2003－DBS－131　(19)
2003／7／16

半導体ディスクを用いた自律ディスクの階層化

花井　知広[*]　渡邊　明嗣[*]　山口　宗慶[*]　田口　亮[‡]　林　直人[‡]
上原　年博[‡]　横田　治夫[†,*]

[*] 東京工業大学 大学院 情報理工学研究科 計算工学専攻
[†] 東京工業大学 学術国際情報センター
〒 152-8552 東京都目黒区大岡山 2-12-1
[‡]NHK 放送技術研究所〒 157-8510 東京都世田谷区砧 1-10-11
[*]{hanai,aki,muu}@de.cs.titech.ac.jp, [†,*]yokota@cs.titech.ac.jp,
[‡]{taguchi.r-cs,hayashi.n-gm,uehara.t-jy}@nhk.or.jp

概 要

我々は，ストレージシステムにおいて負荷分散，故障対策，障害回復などの高度な機能を実現するためのアプローチとして，自律ディスクを提案してきた．しかし，近年の性能要求の高まりにより，記録媒体としてディスク装置のみを用いたストレージでは性能要求に応えられなくなってきている．

本稿では，自律ディスクを半導体ディスクを用いて構成し，従来の自律ディスクのキャッシュとして利用することにより階層化を行い，ストレージの高速化を行う手法を検討する．データの挿入・更新時に耐故障性を保ちつつ半導体ディスクの効率的な利用を行うプロトコルや，効率的なキャッシュ管理アルゴリズムについて提案する．さらにそれらの手法について実験により評価を行い，有効性を確認した．

キーワード： 自律ディスク, 階層化ストレージ, キャッシュ管理, 半導体ディスク

Hierarchical Autonomous Disks with Solid State Disks

Tomohiro HANAI[*]　Akitsugu WATANABE[*]　Muennori YAMAGUCHI[*]
Ryo TAGUCHI[‡]　Naoto HAYASHI[‡]　Toshihiro UEHARA[‡]　Haruo YOKOTA[†,*]

[*] Department of Computer Science Graduate School of Information Science and Engineering
Tokyo Institute of Technology
[†] Global Scientific Information & Computing Center Tokyo Institute of Technology
2-12-1 Oh-Okayama, Meguro-ku Tokyo, 152-8552 JAPAN
[‡] NHK Science & Technical Research Laboratories
1-10-11 Kinuta Setagaya-ku Tokyo, 157-8510 JAPAN
[*]{hanai,aki,muu}@de.cs.titech.ac.jp, [†,*]yokota@cs.titech.ac.jp,
[‡]{taguchi.r-cs,hayashi.n-gm,uehara.t-jy}@nhk.or.jp

Abstract

We have proposed Autonomous Disks to realize high functions (blancing load, tolerating faults, and recovering failures). But recently performance requirement is higher and higher, storage using only (magnetic) disks cannot achieve sufficient performance.

In this paper, in order to improve performance of storage, we propose the use of Autonomous Disks constructed from solid state disks as cache of the one constructed from magnetic disks. Furthermore, we propose efficient protocols for insert and update operation with no loss of fault tolerance, and a efficient cache management algorithm. We also analyze its performance to indicate effectiveness of our approach.

KEYWORD: Autonomous Disks, Hierarchical Storage, cache management, solid state disks

1 はじめに

大規模データ処理システムにおけるストレージ管理コストの増加に伴い，ストレージを全システムの中心に据えるストレージセントリックシステムが注目されている. 我々は，ストレージシステムにおける負荷分散，故障対策，障害回復などの高度な機能を実現するためのアプローチとして，ディスク装置内の制御用プロセッサとキャッシュ用のメモリを利用する自律ディスクを提案してきた．[1, 2, 3]

しかし，ストレージシステムで扱われるデータ量は未だ増加の一途をたどり，その性能に対する要求もより高いものが求められている．例えば我々が提案している自律ディスクを用いたマルチメディアコンテンツサーバ [4] では，ハイビジョンコンテンツを格納・再生するために非常に高いスループットが必要である．しかし，現在のストレージシステムの基礎となる記憶媒体は磁気ディスクであり，その構造から飛躍的な速度向上は望めない状況である．

