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

松沢 敬一[1,2]　品川 高廣[2]

概要：仮想化技術の進展により，複数のアプリケーションを仮想化して単一ホスト計算機上で集約動作させる用途が増加している．また，SSD の普及に伴い，HDD と SSD を併用してよく参照されるデータのみを SSD に載せるストレージ階層管理が効果的になっている．しかし，仮想化環境ではゲスト OS のファイルシステムや仮想ディスクイメージ管理が介在するため，アプリケーションのデータ参照の局所性はホスト OS のボリューム管理層においては薄れてしまう．従来のストレージ階層管理では，ホスト OS からアプリケーション固有の情報を十分に活用することが難しかったり，情報を伝えるためにアプリケーションの改変が必要であったりした．本論文では，ゲスト OS 内のエージェントでアプリケーション固有のデータ配置に関する情報を抽出することにより，ホスト OS においてデータの先読みを効果的におこなって SSD に優先搭載するストレージ階層管理手法を提案する．本方式では，特に DB をはじめとする大容量データを扱うアプリケーションがファイルを単位としたデータ配置をおこなっていることが多いことに着目し，ファイルの配置をホスト OS に伝えることで参照の局所性を活用した先読みを効果的におこなう．提案手法を Linux に実装して TPCx-V ベンチマークを実行したところ，従来のボリュームのアクセス情報のみを用いる階層管理手法に比べ，同容量の SSD を用いた場合で 58.6%の実行性能向上効果を得た．

Proactive Hierarchical Storage Management using In-VM Data Layout

Keiichi Matsuzawa[1,2]　Takahiro Shinagawa[2]

1. はじめに

近年，仮想マシン技術やクラウドサービス等，複数のアプリケーションを仮想化して単一のホスト計算機で動作させる技術が進展・普及している．これらの技術は，従来それぞれ個別に計算機を準備して動かしていたアプリケーションを少数の計算機に集約させることができ，コスト効率の良い運用をおこなうことが出来る．

一方，コンピュータシステムが扱うデータの総量は増加し続けている．従来，データを保存する記憶媒体としては長らく HDD が使用されてきたが，HDD のデータ入出力性能は容量の増加に比べ向上ペースが鈍く，容量あたりの性能は年々低下する傾向にあった [1]．これに対し，近年 SSD をはじめとした NAND Flash を用いた記憶媒体が登場し，その性能の高さから急速に普及を進めている．SSD は普及に伴いビットコストが低下しており，将来的には HDD に漸近すると予測されているが，現時点ではまだ数倍の開きがある [2], [3]．そこで，高価だが高性能な SSD と，安価で大容量の HDD を組み合わせ，性能と容量を両立したストレージを実現するため，これまでストレージ階層管理に関する研究が多数おこなわれてきた [4], [5], [6], [7]．

しかし仮想化環境においては，アプリケーションからのストレージアクセスが実際の HDD や SSD に到達するためには何層ものソフトウェアスタックを通過する必要があり，アプリケーションのレベルでは存在した参照の局所性が実際に HDD や SSD に到達した段階では見えにくくな

1　(株) 日立製作所 研究開発グループ
Hitachi, Ltd. R&D Group.
2　東京大学大学院 情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo

るという問題がある．例えば，アプリケーションはゲストOSのファイルシステムを経由してファイルにアクセスし，その後ゲストOSのディスク管理層やホストOSにおける仮想ディスクイメージ管理，更にホストOSのファイルシステムやディスク管理層などを経由することになり，実際のデータ配置は大きく変わってしまうことがある．

従来のストレージ階層管理手法としては，まずアプリケーションが直接HDDとSSDを管理する手法がある [4], [5]．しかし，この手法ではアプリケーションに改変が必要であるほか，クラウドでゲストOSに物理ディスクを直接アクセスさせると，管理者によるストレージ管理が複雑になる．アプリケーションには依存せずにストレージ階層管理をおこなう手法としては，OSのストレージ管理層でキャッシュをおこなう手法がある [6], [7]．しかし，この手法ではアプリケーションの特性が見えにくくなり，特に仮想化環境では参照の局所性が失われる可能性が高い．

本論文では，ゲストOS内においてアプリケーション固有のデータ配置に関する情報を取得するエージェントを配置することにより，アプリケーション自身は改変することなくそのデータ配置に関する情報をホストOSのストレージ階層管理システムに伝え，データの先読みをおこなって次に参照される可能性が高いデータを効果的にSSDに配置する手法を提案する．本手法では，データベースやメールサーバなどの大容量データにランダムアクセスする傾向があるアプリケーションにおいては，ファイルを単位としたデータ配置をおこなっていることが多く，かつそのファイル内においては参照の局所性があることが多いことに着目し，現在アクセスされているデータが存在するファイルのデータを優先的に先読みしてSSDにキャッシュする．これにより，アプリケーションの参照特性が時間によって変化した場合においてもいち早く追随し，より効果的にデータをSSDにキャッシュすることを目指す．

提案手法をPostgreSQLとLinuxを対象として実装し，TPCx-Vベンチマークで実験したところ，従来のFlashCacheを使ったOSのストレージ階層のみにおいてキャッシュをおこなうシステムと比べて，58.6%高いTransaction/secが達成できることを確認した．

