サーバーベースストレージにおける更新頻度による動的冗長化制御手法

深谷　崇元[†,††]　　Le Hieu Hanh[†]　　横田　治夫[†]

†† 株式会社日立製作所　〒 244-0817 神奈川県横浜市戸塚区吉田町 292 番地
† 東京工業大学　〒 152-8552 東京都目黒区大岡山 2-12-2
E-mail: *takayuki.fukatani.re@hitachi.com*, *hanhlh@de.cs.titech.ac.jp*, *yokota@cs.titech.ac.jp*

あらまし　低信頼な汎用サーバーの内蔵ディスクを、ソフトウェア制御によりサーバー間で冗長化して使用する、サーバーベースストレージが広まっている。サーバー間のデータ冗長化方式として広く使われているサーバー間レプリケーションは、複製データによる容量消費が大きく、容量効率の改善が課題となる。一方、Ceph などの分散ストレージでは、Erasure code(EC) による容量効率改善が可能だが、ネットワーク通信に伴うオーバヘッドがあるため、適用できるワークロードが限られてしまう。本論文では、サーバーベースストレージにおいて、更新頻度によりデータの冗長化方式をサーバー間レプリケーションから EC に切り替える Dynamic Redundancy Control(DRC) を提案する。DRC は、Linux ファイルシステムのファイルメタデータを拡張し、サーバー間で複製したユーザファイルを固定長チャンク単位で EC に切り替え可能とする。更新の少ないチャンクのみを EC 化することで、アクセス性能を維持した容量効率改善を実現する。また、DRC と従来手法とのアクセス性能の比較、および性能と容量効率のトレードオフの評価を行い、提案手法の有用性を確認する。

キーワード　ストレージ，冗長化，Erasure Code

1　はじめに

近年、汎用サーバーに搭載した低信頼な内蔵ディスクを、ソフトウェア制御により高信頼化して使用する、サーバーベースストレージが広まっている [1]- [3]。サーバーベースストレージは、安価な汎用サーバーを使用するため、従来のアプライアンス型の専用ストレージに対し、低コストでシステムを構築できる。また、コモディティの CPU や記憶媒体の性能向上に伴い、当初の課題であった性能も改善しつつあり、適用できるワークロードも広がっている [4]。

サーバーベースストレージを高信頼化するには、複数サーバーをまたがってデータを冗長化する必要がある。低信頼な汎用サーバーを使用するシステムでは、定常的にサーバー障害が発生する。そのため、サーバー障害発生時には、冗長データを持つ別のサーバーでサービスを継続することが求められる。このようなデータ冗長化制御として、アプリケーションや OS によるサーバー間レプリケーション [5]- [8] や、Ceph [1] や HDFS [3] などの分散ストレージによるデータ冗長化が広く使われている。

データを複数のサーバーに冗長化した場合、冗長データの容量削減が課題となる。サーバー間レプリケーションは、ローカルに格納したユーザデータの複製を、一台以上の別のサーバーに格納するため、冗長度に比例して複製データの容量が増えてしまう。一方、分散ストレージは、Erasure Code(EC) [9] により容量効率を改善することができるが、データやメタデータをネットワーク上に分散して格納するため、ネットワーク通信による性能オーバヘッドが発生してしまう。そのため、応答性能やサーバーあたりの処理性能を必要とする性能要件の高い用途ではサーバー間レプリケーションが選ばれることが多く、容量効率の改善が求められている。

本論文では、ユーザデータの冗長化方式を、更新頻度によりサーバー間レプリケーションから EC に切り替える、Dynamic Redundancy Control (DRC) を提案する。DRC は Linux[1] ファイルシステムのファイルメタデータを拡張し、ユーザファイルの冗長化方式を、固定長データのチャンク単位で切り替え可能とする。さらに、チャンク単位の更新頻度をモニタリングし、更新頻度の低いチャンクのみを EC 化することで、性能を維持した容量効率の改善を実現する。

また、提案手法の有用性を確認するため、DRC のプロトタイプを用いて、アクセス性能と、性能と容量効率のトレードオフを評価する。アクセス性能の評価では、DRC の EC 化有無と、従来方式との比較により、DRC の性能面での有用性を示す。性能と容量効率のトレードオフの評価では、ユーザが EC 化するチャンク容量の割合を指定することで、必要な性能に応じて容量効率を改善できることを確認する。

1 : Linux は、Linus Torvalds の米国及びその他の国における登録商標あるいは商標です。その他本稿で使用するシステム名、製品名はそれぞれ各社の商標、または登録商標です。

2 従来技術

2.1 複数サーバーによるデータ高信頼化

2.1.1 サーバー間レプリケーション

サーバー間レプリケーションは、ローカルに格納したプライマリデータの複製データを、一台以上の別のサーバーに格納するデータ冗長化手法である。ユーザは、プライマリデータを持つサーバーに障害が発生した場合でも、複製データを持つサーバーにアプリケーションを移行することで、サービスを継続することができる。PostgreSQL や MongoDB など多くのデータベース管理システムや、Virtual Machine(VM) ハイパバイザー、OS がサーバー間レプリケーションを提供している [5]- [8]。

サーバー間レプリケーション使用時のデータアクセスは、リードはローカルのプライマリデータからの読み込み、ライトはプライマリデータと複製データへの同期更新が基本となる。そのため、リード性能はローカルリードと同等性能となり、特に SSD などの高速な記憶媒体を使用する構成で高い性能となる。一方、ライト性能はサーバー間の同期更新が必要となり、ローカルライトに対し大幅に性能が低下する。そのため、種々の更新性能改善方式が提案されている [10]。

