自律ディスク上の分散ディレクトリの負荷均衡機構を用いたクラスタ再構築

伊藤大輔† 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
daisuke@de.cs.titech.ac.jp, yokota@cs.titech.ac.jp

あらまし 大規模ストレージシステムの管理を効率的に行う一つのアプローチとして, 我々は自律ディスクを提案している. 自律ディスクはネットワークに直接接続されてストレージクラスタを構成し, 集中コントローラを持つことなくクラスタ単位でアクセスを効率よく処理する. さらに負荷分散や冗長化といった様々な管理もクラスタ内で処理することで高いリライアビリティとスケーラビリティをもたらす. これらの管理や大規模クラスタ内の論理ユニットの拡張といった操作はクラスタの構成変更を伴う. よって, アベイラビリティを高めるためにはオンラインでのクラスタ再構築が必要不可欠である. 本稿では負荷均衡機構をもった分散ディレクトリを実装した模擬自律ディスク上にクラスタ再構築プロトコルを実装する. また, 再構築の所要時間や再構築中のクラスタの性能を測定し, 評価する.

キーワード 自律ディスク, ネットワーク接続ディスク, NAS, 分散ディレクトリ, クラスタ, 再構築, スキューハンドリング

Online Cluster Reconfiguration on Autonomous Disks Using Skew Handling Mechanism on Distributed Directory

Daisuke Ito† and Haruo Yokota‡
† Department of Computer Science, Tokyo Institute of Technology
‡ Global Scientific Information and Computing Center, Tokyo Institute of Technology
daisuke@de.cs.titech.ac.jp, yokota@cs.titech.ac.jp

Abstract We have been proposed autonomous disks to enable distributed control in the storage-centric systems. Autonomous disks configure a cluster in a network, and accepts operations using a distributed directories without any centralized server. Autonomous disks also manages load distribution, and has a capability of reconfiguring the cluster automatically. In this paper, we focus on the cluster reconfiguration utilizing the skew handling mechanism. The skew handling mechanism is provided by distributed directories. We also report preliminary results of our experimental system.

Keywords autonomous disks, network disks, NAS, distributed directory, cluster, reconfiguration, skew handling

1 はじめに

我々がコンピュータを通して扱う情報量は急増し続けており, それに伴いコンピュータ上の情報はより重要なものとなる. その結果, 情報の記憶装置であるストレージの管理コストの急騰という新たな問題が生じた. この問題の解決策の一つとして, ストレージをシステムの中心に据える構成が注目されている. このような構成はストレージセントリックと呼ばれ, 高いアクセス透過性をもたらし, またサーバボトルネックを削減する. ストレージセントリックシステムの構成要素として NAS (Network Attached Storage) や SAN (Storage Area Network) が提案され, すでに製品化も行われている.

しかし, ストレージセントリックにするだけで全ての問題を解決できるわけではない. 未解決な問題のうち, 最も重要なものはストレージ間のデータ配置の問題である. データ配置には通常それ専用の独立したサーバを用いる. これは不特定多数のクライアントからのアクセスを受ける際の管理コストを下げる意味で有効な手段である. しかしシステムの規模の増大にともない, データ配置用サーバが新たなボトルネックとなることは避けられず, また, 特定のサーバへの依存は Single Point Of Failure (SPOF) 問題を招く. もちろん分散ディレクトリを用いることで, データ配置用サーバを介することなくデータへアクセスすることは可能である. しかし, データ配置用サーバを用いない場合には, 新たにストレージ間のデータに偏りが生じた際の再配置問題が生じる.

これらの問題は, ストレージがパッシブデバイスとして扱われていることに起因する. パッシブなストレージは, ネットワーク越しにホストからの命令を受けるまで動作を行えない. そのために, 中央集権的な管理用のサーバへの依存を断ち切れない.

一方で, ディスク上の高性能なディスクコントローラを用いた高機能ディスクの研究が活発に行われている [1, 2, 3]. ディスクコントローラ上の演算能力の一部をアプリケーションの実行に用いることでストレージ上のデータを最もデータに近い場所で処理することが可能となり, データマイニングなどのアプリケーションに用いた際に高い性能を発揮することが確認されている.