以下，本論文では，2節でストレージ階層管理に関する従来研究について分類し，3節で提案手法のデザインについて述べ，4節でPostgreSQLへの適用について述べ，6節で実装及び性能評価を示し，7節にてまとめる．

2. 関連研究

ストレージ階層管理の手法は，階層管理を行うコンピュータシステム上の部位やデータ配置のアプローチによって分類できる．

階層管理を行う部位について，最も単純な手法はアプリケーションで直接HDDとSSDを管理し，最適なデータ配置を行う手法である．例えばSSD Bufferpool Extension[4]やFaCE[5]はRDBMSを対象アプリケーションとし，RDBMSが管理するバッファキャッシュを拡張する手段として用い，SSDにデータ配置を行う．これらはアプリケーション固有の情報を用いてデータ配置を行うため，高いキャッシュヒット率が期待できる．一方個々のアプリケーションに対して改造を要する点や，複数のアプリケーションでSSDを共有することを想定していない点で複数のアプリケーションが混在する環境へは適用できない．

より下位に位置するシステムソフトウェアで階層機構を実装する手法には，LinuxのブロックI/O層にキャッシュ管理機構を加えるbcache[6]やLinuxのDevice Mapperを用いてキャッシュ機能を備えたボリュームを構成するFlashCache[7], [8]がある．また，vCacheShare[9]はハイパーバイザのレイヤでVirtual Machine(VM)からのI/O要求に対するキャッシュ機構を実装し，VMから透過的にSSDをキャッシュとして利用する．これらの手法はアプリケーションに透過的にキャッシュ機能を提供可能であり，アプリケーション側の対応が不要である一方，アプリケーション固有の情報を用いたキャッシュヒット率向上が難しい．

両者の短所を補完する手法として，アプリケーションとシステムソフトウェアで連携して階層管理をおこなう研究もある．CLIC[10]やhStorage-DB[11]は，アプリケーションがI/O要求にヒント情報を付与し，システムソフトウェアがそのヒント情報を用いて対象データをキャッシュするかどうか判断可能とすることで，キャッシュヒット率を向上させる．しかし両手法はアプリケーション・システムソフトウェアの双方に変更を要するため，多様なアプリケーションが混在する環境への適用は難しい．

ストレージ階層管理におけるデータ配置のアプローチは，前回アクセス時のデータの保持と，先読みに分類できる．ここまで挙げた関連研究はいずれも前者に相当する．これらは，データアクセスの時間的な局所性に基づき，最近アクセスされたデータをSSDにしばらく保持しておくことで，同じデータに次にアクセスが行われた場合の応答時間の短縮を図る．

後者の先読みに相当する研究としては，KNOWAC[12]やLAM[13]が挙げられる．これらはデータアクセスが長周期で類似したパターンを繰り返すことを前提とし，ブロックI/O層で過去のアクセスパターン統計情報を元に直近のデータアクセスの情報から次にアクセスされるデータを予測し，そのデータをHDDからSSDに先読みする．これにより，以後のデータアクセスに対しデータをSSDから読み込むことでI/Oアクセスの応答時間の期待値を短縮する．この手法は，ブロックI/O層におけるデータアクセスの周期性を前提とするため，バッチ処理のような日々同じ処理を繰り返すワークロードには有効であるが，Webサー

ビスのようにユーザーの動向次第でアクセス位置が変動する周期性が弱いワークロードでは効果が見込めない．

3. 提案手法

3.1 アプローチ

RDBMS，Key-value Store，文書検索，メールサーバ等，大量データを扱うアプリケーションは，アプリケーション固有のデータ構造によりデータを分類し管理する．例えば，RDBMS は固有のデータ構造としてテーブルやインデックス単位でデータを管理する．そしてそのデータを，OS が管理するボリュームやファイルシステム上に格納する．本手法は，このようなアプリケーション仮想化技術を用いて単一計算機上で複数動作させる構成を対象する．

本手法では，アプリケーションが上記データの管理単位ごとに備えているアクセスの局所性を活かし，キャッシュヒット率を向上する．

アプリケーションが管理するデータの格納位置は，実際に記憶媒体上に格納されるまでにゲスト OS によるファイルシステムやホスト OS による仮想ディスクイメージ管理が介在することで変換される．そのため，記憶媒体に近いソフトウェアの機構，例えばホスト OS のボリューム管理機構において I/O のアクセスパターンを分析しても，本来のアプリケーションによるアクセスの局所性が十分反映されず，キャッシュヒット率を十分向上できるような階層移動が行えない．

そこで本手法では，各ゲスト OS 内のアプリケーションがボリューム上に構築するデータのレイアウト情報をホスト OS で集約する．そしてこのレイアウト情報と，ホスト OS のボリューム管理層で取得した I/O アクセス統計情報を合わせて，データ管理単位でアクセス頻度を集計する．そして，高アクセス頻度の傾向を示すデータ管理単位を一括して SSD キャッシュに移動する．

データ管理単位が全体で高アクセス頻度の傾向を示す場合，内部にまだアクセスされていない領域が含まれていても，近い将来にアクセスされることが期待できる．この手順により，データを先読みの形で SSD キャッシュに移動し，以後のデータアクセスを高速化する．

3.2 アーキテクチャ

1 に本手法の全体図を示す．ホスト計算機は HDD と SSD を備えている．ホスト OS はハイパーバイザの機能を備え，内部で複数の仮想マシンが動作する．各仮想マシンはゲスト OS が管理し，その上でアプリケーションが動作する．

