呼び出すと、標準ライブラリがシステムコールで OS 内のファイルシステムを呼び出す。最終的にカーネル内のファイルシステムが NVM 上のデータを load/store することでアクセスが完了する。(b) は提案方式の構成である。(b) ではアプリケーションが標準ライブラリの関数を呼び出すと、標準ライブラリに含まれるライブラリ呼び出し層がファイルシステムライブラリを呼び出す。このファイルシステムライブラリはアクセス対象のファイルが固有ディレクトリ内か判定し、固有ディレクトリ内であれは NVM を直接 load/store することでアクセスを行う。この際にカーネルモードへのコンテキストスイッチが不要であるため、本方式はオーバヘッドを削減できる。固有ディレクトリ外であれば標準ライブラリが処理を継続し、通常の構成同様にシステムコールを呼び出す。

本ライブラリでは、性能に影響するパラメータをアプリケーション毎に調整できる。通常、ファイルシステムでは構築時にブロックサイズなど一部パラメータをチューニング可能であるが、その範囲は限定的であ

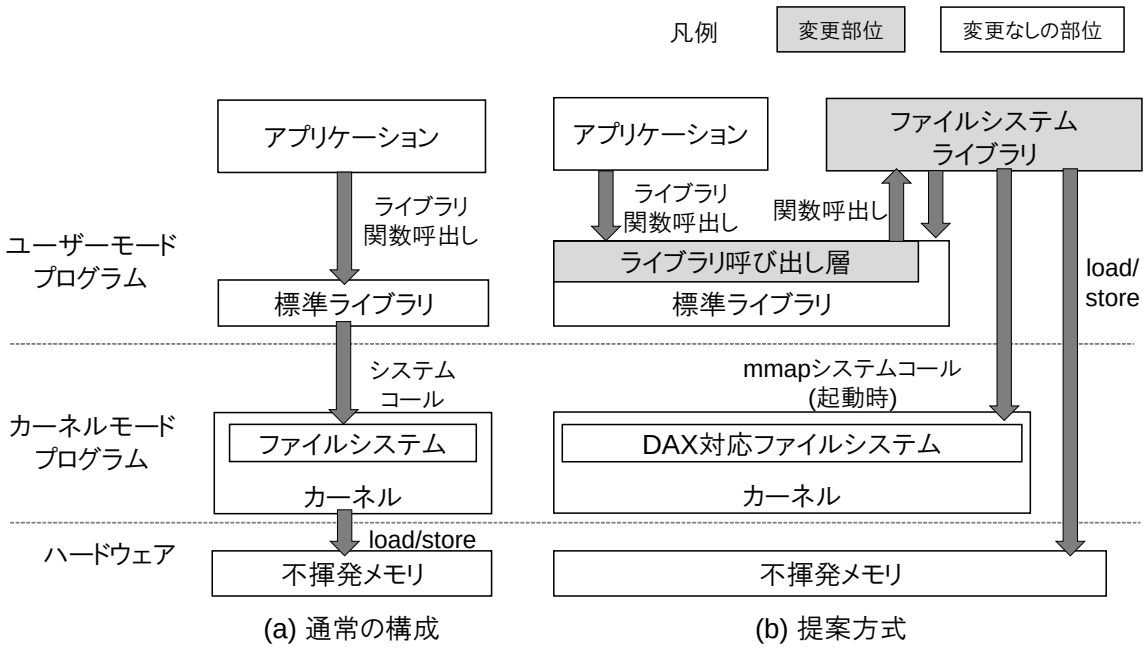


図 4.1: ソフトウェア構成

る。例えばファイル名長やファイルのブロックサイズは、POSIX や OS で取れる値が規定される。その規定要件を満たすために、可変長の文字列やブロックを保持するために処理オーバヘッドが生じる。本ライブラリでは、アプリケーションの配置するファイルサイズやファイル名長はアプリケーションの動作によって事前に想定できるため、最大ファイル名長・ブロックサイズ・最大ブロック数・最大ファイル数を固有ディレクトリの初期化時に個別に調整可能とする。これによりアプリケーションが要求しない仕様を満たすためのオーバヘッドを削減し、さらなる性能向上を図る。

4.4.1 アプリケーションへの適用方法

本方式を組み込んで動作させるアプリケーションでは、改造してライブラリ呼び出し層を組み込んだ標準ライブラリを用いて起動する。その際、プロセス固有ディレクトリとなるパス名を環境変数にて指定する。アプリケーションからは、この固有ディレクトリ以下がファイルシステムライブラリによって提供されるファイルシステムとして認識される。アプリケーションは固有ディレクトリ配下に対し、これまで通り POSIX インターフェースの関数を呼び出すことで、NVM 上に構築されたファイルを読み書きできる。固有ディレクトリ下のデータは同一プロセス内のスレッド間でのみ共有され、他プロセスおよびカーネルからは認識されない。

4.4.2 ライブラリ呼び出し層

標準ライブラリ中のライブラリ呼び出し層では、POSIX インターフェースのうちファイルアクセスに関する関数呼び出しを捉えて、ファイルシステムライブラリを呼び出す。ただし、ファイルシステムライブラリ自身が標準ライブラリを呼び出すケースにおいては、再帰的な無限ループの発生を防ぐため、フラグ変数によりライブラリ呼び出しの有無を切り替えられるようにしている。

4.4.3　ファイルシステムライブラリ

ファイルシステムライブラリは、ライブラリ呼び出し層からの関数呼び出しに対応して、ファイルアクセスの処理を行う。ファイルシステムライブラリは起動時に DAX 対応ファイルシステム上に単一の大サイズファイルを作成し、そのファイルを自身のアドレス空間上に `mmap` したうえで、inode やディレクトリエントリなどファイルシステムを構成するデータ構造を作成・配置・管理する。以後、この `mmap` した NVM の領域をプールと呼ぶ。ファイルシステムを構成するデータは、NVM 上と DRAM 上に分けて配置する。この配置については 4.4.4 項にて述べる。

引数の検証による操作対象ファイル判定

ファイルシステムライブラリは、標準ライブラリのライブラリ呼び出し層から呼び出されたとき、操作対象のファイルが固有ディレクトリ内のファイルであるか、引数を検証する。操作対象のファイルが固有ディレクトリ内である場合、その処理はファイルシステムライブラリで継続される。そうでない場合、標準ライブラリ及びカーネルの処理対象であるため、ファイルシステムライブラリは標準ライブラリに処理を引き継ぐ。

この検証は以下の通り行う。

ファイルパス　処理対象のファイルがパス名で指定される関数に関しては、パス名を絶対パスに変換したうえで、その接頭辞が固有ディレクトリのパス名と一致するかどうかで判定する。

ファイルデスクリプタ　ファイルデスクリプタは通常 OS が割り当てる情報であるため、ファイルシステムライブラリと OS で重複した値を取らないようにしなければならない。そこで、ファイルシステムライブラリは起動時にファイルデスクリプタを一定数予約しておく。そして固有ディレクトリ内のファイル操作に対しては、このデスクリプタを返すようにする。デスクリプタを予約しておくことで、固有ディレクトリ外のファイル操作に対し、OS がこれらのデスクリプタを返すことがなくなる。以後ライブラリ内では、デスクリプタが予約した値の範囲かどうか確認することで、固有ディレクトリ内のファイルを指すか判定可能となる。

4.4.4　データ配置

NVM は DRAM と性能特性が異なる。例えば Intel DCPMM は DRAM と比較してアクセス応答時間・帯域いずれも劣る [119]。そのため、ファイルシステムのアクセス性能を高めるためには、ファイルシステムの管理データのうち、不揮発性を要するデータのみ NVM 上に配置し、参照・更新頻度が高く不揮発性を要しないデータは DRAM 上に配置することが望ましい。

NVM 上に配置するデータ

NVM 上に配置すべき不揮発性を要するデータは、他の情報から再生成できないデータであり、通常ファイルシステムでディスクに格納される下記のデータが相当する。

- ファイルシステム全体の設定情報
- ファイルメタデータ (inode 及びブロックマップ)
- ディレクトリエントリ
- ファイルのデータブロック

- inode 番号の割り当て状況
- 空き領域の管理情報

本ライブラリでは、ファイル数等の統計情報は NVM には配置しない。一般的なファイルシステムでは、統計情報もディスク上のスーパーブロックに格納している。ディスクを前提とし、かつ大量のデータを抱えるファイルシステムでは、ファイル数などの統計情報をディレクトリツリーから集計すると非常に時間がかかるため、これらの統計情報は更新の度に更新差分のみ計算してディスクに書き込んでいるが、頻繁にファイルの増減や更新が行われる条件では、この処理自体の負荷が高い。本ファイルシステムは NVM 上に構築されおり、その容量の上限からディレクトリツリーの探索は現実的な時間内で完了すると考える。そのため、本方式では統計情報等は DRAM 上のみに配置することで、起動時間と引き換えに通常アクセス時の NVM への書き込みの削減を図る。

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

DRAM 上に配置するデータ

DRAM 上には、プロセス実行中のみ必要なデータや、ファイルアクセス時の処理高速化に有用なデータを保持する。前者は、ファイルデスクリプタが該当する。後者の高速化に要するデータは、下記が該当する。

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

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

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

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

4.5 実装

4.5.1 標準ライブラリ及びライブラリ呼び出し層

本実装では、Ubuntu Linux 20.04 LTS 付属の `glibc` のソースコードを改変し、4.5.2 節で述べた関数毎にファイルシステムライブラリ呼び出しのコードを付与した。

POSIX インターフェースにおけるファイルアクセスには、ファイルデスクリプタと I/O ストリームの二つのインターフェースが提供されている。glibc では、前者を低レベル I/O とよび、後者の I/O ストリーム機能は内部で低レベル I/O を呼び出す実装となっている。ライブラリ呼び出しをフックし、機能を加える手法として LD_PRELOAD を用いる手法 [120] があるが、I/O ストリーム機能から低レベル I/O の呼び出しはライブラリ内で静的にリンクされており、LD_PRELOAD で捉えることはできない。そのため、本実装では低レベル I/O の処理部を改変することで、同時に I/O ストリームへの対応も実現した。

4.5.2　ファイルシステムライブラリ

ファイルシステムライブラリは、C++ で実装した。NVM のプールに対するトランザクション処理及びメモリ管理機構の実現には、Persistent Memory Development Kit (PMDK)[111] が提供する NVM 向けのオブジェクト管理ライブラリ libpmemobj++ を用いた。

本ファイルシステムは空のプールまたは以前他プロセスが使用したプールを利用することができる。以前に使用されたプールを用いる場合、起動時にプール内のデータを辿り、4.4.4 節に示す DRAM 上に配置するデータを再構築する。もしディレクトリツリーで参照されない inode が残っていた場合、それは unlink 処理後遅延処理が行われていない inode なので、起動時に削除する。一方で、プロセスの終了時は、遅延されたファイル削除処理を除き何も行わない。ファイルの更新は同期的に行われるため、データの一貫性に問題は生じない。