我々はこの演算能力をストレージの自律的な管理に用いるアプローチが有効であると考えている. ネットワーク上のストレージがそれぞれ自律分散的な管理を行うことで, サーバに依存しない管理が可能となる. 我々はこのような背景の基で自律ディスク [4] を提案している.

自律ディスクはストレージクラスタを構成し, その各ノードがアクティブに動作することでネットワークの通信コストとホストの負荷を下げる. また, クラスタ内での冗長構成とトランザクション的機能の導入により高いリライアビリティを実現できる. さらに, クラスタの構成をオンラインで動的に変更することで高いスケーラビリティを持つ. これらはストレージシステムの管理を容易にし, 管理コスト問題の有効な解決策となる.

我々はこれまでに自律ディスクの様々な機能の提案や, その機能を検証するため PC を用いた模擬自律ディスクの試作を行ってきた [5, 6, 7]. また, 既存の NAS をベースとしたシステムとの比較を行い, その優位性の検証も行った [8].

本稿では先に提案したクラスタ構成のオンライン再構築プロトコルを模擬自律ディスク上に実装し, 自律ディスクが高いスケーラビリティを有することを検証する. まず第 2 章で自律ディスクについて説明し, 特徴を整理する. 次に第 3 章ではクラスタ構成の再構築と負荷の偏り制御の関係について説明する. 第 4 章では模擬自律ディスク上での実験と考察を行い, 最終章で本稿のまとめと今後の課題を述べる.

2 自律ディスク

自律ディスクの構成例を図 1 に示す. ここでは本稿の扱う事柄と関連した重要な特性を 6 つあげる. 詳細は文献 [4] を参照されたい.

ネットワーク上でストレージクラスタを構成 クライアントからのアクセスをクラスタとして処理する. そのため, クライアントから見たクラスタの受け口が多く, 高いスループットを有する.

高機能な分散ディレクトリを採用 各ディスクが分散ディレクトリを持つことで, クラスタへの同時アクセスの平行処理が可能となる. そのため, スループット / レスポンスタイムの両面で有利となる. また, クライアントはクラスタ内の任意のディスクにアクセスするだけで実際にデータが格納されているディスクから結果が返ってくる. さらに, 高機能

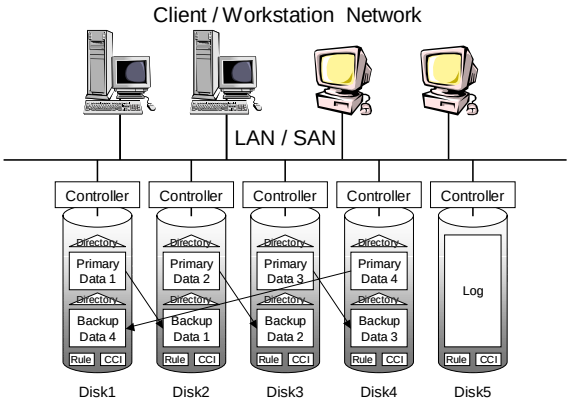

図 1: 5 台の自律ディスクからなるクラスタの構成例

な分散ディレクトリを採用することで, ディレクトリ自身で負荷均衡制御を行うことが可能である.

カスタマイズ可能なコマンド改装　内部コマンドと外部コマンドを持ち, これらはルールでマッピングされる. ルールの記述には Event-Condition-Action (ECA) ルールを使用する. ユーザの管理戦略に合わせてルールをカスタマイズ可能とする.

ストリームインターフェース　従来のブロックアドレスベースのインターフェースとは異なるストリームインターフェースを用いる. この結果, NSIC によって提案された OBSD [9] のようにデータオブジェクトの論理的な取り扱いが可能となる.

ログを介した非同期プライマリバックアップ　高いリライアビリティを実現するためにはクラスタ内にバックアップを保持する必要があるが, バックアップ操作は更新処理のレスポンスを悪化させる. そのため, Write-Ahead-Log (WAL) プロトコルに従い非同期バックアップを行う事でレスポンスの悪化を避ける. バックアップは論理バックアップを想定している. 図 1 の例は, 独立したログディスクを用いた 4 台のデータディスクと 1 台のログディスクからなるクラスタの構成例である.