提案手法は，ホスト OS とゲスト OS の連携によって，HDD と SSD の階層管理を実現する．網掛け部は，本提案手法で新規に作成するソフトウェアを示す．

ホスト OS では，論理ボリューム管理機能と，最適配置プログラムが動作する．論理ボリューム管理機能は HDD と SSD から複数の論理ボリュームを作成する．各論理ボリュームは VM に割り当てられ，ゲスト OS 内のアプリケーションがアクセスする．

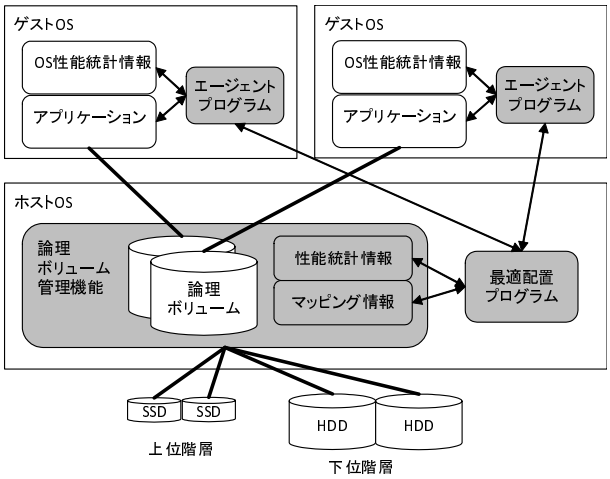


図 1: 提案するストレージ階層管理の構成

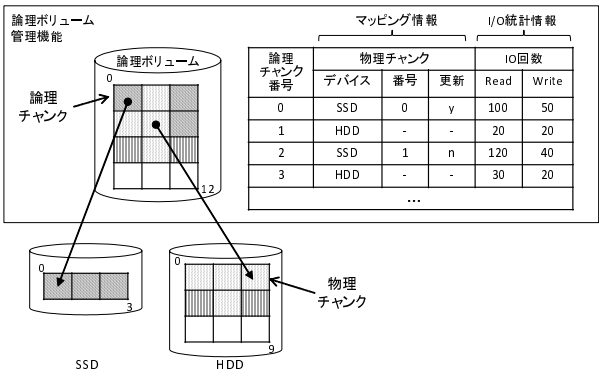


図 2: 論理ボリューム管理機能の構成

最適配置プログラムは各ゲスト OS 内のエージェントプログラムと通信し，アプリケーションが論理ボリューム上に構築するデータレイアウト情報や，統計情報を取得する．そして取得した情報を集計して高アクセス頻度であるデータ管理単位を求め，それらのデータが SSD に配置されるよう論理ボリュームの構成を指示する．

ゲスト OS 内ではエージェントプログラムが動作する．このエージェントプログラムは，アプリケーションのデータレイアウトや性能統計情報を収集し，ホスト OS 中の最適配置プログラムに伝える．

3.3 論理ボリューム管理機能

論理ボリューム管理機能の詳細を図 2 に示す．論理ボリューム管理機能は，ホスト OS が管理する HDD 及び SSD から，ゲスト OS に提供する複数の論理ボリュームを作成する．各論理ボリュームは，固定長の論理チャンクの集合として管理され，HDD 及び SSD の領域も同サイズの

物理チャンクの集合として管理される．論理チャンクと物理チャンクは，論理ボリューム管理機能内のマッピング情報により対応付けられる．ゲスト OS が論理ボリュームに対し I/O 要求を行うと，論理ボリューム管理機能はマッピング情報を参照し，アクセス対象の論理チャンクに対応する物理チャンクを求めて，その物理チャンクに I/O 要求を転送する．

マッピング情報は最適配置プログラムから変更を指示できる．マッピング情報の変更指示要求が出されると，論理ボリューム管理機能は，対象論理チャンクへの I/O を一時的に停止して物理チャンクを参照不可にし，その間に変更元の物理チャンク間から変更先の物理チャンクにデータをコピーする．コピー完了後，マッピング情報を変更して I/O を再開することで，アプリケーションによる論理チャンクへのデータアクセスの一貫性を保ちながらマッピング情報の変更指示に対応できるする．このマッピング変更時のデータコピーを削減するための最適化として，マッピング情報には各論理チャンクに対し最後に実行されたマッピング変更以後のデータ更新の有無を記憶しておき，マッピング変更時は，当該論理チャンクに対するデータ更新が行われていた場合のみデータをコピーする．

以下，このマッピング情報変更指示及びそれに伴う物理チャンク間のデータコピーを HDD/SSD 間の階層移動と呼び，論理チャンクを HDD/SSD のどちらにマッピングするか定めることを HDD/SSD への配置と呼ぶ．

さらに，論理ボリューム管理機能は性能統計情報として，各論理チャンクに対する IO の回数や，論理チャンクアクセス順の LRU を保持する．

3.4 エージェントプログラム

エージェントプログラムは各ゲスト OS 内で動作するプログラムで，ゲスト OS で動作するアプリケーションのデータレイアウト情報や性能統計情報を収集し，ホスト OS の最適配置プログラムに送る．

本手法におけるデータレイアウト情報とは，アプリケーションにおけるデータ管理単位と，そのデータ管理単位が格納された論理ボリューム上の位置 (LBA の集合) の対応関係を列挙したものである．

