Vol.2013-ARC-205 No.2
Vol.2013-OS-125 No.2
2013/4/25

クラウドを前提としたストレージアーキテクチャの提案

豊島 詩織[1,a)]　小田 哲[1,b)]　小西 隆介[1,c)]　湯口 徹[1,d)]

概要：究極的にストレージリソースが集約された世界のストレージアーキテクチャの提案を行う．リソース集約を行うことによる一般的なクラウドのメリットに加えて，大規模なストレージリソースの重複排除を行うことができる．我々はデータに対するポインタの作成に暗号学的ハッシュ関数を利用し，時間や空間，ユーザを超えた効率的なデータの再利用が可能なアーキテクチャ提案する．本稿では提案アーキテクチャ上でオブジェクトストレージを実装しフィージビリティの確認を行った．

キーワード：クラウドストレージ，広域ネットワーク，暗号学的ハッシュ関数

A Proposal of Storage Architecture on the Cloud

Shiori Toyoshima[1,a)]　Satoshi Oda[1,b)]　Ryusuke Konishi[1,c)]　Toru Yuguchi[1,d)]

Abstract: We propose a storage architecture assuming when storage resources are consolidated ultimately. In addition to general benefits of cloud, we can perform deduplication on a large scale on the situation. We propose a architecture which can re-use data over time, space and people. Pointers that indicating data are generated by using cryptographic hash function. In this paper, we implemented object storage on the proposing architecture and confirmed its feasibility.

Keywords: CloudStorage，Wide Area Network，Cryptographic hash function

1. はじめに

1.1 背景

ネットワーク網が進化し，世界中のあらゆる所から様々なリソースにアクセスできる世界が来ている．特に巨大なDC を構築し必要に応じたリソースを必要なだけ提供するタイプのユーティリティ・コンピューティングは，クラウド・コンピューティングと呼ばれ多くの場面で利用されている．これまですべてがネットワークの向こう側で処理されるという不安や，セキュリティの観点などでクラウドへの移行に懐疑的だった企業も，コストメリットや事業継続の観点から，オンプレミスなシステムからプライベートクラウドやパブリッククラウドへの移行が進んできている．

一般的なクラウドは数に対するスケールメリットを生かし，集約効果によるコストの削減を実現している．これらは集約すれば集約するほどその効果は高まるため，これからもコンピューティングリソースの集約が進むことが予想される．

コンピューティングリソースがクラウド側に集約されると，手元にあるローカルの端末は少ない機能でこれまでと同様のことが実現できる．例えば，DaaS(Desktop as a Service) に代表されるようにデスクトップの仮想化が進むと，ローカルの端末はネットワーク機能の他に，キーボードやディスプレイといった入出力機器だけで実現できるようになる．

それでは，ストレージが究極的にクラウドに集約されている状態というのはどういう状況であろうか？今，問題を簡単にするためにネットワークの転送コストが無視出来るほど小さい状況を考える．ネットワークを介したデータは

[1] 日本電信電話株式会社
3-9-11, Midoricho, Musashino-shi, Tokyo 180-8585, Japan
[a)] toyoshima.shiori@lab.ntt.co.jp
[b)] oda.satoshi@lab.ntt.co.jp
[c)] konishi.ryusuke@lab.ntt.co.jp
[d)] yuguchi.toru@lab.ntt.co.jp

無視出来るほど小さい時間 μ で取得可能であり，同時に書込も可能であるとする．

ここで，不揮発なストレージに書かれているあらゆるデータは，ネットワークを介してクラウド上の巨大なストレージに置かれている状況を考えてみよう．そのストレージは，あらゆるユーザが，過去から現在に至るまでに書き込まれたすべてのデータを保存しており，全世界から参照可能である．

ユーザはその巨大なストレージにデータを書き込んでおけばよい．一度データを書き込むと，ユーザはストレージに書かれているデータに対するポインタを持っているだけでよい．データを参照する場合は，ポインタを利用してネットワーク上のデータにアクセスし，取得する．このように一般的なクラウドのメリットと同じようにストレージを一極集中管理をすることで，データを維持する管理コストを減らすことが出来る．