PMDK では NVM 上に構築した DAX 対応ファイルシステム上に、プールファイルを確保し、その内部をアプリケーションで任意に扱うという動作をしている。このプール確保に際し、PMDK と XFS を組み合わせて使う場合、領域確保のみ行うもののデータのゼロクリアが遅延処理される。その時、mmap されたページに初めて書き込みが行われると、ページフォルトが生じてゼロクリアが行われるため、性能が不安定となる。そこで、本実装においては、PMDK のプールファイルの新規作成時に、ファイルの全領域にゼロデータを書き込むようにした。これにより遅延ゼロクリア処理を防ぎ、性能が不安定になることを防ぐ。

ファイルシステムのパラメータとして、inode 数、ファイルブロックサイズ、ファイルあたりのブロック数、最大ファイル名長はファイルシステム構築時に環境変数で与えられるようにしている。これにより、アプリケーションのデータの特性に応じた性能改善が可能となっている。

サポートする機能

本ライブラリでは、POSIX インターフェースのうち、下記に示す基本的なファイルアクセス関数のみ対応する。

ファイルデータアクセス　ファイルの open, close や、データ読み書きを行う関数が相当する。
ファイル・ファイルシステム情報　stat, statvfs 系列の情報取得に関する関数が相当する。
ディレクトリツリー操作　ディレクトリエントリの参照や、rename, unlink に関する関数が相当する。
ファイルロック　fcntl・flock が相当する。
シンボリックリンク　symlink や readlink が相当する。

本設計及び実装では、データ入出力及びメタデータの操作は同期的に行う。これは、ディスク入出力と異なり NVM の load/store は同期的に行われるため、非同期処理を行う際に生じるタスクスイッチのオーバヘッドが生じることを防ぐためである。
POSIX インターフェースでは非同期 I/O も備えているが、本ライブラリでは非同期 I/O には対応しない。主要なアプリケーションでは、非同期 IO 機能の利用はオプションで切り替えられることが多く、実用上は

表 4.2: ハードウェア構成

項目	設定
CPU	Intel Xeon Gold 6240 (2.6 GHz, 18 core) × 2 (1CPU のみ使用)
DRAM	384 GB (DDR4 32 GB モジュール× 12)
不揮発性メモリ	1.5 TB (Intel DCPMM 128 GB モジュール× 12) うち半分の 6 モジュール、768 GB のみ使用
OS Disk	1.6 TB SAS SSD × 2 (RAID1 構成)

問題とならない。
例外的に、`unlink` はディレクトリツリーからファイルを取り除くことは同期的に行うが、ファイルを構成する inode を NVM 上から削除する処理は遅延して行う。これは Linux における、`open` 中ファイルデスクリプタが存在する限り inode を削除しないという仕様に合わせたためである。

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

拡張属性 `getxattr`, `setxattr` 等が相当する。
非通常ファイル `mknod` 関数が作成する非通常ファイルやハードリンクはサポートしない。本ファイルシステムはファイルデータへのアクセス高速化を図るものであるため、非通常ファイル作成は目的と合致しない。
memory mapped file メモリマッピングに対応するアプリケーションは、直接 DAX 対応ファイルシステムを用いればよいため、本ライブラリでは対応しない。
セキュリティコンテキスト `getfilecon` 等が相当する。本ファイルシステムはプロセス内で閉じたディレクトリを提供するため、ファイルシステムレベルで強化されたセキュリティ機能を提供する必然性がない。
ファイルアクセス監視機能 `inotify` 等のファイルアクセス監視機能は対応しない。本ファイルシステムではプロセス間でデータを共有できないため、他プロセスからの更新監視は有効でないためである。

4.6 評価

本節では、従来手法と比較して提案手法の性能評価を行う。

4.6.1 実行環境

実験に用いたサーバのハードウェア構成を表 4.2 に示す。本サーバは CPU を 2 ソケット搭載しているが、測定において NUMA の影響を排除するため、ソフトウェア的に 1CPU とそこにつながる DRAM 及び NVM のみ利用している。

続けてソフトウェアの構成と、ファイルシステムライブラリのパラメータを表 4.3 に示す。XFS の測定においてはページキャッシュを用いる通常の構成に加え、DAX を有効化した構成を測定した。DAX を有効化した状態では、ファイルデータのアクセスにページキャッシュが利用されないため、通常の状態と比較

表 4.3: ソフトウェア構成

分類	項目	バージョン・設定
基本ソフト	OS	Ubuntu Server 20.04 LTS
	Kernel	Linux Kernel 5.1.0
	ファイルシステム	XFS、XFS(DAX)、NOVA、提案手法
ライブラリ	glibc	2.31
	PMDK	1.8
	libpmemobj-cpp	1.10
ベンチマークソフト	tar	1.30
	filebench [30]	1.5-alpha3
	LevelDB [121]	1.22
ファイルシステムライブラリのパラメータ	ファイル数	1,000,000 個
	ファイルブロックサイズ	128 KB
	ファイル当たりブロック数	256 個
	ファイル名長	64 Byte
	ディレクトリあたりエントリ数	64

することでページキャッシュの影響を確認することができる。また、提案手法は DAX を有効化した XFS 上でプールファイルを構築している。

DAX を有効化しない通常の XFS の測定では、ページキャッシュの効果を制限するため、また、DRAM は 1 ソケットに接続される 192 GB のうち、8 GB だけメインメモリとして使用した。これは、メインメモリの搭載量が実験のデータセットより大きい場合、データアクセスがページキャッシュにヒットしてしまい NVM に対するデータアクセス性能を正確に測定できなくなることを防ぐためである。ただしこれでもページキャッシュの効果は完全には排除できないため、ページキャッシュにより特徴的な結果を示す測定項目においては、適宜補足する。

4.6.2　ファイル生成のマイクロベンチマーク

本節では、ファイルの書き込み性能を確認するためのマイクロベンチマークとして、多数のファイルを含む tar 形式のアーカイブファイルを不揮発性メモリ上に展開する実験を行う。

用いたアーカイブファイルを表 4.4 に示す。Linux Kernel ソースファイル [122] は、小サイズのファイルを多数含むため、アーカイブファイルの展開にはメタデータを高速に保存することが要求される。音声認識コーパス [123] は、機械学習用の音声データセットである。こちらは音声データを多数含むために平均ファイルサイズが Linux Kernel ソースファイルに比べて 14 倍大きい。そのため、このアーカイブファイルの格納は、ファイルデータの追記を高速に行えることが要求される。

本実験における測定では、アーカイブファイルのディスク読み込み処理によるストレージボトルネックが発生しないよう、アーカイブファイルのデータが DRAM 上のページキャッシュに格納された状態で実験を行う。それぞれ測定対象のファイルシステムに対し、tar コマンドでアーカイブファイルに格納されたファイルを展開する。また、tar コマンドの結果が確実にメディアに反映されるよう完了後に sync コマンドを実行し、その時間を測定する。この手続きは、各測定において 5 回以上行い、平均値をとった。なお、提案手

表 4.4: 対象アーカイブファイル

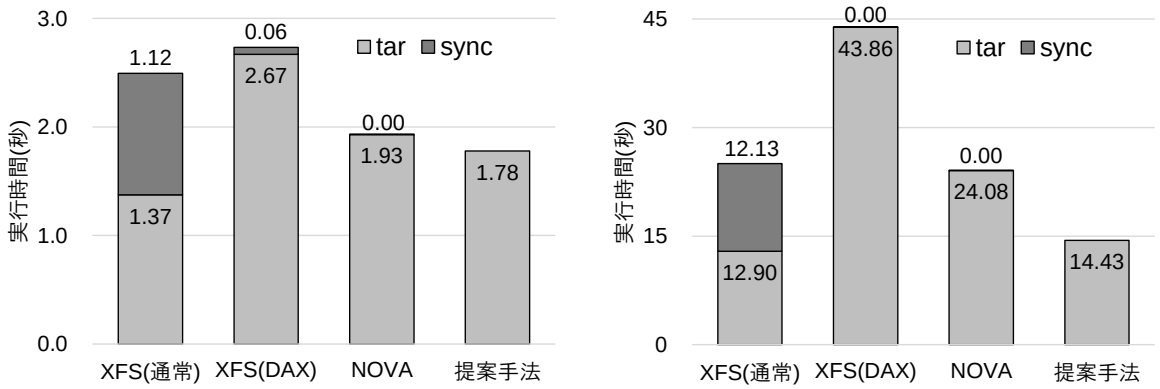
内容	ファイル数	ディレクトリ数	総ファイルサイズ	平均ファイルサイズ
Linux Kernel 5.6.0 ソース	67,302	4,438	1.029 MiB	15.1 KiB
音声認識コーパス	106,116	3,020	23.5 GiB	221 KiB

法ではこの sync コマンド実行は省略している。DAX 有効時の XFS に対する sync コマンドは、データの更新有無を問わず `mmap` を実施したプールファイル全体のデータの flush を試みるため、時間がかかるためである。提案手法はファイル作成時に NVM まで同期的にデータを書き込んでいるので、実用時は sync コマンドの実行は必要ない。

測定結果を図 4.2 に示す。XFS(DAX) と NOVA と提案手法は、ページキャッシュを用いないため、sync の実行時間は tar の実行時間より短く無視できる程度である。いずれのアーカイブファイルにおいても、提案手法は XFS・NOVA よりも短い時間でファイルの格納を完了した。Linux Kernel 5.6.0 ソースファイル (図 4.2(a)) において、提案手法は 1.78 秒と、XFS(DAX) の 66%、NOVA の 92% の時間で完了している。ただしこの比率はデータセットで異なり、音声認識コーパス (図 4.2(b)) においては、提案手法は 14.43 秒と XFS(DAX) の 33%、NOVA の 60% の時間で完了しており、提案手法の効果が大きく出ている。

いずれの方式でもデータの読み書きは CPU から不揮発性メモリの load/store 命令で行うことから方式間の差が付きにくい。そのためこの性能差はメタデータ更新の効率によるものと考えられる。XFS はディスクのセクタ単位のアクセスを前提としたメタデータの構造を取るため、ファイルの作成時により多くの不揮発性メモリへのアクセスを要し時間がかかったと考えられる。NOVA と提案手法の性能差も、同様にメタデータ更新の効率の差によるものと考えられる。これらのファイルアクセスに伴う効率については、後述の filebench の測定においてより詳細な分析を行う。