この対応関係の取得方法はアプリケーション依存であるが，多くはアプリケーションの設定ファイルの参照や，アプリケーションに問い合わせにより取得できる．

アプリケーションがファイルシステム上にデータを格納している場合は，エージェントプログラムはアプリケーションが格納したファイル名を取得し，さらにファイルシステムに各ファイルの論理ボリューム上のデータ格納位置を問い合わせることで対応関係を取得できる．

エージェントプログラムは，性能統計情報としてゲスト OS の CPU 利用率と，アプリケーション固有の性能指標を取得し送信する．例えば RDBMS の場合，クエリの実行回数が固有の性能指標に相当する．

3.5 最適配置プログラム

最適配置プログラムは，論理チャンクの配置及び階層移動を行う．このプログラムは，定期的に情報収集・移動計画・階層移動の 3 ステップを繰り返す．

情報収集のステップでは，チャンクの配置に必要な情報を収集する．各ゲスト OS 内のエージェントプログラムと通信してアプリケーションのデータレイアウトや性能統計情報を取得し，また論理ボリューム管理機能の性能統計情報を取得する．

続く移動計画のステップでは，取得した情報を元に配置を変更する論理チャンク群を決定する．さらに，階層移動の並列度や実行時間の上限を定める．上限を設定するのは，階層移動は 3.3 で記載した通り，データコピーの I/O を伴うために，階層移動を多並列で長時間行うと，却って通常のアプリケーションの I/O 処理時間を悪化させてしまうためである．このステップは，キャッシュ管理を行う最重要のステップである．キャッシュ管理の戦略については 3.6,4 にて論ずる．

最後の階層移動ステップでは，上記移動計画ステップで定めた配置に基づき，階層移動を行う．階層移動は，HDD と SSD 間のデータコピーを伴うため，即座に全チャンクに対して実行できない．そこで移動計画ステップで定めた通りの並列度と実行時間で階層移動を行う．

3.6 キャッシュ管理戦略

本節では，最適配置プログラムの移動計画ステップで行うキャッシュ管理の戦略について論ずる．

本研究の目的はアプリケーションの性能を向上させることである．そのため，性能が最大限向上するようにデータの階層移動を行う．

本手法では，低速な HDD へのアクセス頻度を減らせるよう，SSD が HDD の Writeback キャッシュとして振る舞うようにマッピングを設定する．論理チャンクと HDD 上の物理チャンクは一対一でリニアマッピングする．これは低速な HDD へのランダムアクセスを増加させないよう，論理チャンクと物理チャンクの対応が不連続となることを防ぐためである．HDD 上のデータを SSD にキャッシュしたい場合は，HDD の物理チャンクと対応する論理チャンクを SSD 上の物理チャンクにマッピング変更する．逆にキャッシュから掃き出す場合は，元の HDD 上の物理チャンクにマッピングを戻す．

階層移動のペース（移動する時間の上限や並列度）は，大きくし過ぎないよう調整しなければならない．急激なデータ階層移動を行うと，前述のとおりマッピング変更に伴うデータコピーのオーバヘッドがキャッシュヒットの改

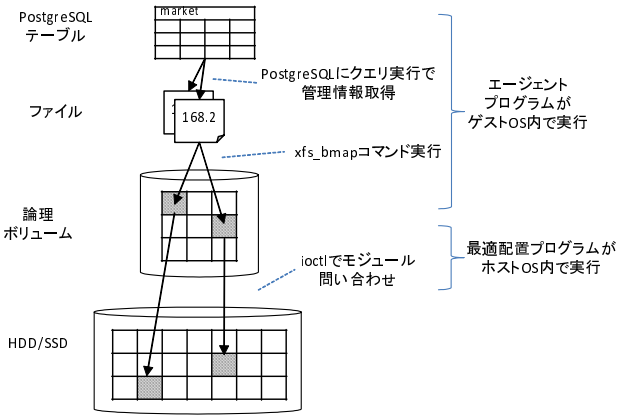


図 3: テーブルと物理チャンクの対応関係の取得

善効果を上回り，結果的にアプリケーション性能を低下させる．そのため，アプリケーションへの要求性能の下限を下回らない範囲で階層移動のペースを定める．

本手法は本質的に先読みであることと，階層移動自体が一時的に I/O 性能に負の影響を及ぼすことから，事前に最適な階層移動のペースを求めることは困難である．そのため，階層移動のペースを調整のため，短時間階層移動を実行後，アプリケーションの性能の変化を観測し，その結果をフィードバックして階層移動のペースを調整する，といった手段を適用する．

4. PostgreSQL への手法適用

本節では，アプリケーションとして RDBMS の一種である PostgreSQL を対象とし，提案手法を適用する場合の構成とそのキャッシュ戦略について述べる．

PostgreSQL は管理対象のテーブルを 1GB 単位で分割し，それぞれファイルとしてファイルシステム上に格納する．そのため，PostgreSQL に提案手法を適用した場合，ゲスト OS 内の PostgreSQL が持つテーブルと，その格納先物理チャンクの位置関係はゲスト OS 内のファイルシステムと論理ボリュームを介して対応付けられる (図 3)．