さらに，ストレージを一極集中管理するメリットはこれだけではない．具体的には，同一のビット列を持つデータの重複排除をすることが出来る．データの重複排除はバックアップやクラウド上のマシンイメージなどで非常に効果が高いと言われている [2][3]．もし，世界に分布するデータの頻度がすべて明らかになり，頻度が高いデータに短いポインタを割り当てることが出来るならば，この処理はハフマン符号化で実現することが出来，全てのデータとそのポインタを保存するのに必要なストレージの容量は，すべてのデータをハフマン木を用いたエントロピー符号化による圧縮を適用した容量になる．

一方，こういった巨大なストレージを実現するためには，スケールアウトが可能な分散ストレージを利用する必要がある．しかし，一般的にこういったストレージは CAP の定理により，一貫性 (Consistency)，可用性 (Availability)，分断耐性 (Partition-tolerance) の 3 つの特性を満足することは出来ない，とされる．

1.2 関連研究

人々が利用するデータは，完全にランダムではなく大きく偏っていることが分かっている．例えば，バックアップの用途ではこの偏りを利用して，差分バックアップを行うことで，実際に利用するデータ量を削減しており，実際に最大 99 %以上の削減が実現できているとされる．また，同一のシステムの時系列的な差分だけではなく，同一の時間に複数の人が利用している環境において重複率を調べる実験も行われている．*Meyer* らは，社内で利用する環境において実際に書き込まれているデータに対して，どれぐらいの重複があるかを検証した [4]．857 のデスクトップ PC のデータを集約しブロック単位の重複排除を行ったところ，32 %のデータ削減ができ，また 4 週間分のバックアップについては 87 %の削減ができている．また，我々が 14 人のデータを集約した所，17 %のファイルが重複しており，結果として約 40 %の容量削減ができた．このように，ストレージの中身は時系列的にも空間的にも重複するがその重複率は利用用途や利用状況によって大きく異なることが分かる．

またストレージの重複除去を行うための多くの技術が提案され，すでに実装されている．例えばファイル毎の重複除去よりも効果の高い方法としてブロックレベルの重複除去がある．ファイルシステムの ZFS はあるファイルをディスクに書き込む前にファイルをブロックに分割する．ブロック単位でハッシュ関数を計算し，メモリ上に保持している過去のハッシュ値と照合，書き込み要否の判断をすることで重複除去を実現している．ブロック単位よりも細かい重複除去の単位として，可変長のチャンクを利用する方法があげられる．これらはリアルタイムで判定をすることは困難であるため，システムのアイドル時間に重複部分を発見し，不要な部分を削除する．Windows Server 2012 ではファイルによって可変サイズの小さなブロック単位で重複排除を行う．OS 内部に組み込まれている機能であるためユーザから見たファイルシステムは従来とほぼ完全に互換性を保っている [1]．またネットワークを介してデータを転送しながら重複排除を行う方法がある．これはインライン方式と呼ばれ書き込み終了時点で重複排除も終了するためバックアップと同時にレプリケーションに移ることができ RPO に優れている [7]．

しかし，すべてのデータについてこれまでのデータと比較して重複部分を検出し重複排除をする方法は一般的には非常にコストが高い．これらを全世界のデータ規模で実現するためには，データ量 n にたいして，$O(log_n)$ 程度の比較で重複排除が実現できる手法でないと，現実的ではない．

このようなデータ比較を行う手法としては，暗号学的ハッシュ関数と CAS(Content Addressing Storage)[6] を組み合わせる手法が挙げられる．つまり，ハッシュ関数を $hash()$，書かれるデータの中身を $value$ とすると，$key = hash(value)$ となるような key をポインタのアドレスとし，その実体に $value$ を書き込む．こういった手法は plan9 におけるストレージ，venti[5] で利用されている．この手法を用いると，同じデータは同じアドレスを持つため，本質的に重複排除が可能となる．そのためハッシュ関数として同じ関数を利用すれば，たとえ分散環境でお互いに共有する情報がなくても，重複を排除することができる．

つまり，ユーザはストレージに書き込みたい場合，まず $key = hash(value)$ を計算し，中央にある巨大なストレージに対して $(key, value)$ のペアを書き込み，ローカルにあるストレージにポインタとして key を書き込む．