XFS(通常) は sync コマンドを含めた実行時間は NOVA や提案手法より長いが、tar コマンドの実行時間だけみると最も短い。これは、XFS(通常) がファイルデータの書き込みを DRAM 上にページキャッシュに行った段階で完了することから、DRAM と不揮発性メモリの性能差によって生じたと考えられる。
ページキャッシュを用いるとプログラムから不揮発性メモリの読み書きにおけるデータのコピーが増加するため、一般には処理効率が低下する。しかし以下二点の理由により、XFS(通常) は XFS(DAX) よりも短い実行時間を達成し、また音声認識コーパスにおいては tar コマンドと sync コマンドの合計実行時間は NOVA に匹敵するものとなったと考えられる。一つは、ページキャッシュが更新データを一度ため込むことで、ファイルブロック割り当てのメタデータ処理が高速化されることである。XFS はページキャッシュ中の dirty データをメディアに書き出す時点で初めてファイルブロックを割り当てるため、ページキャッシュを用いず細目にデータを書き出す XFS(DAX) ではそのつどブロック割り当てのコストがかかる。この作業がまとめて行われることで効率が改善する。もう一つは、データ書き込みの並列化にある。処理中のプロセスの動作についてみると、NOVA と提案手法は、tar プロセスだけが動作するため、1 コアしか動作しない。一方 XFS(通常) の場合は、tar プロセスとは別にページキャッシュ吐き出しやメタデータの非同期更新のためのカーネルスレッドが動作し、2 コア以上が同時に動作する。そのため、XFS 自体は CPU の処理効率が良いとは言えないが、ページキャッシュを介することで処理の複数コアによる並列処理が可能となり、結果としてページキャッシュを介する処理のオーバヘッドを軽減した。この点は、提案方式も DRAM をキャッシュとして用いたり、一部処理を並列化することで応答時間の短縮の余地があると考えられる。



(a) Linux Kernel 5.6.0 ソースファイル　　(b) 音声認識コーパス

図 4.2: tar コマンドによるファイル展開時間

表 4.5: fileserver ワークロード詳細

項目	設定値
実行時間	180 秒
平均ファイルサイズ	64 KB, 256 KB, 1 MB
プロセス数・スレッド数	1 プロセス・1 スレッド (シングル構成)、3 プロセス× 15 スレッド (並列構成)
ファイル数	50000 ファイル/プロセス

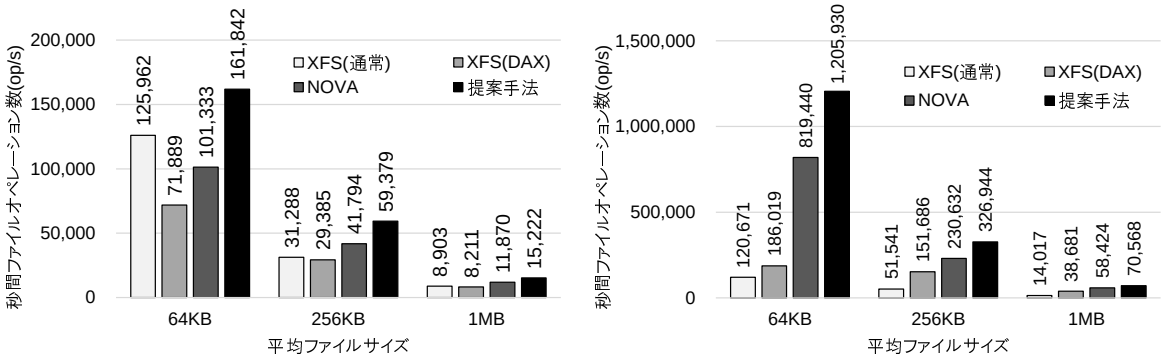
4.6.3　ファイルアクセスの統合ベンチマーク

本節では、ファイルにおけるデータ・メタデータを多数参照・更新するアプリケーションを模擬した性能評価を行うため、多様なファイルアクセスを行うベンチマークソフト filebench[30] を用いた。filebench に含まれる fileserver ワークロードでは、ファイル作成・追記・参照・削除を繰り返し、秒間ファイルオペレーション数を評価する。

測定時のパラメータは表 4.5 に示す通りである。パラメータのうち、平均ファイルサイズと、プロセス・スレッド数については複数の値で測定を行った。

図 4.3 に結果を示す。図 4.3(a) は filebench がシングルプロセス・シングルスレッドで動作するシングル構成の実行結果である。提案手法はいずれのファイルサイズでも最大のオペレーション数を達成した。ファイルサイズが小さいときほどその差は顕著で、平均ファイルサイズ 64 KB においては、ページキャッシュを用いた XFS(通常) が秒間処理数 125,962 ops に対し、提案手法は 161,842 ops と 1.28 倍の処理性能を示している。

図 4.3(b) は filebench が 3 プロセス・15 スレッドと CPU コア数以上の並列度で動作する並列構成の実行結果である。並列構成ではデータ量が増加するため、ページキャッシュの効果が薄れて平均ファイルサイズ 64 KB の XFS(通常) のみシングル構成より性能が低下しているが、それ以外はいずれの測定においても、シングル構成より性能が向上している。ただし、XFS ではその向上幅は小さい。平均ファイルサイズ 64 KB の XFS(DAX) では、シングル構成 (71,889 ops) と並列構成 (186,019 ops) で 2.59 倍にしかなっていない。同一の条件で、NOVA が 101,333 ops から 819,440 ops で 8.08 倍、提案手法が 161,842 ops から



(a) シングル構成 (b) 並列構成

図 4.3: filebench における秒間オペレーション回数

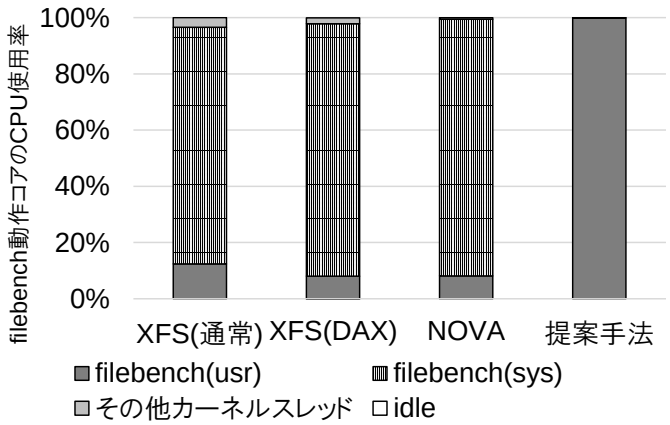


図 4.4: filebench 実行対象コアの CPU 使用率 (シングル構成・平均ファイルサイズ 64 KB)

1,205,930 で 7.45 倍になったことと比べるとわずかである。方式間の性能差を見ると、小サイズほど提案手法は効率がよく、平均ファイルサイズ 64 KB では提案手法は XFS(DAX) の 6.48 倍、NOVA の 1.47 倍という結果となった。ただし、ファイルサイズが増加するにつれ、性能向上効果は相対的に減少する。これはファイルサイズが増加するとデータの読み書きの時間が支配的となる。データの読み書きは方式の差が出にくいため、その結果性能向上効果が減少した。

オペレーション当たりの CPU・メモリ使用状況

ここでは、オペレーション当たりに要する CPU やメモリ帯域の使用状況を評価するため、シングル構成で平均ファイルサイズ 64 KB における CPU の利用率と、CPU の Performance Monitoring Counter の値を確認する。

図 4.4 は、filebench プロセスが動作しているコアの CPU 使用率を示している。この測定においては、filebench プロセスが動作するコアはいずれも使用率 100% で動作している。カーネルで実装される XFS と NOVA では、filebench プロセスがファイルアクセスの延長でカーネル空間にて動作する時間が 8 割以上を占めている。

ここで特筆すべきは XFS で、DAX の有無に依らず、filebench 以外にカーネルスレッドが動作する時間が XFS(通常) で 3.4 %、XFS(DAX) で 2.1 % を占める。これは、ページキャッシュの吐き出しや dirty なメ

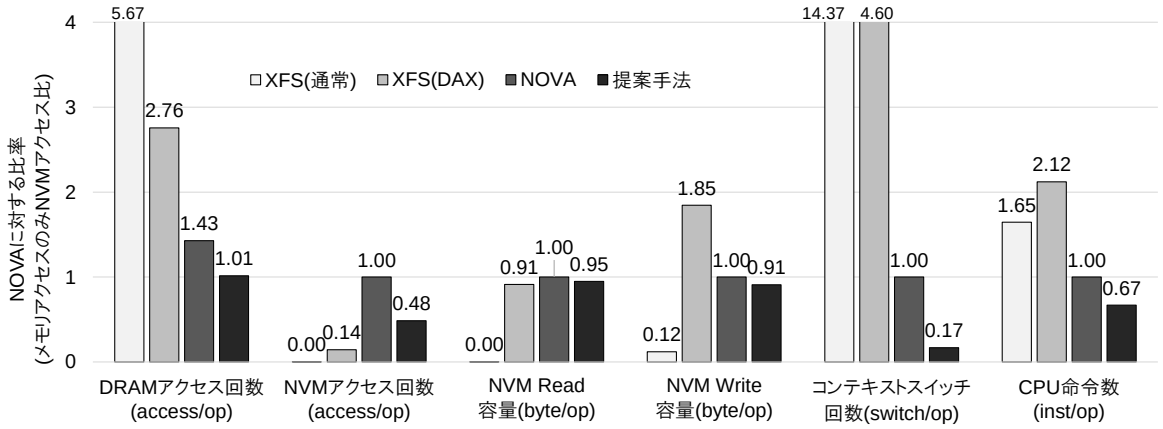


図 4.5: ファイルオペレーションあたりの CPU 動作状況

タデータの吐き出しを行うための VFS 及び XFS の各カーネルスレッドが同期的に動作しているためである。そのため、XFS はタスクスイッチが生じるため CPU の使用効率の点では NOVA や提案手法に劣る。

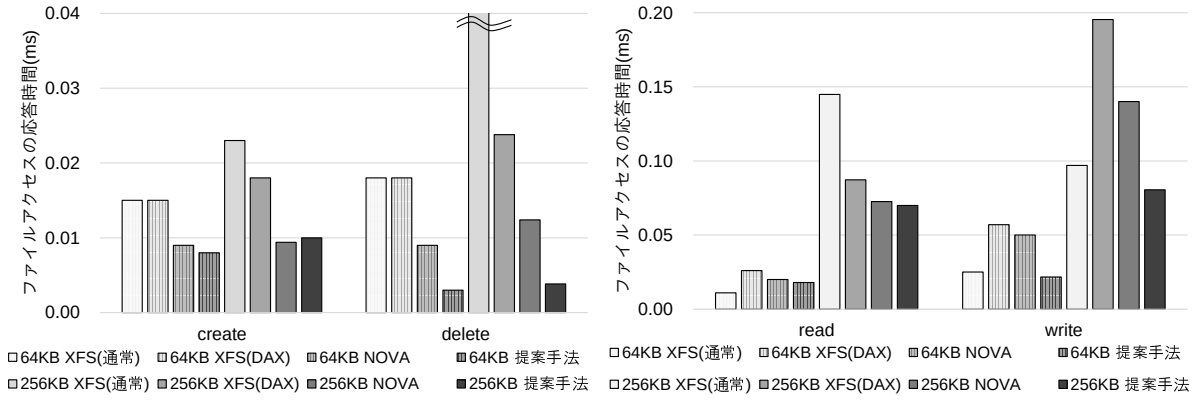
図 4.5 は、1 ファイルオペレーションあたりの CPU の各動作回数を Performance Monitoring Counter で計測し、NOVA を 1 としたときの比率を示したものである。この値が小さいほど、ファイルオペレーションに際し CPU やメモリを使用しないため、効率が高いと言える。提案手法はいずれの項目においても NOVA よりも小さな値を示しており、filebench の高い測定値の裏付けとなっている。