ゲスト OS 内のエージェントプログラムは，以下の情報を返す．

- データレイアウトに関する情報 (図 3)
  - PostgreSQL のクエリを実行し，データベースを構成するテーブル・インデックスの一覧を取得
  - PostgreSQL のクエリを実行し，上記テーブル・インデックスが格納されたファイル名を取得
  - XFS のコマンド xfs_bmap を呼び，上記ファイルと論理ボリューム上のデータの位置関係を取得
- 実行性能に関する情報
  - クエリの実行回数統計情報
  - ゲスト OS の CPU 利用率

上記の構成とエージェントプログラムが返す情報を踏まえ，以下の方針で階層移動を行う．

まず，ゲスト OS のファイルごとに論理チャンクの I/O 回数を集計する．今回 PostgreSQL のテーブル・インデックス単位ではなく，テーブル・インデックスを構成する個々のファイル単位で回数の集計を行った．その理由は，事前に PostgreSQL の I/O の傾向を測定したところ，同一テーブル・インデックス内部においても I/O の偏りと局所性が観測されたことと，巨大なテーブル全体の階層移動はデータ量が多く I/O のオーバヘッドが大きいためである．

次に上記集計結果において，各ファイルを容量あたりのアクセス頻度でスコアリングし，降順にソートする．このスコアは，HDD にマップされた物理チャンクへの Read 回数と Write 回数を 3:1 で重みづけして合算する．RDBMS のクエリ処理において，Read は応答を返すまでクエリを続行できないため同期 I/O として発行されるが，Write は非同期的 I/O として遅延処理されることが多く，Read の応答時間の方がアプリケーションの処理性能に大きく影響するためである．この重みづけの結果，HDD に Read 要求が多数発生しているファイルほど SSD にキャッシュされやすくなる．

このようにスコア順に降順ソートしたファイルに対し，ファイルの格納先の論理チャンクを先頭から順に制限時間 (30 秒のうち事前に定めた一定の割合) まで SSD に階層移動することでデータを SSD にキャッシュする．逆に SSD マッピング済みの物理チャンクは，LRU 順で HDD に掃き出しを行った．

今回は 3.6 に示すような制限時間の動的な調整は行わず，測定前に定数として用いた．

5. 実装

本節では，Linux 及び KVM を用いた仮想化環境で動作する PostgreSQL に対する提案ストレージ階層管理手法の実装について述べる．

5.1 論理ボリューム管理機能

論理ボリューム管理機能は，ホスト OS の Linux 上で動作する Device Mapper カーネルモジュールとして実装した．

本モジュールではロード時に単一のブロックデバイスをキャッシュ用ボリュームとして設定する．実行時には，ホスト計算機上で Device Mapper を用いて，HDD で構成された既存のブロックデバイスを物理ボリュームとして指定する．本モジュールは，指定された物理ボリュームに対応する論理ボリュームとして，同じ容量を持つ SSD キャッシュ機構を備えた新規ブロックデバイスを生成する．

本実験では LVM を用いて，TierB のゲスト OS 毎に前述の容量をもつ HDD ボリュームを作成し，それぞれ対応する論理ボリュームを作成した．

5.2 エージェントプログラム

各ゲスト OS 内で動作するエージェントプログラムは，python のスクリプトとして実装した．本スクリプトは，最適配置プログラムから ssh 経由で実行される．本プログラムは，初回実行時には PostgreSQL 上の全テーブル・インデックスを取得し，対応するファイル名及び論理ボリューム上の各ファイルの配置を返す．以後定期的に実行されるたびに，CPU 利用率等の性能情報を返す．

5.3 最適配置プログラム

最適配置プログラムはホスト OS 内で動作する Python のスクリプトとして実装した．本スクリプトは，30 秒毎に周期的に起動し，4 の戦略に従いデータ階層移動を行う．

6. 評価

提案手法の効果を検証するため，TPCx-V[14] を用いた性能測定を行った．

TPCx-V は複数の仮想化環境におねるデータベース運用を模擬するベンチマークプログラムである．

TPCx-V では 13 個の仮想マシンを用いる．1 つはベンチマーク全体の動作を制御するドライバで，残りの 12 個の仮想マシンが測定対象である．測定対象の仮想マシンは 3 つで 1 つの Group を構成する．各 Group は TierA の仮想マシン 1 台，TierB の仮想マシン 2 台からなる．TierA は TierB にクエリ要求を出す役割を担い，TierB の仮想マシン群が実際にデータベースのクエリを処理し，論理ボリューム上のデータにアクセスする役割を担う．ドライバ及び TierA の仮想マシン群によるストレージへの I/O 要求はわずかであり，実行する処理も軽量であるため，本ベンチマークにおける性能は，各 Group にある TierB の仮想マシン群の応答性能によって決まる．

TPCx-V の特徴として，時間経過に合わせ負荷が変動する．TPCx-V は 1 セットの測定が 10 フェーズで構成されている．各フェーズは 12 分であり，フェーズ毎に定められた比率で各 Group にクエリ要求を出す．その際，最も応答性能が低い Group により全体の性能が評価される．

6.1 実行環境

実験環境は下記の通りである．各仮想マシンに割り当てたメモリの合計容量は 38GB であり，これはホスト計算機の搭載メモリの 95%にあたる．よって仮想マシンのメモリオーバーコミットは行っておらず，またホスト OS におけるページキャッシュの影響はほとんど生じないと考えてよい．