また、サーバー間レプリケーションは、冗長度に比例して複製データの容量が増えるため、容量効率が低いという欠点がある。

2.1.2 分散ストレージによるデータ冗長化

分散ストレージが提供するデータ冗長化機能を用いることで、ユーザデータを複数サーバー間で冗長化することができる。分散ストレージは、多数のサーバーでクラスタを組み、サーバー間でデータやメタデータを分散、冗長化して格納する。分散ストレージは、サーバー数に比例してシステム容量や性能を増やすことができるため、システムとしての容量･性能スケーラビリティを必要とするクラウドコンピューティングやビッグデータ解析の分野で普及が進んでいる。代表的な分散ストレージとしてオープンソースの Ceph や HDFS、ベンダによる VSAN [2] や Nutanix [12] などがある。

分散ストレージでは一般的に、レプリケーションと EC の 2 つの冗長化制御を使用する。分散ストレージのレプリケーションは、サーバー間レプリケーションと同様にプライマリデータの複製データを別のサーバーに格納するが、プライマリデータもネットワーク上で分散格納する点で異なる。一方、EC は複数のサーバーのデータからパリティデータを計算し、パリティデータを別サーバー格納することでデータを冗長化する [9]。データとパリティデータの組みをパリティグループと呼び、サーバー障害発生時には、パリティグループの残りのデータとパリティデータから、ロストしたデータを復号することができる。

レプリケーションと EC は、それぞれ更新性能と容量効率で他の方式に対し優位となっている。データ更新時に複製データの更新のみを必要とするレプリケーションは、パリティ再計算を必要とする EC に対し、更新性能で優位となる。一方、パリティデータによりデータを冗長化する EC は、複製データを必要とするレプリケーションに対し、容量効率で優れる。そのため、両者を組み合わせたハイブリッド方式として、データのアクセス頻度に応じた冗長化制御切り替え [11] [12] [18] や、レプリケーションと EC のキャッシュ階層制御 [1] [2] が提案されている。

分散ストレージは、データやメタデータのアクセスや管理のためにネットワーク通信が必要となり、応答性能やサーバー単位の処理効率が低下するという欠点がある。

2.2 従来技術の課題

応答性能やサーバー単位の性能を必要とする用途では、性能上の理由から、冗長化制御にサーバー間レプリケーションを選ぶことが多い。ローカルボリュームやローカルファイルシステムを複製するサーバー間レプリケーションは、分散ストレージに対しネットワーク通信量が大幅に少なくなる。そのため、ネットワーク通信に伴う遅延や、ネットワーク帯域や CPU 消費といった性能オーバヘッドも小さくなる。

一方で、従来のサーバー間レプリケーションは、複製データによる容量消費が大きく、かつ EC による容量削減もできないという問題がある。EC により容量を削減するためには、異なる複数のサーバー間でパリティグループを構成する必要がある。サーバー間レプリケーションでは、分散ストレージのように複数のサーバーをまたがってデータやメタデータを一括して管理することはなく、サーバーをまたがった EC を構成できない。

サーバー間レプリケーションを使用するような性能要件の高い用途で、性能を維持しつつ EC による容量削減を実現するには、以下に述べる課題を解決する必要がある。

a） **低オーバヘッドな EC の実現**

従来の分散ストレージの EC では、クラスタ内のデータ配置を分散データベースで管理し、その情報を元に複数サーバー間でパリティグループを構成し、EC 化する。このようなネットワークを介した EC では、データやメタデータのアクセスの度にネットワーク通信が必要となり、性能オーバヘッドが大きい。サーバー間レプリケーションでは、ネットワーク通信は更新処理でのみ行っており、分散ストレージの EC をそのまま使用した場合、性能が低下することとなる。従来のサーバー間レプリケーションの性能を維持するためには、ネットワーク通信を削減し、低オーバヘッドな EC を実現する必要がある。

b） **EC 化されたデータに対する更新頻度の低減**

EC 化したデータを更新する場合、パリティの再計算が必要となり、レプリケーションに対し更新性能が低下する。特に更新サイズが小さい場合、オリジナルデータとパリティデータの両方でリード・モディファイ・ライトが発生し、更新性能が大幅に低下する。このような更新性能低下を回避することは EC の仕組み上難しいが、EC 化されたデータに対する更新自体を減らすことで、影響範囲を抑えることができる。そのため、従来のサーバー間レプリケーションの性能を維持するためには、EC 化するデータに対する更新頻度をできるだけ下げることが必要となる。

3 Dynamic Redundancy Control

3.1 概　　要

本研究では、性能を維持した EC による容量効率改善を実現するデータ冗長化制御手法として、DRC を提案する。DRC は、ローカルに格納したユーザデータの冗長化方式を、更新頻度に応じてサーバー間レプリケーションから EC に切り替える冗長化制御である。DRC は、ローカルファイルシステムにデータや EC 用のメタデータを格納することで、データやメタデータアクセス時のネットワーク通信を不要とし、低オーバヘッドな EC を実現する。また、低更新頻度のデータのみを EC 化することで、EC 化に伴う更新性能低下の影響を抑える。