図中、左四つの項目はメモリのアクセスに関する。メモリアクセス回数は、NOVA の NVM アクセス回数を 1 とし、DRAM 及び NVM のアクセス回数を示す。XFS(通常) では測定中のデータがページキャッシュに収まってしまい、メタデータ書き出しのための若干の NVM Write を除けば、NVM のアクセスがほとんど生じない結果となっている。XFS(DAX) では、NVM のアクセス回数は NOVA 比 0.14 と一見小さい値を示すが、NVM Read 容量は 0.91、NVM Write 容量は 1.85 と容量ではむしろ大きい。これは、XFS がセクタ単位にデータを読み書きするため、アクセスあたりの容量が大きくなっていると考えられる。また、スーパーブロックやジャーナルのような 1 セクタよりも小さい情報の更新がセクタ単位となるため、NVM Write の容量が嵩んだと考えられる。NOVA と提案手法を比較すると、容量でみると提案手法は NVM Read で 0.95、NVM Write で 0.91 とその差は filebench の結果ほどは大きくない。しかし NVM アクセス回数では提案手法は 0.48 と NOVA の半分以下である。ファイルデータの読み書きについては両者で NVM のアクセス容量は同等となるはずなので、この差はメタデータの参照・更新における NVM アクセスの減少が主要因と考えられる。

コンテキストスイッチ回数と CPU 命令数は、CPU がどれだけ効率的に動作できるかを示す。XFS はカーネルスレッドと協調して動作するため、必然的にこの値が増える。一方 NOVA と提案手法はコンテキストスイッチが少なく、CPU を効率的に使用できていることを示す。CPU 命令数においても提案手法は他方式より少なく、CPU を効率的に使用できていることを示している。

図 4.6 にて、シングル構成において主要なオペレーションのアクセス応答時間を、平均ファイルおよびファイルシステム別に取得した結果を示す。

(a) メタデータ操作は、ファイルの作成 (create)・削除 (delete) の時間を示している。いずれも XFS は NOVA・提案手法より時間が長い。ファイル作成時間は、NOVA と提案手法は同程度となっている。ファイル削除は、データブロックを解放する処理を要するため、ファイルサイズによって時間が伸びる異なる。ただし提案手法はファイル削除を非同期処理しているため、応答時間は XFS や NOVA よりも短い。



(a) シングル構成におけるメタデータ操作　　(b) シングル構成におけるデータ操作

図 4.6: 個々のファイルアクセス応答時間

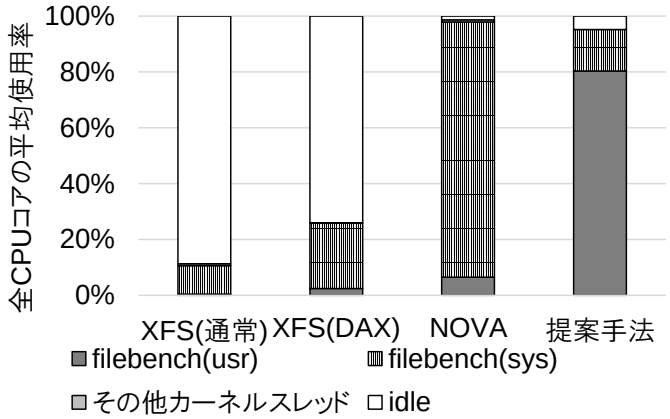


図 4.7: 全 CPU コアの平均使用率 (並列構成・平均ファイルサイズ 64 KB)

(b) データ操作は、ファイルの読み込み (read)・書き込み (write) の時間を示している。ページキャッシュへのアクセスで完結する XFS(通常) を除けば、いずれのケースも提案手法が最も応答時間が短い。特に書き込みで NOVA に対する応答時間が、平均ファイルサイズ 64 KB で 43 %、256 KB で 57 % と大きく差をつけている。これらは、データのブロックサイズが大きいことによるメタデータ更新のコストが低いためと推測される。

ここまでの解析により、提案手法はオペレーションに要する CPU 命令数やメモリアクセス回数が XFS や NOVA よりも少なく、その結果各オペレーションを高速に完了できたことで filebench で高い秒間オペレーション性能を達成できたことを確認した。

処理の並列度

ここでは、並列構成の測定結果から、並列処理の効率を比較する。

図 4.7 は、filebench 動作中の全 CPU コアの平均使用率を示したものである。idle はその CPU コアでどのスレッドも動作しておらず、すなわち CPU が待機中であることを示す。XFS は DAX 有効時で高々 26 % しか CPU を利用できておらず、filebench のスレッド数を増加しても性能が NOVA や提案手法ほど増加しない様子が確認できる。NOVA と提案手法ではいずれも CPU が 95 % 以上の時間動作しており、並列度

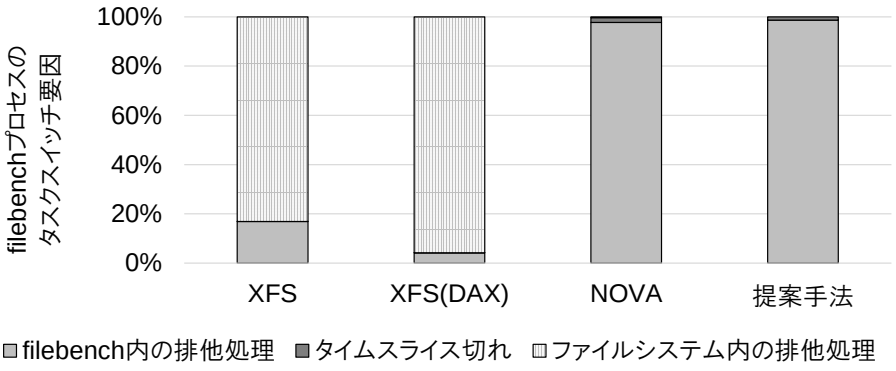


図 4.8: filebench プロセスのスリープ要因 (並列構成・平均ファイルサイズ 64 KB)

を損なわず filebench の各スレッドが並列に動作できていることを示している。なお、NOVA と提案手法にも 5 % 以下の idle 時間があるが、filebench 自体が行うスレッド間の排他処理によるものであり、ファイルアクセスによるものではない。

実際 XFS においてファイルシステムの処理によって並列動作が行えていないことを確認するため、filebench の動作中に ftrace [124] を用いて filebench の各スレッドのスリープ要因を解析した。ftrace はプロセスが指定した関数を通過する際のカーネル内のスタックトレースを出力できるので、filebench が schedule 関数 (スリープする際に呼ばれる関数) を呼ぶ際のスタックトレースを 40 万回分超取得し、関数名から要因を分析して集計した。

図 4.8 はスリープ要因の分析結果を示す。NOVA では、ファイルアクセスに由来するスリープは 0.4 %、提案手法では 0 % であり、両方式はファイルアクセスのためのスレッド間の排他制御によるスリープやタスクスイッチはほぼ生じないことが確認できた。両方式ともスピンロックによる排他制御は行っているが、これはスリープを伴う排他よりはるかに軽量であり、両方式が効率よく動作できている理由を示している。一方 XFS ではファイルシステム内のプロセス間・スレッド間の排他処理によるスリープおよびタスクスイッチが大半を占める。XFS においては、ジャーナルアクセス等ファイルシステム全体の資源を排他的にアクセスするため、特にメタデータ操作を伴う open/unlink/write システムコールの延長でスリープが生じていた。

一般にスリープを伴うタスクスイッチは、CPU キャッシュや TLB の使用効率が低下し性能オーバヘッドが大きい。そのため、従来の記憶メディアを前提にタスクスイッチを繰り返しながらファイルアクセスを行う従来のファイルシステムは、NVM の活用においては効率が悪いと言える。その点 NOVA と提案手法はメディアアクセスが高速なことを前提に軽量なスピンロックを用いることで、効率よく動作できている。また、NOVA で 0.4 % 生じたファイルシステム内のタスクスイッチスリープであるが、これらは NOVA 自体ではなくカーネルの VFS 層で生じており、すなわちカーネルのファイルシステムを用いることにより生じうるオーバヘッドである。その点提案方式は VFS 層含めたスリープが生じないため、NOVA 以上にオーバヘッドを削減できる方式といえる。

性能ボトルネックの確認

今回使用した NVM である DCPMM は前述の通りアクセス応答時間と帯域のいずれも DRAM に比べ性能が低い。そのため、現状 CPU の演算性能とバランスがとれておらず、CPU の演算性能より先に NVM の性能が上限に達していると考えられる。

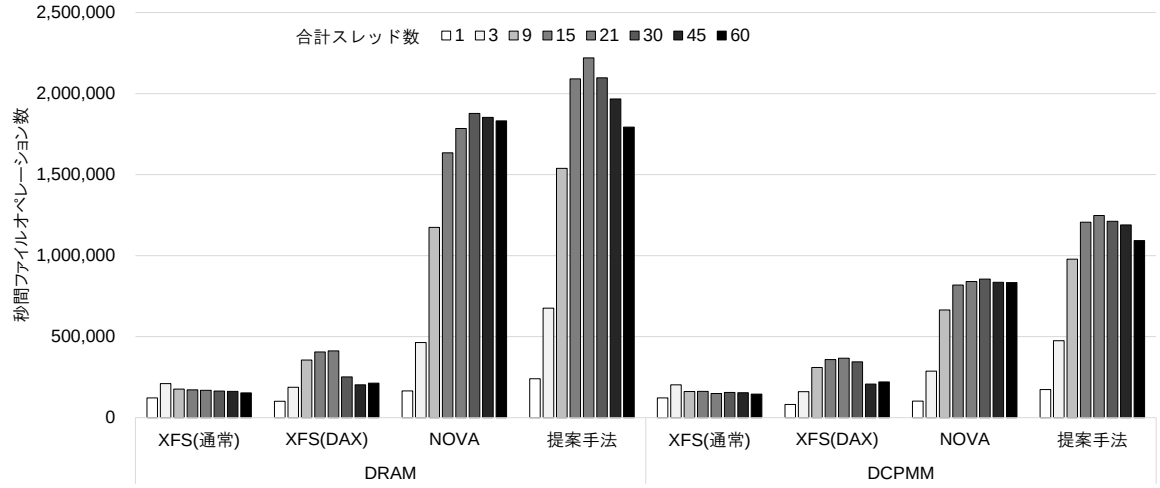


図 4.9: メモリ種別・スレッド数に対する filebench の実行結果 (並列構成・平均ファイルサイズ 64 KB)

図 4.9 は、プロセス数 3、平均ファイルサイズを 64 KB に固定し、ファイルシステムを構築するメディアと合計スレッド数を変えながら filebench を実行した結果である (合計スレッド数 1 のみ 1 プロセス)。使用したサーバの CPU は 18 物理コアを搭載し、HyperThreading により計 36 仮想 CPU で動作する。そのため CPU の演算性能からすると、18〜36 スレッドの間で性能が頭打ちになると考えられる。実際、メディアとして DRAM を使用すると、ページキャッシュ動作を伴う XFS(通常) を除くと合計スレッド数 30 が最高性能を示している。一方メディアとして DCPMM を使用すると、スレッド数 15 程度でほぼ頭打ちになる。この状態では、CPU のコア数と filebench のスレッド数を増加してもメモリアクセスの応答時間が遅延するために CPU の使用率が 100 % 近くに達し性能が上がらなくなる。