ホスト計算機

- CPU: Intel Xeon E5-2620 × 2
- メモリ: 40GB
- OS: CentOS 7.2.1511 (Linux 3.10.0)
- HDD: SEAGATE ST31500341AS 1.5TB 7200rpm × 6 (RAID0+1)
- SSD: Intel SSD 750 400GB (うち 100GB 利用)

仮想マシン当たりの構成

- vCPU : Driver:1, TierA:2, TierB:6
- メモリ : Driver:2GB, TierA:3GB, TirB:3GB
- OS: CentOS 7.2.1511 (Linux 3.10.0)
- DB: PostgreSQL 9.2.15
- ファイルシステム: XFS

TPCx-V は Group 毎で参照するデータセット量に傾斜をつける．そのため各 TierB に割り当てる論理ボリュームのサイズも同様に傾斜させ，Group 毎に 45GB, 90GB, 135GB, 180GB とした．各論理ボリュームは仮想マシン内で XFS でフォーマットし，PostgreSQL のデータを格納する．PostgreSQL のデータの総量は約 700GB であり，各仮想マシンで論理ボリューム容量の約 80%を占める．

6.2 実験結果

6.2.1 従来手法との比較

提案手法の有効性を検証するため，ブロック I/O 層でのみストレージ階層管理を行う FlashCache との性能比較実験を行った．測定時のパラメータは下記のとおりである．

- SSD のキャッシュ容量: 100GB
- チャンクサイズ: 1MB
- 提案手法の階層移動パラメータ:
  - 移動並列度：32
  - 移動実行時間：30 秒毎に 4 割 (12 秒) の間移動
  - 移動判定基準：30 秒毎に 200 回以上の R/W が生じたファイルが対象

本評価実験では TPCx-V を 3 セット分計 30 フェーズを連続実行し，トランザクション秒間実行回数の推移を提案手法・FlashCache・HDD のみ，の 3 通りの構成について計測した．いずれの測定中も，プロセッサ稼働率は 100%に達していなかった．一方 HDD の稼働率は常時 99%を超えており，アプリケーション実行は HDD がボトルネックとなり律速されていると考えてよい．

トランザクション秒間実行回数の推移を図 4 に示す．測定開始後，時間が経過した 3 セット目における秒間実行回数の平均で比較すると，FlashCache における秒間実行回数は HDD のみの場合の 2.06 倍，提案手法は HDD のみの場合の 3.26 倍となった．また，提案手法は常時 FlashCache よりも常時で大きな秒間事項回数を示し，平均で 58.6%上回っていた．

TPCx-V ではフェーズの境界 (図中縦線) で Group 毎の負荷比率を変化する．そのため，この境界直後では各測定とも秒間実行回数が変化する．FlashCache の場合はそれが顕著であり，フェーズ境界で秒間実行回数が大きく減少し，その後フェーズ内で徐々に増加する，という動作を繰

り返す．

提案手法もフェーズ境界では秒間実行回数が変動するものの，総じて変動幅は FlashCache より小さく，またフェーズ内の変化も小さい．これらの事実より，提案手法は同容量の SSD を用いた場合に，FlashCache に比べより高い性能をより安定して出すことができていると言える．

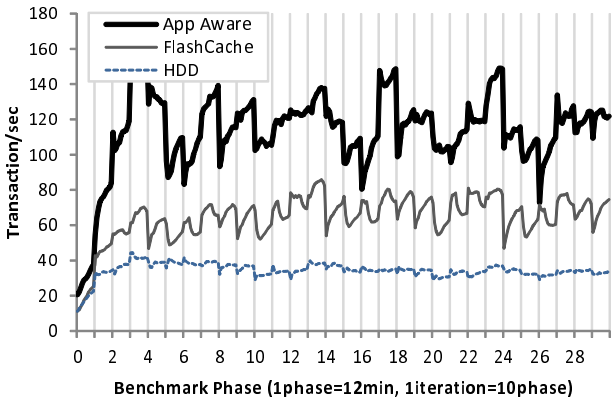


図 4: TPCx-V 実行時のトランザクション実行性能

続けて，両手法における HDD, SSD へのアクセス傾向について論ずる．図 5 は TPCx-V 実行時間中の HDD, SSD への秒間の I/O 実行回数 (tps) と Read/Write 別転送セクタ数 (sector) を示す．提案手法で HDD の読み込みセクタ数と SSD の書き込みセクタ数が大きく振動しているのは，階層移動が 30 秒毎に 12 秒すつ行うのに対し，測定のサンプリング間隔が 10 秒と短いためである．

HDD への I/O 実行回数は，両手法とも時間経過を問わず 800tps 前後で推移している．この数値は今回利用した HDD の上限性能であり，HDD がトランザクション処理のボトルネックとなっていることを示す．ただし，I/O の内訳は両手法で異なり，提案手法は HDD の Read セクタ数及び SSD の Write セクタ数が FlashCache に比べ多い．これは提案手法が 1MB 単位でチャンクを階層移動する際，大サイズの I/O 要求を行うためである．また，測定開始直後は HDD の Read セクタ数が多く，Write セクタ数が少ないが，徐々に Read セクタ数は減少して Write が増加する．これはデータの階層移動が進みキャッシュヒット率が向上してきたためと考えられる．