DRC の概要を図 1 に示す。DRC は POSIX ベースの Linux ファイルシステム (Linux FS) の上位モジュールとしてユーザ透過に動作する。DRC は、レプリケーションを相互に行う 2 台のサーバー間でレプリケーションペアを組み、複数のレプリケーションペアからなる冗長化クラスタを構成する。レプリケーションペアを組むサーバーは、冗長化対象となる全てのユーザファイルとディレクトリの複製データを相互に持ち合い、全ての更新操作を同期して処理する。レプリケーションペアを組むどちらか一方のサーバーで障害が発生した場合、もう一方のサーバーでサービスを引き継ぐことができる。

DRC は、ローカルファイルシステムのファイルメタデータに EC 用の構成情報を格納し、ユーザファイルを固定長のチャンク単位で EC 化可能としている。DRC は、Linux FS のファイルメタデータを拡張し、チャンクの EC 用の構成情報をファイルごとに管理する。これらの構成情報はローカルアクセスにて参照することができるため、アクセス時にネットワーク通信のオーバヘッドが発生しない。チャンクサイズは 512KB としており、EC 化対象のファイルも 512KB 以上となる。512KB 以下のファイルに対しては、メタデータ管理やモニタリングは行わず、従来のサーバー間レプリケーションをそのまま適用する。一般的にデータサイズの大きなファイルが全データ容量の大部分を占めることが知られており、EC 化対象を 512KB 以上としても十分な効果が得られる [13]。一方、小さいファイルは一時ファイルなど更新頻度が高いファイルが多く含まれており、EC 化による更新性能低下の影響が大きい。

また、DRC は、複数サーバー間で連携し、低更新頻度のチャンクを優先して EC 化することで、EC 化による更新性能低下の影響を抑える。DRC は、異なるサーバーから EC 化していない低更新頻度のチャンクを選択し、パリティデータを別のサーバーにパリティファイルとして格納することでデータを冗長化する。パリティファイル作成後は、複製データの容量を解放することで、冗長データ容量を削減する。また、ユーザはユーザデータの容量に対し、EC 化するチャンク容量の割合を指定することで、必要な性能に応じた容量効率改善を実現できる。

3.2 アーキテクチャ

DRC は、FUSE をベースとしたユーザスペースファイルシステムであり、内部的に ext4 や XFS といった Linux FS を使用する [15] [16]。Linux FS との通信には標準的な Linux API を用いており、標準的な Linux サーバーであれば DRC を使用することができる。

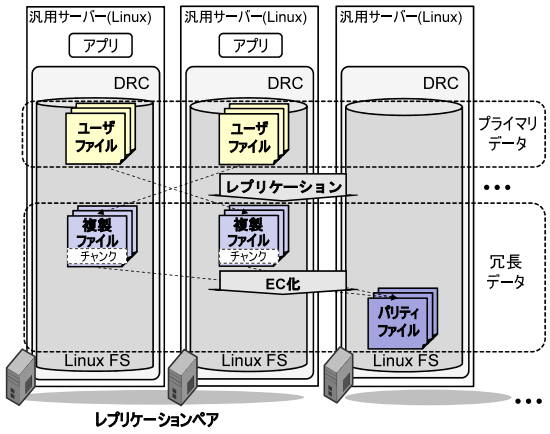

図 1　DRC 概要図

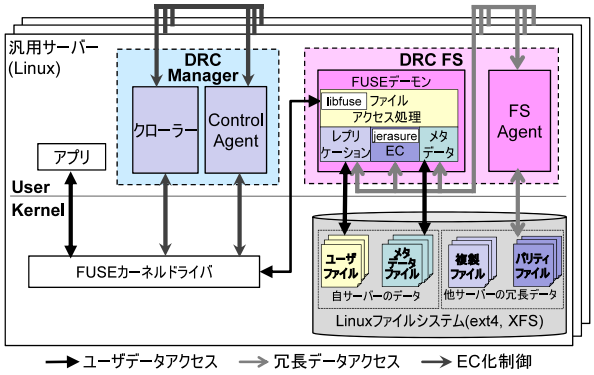

図 2　DRC アーキテクチャ

図 2 に DRC のアーキテクチャ図を示す。DRC はアクセス要求を処理する DRC FS と、EC 化を制御する DRC Manager の二つのモジュールからなる。

DRC FS は、FUSE カーネルドライバからのユーザデータアクセスを処理する FS デーモンと、他のサーバーからの冗長データアクセスを処理する常駐プロセスの FS Agent からなる。FS デーモンは FUSE カーネルドライバからのアクセス要求に対し、Linux ファイルシステムのファイル操作に加え、他のサーバーの FS Agent と連携したレプリケーション・EC 制御と、メタデータ管理を行う。FS デーモンは、ユーザファイルの更新要求受信時に、他のサーバーの FS Agent を介して、複製ファイルまたはパリティファイルを更新する。複製ファイルの更新は、更新系のファイル操作をサーバー間で転送することで実現する。一方、パリティファイルの更新は、jerasure [17] を用いて再計算したパリティデータを、パリティファイルを格納するサーバーの FS Agent を介して、パリティファイルに上書きする。メタデータ管理は、EC 化された全てのチャンクに対し、パリティグループの構成情報をファイルメタデータに格

納する。メタデータ管理の詳細については、3.3 節で後述する。

DRC Manager は、コマンドとして動作するクローラーと、他のサーバーからの制御要求を処理する常駐プロセスである Control Agent からなる。クローラーは定期的に起動し、他のサーバー上の Control Agent と連携してパリティグループを構成し、EC 化する。EC 化制御の詳細は、3.4 節で後述する。