よって、現状の CPU の演算性能と NVM の性能バランスにおいては、NVM のアクセス回数を減らすことが性能優位につながると考えられる。また、オペレーションあたりの NVM のアクセス回数が少ない提案手法が XFS や NOVA に比べ優位な結果となった要因と考えられる。

4.6.4 Key-Value Store のバックエンドストレージとしての利用

本節では、ミドルウェアのバックエンドストレージとして各ファイルシステムを用いた場合の性能評価を行う。本測定では、対象のミドルウェアとして組み込み型 Key-Value Store である LevelDB [121] を用いた。LevelDB はキーと値で構築されるレコードの集合を、カレントディレクトリ内に複数のファイルを作成し、キーの昇順で格納する。LevelDB は共有ライブラリとしてアプリケーションに組み込まれるため、レコードのアクセスにおいてプロセス間通信は行わず、データ構造も単純なためストレージのアクセス性能が Key-Value Store の性能に反映されやすい。

ベンチマークには、LevelDB に標準添付されているテストプログラムを用いた。このテストプログラムでは、表 4.6 に示すワークロードをそれぞれ行い、その時間を測定する。本測定においては、ページキャッシュの影響を防ぐために DRAM を 8GB のみ使用する。そしてデータ量が DRAM よりも多くなるよう、データ総量は 10 GB で統一し、レコードサイズ 1 KB (レコード数 10,000,000)、レコードサイズ 10 KB (レコード数 1,000,000)、レコードサイズ 100 KB (レコード数 100,000) の三通りのデータで測定を行った。

各ファイルシステム・レコードサイズにおけるオペレーションあたり平均応答時間を図 4.10 に示す。書き込みに分類されるオペレーションは、レコードサイズが増加するとストレージへの書き込み量も増加する

表 4.6: LevelDB の測定内容

オペレーション分類	オペレーション種別	設定値
書き込み	fillseq	キーの昇順に新規レコード書き込み
	fillrandom	キーをランダムに新規レコード書き込み
	overwrite	キーをランダムに既存レコードを更新
読み込み	readseq	キーの昇順にレコードを読み込み
	readrandom	キーをランダムにレコードを読み込み
削除	delseq	キーの昇順にレコード削除
	delrandom	キーをランダムにレコード削除

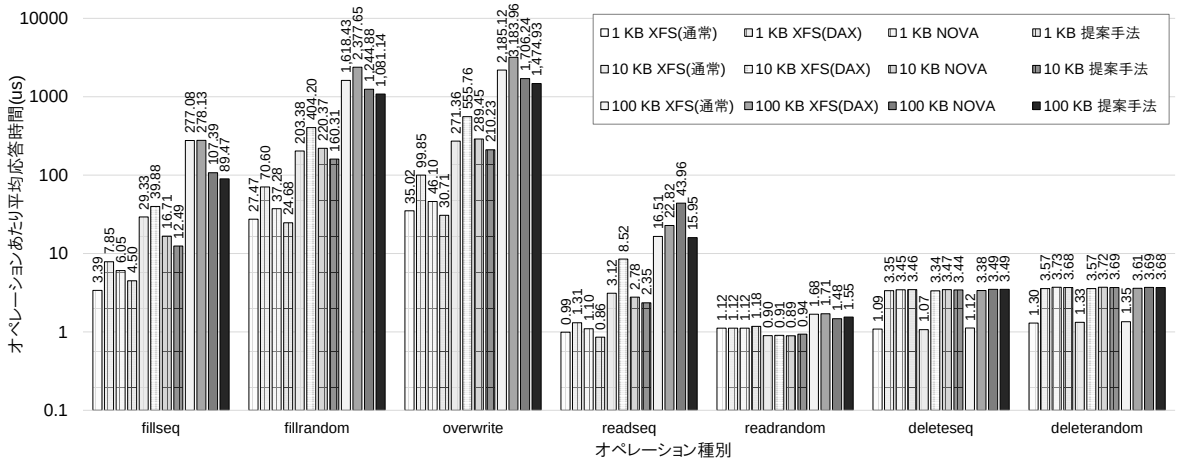


図 4.10: LevelDB 標準ベンチマークにおける実行結果

ため、応答時間も増加する。

今回実行したオペレーションは、メディアへの同期的な書き込みは行わない。そのため、ページキャッシュを用いる XFS(通常) では不揮発性メモリへのアクセスがほとんど行われない。その結果、レコードの削除のような小サイズの更新を伴うオペレーションでは他の方式より飛びぬけて短い応答時間となっている。

XFS(通常) を除くと、提案手法はいずれのオペレーションにおいても XFS(DAX) や NOVA よりも短い時間で応答している。まず書き込みに分類されるオペレーションでは、レコードサイズに応じて応答時間が長くなる。ただし、レコードサイズを問わず提案手法、NOVA、XFS(DAX) の順で応答時間が短い。同一のレコードサイズにおいても、fillseq、fillrandom、overwrite と徐々に応答時間が長くなるが、fillseq と fillrandom はレコードがファイルの連続ブロックに格納されるのに対し overwrite では不連続なブロックが更新されることと、fillseq では検索インデックスがキー順に作成されるが fillrandom と overwrite では検索インデックスがランダムに更新されることで、ファイルアクセスの回数が変化したことがこの結果につながっている。そのため、前述の filebench の結果の通り、ファイル write の性能に対応した応答時間順となった。

次に読み込みに分類されるオペレーションについて結果を見ると、readseq では書き込み同様にレコードサイズに応じて応答時間が延び、かつ提案手法がいずれも最短の応答時間であるという結果が出た。readrandom はファイルシステム毎の差が出にくく、レコードサイズ毎の差も小さい。また、レコードサイ

ズが中間である 10 KB の測定値が最短であるという結果が出ている。これは LevelDB が保持するキャッシュの仕組みによる。LevelDB はインデックス情報のキャッシュと、レコードのキャッシュをそれぞれ保持する。前者はレコードサイズに依らず、かつ上限が小さいため、レコード数の多い 1 KB レコードの測定ではキャッシュヒットしにくく、10 KB の方が高い性能を得た要因となっている。一方レコードのキャッシュは DRAM 容量で決まるため、いずれのレコードサイズの測定でも、レコードの総データ量を統一している以上キャッシュヒット率は同程度である。そのため、キャッシュミス時のディスク読み込み量が増える大サイズレコードの方が平均応答時間が伸びる傾向になる。LevelDB ではキーの削除はレコードに削除済みフラグを付与して完了するため、応答時間はレコードサイズの影響を受けない。レコードサイズ 100KB において、NOVA の readseq の応答時間がレコードサイズ以上に増加して XFS(DAX) より長い理由は不明である。NOVA は Log Structured なデータ更新を行う構造を取るため、overwrite のオペレーションによりファイルが不揮発性メモリ上で断片化している可能性が考えられるが、実行中のファイルシステムの内部構造の解析を要するため未実施である。

削除に分類されるオペレーションでは、レコードサイズやアクセスパターンによる大きな差異は見られなかった。これは、LevelDB におけるレコードの削除は、レコードに対し削除フラグを立てるという動作で完了するため、ファイルの更新データ量がレコードサイズに依存しないためである。

全オペレーションを通してみると、総じて提案手法は他方式に比べ短い時間で応答できており、その効果の高さを実証した。特に、レコードの追加・更新や readseq のようなレコードのスキャンのように応答時間が長いオペレーションを短い時間で応答できることから、アプリケーションの性能改善に有効に働くことが期待できる。

一方、現状の DRAM と不揮発性メモリの性能差により、一見データのコピーを増加させるページキャッシュが却って性能向上につながるケースも見られる。このほかにも、DAX 対応ファイルシステムにおいてページキャッシュが有利に働くケースがあることは、他の研究でも言及されている [125, 126]。アプリケーションのアクセスパターンやデータ構造を踏まえ、ページキャッシュを選択的に使用していくことも、さらなる性能改善には有効と考える。

4.7 制限事項

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

4.7.1 マルチプロセス対応

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

一つ目の理由は性能トレードオフである。本方式では、ファイルシステムを構成する一部の管理データを DRAM に配置している。これらの管理データをプロセス間で共有するには、アドレス空間を共有し、かつ適宜排他処理を挿入する必要がある。一方で、プロセス毎に管理データを持つ場合、一プロセスが加えた操作に対し、全プロセスが同期して管理データを更新する必要があり性能オーバヘッドが大きい。いずれにしても、性能オーバヘッドが増大する。

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

前述の通り、多くのアプリケーションではプロセス内でディレクトリ内のデータを排他的に使用するが、

ソフトウェアのビルドなど複数プロセスが並列に動作し同じディレクトリのデータを参照するアプリケーションもある。これらの用途に関しては、今後対応を行い性能改善効果の評価を続ける。

4.7.2　標準ライブラリ不使用アプリケーション

提案手法は標準ライブラリを改変することで、アプリケーションのファイルシステムアクセスを捉えている。そのため標準ライブラリを用いないアプリケーションには適用できない。例として、ランタイムをコンパイル時に静的リンクする Go 言語の生成バイナリが相当する。これらのアプリケーションは、実行バイナリの解析によりシステムコール呼び出し部を書き換えることで、本提案手法と同等の処理を行えると考えられる。

4.8　まとめ

本章では、不揮発性メモリの高性能を活かし、アプリケーションのファイルアクセス性能を向上させるストレージのソフトウェアインターフェースの検討を行い、またソフトウェアスタックの構成手法について述べた。不揮発性メモリの性能をアプリケーションで活かすためには、ファイルアクセスにおけるソフトウェアオーバヘッドを削減することに加え、またアプリケーションを改変せず適用できる方式でなければならない。提案手法では、多くのアプリケーションがデータ格納のために一部ディレクトリを排他的に使用することに着目し、不揮発性メモリ上にライブラリファイルシステムを構築し、プロセスで占有できる固有のディレクトリを提供する。提案方式では、この固有ディレクトリへのファイルアクセスに対しカーネルモードへのコンテキストスイッチが不要であり、プロセス間のデータ共有を行わないことで排他処理を削減し、ソフトウェアオーバヘッドを削減する。また、アプリケーションに特化したチューニングを可能とすることでさらに性能向上効果を高められる。

提案手法による性能向上効果を確認するため、本章では tar・filebench・LevelDB の 3 つのアプリケーションにおいて従来手法とファイルアクセスの性能を比較した。その結果、いずれのアプリケーションにおいては提案手法は従来手法より高い性能を示した。特に filebench においては、小サイズファイル・マルチプロセス・マルチスレッド構成において、XFS 比で 6.48 倍、既存研究の NOVA 比で 1.47 倍と、従来手法より大幅な性能向上を達成した。