FlashCache は TPCx-V のフェーズ境界の周期に合わせ，I/O 特性が周期的に振動している．HDD の Write セクタ数が特にフェーズ境界直後に急上昇する．これは TPCx-V のフェーズ移行に伴いアクセス傾向が急激に変化した結果，キャッシュ対象の大規模な入れ替えが生じ，HDD 中の Dirty データの掃出しが生じたためと考えられる．また提案手法は SSD の転送セクタ数こそ FlashCache よりも多いが，I/O 実行回数では提案手法の方が常時少なく，大サイズの I/O を効率的に行えていることがわかる．

これらの分析より，提案手法はボリューム層で階層管理を行う従来手法に比べ，より効率的にデータの配置及び階層移動を行い，アプリケーションの実行性能を向上できることが分かった．

6.2.2 階層移動実行時間のアプリケーション性能への影響評価

本節では，最適配置プログラムの移動計画において動的な移動ペース変更を行うための事前測定として，階層移動パラメータの性能への影響を評価する．本測定では，30 秒毎の最適配置プログラム実行間隔に対し，パラメータとしてデータ階層移動を行う時間の割合を 0.2(6 秒)〜0.8(24 秒) まで変化させ，トランザクション秒間実行回数を測定した．

測定結果を図 6 に示す．最も高い処理性能を示したのは割合が 0.4 の場合であり，割合が大きすぎても小さすぎても性能が低下するという結果を得た．

割合が 0.2 と 0.4 の測定結果を比較すると，前者は秒間実行回数の増加ペースが他の測定に比べ低い．これは階層移動を行う時間が少ないため，負荷の変動への追従が遅れた結果と考えられる．反対に割合が 0.8 と大きい場合の性能を他の測定と比較すると，秒間実行回数が常に小さい．これは階層移動を行う時間が長いためにデータコピーのオーバヘッドが大きく，PostgreSQL のトランザクション処理性能が低下したためと考えられる．

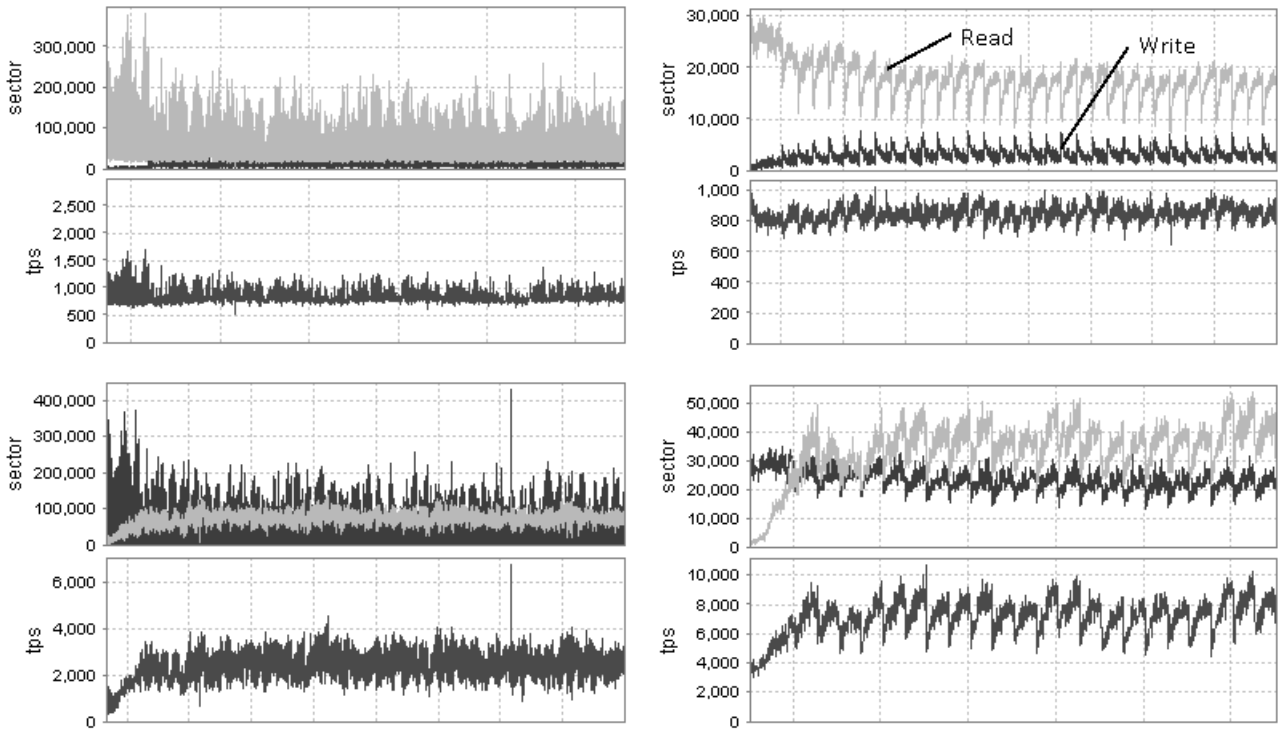
本評価実験により，階層移動自体がアプリケーションの処理性能に影響を及ぼすため，性能向上には最適な階層移動のパラメータ設定が必要であることが確認できた．階層移動に伴うパラメータを自動的に調整し，性能を最大化することは今後の課題である．