そこで本稿では磁気ディスクを用いて構成された自律ディスククラスタ上に，キャッシュ用の自律ディスククラスタを半導体ディスクを用いて構成することによる，自律ディスクを用いたストレージシステムの高速化を検討する．また，アベイラビリティを保ちつつ効率的な挿入・更新処理を行うプロトコル，効率的なキャッシュ管理アルゴリズムについても検討を行う．その上で提案手法を試作システム上に実装し，性能測定により提案手法を評価する．

以下に本稿の構成を述べる．まず 2 章で自律ディスクの概要について述べる．次に 3 章で半導体ディスクを用いた階層化の手法について述べる．4 章では実験による提案手法の評価を行う．最後に 5 章でまとめと今後の課題を述べる．

2 自律ディスクの概要

まず自律ディスクの概要について説明する．図 1 に標準的な自律ディスクのクラスタの例を示す．自律ディスクはネットワーク環境でクラスタを構成することを前提としている．ホスト (クライアント) はデータにアクセスするためにクラスタ内の任意のノードに要求を発する．標準的な構成ではクラスタ内の各ノードは，プライマリディレクトリと他ノードのバックアップを行うバックアップディレクトリを持つ. データに対するアクセスは分散ディレクトリをトラバースすることにより，リクエストを適切なノードに転送することにより行われる．このような前提のもとで，自律ディスクはデータ分散，ホストからの均質的なアクセス，同時実行制御，偏り制御，耐故障性，異種性といった性質を持つ [1].

3 自律ディスクの階層化

磁気ディスクより高速なメディアを用いたストレージデバイスとしては半導体ディスクが存在する．しかし半導体ディスクはアクセス速度が高速であるが，容量あたりの単価が非常に高く，ストレージシステム全てを半導体ディスクで構成するのは現実的ではない．そこで現在の一般的なストレージシステムでは，磁気ディスクを用いたストレージシステムのキャッシュとして半導体ディスクが用いられている．この手法は自律ディスクに対しても有効であると考えられるので，磁気ディスクを用いた自律ディスククラスタに対して，半導体ディスクを用いてキャッシュシステムを構成することを考える．そのような階層化ストレージを構成する上で，ストレージは以下の条件を満たすものとする．

- ストレージシステム全体として既存の自律ディスクと同じ機能を提供する．つまり，ノードの追加・削除がオンラインで可能であり，管理コストが増加しない.
- 半導体ディスクは揮発性メモリを利用するものとする．つまり，電源断などの障害が発生した場合には半導体ディスク内のデータが失われて

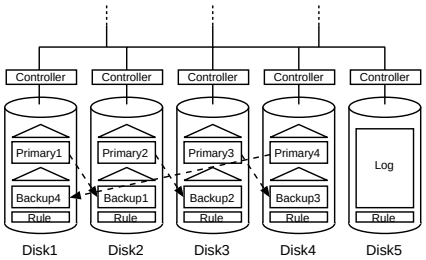

図 1: 自律ディスク

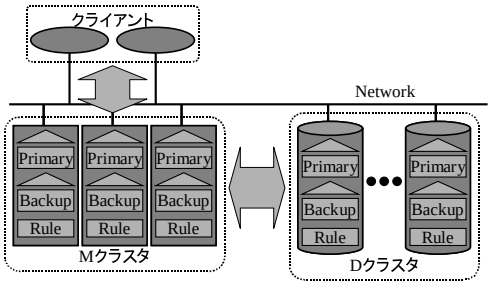

図 2: 階層化アーキテクチャ

しまう可能性がある．このため半導体ディスク内のデータは冗長化されなければならない．

以上の条件のもとで，磁気ディスクのみを用いて構成された自律ディスククラスタと，半導体ディスクのみを用いて構成された自律ディスククラスタを考える．以下，磁気ディスクのみを用いて構成されたクラスタを D クラスタ (Disk-cluster)，半導体ディスクのみを用いて構成されたクラスタを M クラスタ (Memory-cluster) と呼ぶことにする．D クラスタと M クラスタは同一のネットワーク上に存在し，M クラスタがクライアントからの要求を受ける．D クラスタは直接クライアントとの通信は行わない．このアーキテクチャの概要を図 2 に示す．D クラスタ，M クラスタともに同一の自律ディスクソフトウェアを用い，異なるのはハードウェア構成とルールのみである．このような構成にすることにより，以下に述べる利点が得られる．D クラスタや M クラスタの構成変更をそれぞれ独立に行うことができる．また，M クラスタではキャッシュの配置管理が自律ディスクの持つ分散ディレクトリによって行われる．これによりキャッシュが保持されるべきノードが一意に決まるので，分散キャッシュで問題になるキャッシュの一貫性問題を避けることができる．