実装においては従来の POSIX インターフェースを保持し、標準ライブラリ以降の処理に不揮発性メモリ対応を組み込む構成をとった。これにより、標準ライブラリを用いるアプリケーションであれば適用可能となる。その点で、本手法は、多くのアプリケーションに透過的に利用できる方式といえる。

本研究における今後の課題として、現方式の制限事項である、マルチプロセスや標準ライブラリの動的リンクを行わないアプリケーションへの対応がある。これらへの対応を行うことで、提案方式の適用範囲がさらに拡大することが期待できる。性能面では、ページキャッシュを含めた DRAM と不揮発性メモリ間のさらなるのデータ構造の配置の改善・検討により、向上できることが期待できる。

第 5 章

結論

5.1 本論文の成果

本論文では、高性能な記憶メディアを活かした実用的でアクセス性能の高いストレージの実現のため、アプリケーションに透過的に適用可能であり、データを最適配置できる階層ストレージについて論じた。

第 1 章では、本研究の背景として、記憶メディアの選択肢が増加したことと、データやアプリケーションが長期間用いられるようになったことを示した。ここから、本研究の動機となるアプリケーションに透過的にデータを移動する階層ストレージ技術の必要性を述べた。また、本研究の目的と課題について述べ、以降の各論で述べる研究の位置づけを明らかにした。

第 2 章では、新旧ファイルサーバを対象とし、機器移行のためのデータ複製手法を提案した。ファイルサーバの移行においては、格納データを新旧サーバ間で複製する必要がある。このデータ複製は長期間を要することから、ファイルサーバ上のデータを参照するアプリケーションに対し、複製中にファイル共有サービスを停止させることは現実的ではない。本章では、新ファイルサーバにおいて Post-copy 方式のファイル単位複製を備えることで、既存の機器である旧ファイルサーバやクライアントの改変を行わない透過的なサーバの移行を可能とした。また、複製中におけるクライアントアクセスの応答時間を、複製の進捗に伴い早期に短縮するため、アプリケーションによるファイルアクセスの手順に基づき、データの複製順の検討を行った。本移行方式では、ファイル共有サービスの停止時間や、アクセスの応答時間を数十秒以下と、実用上十分な短時間に抑えること確認した。

第 3 章では、クラウドコンピューティング環境を想定し、仮想マシン上で動作中のアプリケーションに対し、アプリケーション動作を改変することなく適用できる SSD・HDD の階層ストレージの構築方法を提案した。提案手法では、仮想マシン環境の階層ストレージにおいてキャッシュヒット率を下げる要因であるストレージソフトウェアの階層に着目した。そこで、提案手法では、仮想マシン内で得られるファイルのレイアウト情報をホスト OS に伝えることで、仮想マシン内における論理的に連続なファイルブロックの配置ををホスト OS にて認識できるようにした。この情報に基づき、SSD・HDD 間でデータのプリフェッチを行うことで、アクセスの局所性に基づき HDD に対する低速なアクセスを削減してデータへのアクセスの平均応答時間を短縮する。また、アプリケーションの実行状況によってプリフェッチの実行時間を動的に調整することで、HDD に対する過剰なプリフェッチによる性能低下を抑えることができた。これらの手法により、既存のアプリケーションに対し変更を行うことなく、キャッシュヒット率の改善ならびにアプリケーションの性能向上を実現した。

第 4 章では、近年実用化された不揮発性メモリの性能を活かし、高速にデータを保存するためのストレージインターフェースとソフトウェアスタックを提案した。本研究では、POSIX インターフェースにおいて、プロセス間でグローバルに一貫性を持たせた名前空間の保持を行うために、コンテキストスイッチやプロセ

ス間の排他処理等多くのオーバヘッドが生じている点に着目した。実用アプリケーションの多くは、このようなプロセス間でのデータや名前空間の共有を要しないストレージの使用法を取っている。そこで提案手法では、ユーザー空間においてプロセス内に閉じた固有ディレクトリ構築し、ユーザー空間ファイルシステムライブラリを介して提供することでオーバヘッドを削減する。本ライブラリは POSIX インターフェースを介して利用できるため、既存アプリケーションに対し変更を要することなく不揮発性メモリの高い性能を享受できる。その結果、本手法は複数のアプリケーションにおいて不揮発性メモリ専用に構築されたファイルシステムよりも高い性能を達成した。

本研究では、すでに広く使用されているストレージのインターフェースであるファイルと POSIX インターフェースに着目した階層ストレージを提案した。各章で述べたいずれの提案手法も、広く使われる既存のインターフェースを保持することで、既存のアプリケーションを変更することなく透過的に利用な階層ストレージを実現した。また、各提案手法において、アプリケーションのアクセスパターンに応じてデータを配置する記憶メディアやデータの移動順を定めることで、アプリケーション実行中にデータが適切なメディアに格納できるようになった。

このように、本研究ではアプリケーションの改造不要で透過的に適用可能であることと、アプリケーション実行中にデータを適切なメディアに配置するという二つの課題を解決した階層ストレージを実現した。本研究の成果によって、階層ストレージを介して様々なアプリケーションが容易に高性能な記憶メディアの恩恵を受け、性能向上効果を享受できると確信している。

5.2　今後の課題

不揮発性の記憶メディアは、現在も種々研究開発が行われている。提案手法は HDD・SSD・不揮発性メモリと本研究の実施時点で主要な記憶メディアを想定しているが、今後も新たな特性を持つメディアが市場に現れることは想像に難くない。その時、性能特性やインターフェースの差異によっては、本論文の提案手法がそのまま適用できない可能性がある。今後提案手法の適用範囲を拡大するためには、これら新たな性能特性モデルやインターフェースを備えた記憶メディアに対しても適用考慮し、手法を汎用化することが必要である。

謝辞

本研究の推進にあたり、手厚くご指導頂きました指導教員の品川高廣准教授には深く感謝いたします。企業に勤めながらの研究活動のため、思うように成果が上がらない時期もありましたが、その間研究の方向性や論文執筆のための議論に粘り強くお付き合い頂きました。お蔭様で、在学中に国際学会講演や学術雑誌掲載と様々な経験を積むことができました。ご指導頂いたことは、今後の研究においても、また企業での業務においても役立つものと確信しています。

システム情報学専攻の中村宏教授には、大学院の入試説明会にてお会いした折、社会人博士課程での学生生活・研究推進の不安について相談に乗って頂き、入学を後押しして頂きました。その後もお会いするたびに言葉をかけて頂き、励みになったことを覚えています。

品川研究室の皆様にも大変お世話になりました。深井貴明氏には、仮想マシンの動作に関する知見や、研究ストーリーの組み立て等多数のご助言を頂きました。味曽野雅史氏には、関連研究調査の折に様々な質問・コメントを頂き、議論を通して理解を深めることができました。また、実験機器の手配やスケジュール調整等、私が研究やゼミ参加を恙なく行えるよう配慮して下さいました。その他研究室の皆様には、日頃接することのない研究分野に触れる機会を与えて頂き、自分の見分を深める良いきっかけとなりました。

日立製作所の皆様にも大変お世話になりました。特に岩嵜正明氏、早坂光雄氏には、入学前後を通じて研究推進・論文執筆に際し多数のご助言を頂きました。また、須藤敦之氏には、社内における通学の体制づくりにご助力頂きました。その他職場の皆様には、社会人博士の先輩としてのご助言を頂いたり、私の発表練習にお付き合い頂いたりという形で研究をご支援頂きました。何よりも、大学院通学が業務に影響を与えてしまうこともあったにも関わらず、皆様温かい目で励ましや労いの言葉をかけて下さるなど、精神的に大きな支えとなりました。

最後になりますが、品川研究室の皆様、日立製作所の皆様、並びに家族のご助力により、楽しく社会人博士課程を送ることができました。この場を借りまして、厚くお礼を申し上げます。

参考文献

[1] IDC. Data Age 2025: The digitization of the world from edge to core. https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf, 2018.

[2] Jim Handy. Where are we in the current flash cycle? In Presentation of Flash Memory Summit 2019, 2019.

[3] Bob Fontana and Gary Decad. Storage media overview: Historic perspectives. In *Proceedings of IEEE 32nd Symposium on Massive Storage Systems and Technology*, MSST 2016, 2016.

[4] Bob Fontana and Gary Decad. A ten year (2008-2017) storage landscape. LTO tape media, HDD, NAND. In *Proceedings of IEEE 34th Symposium on Massive Storage Systems and Technology*, MSST 2018, 2018.

[5] 堀内 義章. 2020 年のストレージと HDD の業界展望. http://www.idema.gr.jp/common/pdf/news/tenbo2020.pdf, 2019.

[6] Chris Mellor. Western digital aims to boost nvme ssd share with this one cunning virtualization trick. https://blocksandfiles.com/2018/11/12/wd-needs-to-boost-its-nvme-ssd-ship-share/, 2018.

[7] Jim Handy and Tom Coughlin. MRAM, XPoint, ReRAM, PM Fuel to Propel Tomorrow's Computing Advances. https://www.snia.org/sites/default/files/PM-Summit/2019/presentations/13-PMSummit19-Handy-Coughlin.pdf, 2019.

[8] Intel Inc. Intel Optane DC Persistent Memory. https://www.intel.com/content/www/us/en/products/memory-storage/optane-dc-persistent-memory.html.

[9] Technical Committee T13 AT Attachment. *Zoned Device ATA Command Set (ZAC)*, 2015. Rev. 0.5.

[10] Chris Mellor. Hard disk drive shipments fell 50% between 2012 and 2019. https://blocksandfiles.com/2020/01/14/disk-drive-shipments-50-per-cent-fallfrom-2012-to-2019/, 2019.

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

[12] Python Software Foundation. Python 標準ライブラリ posix. https://docs.python.org/ja/3/library/posix.html, 2020.

[13] まつもとゆきひろ. Ruby 2.7.0 リファレンスマニュアル file クラス. https://docs.ruby-lang.org/ja/latest/class/File.html, 2020.

[14] The PHP Group. PHP マニュアル 関数リファレンス ファイルシステム. https://www.php.net/manual/ja/book.filesystem.php, 2020.

[15] Oracle Corporation. OpenJDK. http://openjdk.java.net/, 2020.

[16] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E. Porter. A study of modern linux API usage and compatibility: What to support when you're supporting. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, 2016.

[17] Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh. POSIX abstractions in modern operating systems: The old, the new, and the missing. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, 2016.

[18] Dwayne Richard Hipp. SQLite. https://www.sqlite.org/index.html, 2020.

[19] The PostgreSQL Global Development Group. PostgreSQL. https://www.postgresql.org/, 1996.

[20] solid IT gmbh. DB-Engines Ranking. https://db-engines.com/en/ranking, 2020.

[21] AWS. Amazon Elastic File System. https://aws.amazon.com/jp/efs/, 2020.

[22] Google. Cloud Filestore. https://cloud.google.com/filestore, 2020.

[23] Larry Freeman. What's old is new again - storage tiering. https://www.snia.org/sites/default/education/tutorials/2012/spring/storman/LarryFreeman_What_Old_Is_New_Again.pdf, 2012.