なお，先ほどの CAP の定理においては，ストレージ側で一貫性を考慮しないことによって，スケールアウトを実現している．しかし，適切に設計された暗号学的ハッシュ関

数を用いているかぎり，同じ *key* にたいして異なる *value* が書かれることは，ある一定以下の確率で抑えられる．

1.3 暗号学的ハッシュ関数

ハッシュ関数の出力ビット長が長ければ長いほど演算に要する時間は長くなる．また，ハッシュの出力長はそのままクライアント側が保持しておくデータ長に比例する．このためシステム的な要件からは，なるべく出力ビット長が短いハッシュ関数を選択するべきである．一方，本方式は暗号学的ハッシュ関数の衝突困難性を仮定している．正しく設計された暗号学的ハッシュ関数は，互いに同じハッシュ値を出力する異なる入力を見つけることが困難であるが，本方式のような使い方をした場合，これまでに書き込まれたあらゆるデータに対するハッシュ値とも異なることが期待出来なくてはならない．本節では 256bit の出力長をもつハッシュ関数を本方式に適用すると，どの程度までこの方式を利用可能かを見積もる．

ハッシュ関数がランダムオラクルであると仮定し，提案システムに対して，2^{110} 個の異なるデータを入力した時に，そのどれもが互いにハッシュ値が衝突しない確率は，99.999999999%を上回る．つまり，2^{110} 個のデータを入力された程度では，衝突は起こらないものと見積もることが出来る．これは，次節で議論するが，ブロックのサイズを 128KByte（2^{15}Byte）としたとき，2^{128}Byte 程度のデータを保持できると見積もることができる．現在，世界中で流通しているデータは 2.8ZByte(2^{71}Byte) 程度であるとされる．毎年，データの量が 2 倍になると仮定すると，あらゆる世界中のデータをすべてこのストレージに入れて，57 年利用し続けたと仮定しても，まだ衝突しないと見積もることが出来る．

なお, 現在 checksum など多く使われている MD5(128bit) では，2^{64}Byte なので，既に衝突している可能性が無視できなく, [*1]SHA-1(160bit) では，2^{80}Byte 程度なので，2020 年代には衝突する可能性を無視できなくなる．このため，256bit の出力長とすることは妥当であると考えられる．

2012 年 12 月 NIST は，新しいハッシュ関数アルゴリズムの標準，SHA-3 として Keccak を制定した．SHA-3 は 256bit の出力長を持つため，今回の要件を満たしている．しかし，まだ制定されて時期が浅いこともあり各種言語や環境における実装が進んでいない．そのため，本検討では SHA-256 をハッシュ関数として用いた．

また，本方式の場合，アルゴリズムを変更したり，ビット長を伸ばすことも容易である．具体的には，$key = hash(value)$ としているところを，$key' = hash_1(value)|hash_2(value)$ とすればよい．これらは *value* を知っていれば誰でも計算可能であるため，システム側がアイドル時間に徐々に更新を行なってもよい．

このとき，*key*， *key*’のそれぞれにたいして *value* を設定するなど両方の *key* でアクセスすることが出来るマイグレーション期間を用意することも可能である．

1.4 本論文の構成

本節では、ネットワークの進化が進みあらゆる所からアクセスすることが出来、世界中のあらゆるデータを保存するのに十分な大きさを持つストレージの実現性について議論を行った．第 2 節では，まずそのアーキテクチャを定義しそのアーキテクチャが実現できた場合に，トータルで保存される容量以外にメリットとなることについて議論する．第 3 節では，これらをクラウド環境で利用しようとしたときに課題となる点をあげ，それらを解決するためのアーキテクチャを提案する．第 4 節では，提案アーキテクチャのプロトタイプを実装し，第 5 節では 2 節で議論したメリットが想定する環境において確認できることを示す．

2. 提案アーキテクチャ

2.1 基本的なアイデア

我々が提案するもの自体はファイル操作やバックアップシステムといったサービスを提供しない．むしろこうしたアプリケーションが使うためのストレージアクセス方法である．提案アーキテクチャにおいて，全員で共有されるのは，固定長ブロックのビット列である (実データ)．ビット列やその組み合わせの意味，コンテキストを表す情報は，そのビット列を一意に表すポインタとともに，各々が管理する (メタデータ)．

以下にこれから用いる用語について説明する．

- ブロックサイズ
  - 元データの分割単位．ここでは 128KByte.
- 実データ
  - 保存したいデータのバイナリ形式．ここでは大きさ 128KByte．*value*.
- メタデータ
  - 実データのハッシュ値．*key*.
  - ユースケースによりファイルの作成時間，更新時間，アクセス時間，ファイルサイズ，ファイル名，ACL が含まれる．
- 実データストレージ
  - 一つだけ存在する．今までのデータを全て保存するストレージ.
- メタデータストレージ
  - メタデータを保存するストレージ
- 実データ重複
  - 既に実データブロックが存在すること.