3.1 キャッシュ置換アルゴリズム

キャッシュシステムの構成には，キャッシュ容量に対してエントリ数が溢れた際に，どのエントリをキャッシュから追い出すかを決定するための置換アルゴリズムが重要である．エントリにアクセスした際の記録や追い出すべきエントリを決定する際のオーバーヘッドが小さいものが必要である．また追い出すべきエントリとしては，頻繁にアクセスされるエントリではなく，アクセス頻度が低いエントリを選ぶべきである．現在のマイクロプロセッサ内部では LRU(Least Recently Used) アルゴリズムがキャッシュ制御アルゴリズムとして広く用いられている．一方，自律ディスクでは分散ディレクトリとして Fat-Btree などの分散 Btree を利用する．そこで本稿では Btree の葉ノードにおいて効率的に LRU アルゴリズムを利用する手法を提案する．ここで，Btree のインデックスノード (中間ノード) は葉ノードを追い出してからでないと追い出すことができない．また，葉ノード 1 つの容量がインデックスノード 1 つの容量よりも大きい場合を考える．よってインデックスノードの使用容量は無視し，葉ノードの使用容量のみを考慮する．

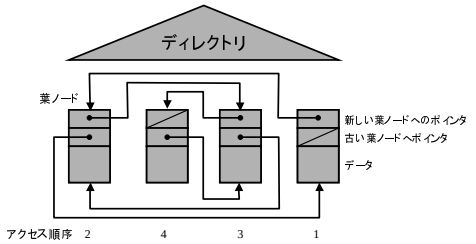

図 3: 葉ノードによる LRU リストの構成

M クラスタ内で追い出すべきエントリを選ぶ際には，M クラスタ全体の中で最も使われていないエントリを選ぶのがキャッシュヒット率の観点からは望ましい．しかしそれでは追い出し要求が発生する度に M クラスタ全体に問い合わせを行う必要があり，スケーラビリティが問題となる．そこでクラスタの各ノード内でのみ LRU アルゴリズムを行い，追い出し要求に対処するものとする．ノード間にキャッシュ利用率の偏りが発生した場合は，自律ディスクの持つ負荷均衡化機構により偏りを除去する．葉ノードは図 3 のように最近アクセスされた順に LRU リストと呼ばれる双方向リンクリストを構築する．このために葉ノードは 1 つ新しくアクセスされた葉ノードへのポインタと，1 つ古くアクセスされた葉ノードへのポインタを持つ．Btree へ新たに葉ノードが挿入されると，LRU リストへは先頭に挿入される．Btree から葉ノードが削除された場合には，LRU リストからも削除される．読み出し，更新などの葉ノードへの

その他のアクセスの場合は，その葉ノードはいったん LRU リストから削除され，LRU リストの先頭に挿入される．

この LRU リストの維持にかかるコストは，以下の理由から一定時間で行うことができる．LRU リストからの削除は双方向リンクリストであるので削除ノードの両側のノードのポインタを書き換えるだけである．LRU リストへの挿入は，先頭への挿入しか発生せず，これも一定時間で行うことができる．これより，葉ノードにアクセスが発生した場合に LRU リストを維持するための処理は葉ノード数に対して $O(1)$ で行うことができる．どの葉ノードを追い出すかを決定する場合は，LRU リストの最後のノードを選ぶ．この操作も葉ノード数に対して $O(1)$ で行うことができる．

よって，この葉ノードに対する LRU アルゴリズムを用いることによって，キャッシュ管理にかかるコストを非常に小さく抑えることができる．

3.2 挿入・更新プロトコル

次に，自律ディスクの階層化構成における挿入・更新プロトコルの設計を行う．プロトコルの設計は以下の方針で行う．まず，単一故障に耐えられるように設計する．このためには，データは 2ヶ所以上に冗長化して保持されなければならない．また，半導体ディスクを揮発性であると仮定しているので，データを永続化するためには必ず磁気ディスクへ書き込まなければならない．

以上の条件に従って，3 つの挿入・更新プロトコルを提案する．ただし本来の自律ディスクはログを用いた非同期バックアップを行うことで，プライマリとバックアップの同期更新を避け Insert 処理のコストを抑えているが，ここでは簡単のために同期更新を行う場合のプロトコルについて説明する．