[24] Bianca Schroeder and Garth A. Gibson. Understanding disk failure rates: What does an MTTF of 1,000,000 hours mean to you? *ACM Transactions on Storage*, Vol. 3, No. 3, 2007.

[25] Intel Inc. Planning guide: Updating it infrastructure. http://www.intel.com/content/dam/www/public/us/en/documents/guides/server-refresh-planning-guide.pdf, 2013.

[26] syncsoft. The 2018 state of resilience. http://rc.visionsolutions.com/WP_2018_State-of-Resilience-Report, 2017.

[27] TechTarget. Snapshot 1: New NAS buys motivated by performance and outdated hardware. *Storage Magazine*, Vol. 16, No. 2, p. 12, 2017.

[28] TechTarget. NAS trifecta: Price, features and performance. *Storage Magazine*, Vol. 16, No. 8, p. 14, 2017.

[29] Standard Performance Evaluation Corporation. SPEC SFS 2014 SP2 User's Guide. https://www.spec.org/sfs2014/docs/usersguide.pdf, 2014.

[30] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *;login: The Usenix Magazine*, Vol. 41, No. 1, pp. 6–12, 2016.

[31] Andrew Tridgell and Paul Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, ANU Research Publications, 1996.

[32] Microsoft TechNet. Command-line reference robocopy. https://technet.microsoft.com/en-us/library/cc733145(v=ws.11).aspx, 2016.

[33] EMC Corporation. EMC VNX Series VNX File System Migration Version 2.0 for NFS and CIFS, 2013.

[34] Marc Eshel, Roger Haskin, Dean Hildebrand, Manoj Naik, Frank Schmuck, and Renu Tewari. Panache: A parallel file system cache for global file access. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST '10, pp. 155–168, 2010.

[35] IBM. IBM Spectrum Scale. https://www.ibm.com/marketplace/scale-out-file-and-object-storage, 2020.

[36] IBM. Active File Management. https://www.ibm.com/support/knowledgecenter/en/STXKQY_

5.0.4/com.ibm.spectrum.scale.v5r04.doc/bl1hlp_filesafm.htm, 2020.

[37] Naren Rajasingam and Ravikumar Ramaswamy. Data migration method using AFM. https://www.ibm.com/support/knowledgecenter/STXKQY_5.0.4/com.ibm.spectrum.scale.v5r04.doc/bl1ins_uc_migrationusingafmmigrationenhancements.htm, 2019.

[38] Alain Azagury, Michael E. Factor, Julian Satran, and William Micka. Point-in-time copy: Yesterday, today and tomorrow. In *Proceedings of the 19th IEEE/10th NASA Goddard Conference on Mass Storage Systems and Technologies*, MSST 2002, pp. 259–270, 2002.

[39] Ted Anderson, Leo Luan, Craig Everhart, Manuel Pereira, Ronnie Sarkar, and Jane Xu. Global namespace for files. *IBM Systems Journal*, Vol. 43, No. 4, pp. 702–722, 2004.

[40] Yoshiko Yasuda, Shinichi Kawamoto, Atsushi Ebata, Jun Okitsu, and Tatsuo Higuchi. Concept and evaluation of X-NAS: a highly scalable NAS system. In *Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies*, MSST 2003, pp. 219–227, 2003.

[41] Wataru Katsurashima, Satoshi Yamakawa, Takashi Torii, Jun Ishikawa, Yoshihide Kikuchi, Kouji Yamaguti, Kazuaki Fujii, and Toshihiro Nakashima. NAS switch: a novel CIFS server virtualization. In *Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies*, MSST 2003, pp. 82–86, 2003.

[42] F5 Networks, Inc. ARX series datasheet. https://www.f5.com/pdf/products/arx-series-ds.pdf, 2013.

[43] Data Dynamics Inc. StorageX 8.0. https://www.datadynamicsinc.com/launch/, 2017.

[44] EMC Corporation. EMC Rainfinity file management appliance getting started guide, 2009.

[45] Datadobi. DobiMigrate. https://datadobi.com/migrate/, 2018.

[46] Microsoft. Azure FXT Edge Filer. https://azure.microsoft.com/en-us/services/fxt-edge-filer/, 2019.

[47] Spencer Shepler, Mike Eisler, and David Noveck. Network file system (NFS) version 4 minor version 1 protocol. http://www.rfc-editor.org/rfc/rfc5661.txt, 2010.

[48] Justin Parisi and Bikash Roy Choundhury. Parallel network file system configuration and best practices for Clustered Data ONTAP 8.2 and later. Technical Report 4063, NetApp, 2016.

[49] Microsoft TechNet. Usage of file server migration toolkit. https://social.technet.microsoft.com/wiki/contents/articles/32299.usage-of-file-server-migration-toolkit.aspx, 2015.

[50] Jun Nemoto, Atsushi Sutoh, and Masaaki Iwasaki. Directory-aware file system backup to object storage for fast on-demand restore. *International Journal of Smart Computing and Artificial Intelligence*, Vol. 1, No. 1, pp. 1–19, 2017.

[51] Michael Vrable, Stefan Savage, and Geoffrey M. Voelker. Cumulus: Filesystem backup to the cloud. *ACM Transactions on Storage*, Vol. 5, No. 4, pp. 14:1–14:28, 2009.

[52] Microsoft TechNet. Windows server migration tools and guides. https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2008-R2-and-2008/dd759159(v=ws.11), 2009.

[53] Dell Inc. Dell FluidFS NAS Solutions Administrator's Guide. https://www.dell.com/support/manuals/jp/ja/jpbsd1/powervault-nx3610/pvfluidfsv3ag-v2/introduction, 2013.

[54] Arpan Merchant. SnapMirror configuration and best practices guide for Clustered Data ONTAP.

Technical Report 4015, NetApp, 2018.

[55] Tim Bisson, Yuvraj Patel, and Shankar Pasupathy. Designing a fast file system crawler with incremental differencing. *ACM SIGOPS Operating Systems Review*, Vol. 46, No. 3, pp. 11–19, 2012.

[56] Andrew W. Leung, Shankar Pasupathy, Garth Goodson, and Ethan L. Miller. Measurement and analysis of large-scale network file system workloads. In *Proceedings of the 2008 USENIX Annual Technical Conference*, ATC '08, pp. 213–226, 2008.

[57] Dave Hitz, Bridget Allison, Andrea Borr, Rob Hawley, and Mark Muhlestein. Merging NT and UNIX filesystem permissions. In *Proceedings of the 2nd USENIX Windows NT Symposium*, pp. 87–96, 1998.

[58] Ellie Berriman and Binguxe Cai. NetApp storage system multiprotocol users guide. Technical Report 3490, NetApp, 2011.

[59] Steven M. French. A new network file system is born: Comparison of SMB2, CIFS, and NFS. In *Proceedings of the Linux Symposium 2007*, Vol. 1, pp. 131–140, 2007.

[60] Thomas Haynes. Network file system (NFS) version 4 minor version 2 protocol. `https://www.rfc-editor.org/rfc/rfc7862.txt`, 2016.

[61] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To FUSE or not to FUSE: Performance of user-space file systems. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, FAST '17, pp. 59–72, 2017.

[62] Boah. Harrosh. ZUFS - zero-copy (low latency) user-mode fs. In *Presentation at Linux Plumbers Conference*, 2017.

[63] Philippe Deniel, Thomas Leibovici, and Jacques-Charles Lafoucriere. GANESHA, a multi-usage with large cache NFSv4 server. In *Proceedings of the Linux Symposium 2007*, Vol. 1, pp. 113 – 124, 2007.

[64] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, FAST '11, pp. 1–14, 2011.

[65] Allen B. Downey. The structural cause of file size distributions. In *Proceedings of the 9th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS 2001, pp. 361–370, 2001.

[66] John R. Douceur and William J. Bolosky. A large-scale study of file-system contents. In *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '99, pp. 59–70, 1999.

[67] Dell EMC. VNX: What tools does Dell EMC recommend to migrate data between arrays? `https://community.emc.com/docs/DOC-63414`, 2018.

[68] NetApp. NetApp XCP migration tool. `https://xcp.netapp.com/`, 2019.

[69] Takashi Sato. ext4 online defragmentation. In *Proceedings of the Linux Symposium 2007*, Vol. 1, pp. 179–186, 2007.

[70] Yongseok Oh, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Caching less for better performance: balancing cache size and update cost of flash memory cache in hybrid storage systems. In *Proceedings of 10th USENIX Conference on File and Storage Technologies*, FAST '12, pp. 313–326, 2012.

[71] Timothy Pritchett and Mithuna Thottethodi. SieveStore: a highly-selective, ensemble-level disk

cache for cost-performance. In *Proceedings of 37th Annual International Symposium on Computer Architecture*, ISCA '10, pp. 163–174, 2010.

[72] Qing Yang and Jin Ren. I-CASH: intelligently coupled array of SSD and HDD. In *Proceedings of IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA 2011, pp. 278–289, 2011.

[73] Mustafa Canim, George A. Mihaila, Bishwaranjan Bhattacharjee, Kenneth A. Ross, and Christian A. Lang. SSD bufferpool extensions for database systems. *In Proceedings of VLDB Endowment*, Vol. 3, No. 1-2, pp. 1435–1446, 2010.

[74] Woon-Hak Kang, Sang-Won Lee, and Bongki Moon. Flash as cache extension for online transactional workloads. *The VLDB Journal*, Vol. 25, No. 5, pp. 673–694, 2016.

[75] Jeanna Matthews, Sanjeev Trika, Debra Hensgen, Rick Coulson, and Knut Grimsrud. Intel Turbo Memory: nonvolatile disk caches in the storage hierarchy of mainstream computer systems. *ACM Transactions on Storage*, Vol. 4, No. 2, pp. 4:1–4:24, 2008.

[76] Mohan Srinivasan. FlashCache: a write back block cache for Linux. `https://github.com/facebookarchive/flashcache`, 2011.

[77] Thomas M. Coughlin. New storage hierarchy for consumer computers. In *Proceedings of 2011 IEEE International Conference on Consumer Electronics*, ICCE 2011, pp. 483–484, 2011.

[78] Kent Overstreet. Bcache. `https://bcache.evilpiepirate.org/`.

[79] Fei Meng, Li Zhou, Xiaosong Ma, Sandeep Uttamchandani, and Deng Liu. vCacheShare: automated server flash cache space management in a virtualization environment. In *Proceedings of 2014 USENIX Annual Technical Conference*, ATC '14, pp. 133–144, 2014.

[80] Jun He, Xian-He Sun, and Rajeev Thakur. KNOWAC: I/O prefetch via accumulated knowledge. In *Proceedings of 2012 IEEE International Conference on Cluster Computing*, CLUSTER '12, pp. 429–437, 2012.

[81] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas J. A. Harvey, and Andrew Warfield. Characterizing storage workloads with counter stacks. In *Proceedings of 11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '14, pp. 335–349, 2014.

[82] Gokul Soundararajan, Madalin Mihailescu, and Cristiana Amza. Context-aware prefetching at the storage server. In *Proceedings of 2008 USENIX Annual Technical Conference*, ATC '08, pp. 377–390, 2008.