3.3 メタデータ管理

DRC は、EC 化したチャンクのパリティグループの構成情報を、ローカルファイルシステムにファイルメタデータとして格納する。パリティグループの構成情報は、ファイル更新時のパリティデータ更新に必要なパリティファイルの参照ポインタと、サーバー障害時のチャンクリカバリに必要なパリティグループ内の全てのチャンクとパリティファイルの参照ポインタを含む必要がある。DRC は、このようなパリティグループの構成情報をファイルメタデータとして管理するために、拡張ファイルメタデータ [14] と、サーバーをまたがった識別子であるグローバル ID を使用する。以下では、拡張ファイルメタデータとグローバル ID のそれぞれについて説明する。

3.3.1 拡張ファイルメタデータ

DRC は、ユーザーファイル、複製ファイル、パリティファイルそれぞれに対し、拡張ファイルメタデータを付与し、パリティグループの構成情報として使用する。拡張ファイルメタデータは、POSIX 標準のファイルメタデータであるファイル拡張属性と、メタデータ用のシステムファイルであるメタデータファイルを使用した、ファイルメタデータの格納領域である。拡張ファイルメタデータを用いることで、ファイルに対し任意のメタデータを付与することができる。

ファイル種別それぞれの、拡張ファイルメタデータの内容と格納場所を表 1 に示す。ユーザーファイルの拡張ファイルメタデータには、ファイル内の全ての EC 化済チャンクのパリティ ID を格納する。データの格納先は、ファイル内の任意の数のチャンクの構成情報を格納できるように、メタデータサイズの上限がないメタデータファイルを使用する。メタデータファイルは、ファイル内で最初のチャンクを EC 化する際に作成し、メタデータを付与するファイルのファイル拡張属性にファイルハンドルを記録する。同様に、複製ファイルにも、ユーザファイルと同等のファイルメタデータを付与し、サーバー障害時のチャンクリカバリを可能とする。パリティファイルには、パリティグループに含まれるすべてのチャンクの ID を格納し、チャンクリカバリ時にパリティグループ内のチャンクを参照するために使用する。

拡張ファイルメタデータを使用した EC 化済チャンクのデータ更新とリカバリを図 3 を用いて説明する。図 3 は、ファイル A を含むファイルシステム 1（FS1）をサーバー 1 で、ファイル B を含むファイルシステム 2(FS2) をサーバー 2 で、それぞれ運用する構成を示している。ファイル A のチャンク 1 と、ファイル B のチャンク 2 はパリティグループを構成しており、パリティファイルをサーバー 3 に格納している。これらのファイルシステムでは、ユーザファイルに対し、複製ファイル、パリティファイル、メタデータファイルを、それぞれ repli、parity、meta と名付けた異なるシステムディレクトリに格納している。

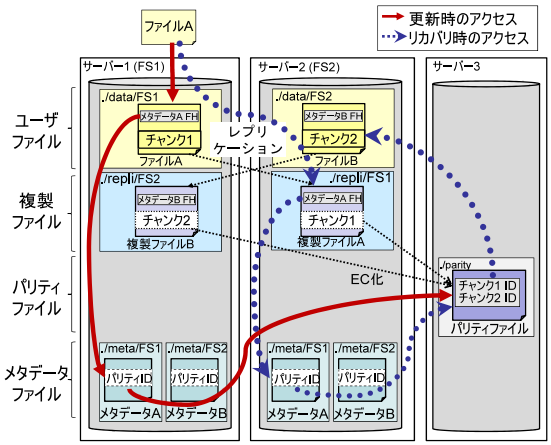

図 3　拡張ファイルメタデータを使用したデータ更新・リカバリ

チャンク 1 に更新が発生した場合、DRC はチャンク 1 とパリティファイルを同時に更新する必要がある。DRC は、ファイル A のファイル拡張属性に記録したファイルハンドルからメタデータファイルにアクセスし、チャンク 1 が EC 化されていると判断する。その後、メタデータファイル中にあるパリティ ID を用い、パリティファイルにアクセスし、パリティデータを更新する。また、サーバ 1 の障害時のリカバリ時には、更新時と同様に複製ファイルのメタデータファイルからパリティファイルのパリティ ID を調べ、パリティファイルにアクセスする。次に、パリティファイルのファイル拡張属性に記録したチャンク ID からチャンク 2 にアクセスする。パリティファイルのパリティデータとチャンク 2 のデータからチャンク 1 を復号する。

3.3.2 グローバル ID

複数のサーバー間でチャンク、パリティを管理するには、サーバーをまたがった ID 管理が必要となる。DRC は、サーバー、ファイルシステム、ファイル、チャンク、パリティそれぞれに対し、システム内で一意な識別子であるグローバル ID を管理する。グローバル ID は対象とするオブジェクトへの参照情報も含んでおり、ID を知ることで、該当オブジェクトにアクセス可能となる。

これらグローバル ID の構成を図 4 に示す。ノード ID はサーバーをシステム内で一意に識別するための 2 バイトの ID である。DRC は、サーバー追加時に、他のサーバーと重複しないノード ID を新規サーバーに割り当てる。ファイルシステム ID は、ノード ID に対してサーバー内で一意な 6 バイトの識別子

表 1　ファイル種別ごとの拡張ファイルメタデータ

ファイル種別	ファイル拡張属性	メタデータファイル
ユーザファイル	メタデータファイルのファイルハンドル	EC 化済チャンクのパリティID
複製ファイル	メタデータファイルのファイルハンドル	EC 化済チャンクのパリティID
パリティファイル	チャンク ID	-