3.2.1 Write-Through-Sync プロトコル

この手法では，M クラスタはプライマリディレクトリのみを持ち，バックアップディレクトリを持たない．Insert 時には M クラスタのプライマリ，D クラスタのプライマリとバックアップに同期的に書き込みを行い，クライアントに完了通知を返す．この

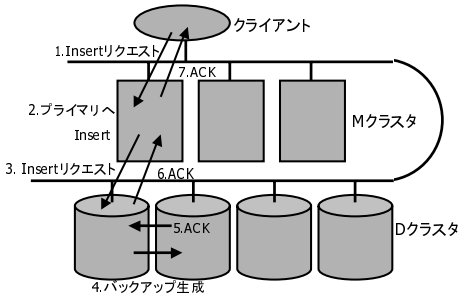

図 4: Write-Through-Sync プロトコル

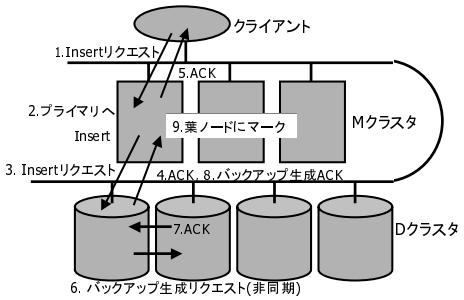

図 5: Write-Through-Async プロトコル

プロトコルにおける処理の流れを図 4 に示す．

1. ホストが M クラスタの任意のノードに対して Insert リクエストを送る．
2. M クラスタのプライマリディレクトリにリクエストデータを書き込む．その後 M クラスタは D クラスタに同様の内容の Insert リクエストを送る．
3. D クラスタはプライマリディレクトリとバックアップディレクトリに同期的に Insert 処理を行

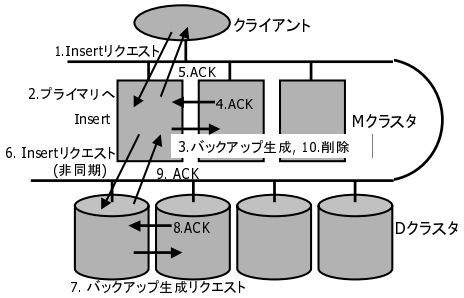

図 6: Delayed-Write プロトコル

い，M クラスタへ完了通知を返す．

4. M クラスタはクライアントへ完了通知を返す．

キャッシュからエントリを追い出す際には，単に追い出すべきエントリをプライマリから削除するだけでよい．このプロトコルは最も単純な手法であり実現は容易であるが，3 箇所に同期的に書き込みを行うためリクエスト処理にかかる時間が大きくなる．

3.2.2 Write-Through-Async プロトコル

この手法でも，M クラスタはプライマリディレクトリのみを持ち，バックアップディレクトリを持たない．Insert 時には M クラスタのプライマリと D クラスタのプライマリへ同期的に書き込みを行い，クライアントに完了通知を返す．その後 D クラスタのプライマリから非同期にバックアップを生成する．このプロトコルにおける処理の流れを図 5 に示す．

1. ホストが M クラスタの任意のノードに対して Insert リクエストを送る．
2. M クラスタのプライマリディレクトリにリクエストデータを書き込む．その後 M クラスタは D クラスタに同様の内容の Insert リクエストを送る．
3. D クラスタはプライマリディレクトリ Insert 処理を行い，M クラスタへ完了通知を返す．
4. M クラスタはクライアントへ完了通知を返す．
5. その後 D クラスタは任意のタイミングでバックアップディレクトリへ Insert 処理を行い，M クラスタへバックアップ生成完了通知を返す．
6. M クラスタはバックアップ生成完了通知を受け取ると，対応する葉ノードにマークをつける．

この手法では Write-Through-Sync プロトコルに比べ，D クラスタでのバックアップ生成がクライアントからのリクエストとは非同期に行われるため，レスポンスタイムが短縮される．

しかしこの手法では M クラスタにおいてキャッシュの追い出し処理が発生した場合，追い出されるエントリが D クラスタ内にプライマリ，バックアップとも存在しなければならない．上記の最後のステップ 6 によって既に葉ノードがマークされている場合は，そのまま追い出すことができる．マークされていない場合は D クラスタにバックアップを生成してから M クラスタから追い出す必要がある．