[83] Gong Zhang, Lawrence Chiu, Clem Dickey, Ling Liu, Paul Muench, and Sangeetha Seshadri. Automated lookahead data migration in ssd-enabled multi-tiered storage systems. In *Proceedings of IEEE 26th Symposium on Mass Storage Systems and Technologies*, MSST '10, pp. 1–6, 2010.

[84] Xin Liu, Ashraf Aboulnaga, Kenneth Salem, and Xuhui Li. CLIC: client-informed caching for storage servers. In *Proceedings of 7th USENIX Conference on File and Storage Technologies*, FAST '09, pp. 297–310, 2009.

[85] Tian Luo, Rubao Lee, Michael Mesnier, Feng Chen, and Xiaodong Zhang. hStorage-DB: heterogeneity-aware data management to exploit the full capability of hybrid storage systems. *In Proceedings of VLDB Endowment*, Vol. 5, No. 10, pp. 1076–1087, 2012.

[86] Chaitanya Yalamanchili, Kiron Vijayasankar, Erez Zadok, and Gopalan Sivathanu. DHIS: discriminating hierarchical storage. In *Proceedings of The Israeli Experimental Systems Conference*, SYSTOR 2009, pp. 9:1–9:12, 2009.

[87] Ioan Stefanovici, Eno Thereska, Greg O'Shea, Bianca Schroeder, Hitesh Ballani, Thomas Karagiannis, Antony Rowstron, and Tom Talpey. Software-defined caching: managing caches in multi-tenant data centers. In *Proceedings of 6th ACM Symposium on Cloud Computing*, SoCC '15, pp. 174–181, 2015.

[88] Erick Bauman, Gbadebo Ayoade, and Zhiqiang Lin. A survey on hypervisor-based monitoring: approaches, applications, and evolutions. *ACM Computing Surveys*, Vol. 48, No. 1, pp. 10:1–10:33, 2015.

[89] Vasily Tarasov, Deepak Jain, Dean Hildebrand, Renu Tewari, Geoff Kuenning, and Erez Zadok. Improving I/O performance using virtual disk introspection. In *Proceedings of 5th USENIX Workshop on Hot Topics in Storage and File Systems*, HotStorage '13, 2013.

[90] Youhui Zhang, Yu Gu, Hongyi Wang, and Dongsheng Wang. Virtual-machine-based intrusion detection on file-aware block level storage. In *Proceedings of 18th International Symposium on Computer Architecture and High Performance Computing*, pp. 185–192, 2006.

[91] Leo Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving virtualized storage performance with sky. In *Proceedings of 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '17, pp. 112–128, 2017.

[92] Bryan D. Payne, Martim Carbone, Monirul I. Sharif, and Wenke Lee. Lares: an architecture for secure active monitoring using virtualization. In *Proceedings of 2008 IEEE Symposium on Security and Privacy*, pp. 233–247, 2008.

[93] Martim Carbone, Matthew Conover, Bruce Montague, and Wenke Lee. Secure and robust monitoring of virtual machines through guest-assisted introspection. In *Proceedings of 15th International Conference on Research in Attacks, Intrusions, and Defenses*, RAID'12, pp. 22–41, 2012.

[94] Installing and configuring vmware tools. `https://www.vmware.com/pdf/vmware-tools-installation-configuration.pdf`.

[95] About the virtual machine agent and extensions for windows VMs. `https://docs.microsoft.com/en-us/azure/virtual-machines/windows/classic/agents-and-extensions`.

[96] The /proc Filesystem. `https://www.kernel.org/doc/Documentation/filesystems/proc.txt`.

[97] Device-mapper resource page. `http://www.sourceware.org/dm/`.

[98] Chuck Paridon. Storage performance benchmarking guidelines - Part I: workload design. `https://www.snia.org/sites/default/files/PerformanceBenchmarking.Nov2010.pdf`, 2010.

[99] Windsor W. Hsu, Alan Jay Smith, and Honesty C. Young. I/O reference behavior of production database workloads and the TPC benchmarks — an analysis at the logical level. *ACM Transactions on Database Systems*, Vol. 26, No. 1, pp. 96–143, 2001.

[100] Andrew Bond, Douglas Johnson, Greg Kopczynski, and H. Reza Taheri. Profiling the performance of virtualized databases with the TPCx-V benchmark. In *Proceedings of 7th TPC Technology Conference on Performance Evaluation & Benchmarking*, TPCTC 2015, pp. 156–172, 2015.

[101] Michael Strassmaier. Intel Optane DC Persistent Memory Performance Review. In *Presentation at Storage Developers Conference 2019*, SDC '19, 2019.

[102] Storage Networking Industry Association. NVM programming model. `https://www.snia.org/tech_activities/standards/curr_standards/npm`, 2017.

[103] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pp. 133–146, 2009.

[104] Xiaojian Wu and A. Reddy. SCMFS: A file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pp. 1–11, 2011.

[105] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of 14th USENIX Conference on File and Storage Technologies*, FAST '16, pp. 323–338, 2016.

[106] Stephan Bates and Neal Christensen. PM support in Linux and Windows. In *Presentation at SNIA Persistent Memory Summit*, PM SUMMIT '18, 2018.

[107] Sudarsun Kannan, Ada Gavrilovska, and Karsten Schwan. pVM: Persistent virtual memory for efficient capacity scaling and object storage. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pp. 13:1–13:16, 2016.

[108] Taeho Hwang, Jaemin Jung, and Youjip Won. HEAPO: Heap-based persistent object store. *ACM Transactions on Storage*, Vol. 11, No. 1, pp. 3:1–3:21, 2014.

[109] Jinglei Ren, Qingda Hu, Samira Khan, and Thomas Moscibroda. Programming for non-volatile main memory is hard. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, APSys '17, pp. 13:1–13:8, 2017.

[110] Xianzhang Chen, Edwin H.-M. Sha, Qingfeng Zhuge, Ting Wu, Weiwen Jiang, Xiaoping Zeng, and Lin Wu. UMFS: An efficient user-space file system for non-volatile memory. *Journal of Systems Architecture*, Vol. 89, pp. 18 – 29, 2018.

[111] PMDK team. Persistent Memory Development Kit. `https://pmem.io/`.

[112] Virendra J. Marathe, Margo Seltzer, Steve Byan, and Tim Harris. Persistent memcached: Bringing legacy code to byte-addressable persistent memory. In *Proceedings of 9th USENIX Workshop on Hot Topics in Storage and File Systems*, HotStorage '17, 2017.

[113] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. SplitFS: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pp. 494–508, 2019.

[114] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. Performance and protection in the zofs user-space nvm file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pp. 478–493, 2019.

[115] SPDK. Storage Performance Development Kit. `https://spdk.io/`.

[116] Krzysztof Czyrylo. Using persistent memory to build a high-performance, fully user space file system. In *Presentation at Open Source Summit Europe*, OSSEU '17, 2017.

[117] MongoDB, Inc. MongoDB. `https://www.mongodb.com/`.

[118] MariaDB Foundation. MariaDB. `https://mariadb.org/`.

[119] Intel Inc. Intel 64 and ia-32 architectures optimization reference manual. `https://software.intel.com/en-us/articles/intel-sdm`.

[120] Zhan Shi, Dan Feng, Heng Zhao, and Lingfang Zeng. USP: A lightweight file system management

framework. In *Proceedings of IEEE Fifth International Conference on Networking, Architecture, and Storage*, NAS 2010, pp. 250–256, 2010.

[121] Sanjay Ghemawat and Jeff Dean. LevelDB. `https://github.com/google/leveldb`.

[122] The Linux Kernel Organization, Inc. The Linux Kernel Archives. `https://www.kernel.org/`.

[123] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. Librispeech: An asr corpus based on public domain audio books. pp. 5206–5210, 04 2015.

[124] Steven Rostedt. ftrace - function tracer. `https://www.kernel.org/doc/Documentation/trace/ftrace.txt`.

[125] Jonathan Corbet. The future of the page cache. `https://lwn.net/Articles/712467/`.

[126] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the Intel Optane DC Persistent Memory Module, 2019.

発表文献

原著論文

1. Keiichi Matsuzawa, Mitsuo Hayasaka, and Takahiro Shinagawa. Practical File Server Migration. *ACM Transaction on Storage*, Vol. 16, No. 2, Article 13, 2020.
2. 松沢 敬一, 林 真一, 大谷 俊雄, 岩嵜 正明. RDBMS におけるクエリ実行計画情報を用いたストレージ階層制御による高速化手法. 情報処理学会論文誌, Vol. 55, No. 7, pp. 1637-1644, 2014.

国際会議

1. Keiichi Matsuzawa, Mitsuo Hayasaka, and Takahiro Shinagawa. The Quick Migration of File Servers. *In Proceedings of The 11th ACM International Systems and Storage Conference (SYSTOR 2018)*, pp.65-75, 2018.
2. Keiichi Matsuzawa and Takahiro Shinagawa. VM-Aware Adaptive Storage Cache Prefetching. *In Proceedings of The 9th IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2017)*, pp.65-73, 2017.

研究報告

1. 松沢 敬一, 品川 高廣. 低アクセスレイテンシを実現する 不揮発性メモリ向けユーザー空間ファイルシステム, 第 31 回コンピュータシステム・シンポジウム (ComSys2019), 2019.
2. 松沢 敬一, 早坂 光雄, 品川 高廣. オンデマンド及びバックグラウンド複製によるファイルサーバ移行時の停止時間削減, 第 29 回コンピュータシステム・シンポジウム (ComSys2017), 2017.
3. 松沢 敬一, 品川 高廣. 仮想マシン内のデータ配置を活用した先読み型ストレージ階層管理, 第 28 回コンピュータシステム・シンポジウム (ComSys2016), 2016.
4. 松沢 敬一, 林 真一, 大谷 俊雄. Application-aware なデータ階層管理によるアプリケーション処理高速化方式, 情報処理学会 第 120 回 システムソフトウェアとオペレーティング・システム研究会, 2012.
5. 松沢 敬一, 揚妻 匡邦, 亀井 仁志, 中野 隆裕. ファイルサーバ向け仮想化機能の設計と実装（２）, 情報処理学会 第 71 回全国大会, 2009.
6. 松沢 敬一, 揚妻 匡邦, 中野 隆裕, 岩嵜 正明. Linux カーネルにおける性能情報取得機構, 情報処理学会 第 68 回全国大会, 2006.
7. 松沢 敬一, 新堂 克徳, 越塚 登, 坂村 健. 管理者による視覚型認証を支援する IC カードを用いた本人確認システム, 情報処理学会 第 66 回全国大会, 2004.

8. 松沢 敬一, 天羽 賢一, 岩崎 慶, 西田 友是. 干渉光を考慮した変形する石鹸泡の高速なレンダリング手法, 画像電子学会 VC シンポジウム, pp.93-98, 2003.

特許

1. Keiichi Matsuzawa, and Hitoshi Kamei. Computer system, computer, and method to manage allocation of virtual and physical memory areas. US10235282. 2016.
2. Keiichi Matsuzawa. Data synchronization among file storages using stub files. US9460106. 2010.