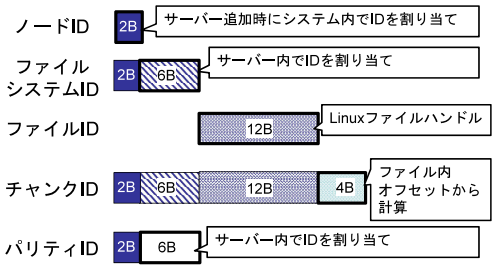

図 4　グローバル ID

を付けた ID となる。ファイル ID はファイルシステム ID に対して、Linux FS のファイルハンドル [23] を追加した ID である。チャンク ID はファイル ID に対して、ファイル内のチャンクのインデックスを追加した ID で、インデックスはファイル内のオフセットから動的に計算できる。パリティID は、ノード ID に対してサーバー内で一意な 6 バイトの ID を追加している。以上で示した通り、グローバル ID はノード ID を除き、ノード ID を先頭につけたサーバーごとの排他的な ID 空間を持つ。この特性により、ノード ID のみ重複なく生成することができれば、他のサーバーと通信することなくグローバル ID を生成することができる。

また、グローバル ID は、対象オブジェクトへの参照ポインタとしても使用することができる。DRC は、ノード ID とサーバーのホスト名の対応関係を、すべてのサーバーで共有し、ノード間の通信に使用する。ファイルシステム ID も、ID 内に含まれるノード ID からアクセス先のサーバーとの通信が可能となり、サーバー上の FS Agent または Control Agent を介しファイルシステムにアクセスすることができる。同様にファイル ID とチャンク ID も、ID 内のノード ID から対象とするサーバーがわかるため、ファイルハンドルベースの Linux API [23] を用いて対象ファイルにアクセスできる。パリティID は、パリティファイルの名前にパリティID を使用し、あらかじめ決められたディレクトリに格納することで、ファイルパスを用いたパリティファイルへのアクセスを可能としている。

3.4　EC 化制御

DRC Manager は、ユーザーファイルの更新頻度をチャンク単位でモニタリングし、他のサーバーの DRC Manager と連携して更新頻度の低いチャンクを EC 化する。EC 化は、全データ容量に対する EC 済チャンクの容量が、ユーザが事前に指定した割当に達するまで行う。このように EC 化するチャンク容量の割合をユーザが指定することで、必要な性能要件に応じた容量効率の改善を可能としている。

EC 化制御は、モニタリング、クローリング、EC 化の処理からなる。以下ではそれぞれの処理について述べる。

3.4.1　モニタリング

DRC FS は、ユーザファイルの更新処理にて、チャンク単位の更新回数をモニタリングし、統計情報として管理する。DRC FS はチャンクごとに 32 バイトの統計情報をメモリ上に格納し、DRC Manager に対し専用 ioctl 経由で提供する。統計情報によるメモリ使用量は、1TB の容量を使用した場合でも最大 64MB にとどまり、近年のメモリの大容量化を踏まえれば問題ないレベルと言える。

3.4.2　クローリング

DRC Manager は定期的にクローラーを起動し、事前に用意した評価関数を用い、ファイルシステム内の全てのチャンクに対し EC 化の優先度を数値化する。クローラーはサーバーごとに起動し、チャンク ID と評価関数の結果をチャンクリストとして記録する。評価関数には、前回のクローリング以降に発生したチャンクの更新回数の逆数を用い、低更新頻度のチャンクを優先して EC 化する。

3.4.3　EC　　化

最小のノード ID をもつサーバーは、EC 化制御を行うマスタノードとして動作し、クローリング完了後に全サーバーの EC 化を一括して制御する。マスターノードは、各サーバーからチャンクリストを取得し、評価関数の結果が大きいチャンクを、異なるサーバーからパリティグループのデータ符号数分選ぶ。また、残りのサーバーのうち、空き容量が大きいサーバーをパリティ格納サーバーとしてパリティ符号数分選び、先ほど選んだチャンクと合わせてパリティグループを構成する。マスタノードは、これらパリティグループの情報を、パリティ格納サーバーのうちの一台の Control Agent に送信し、EC 化を指示する。EC 化は、EC 化するチャンクの割合が、ユーザが指定した値に達するまで行う。

EC 化指示を受けたパリティ格納サーバーの Control Agent は、他のサーバーの Control Agent と連携し、EC 化を実行する。パリティ格納サーバーの Control Agent は、パリティグループ内の全チャンクのデータを各サーバーから取得し、パリティデータを計算してパリティファイルを作成する。また、パリティ格納サーバーの Control Agent は、パリティファイル作成時にパリティID を採番し、パリティグループ内の各サーバーに送信する。各サーバーの Control Agent は、受信したパリティID をファイルメタデータに記録することで、チャンクを EC 状態に遷移し、使わなくなったチャンクの複製データを fallocate システムコール [19] を用いて解放する。

4　評　　価

提案手法の有効性を確認するため、DRC のプロトタイプを用いた評価を行った。評価は、DRC の EC 化有無および従来手法とのアクセス性能比較と、EC 化割合を変えることによる性能と容量のトレードオフの確認を、実機測定により行った。

4.1　評 価 環 境

評価環境を表 2 に示す。評価では汎用サーバー（HP Proliant DL160) を 3 台を使用し、そのうち 1 台でベンチマークツールの filebench [24] を動作させた。ストレージ制御には、DRC の