3.2.3 Delayed-Write プロトコル

この手法では，M クラスタはプライマリディレクトリとバックアップディレクトリを持つ．Insert 時には M クラスタのプライマリとバックアップに同期的に書き込みを行い，クライアントに完了通知を返す流れである．このプロトコルにおける処理の流れを図 6 に示す．

1. ホストが M クラスタの任意のノードに対して Insert リクエストを送る．
2. M クラスタのプライマリディレクトリとバックアップディレクトリに同期的にリクエストデータを書き込む．その後 M クラスタはクライアントへ完了通知を返す．
3. M クラスタは任意のタイミングで D クラスタに同様の内容の Insert リクエストを送る．
4. D クラスタはプライマリディレクトリとバックアップディレクトリで同期的に Insert 処理を行い，M クラスタへ完了通知を返す．
5. D クラスタは M クラスタから完了通知を受け取ると，対応する葉ノードにマークをつける．その後，バックアップディレクトリからエントリを削除する．

この手法では Insert 時のプライマリとバックアップをともに M クラスタ内に生成することにより，レスポンスタイムを短縮する．M クラスタにおいてキャッシュの追い出し処理が発生した場合，追い出されるエントリが D クラスタ内にプライマリ，バックアップともに存在しなければならない．これを保証するために，上記の最後のステップ 5 によって既に葉ノードがマークされていない場合は D クラスタにプライマリとバックアップを生成した後に M クラスタから追い出す必要がある．マークされている場合は，そのまま追い出すことができる．

3.3 読み出し時の動作

読み出し (Search) 時には以下のプロトコルで処理を行う.

1. ホストが M クラスタの任意のノードに対して Search リクエストを送る.
2. M クラスタのプライマリディレクトリをトラバースする. リクエストされたエントリが M クラスタ内に存在するならば, それを読み出してクライアントへ返し処理終了.
3. M クラスタ内にリクエストされたエントリが存在しない場合は, D クラスタへリクエストを転送する.
4. D クラスタはディレクトリをトラバースし, リクエストされたエントリを読み出して M クラスタへ返す.
5. M クラスタは D 受信したデータをクライアントへ返す. D クラスタにリクエストされたエントリが存在したならば, 同時に M クラスタのプライマリディレクトリに挿入する.

この手法では, リクエストされたストリームがストレージ内に存在しない場合, 必ず D クラスタまでリクエストが伝播し効率が悪い. しかし通常のストレージでは存在しないストリームに対して Search リクエストが行われることはあまりないため, 問題ないと考えられる.

4 性能評価

ここでは前章で述べた階層化手法の実験による評価を行う. まず前述の挿入・更新プロトコルとキャッシュ置換アルゴリズムを, 我々が現在 PC 上に Java を用いて実装を行っている試作システム上に実装を行った. その上でリクエスト処理にかかる時間を計測し, 階層化を行わない場合, 階層化を行った場合の各プロトコルの評価を行う.

4.1 測定手法

性能測定に用いた実験環境を表 1,2,3 に示す. 半導体ディスクは Fibre Channel で接続され, OS からは通常の SCSI ディスクデバイスとして扱われる. このため, M クラスタと D クラスタでは同じソフトウェアを用いることができ, ルールのみを切り換えて実験を行う. クライアントには D クラスタのノードと同じ構成の PC を 1 台用いる.

表 1: M, D クラスタ共通の構成

ネットワーク	1000BASE-SX, Switching Hub
Java 環境	Sun JDK 1.4.1

表 2: D クラスタの構成

ノード数	6 台
CPU	Intel Pentium III 933 MHz
メモリ	PC133 SDRAM 256MB
HDD	Seagate Barracuda IV 20.4GB 7200rpm
OS	Linux 2.2.17

表 3: M クラスタの構成

ノード数	3 台
CPU	Intel Xeon 2.40GHz, Hyper Threading
メモリ	PC2100 DDR SDRAM 1024MB
HDD	Seagate Cheetah X15 36LP 36.7GB 15000rpm
半導体ディスク	BiTMICRO E-Disk 4GB Fibre-Cannel
FC-HBA	QLogic SANblade QLA2310
OS	Linux 2.4.20 SMP

この環境に対して, 以下の手法で性能測定を行う. Insert, Search リクエストをそれぞれ一定回数発行し, 処理完了までの所要時間を計測する. リクエストは 6000 個のキー集合を用い, 連続して 30000 回行う. 格納されるストリームのキーは 16 バイトのランダムな文字列であり, ストリームのサイズは 1K バイトとする. キーの偏りによる性能の変化を評価するために, ストリーム ID の利用頻度を Zipf 分布とし, 偏りの度合いを表すパラメータ θ を 0.0〜1.0 に変化させて測定を行う.