クラスタ構成情報 (Cluster Configuration Information : CCI)　クラスタ内の通信回数を削減するために, 全クラスタメンバが CCI をローカルに保持する. 全メンバは等しい内容の CCI を保持する. 現状では CCI として以下の情報を記述している.

- クラスタメンバのエンドポイント
- クラスタメンバの属性 (データディスク / ログディスク)
- プライマリ – バックアップ間のマップ

3　クラスタ再構築

3.1　ストレージクラスタ構成の変更

大規模なストレージシステムの運用においては, 多数のストレージリソースを複数のストレージクラスタに分割して, それぞれを異なる用途に割り当てることが考えられる. このとき, 要求に応じてストレージクラスタとしてのバンド幅や記憶容量を変更することで, ストレージリソースの有効活用につながる[1]. また, システムとして規模の拡張も頻繁に起こりうる. さらに, アベイラビリティ確保のためには, 故障が生じた際の早急な再構成も重要である. これらはストレージクラスタ構成の動的な変更を必要とする.

クラスタ構成の変更は, CCI の変更によって行われる. 同時に, クラスタ構成の変更自体もルールで CCI をカスタマイズすることで, ユーザの戦略にあわせた運用が可能となる. CCI を全ディスクで共有しているため, クラスタ構成の変更の際には全ディスクで CCI の変更を同期させる必要が生じる. このような背景の元で, 我々はクラスタの構成変更時に CCI を安全に更新するためにクラスタ構成変更プロトコルを提案している [6].

3.2　クラスタ再構築プロトコル

クラスタ再構築プロトコルでは構成変更の要因を戦略変更とディスク故障に分けて考える. 戦略変更はさらにデータディスクの追加 / 削除, ログディスクの追加 / 削除, バックアップの追加 / 削除の 6 つに分けられ, ディスクの故障はデータディスクの故障, ログディスクの故障の 2 つに分けられる. これら 8 種類のクラスタの状態変化にそれぞれ 1 つずつプロトコルが対応する.

CCI の更新の同期を保証するためには, 更新を司るコーディネータが必要である. 自律ディスクではクラスタ内に特別なノードを想定しないため, クラ

[1] SAN ではこのような構成をバーチャリゼーションと呼んでいる.

スタ内の任意のノードが動的にコーディネータとなる. これをダイナミックコーディネータと呼ぶ. ダイナミックコーディネータの制御の元で, 2 フェーズコミット (2PC) に類似した多重フェーズ構成同期を行う.

以下にデータディスク追加プロトコルを示す. ここではデータディスク D_a をクラスタに追加することを想定している.

Step 1: D_a はクラスタ内の任意のディスク D_c をコーディネータに任命する.

Step 2: D_c は D_a に許可 / 不許可を返す. 許可の場合は次のステップへ進む. 不許可の場合は D_a は適当な時間待機してから Step 1 からやり直す.

Step 3: D_c はクラスタ内の全ディスクに対し, CCI 変更準備メッセージおよび D_a の ID を送る.

Step 4: クラスタ内の全ディスクは D_c に準備完了 / 拒否を返す. 全ディスクが準備完了の場合は次のステップへ進む. 一台でも拒否の場合は D_c はクラスタ内の全ディスクに処理の取り消しを発し, D_a に不許可を返し, D_a は適当な時間待機してから Step 1 からやり直す.

Step 5: D_c はクラスタ内の全ディスクに対し, CCI 変更メッセージを送る.

Step 6: クラスタ内の全ディスクは D_c に成功 / 失敗を返す. 全ディスクが成功の場合は次のステップへ進む. 一台でも失敗の場合は D_c はクラスタ内の全ディスクに処理の取り消しを発し, D_a に不許可を返し, D_a は適当な時間待機してから Step 1 からやり直す.

Step 7: D_c は D_a に対し, 新しい CCI を送り, クラスタへの参加を許可する.

プロトコル中の Step. 3, 5 が多重フェーズ構成同期であり, データディスクの追加の場合は 2PC と同じく 2 フェーズからなる. データディスクの追加以外のプロトコルでは, コーディネータの任命や CCI の変更のタイミング等が異なるが, 同様の過程を踏む. 詳細は文献 [6] を参照されたい.