表 2 評価環境

設定値	値
マシン	HP Proliant DL160 G6
CPU	Intel(R) Xeon E5620 2.40GHz、4 コア
メモリ	10GB
ディスク	7.2Krpm SATA HDD (HDD 構成)、 RAM ディスク (RAM ディスク構成)
NIC	Intel 82576 Gigabit Network Connection
OS	Ubuntu 16.04.4
ストレージ制御	Ceph version 13.2.2(Ceph FS、ceph-fuse) DRC プロトタイプ (ext4)
ベンチマーク	filebench1.5-alpha3

プロトタイプと、比較対象として Ceph を用いた。Ceph は、ファイルアクセスを提供する Ceph FS および ceph-fuse を用いた。Ceph FS の冗長化方式は、DRC のレプリケーションと条件を合わせるため 2 重化レプリケーションとした。また、ディスク性能の違いによる影響をみるため、ディスクがボトルネックとなりやすい HDD 構成と、ディスク性能が十分にある RAM ディスク構成の 2 つのディスク構成を評価した。メモリ容量は、HDD 構成と RAM ディスク構成共に 10GB とし、HDD 構成ではカーネルオプションで、RAM ディスク構成は RAM ディスクサイズでそれぞれ調整している。

4.2 アクセス性能

DRC のレプリケーショと EC、および Ceph FS のアクセス性能を比較評価した。ここでいうレプリケーションは全チャンクをレプリケーションした構成、EC は全チャンクを 2D+1P にて EC 化した構成である。ワークロードは 5 多重の 64KB シーケンシャルリード/ライト、1 多重の 8KB ランダムリード/ライト、および 1 多重のログライト、10 多重の DB ライト、200 多重の DB リードからなる OLTP を用いた。データセットサイズは、HDD 構成ではメモリ容量の 2 倍、RAM ディスク構成ではリソース量の制約から RAM ディスクを除くメモリ容量と同サイズとした。

評価結果を図 5、図 6 に示す。図中では、各ワークロードごとに DRC のレプリケーションに対する性能比を、DRC の EC、Ceph FS それぞれについて示している。また、DRC のレプリケーションの性能値を該当する棒グラフのラベルで示している。

HDD 構成では、DRC のレプリケーションと EC で、リード性能は同等だが、シーケンシャルライト、ランダムライト、OLTP で EC がレプリケーションに対し 77%、90%、27%低い性能となった。リード性能は、レプリケーション、EC ともにローカルファイルの読み込みとなるため同等性能となるが、その他の更新処理を伴うワークロードでは、EC 化に伴う更新処理オーバヘッドにより性能が低下している。

また、HDD 構成での Ceph FS との比較では、リード性能は Ceph FS が高く、ライト性能は Ceph FS が DRC のレプリケーションと EC の中間、OLTP 性能は EC 化した場合でも DRC の方が高くなっている。リード性能は、ディスクがボトルネックとなる HDD 構成では、Ceph FS の方がサーバー間でディスク負荷が分散できるため性能が高くなっている。その他の更新処理を伴うワークロードでは、レプリケーション同士の比較では DRC が優位だが、EC 化した DRC と Ceph FS ではワークロードにより優劣が異なる。これは、Ceph FS はライト要求に対しディスク IO 数が増えるオーバヘッド [20] が、DRC では EC 化の伴う更新処理のオーバヘッドがあり、ワークロードごとにそれぞれのオーバヘッドの影響度合いが異なり、優劣が変わっていると推測される。

RAM ディスク構成でも DRC のレプリケーションと EC のリード性能は同等となるが、シーケンシャルライト、ランダムライト、OLTP 性能のそれぞれで EC の性能が 6%、39%、36%低下している。HDD 構成に比べて性能差の傾向が変わっているが、これはディスク性能が高くなった結果、ボトルネック部位がシーケンシャルライトではネットワークポート、OLTP では CPU に変わったのと、ランダムライトではディスク IO の応答時間が変わったためである。

RAM ディスク構成では Ceph FS に対する DRC のリード性能がレプリケーションと EC 共にシーケンシャルリードで 16 倍、ランダムリードで 2.7 倍となり、OLTP 性能がレプリケーションで 11 倍、EC で 7.4 倍と、DRC が圧倒的に優位となっている。ライト性能は、シーケンシャルライトはレプリケーションと EC 共に同等で、ランダムライトがレプリケーションで 28%、EC で 55%、DRC の性能が低くなっている。リード性能と OLTP 性能で DRC が優位となるのは、Ceph FS でネットワークポートが性能ボトルネックとなるのに対し、DRC は CPU 限界まで追い込むことができているためである。ライト性能は、シーケンシャルライトでは Ceph FS、DRC 共にネットワークポートが性能ボトルネックとなっているが、ランダムライトでは IO ごとの処理時間が DRC の方が大きく性能差として見えている。これは、バッファ処理など実装レベルの違いが原因と考えられる。

以上の結果から、DRC は EC 化によりライト性能が低下し、特にディスク性能が低い構成にて応答時間が大幅に劣化することがわかる。Ceph FS との比較では、同じレプリケーションを使用した場合、HDD 構成のリード性能と RAM ディスク構成のランダムライト性能以外では DRC が優位となり、DRC を EC 化した場合でも、RAM ディスク構成ではランダムライトを除き DRC の方が優位となる。特に RAM ディスク構成では、DRC のリード性能が Ceph FS に対し圧倒的に優位となり、OLTP 性能でも数倍以上の性能となっている。近年では、SSD の低価格化が進み、高いディスク性能を持つ構成が一般的になってきている。このような構成では、DRC は Ceph FS に対し、優位であると言える。