4.2 予備実験

各操作の性能測定を行う前に, M クラスタと D クラスタのそれぞれの性能を測定する. それぞれのクラスタで独立に従来の自律ディスククラスタを構成し, 偏りのない状態で Insert, Search 操作の性能を計測した結果が表 4 である. この結果より, この実験システムにおいて M クラスタは D クラスタに比べ, Insert で 47%, Search で 55%程度高速であることがわかる.

表 4: M, D クラスタの性能測定

	Insert	Search
M クラスタ	123370 [ms]	52190 [ms]
D クラスタ	181513 [ms]	80681 [ms]

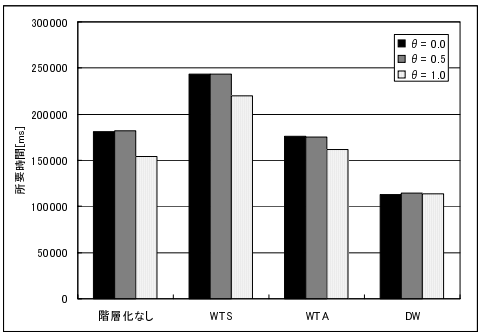

図 7: Insert 操作の実行時間

4.3 Insert 性能の測定

前述の手法によって Insert 操作の性能を測定した結果を図 7 に示す．測定の際には M クラスタにおける最大キャッシュエントリ数を 200 としている．

Delayed-Write プロトコルを用いた場合は階層化を行わない構成に比べ，偏りがない場合で 60%程度高速化されている．これは Delayed-Write プロトコルがプライマリとバックアップをともに M クラスタ内に Insert することにより，Insert リクエストのレスポンスタイムが短縮されたことがわかる．一方，Write-Through-Sync プロトコルでは 3 つのノードに書き込みを行ってからクライアントへレスポンスを返すため，階層化しない場合に比べレスポンスタイムが悪化していることがわかる．Write-Through-Async プロトコルではプライマリを M クラスタに書き込むため，理論上は階層化なしの場合に比べレスポンスタイムが短縮される．しかし，D クラスタ内での非同期バックアップ生成のコストや，M クラスタにおけるキャッシュ管理のオーバーヘッドによって，階層化なしの場合と同等の速度になっている．

Insert リクエストにおいて各キーの発生頻度の偏りが大きい場合は，Delayed-Write プロトコルの場合を除いて，速度が向上している．これは各キーの発生頻度に偏りがある場合では，各ノード上のディスクキャッシュにヒットする頻度が向上するためであると考えられる．

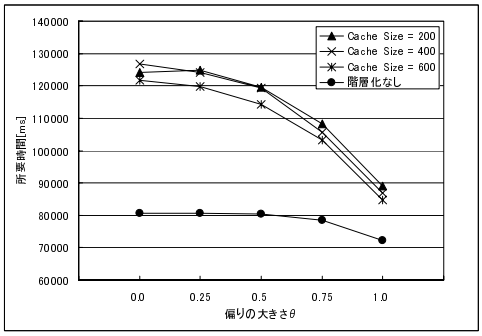

図 8: アクセス偏りと Search 操作の実行時間

4.4 Search 性能の測定

前述の手法により Search 操作の性能を，M クラスタにおける最大キャッシュエントリ数を変化させながら測定した結果を図 8 に示す．

各キーの発生頻度に偏りが大きい場合では，M クラスタがキャッシュとして効率的にはたらくため，キャッシュ容量を増やすにつれて速度が向上していることがわかる．階層化を用いない手法についても，偏りが大きくなるにつれて各ノード上のディスクキャッシュにヒットする頻度が向上するために速度が向上している．しかし，より偏りが大きい場合においては階層化手法と逆転する可能性がある．

一方，偏りがない場合では同一のストリームが高頻度でアクセスされることがないため，キャッシュ容量が増えるに従って速度が低下してしまっている．これは，キャッシュ容量が増えるにつれて M クラスタにおけるディレクトリのトラバースコストが増加するためである．さらにキャッシュ容量を増加させるとキャッシュヒット率の向上により速度が向上している．