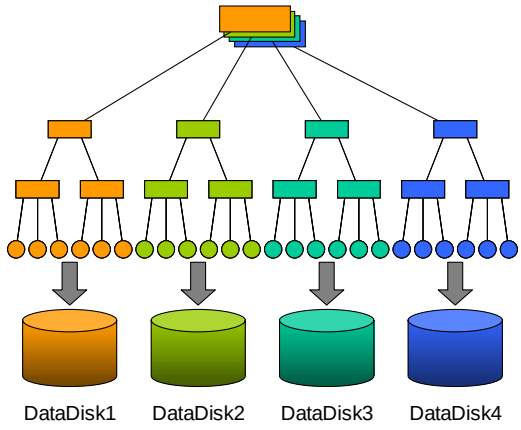

図 2: 値域分割された BTree の例

3.3 自律ディスクの負荷均衡機構

クラスタの構成変更の際には, 負荷を均衡化させるようにデータを再配置しないとその後の性能低下につながる. しかし, 3.2 のデータディスク追加プロトコルはデータの再配置を含んでいない. その理由は自律ディスクが内包する負荷均衡機構を用いているためである.

自律ディスクは更新頻度の高い環境で用いられることを想定して, 負荷均衡機構を持った分散ディレクトリを採用することとしている. クラスタ再構築直後の状態を負荷の偏りの極端に大きい場合とみなして扱うことで, クラスタ再構築とデータの再配置を分割して扱える. 結果として複数台のディスクを同時に追加 (および削除) する際にも, 1 台のディスクを追加 (および削除) するプロトコルを繰り返し用いることで対応できる.

このような分散ディレクトリとして, 現在は値域分割した BTree を用いている. 図 2 に 4 台のデータディスクからなるディレクトリ構造の例を示す. このような構成の BTree では, 論理的に隣接したディスク間で葉ページ毎のデータ移動を行うことで負荷の分散が行える. 本稿で用いた負荷均衡アルゴリズムについては 4.2 に詳しく記す. また, 将来的にはより高性能な Fat-Btree [10] の採用を計画している.

4 模擬自律ディスク上の実験

本章では自律ディスクのクラスタ再構築の優位性を検証するために実験を行う. まずクラスタ再構

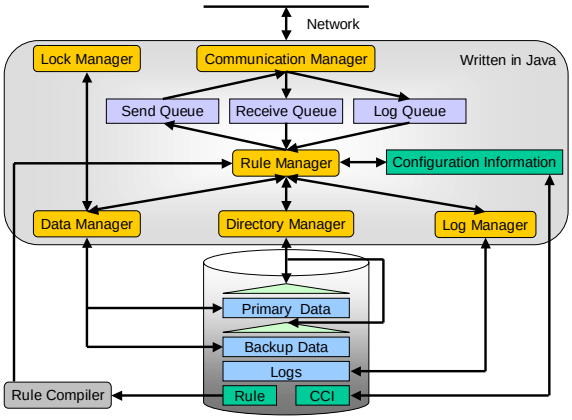

図 3: 模擬自律ディスクの構成

表 1: 実験環境の緒元

CPU	Intel Pentium3 933MHz
Chipset	Intel i815 (ATA100)
Memory	PC133 (CL=3) 256MB
HDD	Seagate ST320424A (7200rpm, Barracuda ATA II 100)
Network	AMD PCnet32 (100M)
OS	Linux 2.2.17, glibc–2.1.3
Java	IBM JRE1.3.0 (IBM build cx130–20010626 (JIT enabled: jitc))
Hub	100BaseT Switch

築の所要時間を実測し, 先に発表した見積もり [8] と比較検討する. 次にオンラインクラスタ再構築に着目し, クライアントからのアクセスのある状態でのクラスタ再構築の実験を行う.

4.1 模擬自律ディスク

我々は未だプログラム可能なプロセッサの搭載されたディスクを自由に使う事はできない. そのため PC を用いた模擬自律ディスクの実装および実験を進めている [7]. 模擬自律ディスクは Java を用いて記述されているにも関わらず, UNIX 上の NFS プロトコルをを用いたシステムと比較しうる性能を有している [8]. 図 3 に模擬自律ディスクのブロック図を, また表 1 に実験システムの緒元を示す. 各機能ブロックの詳細は文献 [7] を参照されたい.