4.3 性能と容量効率のトレードオフ

EC 化による性能と容量効率のトレードオフを評価するため、EC 化するチャンクの割合に対する性能と冗長データの容量の変化を評価した。評価では、前節で用いた OLTP 測定と同じ環境を用いた。

これらの評価結果を、HDD 構成、RAM ディスク構成のそれ

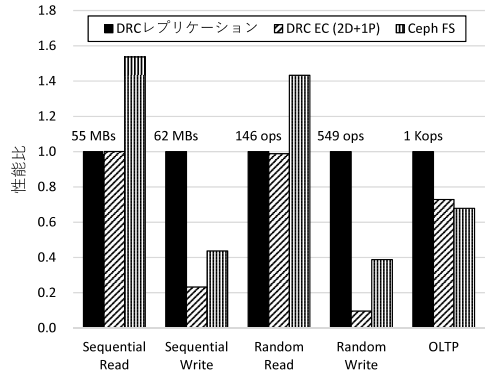

図 5　HDD 構成のアクセス性能

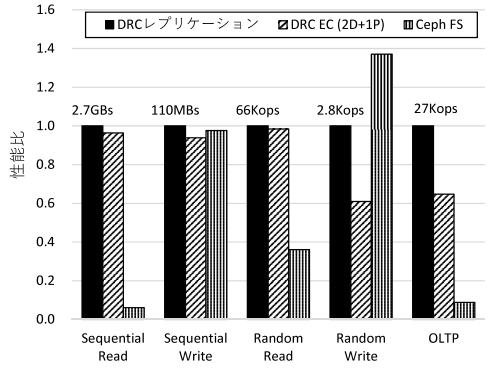

図 6　RAM ディスク構成のアクセス性能

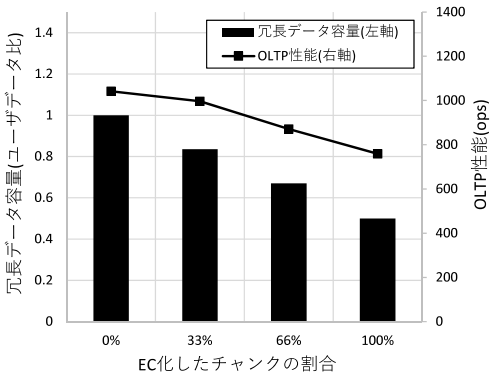

図 7　HDD 構成の性能・容量効率トレードオフ

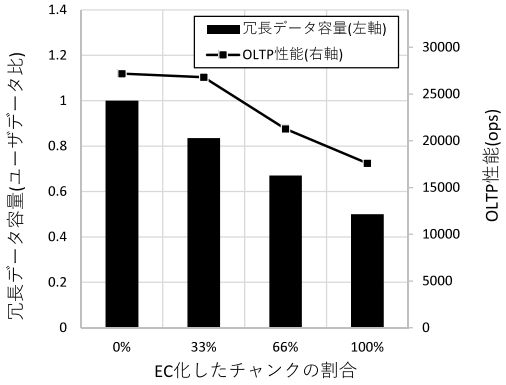

図 8　RAM ディスク構成の性能・容量効率トレードオフ

ぞれについて、図 7 と図 8 に示す。図中では左軸にユーザデータに対する冗長データ容量の比率を、右軸には OLTP 性能をそれぞれ示している。

図が示す通り、HDD 構成、RAM ディスク構成ともに EC 化するチャンクの割合を増やすに従い、性能と冗長データの容量が共に低下することがわかる。これは、EC 化するチャンクを増やすことで、性能は低下するが容量効率は改善されることを示しており、性能と容量効率のトレードオフが実現できていることを意味する。

また、本評価では、データ更新時にランダムに更新箇所を選ぶ filebench を用いているが、実世界ではデータ更新により大きな局所性があることが期待できる。その場合、EC 化するチャンクの割合が低いうちは、EC 化されたチャンクへの更新も少なく、性能低下も本評価結果より小さくなると予想される。

以上の結果より、DRC では EC 化するチャンク割合を指定することで、性能要件に合わせた容量改善が実現できると言える。

5　関 連 研 究

データの冗長方式を動的に切り替える技術の研究が、分散ストレージの分野でなされてきた [11] [12] [18]。これらの研究は、データやメタデータをサーバー間で分散することを前提としており、データやメタデータアクセス時のネットワーク通信量が大きくなっている。本研究ではローカルに格納したファイルメタデータに EC 用のメタデータを格納することで冗長化方式の切り替えを実現しており、従来研究に対しネットワーク通信のオーバヘッドを削減している。

ファイルメタデータを拡張し、データバックアップやマイグレーションに使用する研究が、Network-Attached Storage(NAS) の分野でなされてきた [21] [22]。これらの研究では、NAS に格納したファイルのファイル拡張属性に、別のサーバーのデータへの参照ポインタを格納し、ファイルをスタブ化している。参照ポインタには、オブジェクトストレージのオブジェクト ID や、マイグレーション元ファイルのファイルパスを使用する。また、同じく NAS の分野で、任意サイズのファイルメタデータの実現方式として、ファイルハンドルベースの Linux API を使用した拡張ファイルメタデータが提案されている [14]。本研究では、参照ポインタにグローバル ID を使用することでサーバー間のパリティ、チャンクの相互参照を可能としている点と、ファイルメタデータの拡張により EC を実現している点で、従来手法と異なる。