7. まとめと今後の課題

本論文ではアプリケーションが単一のホスト計算機上の複数の仮想マシンで動作する構成において，データを SSD キャッシュに先読みすることで I/O 時間を短縮し，アプリケーションの実行性能を向上させるストレージ階層管理手法を提案した．

提案手法は，アプリケーション固有のデータ管理におけるアクセスの局所性に着目し，これらのデータがゲスト OS 内でファイル単位で格納されていることを利用する．ホスト OS において，各ゲスト OS 内のファイルがボリューム上で格納されている位置を取得し，ファイル単位でボリュームの領域毎のアクセス頻度の集計，及び HDD・SSD 間のデータ階層移動を行う．これによりファイル単位でデータを SSD に先読みできるため，アプリケーションによるデータアクセスの局所性によって SSD キャッシュのヒット率及びアプリケーション実行性能が向上する．

提案手法を実際に Linux 及び PostgreSQL 上に適用し，TPCx-V ベンチマークを実行したところ，従来のボリュー



(a) 提案手法　　(b) FlashCache

図 5: TPCx-V 実行中の HDD(上段) 及び SSD(下段) アクセス傾向

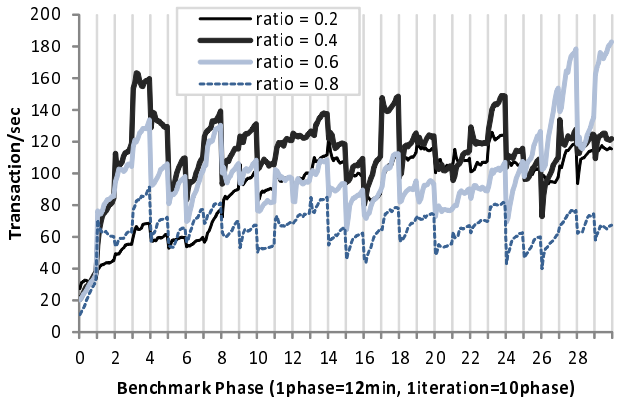


図 6: 階層移動実行時間とトランザクション処理性能

ムのアクセス情報のみ用いる階層管理手法に比べ，高い実行性能向上効果を得られることが確認できた．

今後の課題として，今回定数として設定した階層移動パラメータの動的チューニングがある．パラメータの変更に伴うアプリケーションの実行性能の変化をフィードバックし，ワークロードに適したパラメータに設定することで，更に性能を向上させることが期待できる．

参考文献

[1] Freitas, R., Slember, J., Sawdon, W. and Chiu, L.: GPFS scans 10 billion files in 43 minutes. IBM Whitepaper (2011).

[2] Fontana, R. and Decad, G.: Storage Media Overview: Historic Perspectives, *32nd International Conference on Massive Storage Systems and Technology (MSST)* (2016).

[3] Evolution of All-Flash Array Architectures (online), available from ⟨http://wikibon.org/wiki/v/Evolution_of_All-Flash_Array_Architectures⟩ (accessed 2016-10-10).

[4] Canim, M., Mihaila, G. A., Bhattacharjee, B., Ross, K. A. and Lang, C. A.: SSD Bufferpool Extensions for Database Systems, *Proc. VLDB Endow.*, Vol. 3, No. 1-2, pp. 1435–1446 (2010).

[5] Kang, W.-H., Lee, S.-W. and Moon, B.: Flash As Cache Extension for Online Transactional Workloads, *The VLDB Journal*, Vol. 25, No. 5, pp. 673–694 (2016).

[6] Bcache (online), available from ⟨https://bcache.evilpiepirate.org/⟩ (accessed 2016-10-10).

[7] FlashCache (online), available from ⟨https://github.com/facebookarchive/flashcache⟩ (accessed 2016-10-10).

[8] 加藤　純，佐藤　充：ブロックストレージシステムにおけるキャッシュの高速化，研究報告システムソフトウェアとオペレーティング・システム（OS）2016-OS-138，No. 9 (2016).

[9] Meng, F., Zhou, L., Ma, X., Uttamchandani, S. and Liu, D.: vCacheShare: Automated Server Flash Cache Space Management in a Virtualization Environment, *2014 USENIX Annual Technical Conference*, USENIX ATC 14, pp. 133–144 (2014).

[10] Liu, X., Aboulnaga, A., Salem, K. and Li, X.: CLIC: Client-informed Caching for Storage Servers, *Proccedings of the 7th Conference on File and Storage Technologies*, FAST '09, pp. 297–310 (2009).

[11] Luo, T., Lee, R., Mesnier, M., Chen, F. and Zhang, X.: hStorage-DB: Heterogeneity-aware Data Management to Exploit the Full Capability of Hybrid Storage Systems, *Proc. VLDB Endow.*, Vol. 5, No. 10, pp. 1076–1087 (2012).

[12] He, J., Sun, X.-H. and Thakur, R.: KNOWAC: I/O Prefetch via Accumulated Knowledge, *Proceedings of the 2012 IEEE International Conference on Cluster Computing*, CLUSTER '12, pp. 429–437 (2012).

[13] Zhang, G., Chiu, L., Dickey, C., Liu, L., Muench, P. and Seshadri, S.: Automated Lookahead Data Migration in SSD-enabled Multi-tiered Storage Systems, *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies*, MSST '10, pp. 1–6 (2010).

[14] TPCx-V (online), available from ⟨http://www.tpc.org/tpcx-v/⟩ (accessed 2016-10-10).