4.2 負荷均衡機構の実装

分散ディレクトリにおける各 PE (ここではディスク) の負荷としてアクセス頻度等を考慮にいれた手法 [11, 12, 13] も検討されているが, 本稿では第一ステップとして各データディスク内のページ数を負荷として採用する. また, 3.3 で紹介した負荷均衡機構はファイルシステムを扱う内部コマンド (IS コマンド) として実装される [4]. ディスク間の通信を行う際には, 自律ディスクの作法に従い外部コマンド (EIS コマンド) を用いたストリーム通信を用いる.

本稿では文献 [10] で紹介されたアルゴリズムを参考に, 図 4 に示す `skew_handler` ルールと `migrate_node` ルールを用いた. この手法はトークン・バス方式のようにクラスタ内の全メンバの負荷リストを論理近傍に回していき, その平均に閾値を加えた値が自らの負荷よりも大きい場合に偏りが検出される. 偏りが検出された場合は, 左右の論理近傍ディスクのうち負荷の低い方に葉ページを移す. `skew_handler` ルールの引数となるストリームの実体はクラスタ内の全メンバの負荷のリストである. また, `migrate_node` ルールの第一引数となるストリームは, 葉ページを直列化したものである.

`skew_handler` ルール自体がディスクへのアクセスを伴うため, アクセス処理の妨げとなる. そのため, 16 行目にウエイトを入れてある. ウエイトの間隔は, 文献 [5] で採用したバックアップ間隔の動的調整と同様にルールで行うことも考えられる. なお, `skew_handler` では偏りを検出した際にはウエイトを置かない. これは主にデッドロック回避のための処置であるが, 偏り検出時にはウエイトを挟まずに論理近傍ディスクの偏り検出が始まるため, 結果として再構築所要時間が短くなる.

4.3 アクセスのない状態での再構築所要時間

ここではクライアントからのアクセスがない状態の n 台のデータディスクからなるクラスタに 1 台のデータディスクを追加する際の, 再構築所用時間を測定する. 実験環境を以下に示す.

1. n は 2 から 5 まで変化させる.
2. `skew_handler` のウエイト値は 50ms または 500ms とする.
3. あらかじめ 400 タプルのデータをクラスタ内に均等に格納しておく.

```
Rule_Detect_Skew_&_Invoke_Migration:
 1: when skew_handler(Stream);
 2: update_load(Stream, MyID);
 3: if   read_list(Stream, MyID) > average_load(Stream) + Threshold;
 4: then if (read_list(Stream, LeftNeighbourID) > read_list(Stream, RightNeighbourID))
 5:              || MyID == TheLeftMostID;
 6:      then migrate_nodes(TheRightmostDataPages, RightNeighbourID),
 7:           update_load(Stream, MyID),
 8:           if   MyID == TheRightMostID;
 9:           then send(TheLeftMostID, skew_handler(Stream));
10:           else send(RightNeighbourID, skew_handler(Stream)),
11:           break;
11:      else migrate_nodes(TheLeftmostDataPages, LeftNeighbourID),
12:           update_load(Stream, MyID),
13:           if   MyID == TheRightMostID;
14:           then send(TheLeftMostID, skew_handler(Stream));
15:           else send(RightNeighbourID, skew_handler(Stream)),
16:           break;
16: wait(interval);
17: if   MyID == TheRightMostID;
18: then send(TheLeftMostID, skew_handler(Stream));
19: else send(RightNeighbourID, skew_handler(Stream));


Rule_Migrate_Nodes:
 1: when migrate_node(Stream, ToID);
 2: delete_local(Stream),
 3: send(ToID, fouce_insert(MyID, Stream));
 4: if   ToID.contains(parent(Stream));
 5: then parent(Stream).update_entry(Stream);
 6: else migrate_node(parent(Stream), ToID);
```

図 4: skew_handler ルールと migrate_node ルール

表 2: アクセスのない状態でのクラスタの性能

rule	response time (ms)	throghput (Mbps)
Insert	7.52	19.7
Delete	6.49	—

4. ページサイズは 8KB, タプルサイズは 4KB とする.

5. 偏り検出の閾値は 5 とする. この値は本実験システムのディスク 1 台の最大ページ数の 1% である.