6　今後の課題

今後の課題として、DRC のリカバリ性能の評価と、EC 化制御における評価関数の拡張の 2 点がある。

リカバリ性能の評価では、EC 化によるリカバリ性能と障害時リード性能への影響を評価し、必要であれば性能を改善する。また、本論文で評価したアクセス性能と同様に、これらの性能と容量効率のトレードオフが実現できているかを確認する。

また、本論文では EC 化制御で使用する評価関数として更新頻度を用いているが、他にリカバリ性能やファイル種別を考慮した評価関数が考えられる。評価関数を変えることによる性能や容量効率への影響を明らかとすることが課題である。

7 おわりに

本論文では、サーバーベースストレージにおいて、ユーザデータの冗長化制御を更新頻度に応じてサーバー間レプリケーションから EC に切り替える、DRC を提案した。また、DRC のプロトタイプを用いた評価により、DRC が従来手法に対し性能優位であり、特にディスク性能が十分な構成で、リード性能と OLTP 性能で数倍以上の性能が実現できることを確認した。また、ユーザが EC 化する割合を指定することで、性能と容量効率のトレードオフを実現できることを確認した。これらの結果から、提案手法を用いることで、性能要件が高い用途においても、従来のサーバー間レプリケーションの性能を維持した、EC 化による容量改善が可能となることを示した。

文　献

[1] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, “Ceph: a scalable, high-performance, distributed file system,” 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’06), 2006

[2] vmware, “VMware vSAN 6.7 Technical Overview,” https://storagehub.vmware.com/t/vmware-vsan/, accessed on 2018/12/16

[3] D. Borthakur, “The Hadoop Distributed File System: Architecture and Design,” https://hadoop.apache.org/

[4] RedHat, “Red Hat Ceph Storage on Servers with Intel Processors and SSDs”, 2016, https://www.redhat.com/en/resources/red-hat-ceph-storage-servers-intel-processors-and-ssds, accessed on 2018/12/16

[5] R. Philipp, “DRBD v8 - Replicated Storage with Shared Disk Semantics,” Proceedings of the 12th International Linux System Technology Conference, 2005

[6] J. Spillner, G. Toffetti, and M. R. López, “Cloud-Native Databases: An Application Perspective,” , ESOCC 2017: Advances in Service-Oriented and Cloud Computing, 2017

[7] VMware docs, ”How vSphere Replication Works,” https://docs.vmware.com/, accessed on 2018/12/16

[8] Microsoft Docs, ”Storage Replica overview,” accessed on 2018/12/16

[9] J. S. Plank, “Erasure Codes for Storage Systems: A Brief Primer,” article in ;login: issue: December 2013, Volume 38, Number 6

[10] 小林 大, 横田 治夫, “負荷変動と応答性能維持を考慮した高通う並列ストレージシステムのための複製利用と更新要求制御,” 情報処理学会論文誌 49, 2008

[11] M. Xia, M. Saxena, M. Blaum, and D. Pease, “A Tale of Two Erasure Codes in HDFS,” 13rd USENIX Conference on File and Storage Technologies (FAST ’15), 2015

[12] I. Cano, S. Aiyar, V. Arora, M. Bhattacharyya, A. Chaganti, C. Cheah, B. Chun, K. Gupta, and V. Khot, “Curator: Self-Managing Storage for Enterprise Clusters,” 14th USENIX Conference on Networked Systems Design and Implementation (NSD I ’17), 2017

[13] G. Wallace, F. Douglis, H. Qian, P. Shilane, S. Smaldone, M. Chamness, and W. Hsu, “Characteristics of Backup Workload in Production Systems,” 10th USENIX conference on File and Storage Technologies (FAST ’12), 2012

[14] T. Fukatani, A Sutoh, and T. Nakano, “A Method to Adapt Storage Protocol Stack Using Custom File Metadata to Commodity Linux Servers,” International Journal of Smart Computing and Artificial Intelligence, Vol. 2, No 1, 2018

[15] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu, “A Study of Linux File System Evolution,” 11th USENIX conference on File and Storage Technologies (FAST ’13), 2013

[16] B. K. R. Vangoor, V. Tarasov, and E. Zadok, “To FUSE or Not to FUSE: Performance of User-Space File Systems,” 15th USENIX conference on File and Storage Technologies (FAST ’17), 2017

[17] J. S. Plank, “Jerasure: A library in C/C++ facilitating erasure coding for storage applications,” Technical report CS-07-603, University of Tennessee, 2007

[18] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. “Erasure coding in windows azure storage,” 2012 USENIX conference on Annual Technical Conference (ATC ’12), 2012

[19] J. Corbet, “Punching holes in files,” Nov. 2010, accessed on 2018/12/16

[20] D. Y. Lee, K. Jeong, S. H. Han, J. S. Kim, J. Y. Hwang and S. Cho, “Behaviors of Storage Backends in Ceph Object Store,” 33rd International Conference on Massive Storage Systems and Technology (MSST 2017), 2017

[21] J. Nemoto, A. Sutoh, and M. 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, 2017

[22] K. Matsuzawa, M. Hayasaka, and T. Shinagawa, “The Quick Migration of File Servers,” 11th ACM International Systems and Storage Conference (SYSTOR 2018), 2018

[23] open_by_handle(3) -Linuxman page; https://linux.die.net/man/3/open_by_handle.

[24] V. Tarasov, E. Zadok, and S. Shepler, “Filebench: A Flexible Framework for FileSystem Benchmarking,” article in ;login; magazine, Spring 2016, Vol. 41, No. 1