4.5 考察

Insert 操作の性能については，階層化手法における Delayed-Write プロトコルが階層化しない場合を大きく上回り，階層化手法の有効性を示した．

一方，Search 操作の性能についてはリクエストに偏りがない場合では階層化手法は階層化を行わない

手法に大きく離され，偏りがある場合についても差は小さくなるが階層化を行わない手法の方が高速であるという結果になった．ここで階層化手法における Search 操作の性能が階層化を行わない場合に比べて高速になる条件を考える．M クラスタにおけるキャッシュヒット率を r，M クラスタでの Search 操作の所要時間を T_M，D クラスタでの Search 操作の所要時間を T_D とすると，Search 操作に対する平均所要時間は $rT_M + (1-r)(T_M + T_D) = T_M + (1-r)T_D$ である．これが T_D よりも小さくなるためには，$r > \frac{T_M}{T_D}$ である必要がある．今回実験を行ったシステムでは $\frac{T_M}{T_D} \simeq 0.65$ でありこの条件を達成するのは難しいが，$\frac{T_M}{T_D}$ がより小さいシステム，つまり M クラスタと D クラスタの速度差がより大きいシステムではこの条件を達成することは容易であるので，階層化により高速な Search 操作を行うことができる．

5 まとめと今後の課題

本稿では，半導体ディスクを用いてキャッシュ用の自律ディスククラスタを構成することによって，自律ディスクの特性を損なうことなく自律ディスクを用いたストレージを高速化する手法を提案した．自律ディスクの持つアベイラビリティを損なわない効率的な挿入・更新プロトコルや，Btree 上で効率的に LRU キャッシュ置換アルゴリズム実現する手法を提案した．

また，それらのプロトコルやアルゴリズムの有効性を実験システム上に実装し，性能測定によりその効果を確認した．実験では挿入・更新操作における Delayed-Write プロトコルは階層化を行わない場合に比べ，60%高速であるという結果が得られた．読み出し操作については実験システムの構成上の理由で階層化を行わない場合よりも性能が低下したが，アクセスに偏りがある場合についてはその有効性を確認することができた．

本稿では階層化自律ディスククラスタのアーキテクチャを示したが，M クラスタと D クラスタの性能や台数比よっては M クラスタがボトルネックになることが考えられる．どのような場合に M クラスタがボトルネックになりえるのかの理論的な考察が必要である．

また，M クラスタ内には全てのストリームが存在しないので，分散ディレクトリに対するレンジクエリは M クラスタ内では処理することができない．そのため，レンジクエリの効率的な処理手法の検討も今後の課題として挙げられる．

今回の性能評価は比較的小さなストリームを用いて行ったが，ストリームサイズが大きい場合にも本稿の手法は有効であり，スループットを大きく向上させることができると考えられる．我々は自律ディスクにおいて大きなストリームを効率的に扱う手法として間接ディレクトリ方式 [5] を提案しているが，これと組み合わせた場合の性能評価が必要である．

本稿で提案した階層化手法の最終的な目標は，半導体ディスク・磁気ディスク・テープドライブなどを用いて 3 階層以上のストレージを構成することである．

謝辞

本研究の一部は，文部科学省科学研究費補助金基盤研究 (12680333, 13224036, 14019035) および情報ストレージ研究推進機構 (SRC) の助成により行なわれた．

参考文献

[1] Haruo Yokota. Autonomous Disks for Advanced Database Applications. In *Proc. of International Symposium on Database Applications in Non-Traditional Environments (DANTE'99)*, pages 441–448, Nov. 1999.

[2] 安部洋平 , 横田治夫. Java による耐故障ネットワークディスクのルール処理の実装. In **第 *11* 回データ工学ワークショップ論文集**, *DEWS2000 3B–2*. 電子情報通信学会データ工学研究専門委員会, 2000.

[3] 阿部 亮太 , 横田 治夫. 自律ディスクにおける故障時の動作とそのルール処理の実装. In **第 *12* 回データ工学ワークショップ論文集**, *DEWS2001 2B–3*. 電子情報通信学会データ工学研究専門委員会, 2001.

[4] 渡邊 明嗣 , 花井 知広 , 山口 宗慶 , 横田 治夫. 自律ディスクを用いたマルチメディアコンテンツサーバ. Number DE2002-86, DC2002-22, 2002.

[5] 花井知広 , 横田治夫. 自律ディスクを用いた Web サーバにおける負荷偏りの影響. In **情処学会研究会報告, データベースシステム** *DBS-128-29*. 情報処理学会, 2002.