この環境の元での実験結果を図 5 に示す. また, 以下の考察の基礎データとして, 6 台のデータディスクからなるクラスタ上での Insert ルールのレスポンスタイムおよびスループット, Delete ルールのレスポンスタイムを表 2 に示す.

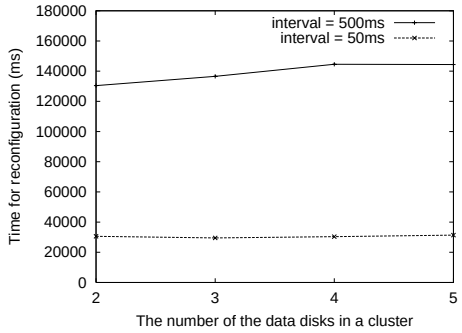

図 5: アクセスのない状態でのクラスタ再構築所要時間

考察 クラスタ内のデータの葉ページの総数を A_{leaf}, 追加前のディスクの台数を n 台として, i 台目から $i+1$ 台目のディスクへのデータ移動量 $M(i)$ は以下の式で表される.

$$M(i) = \frac{A_{\text{leaf}}}{n(n+1)} i \qquad (1)$$

よって, 再構築に伴う総データ移動量は次式で表される.

$$\sum_{i=1}^{n} M(i) = \frac{A_{\text{leaf}}}{2} \quad (2)$$

図 4 のルールを用いた場合には閾値が入るため, 完全に負荷が均一になるわけではない. そのため, この環境下ではデータディスクの追加に伴い約 200 ページ分のタプルが移動することになる.

我々の見積もり [8] では, クラスタ再構築の所要時間 T_{ADisk} は 1 ページの移動の所要時間 $T_{\text{ADisk, migrate}}$ とクラスタ内の総ページ数 A_{leaf} を用いて以下の式で表される.

$$T_{\text{ADisk}} = T_{\text{ADisk, migrate}} \times A_{\text{leaf}}/2 \quad (3)$$

本稿では図 4 に示したウエイトを置いたルールを用いたため, $T_{\text{ADisk, migrate}}$ は `Insert` ルール (タプルの挿入) の所要時間 $T_{\text{ADisk, Insert}}$ と `Delete` ルール (タプルの削除) の所要時間 $T_{\text{ADisk, Delete}}$ および `skew_handler` ルールのウエイト値 `interval` を用いて以下の式で表される.

$$T_{\text{ADisk, migrate}} = T_{\text{ADisk, Insert}} + T_{\text{ADisk, Delete}} + \texttt{interval} \quad (4)$$

式 3, 4 に表 2 の値を代入して, `interval` = 50ms のとき $T_{\text{ADisk}} = 13,800$ms, また `interval` = 500ms のとき $T_{\text{ADisk}} = 103,000$ms となる. 図 5 の実測値は特に `interval` = 50ms の場合にこれより大きく悪化している.

この理由を探るために, 模擬自律ディスクの `RuleManager` 内部を細かく調べてみた. すると `skew_handling` ルールと `migrate_node` ルールは共に JavaVM の測定限界以下の 2 ~ 3ms 程度で終了しており, 遅延の原因とは考えられない. ところが `skew_handling` ルールの受信間隔は `interval` = 50ms の場合に 700ms 程度となる事があり, $T_{\text{ADisk, migrate}}$ に 120ms かかっていることになる. これはガベージコレクション (GC) やソケット通信の遅延といった要因よる遅延とはオーダーが異なり, ロックの競合が生じていると考えられる.

4.4 アクセスのある状態での再構築所要時間およびクラスタの性能劣化

ここではクライアントからのアクセスのある状態での再構築所要時間および再構築中のクラスタの `Insert` ルールのレスポンスタイムとスループットを測定する. 実験環境を以下に示す.

1. クライアントは 1 台, 毎秒 20`Insert` の負荷をかける.
2. 5 台のデータディスクからなるクラスタに 1 台のデータディスクを追加する.
3. `skew_handler` のウエイト値は 50ms とする.
4. クラスタにはあらかじめ 400 タプルのデータを均等に格納しておく.
5. ページサイズは, 8KB, タプルサイズは 4KB とする.
6. 偏り検出の閾値は 5 とする.

表 3: アクセスのある状態での再構築所要時間

	times (ms)	total moved pages
access (20 `Insert`/sec)	115429	340

表 4: アクセスのある状態でのレスポンスタイムおよびスループット

rule	response time (ms)	throughput (Mbps)
`Insert`	6.80	19.4

実験結果の再構築所要時間を表 3 に, 再構築中のクラスタの性能を表 4 に示す. なお, この場合は負荷均衡処理中にもランダムにページが挿入されるため, ページの移動数を予測することが難しい. そのため, 表 3 に負荷均衡処理完了後の総移動ページ数を付しておく.

考察 先の再構築所要時間とは逆に, 性能悪化は最小限に抑えられた. 表 3 および図 5 から, アクセスのある状態では移動ページ数 1.7 倍に対し所要時間は 3.7 倍と線形以上に悪化しているが, 表 4 からクライアントから見たレスポンスタイムの悪化は殆んど生じておらず, 台数増加に伴うスループット向上がはっきりとわかる. これは負荷均衡機構がうまく機能していることを示している. 結果として, クラスタ再構築プロトコルはクライアントから見たクラスタの性能に殆んど影響を与えることなく機能することが分かる.

5 おわりに

本稿では, 模擬自律ディスクに負荷均衡機構を実装し, これを用いたクラスタ再構築の実験を行った. その結果, クラスタ再構築の所要時間は予想よりも大幅に長くかかったが, 一方でクライアントから見たクラスタの性能に殆んど影響を与えることなく機能することが分かった. これはアクセスの多い大規模クラスタの管理コストを下げるための極めて有効な特性である.

今後の課題として, 模擬自律ディスク内部の解析を進め, クラスタ再構築の所要時間の大幅な遅延の原因を突き止め, 今後の実装に生かすことがあげられる. また, 分散ディレクトリに, より高性能な Fat-Btree を実装し, 自律ディスクと組み合わせた際のメリットを確認する.

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) 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] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. A Case for Intelligent Disks (IDISKs). *SIGMOD Record*, 27(3):42–52, Sep. 1998.

[2] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active Disks: Programming Model, Algorithms and Evaluation. In *Proc. of the 8th ASPLOS Conf.*, Oct. 1998.

[3] Erik Riedel, Garth Gibson, and Christos Faloutsos. Active Storage for Large-Scale Data Mining and Multimedia Application. In *Proc. of the 24th VLDB Conf.*, pages 62–73, 1998.

[4] 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.

[5] 阿部 亮太. ルール処理機能を有する高機能化ディスクの構成に関する研究. Master's thesis, 東京工業大学, 2001.

[6] 伊藤 大輔 , 横田 治夫. 自律ディスクにおけるディスク故障時/追加時のクラスタ再構築法. In **第** *12* **回データ工学ワークショップ論文集**, *DEWS2001 2B–4*. 電子情報通信学会データ工学研究専門委員会, 2001.

[7] 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.

[8] 伊藤大輔 , 横田治夫. 自律ディスクのオンラインクラスタ再構築の評価. In **信学技法** *FTS2001-27*, pages 17–24. 電子情報通信学会, 10 2001.

[9] National Storage Industry Consortium (NSIC). Object based storage devices: A command set proposal. http://www.nsic.org/nasd/1999-nov/final.pdf, Nov 1999.

[10] Haruo Yokota, Yasuhiko Kanemasa, and Jun Miyazaki. Fat-Btree: An Update-Conscious Parallel Directory Structure. In *Proc. of the 15th Int'l Conf. on Data Engineering*, pages 448–457, 1999.

[11] Mong Li Lee, Masaru Kitsuregawa, Beng-Chin Ooi, Kian-Lee Tan, and Anirban Mondal. Towards self-tuning data plaement in parallel database systems. *SIGMOD Record*, 29(2):225–236, Sep. 2000.

[12] Hisham Feelifl, Masaru Kitsuregawa, and Beng-Chin Ooi. A fast convergence technique for online heat-balancing of btree indexed database over shared-nothing parallel systems. In *11th Int'l Conf. on Database and Expert Systems Applications*, Sep 2000.

[13] 渡辺明嗣 , 横田治夫. ディレクトリ探索コスト重視並列ディスク偏り制御の評価. In **信学技法** *DE2001-110*, pages 17–24. 電子情報通信学会, 10 2001.