ハッシュ関数にかける実データの単位について考える．

[*1] これとは別に既に $MD5$ は衝突困難性を満たしているとは考えられていない

今回はハッシュ関数として SHA-256 を用いるため符号化後のデータ長は全て 256bit となる．圧縮率の観点や生成されるメタデータ数の観点ではデータについてより長い単位で扱うのが理想的である．しかし一方で重複率の観点からはより短いデータのほうが再利用率が高まると考えられる．我々が行った社内データ重複率の検証結果からデータの分割サイズについて 128KByte 毎とする．元データは固定長のブロックに分割され，実データとメタデータとして保存される図 2.1.

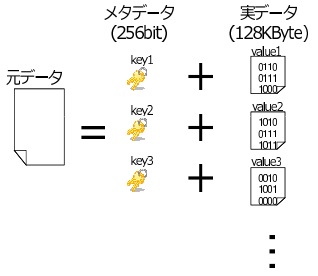

図 1 元データの保存方法

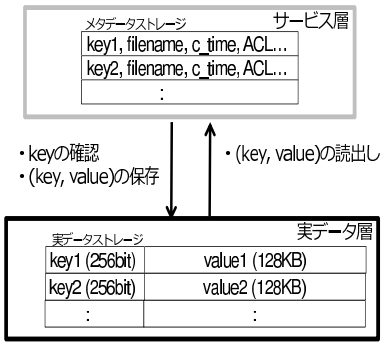

図 2 2 層のアーキテクチャ

- サービス層
  - ユーザインタフェース
  - *key* の計算
  - 実データストレージへの問い合わせ
  - 実データストレージへのデータ保存
  - メタデータの管理
- 実データ層
  - 実データの管理
  - 全ユーザから参照される追記型 key-value ストア

アーキテクチャを図 2.1 に示す．

2.2 サービス層

サービス層ではユーザから保存する元データを受け取りブロックサイズ毎に分割，実データ（*value*）とする．それぞれの *value* について $key = hash(value)$ を計算する．その *key* を用いて既に実データストレージに保存されているか確認し，存在しない場合にのみ $(key, value)$ を送信する．また *key* をメタデータストレージへ保存する．存在する場合，つまり実データの 2 回目以降の書き込みはそれを指し示す識別子のみで通信できるようにする．一つのファイルを複数のブロックに分割して保存する場合は，メタデータストレージに対して *key* のリストを保存する．サービス層においてメタデータを複数人で共有することにより，ファイル共有をすることができる．

またメタデータに ACL を付与しセキュリティを担保することも考えられる．この世界では *key* と *value* は等価である．そのため *key* を知る人は *value* を知る人であり万が一第三者に *key* が知られた場合には実データが読みだされる可能性がある．そのため *key* の暗号化など何らかのセキュリティ対策が必要となる．

2.3 実データ層

今までやり取りされた全ユーザの $(key, value)$ を全て保存する．*key* に応じて複数台に分散して保存できるためスケールアウトし易い．既存のファイルシステムでは，同じ中身だが形式が異なるファイルがいくつも存在し得るが，ユーザが保存したいデータにおける形式等の情報を含むメタデータは別に管理をしていることから，実質重複が排除されている

同じ元データでハッシュ関数が同一ならばだれが計算しても同じ *key* を計算することができ，また違うものは違うことが担保される．このように時間・空間を超えて実データのユニーク性が確保されている．一方元データが変われば *key* も変化する，append only なストレージである．

2.4 提案手法の利用例

提案手法のストレージアクセス方法を実装すれば，単なるデータ保管庫という用途だけでない使い方が可能となる．例えばデータの保存目的のオンラインオブジェクトストレージという利用でも，他のユーザや過去を超えてブロック単位でデータを再利用することができるのでディスクスペースやネットワーク帯域の効率的な利用を図ることができる．

またメタデータが git などのバージョン管理システムで管理されている場合，いつの状態にでも遡ることができるファイルシステムとして利用することができる．

実際にアプリケーションに適用する例としてはオブジェクトストレージが考えられる．離れた拠点にいるユーザ同士がメタデータサーバを共有することで共有フォルダの使い方が可能となる．逆にメタデータサーバを分けることでデータの共有を許す範囲を限定でき，例えばプロジェクト毎に利用者を設定するといったことが可能となる．別拠点のユーザとファイルを共有する場合でも，実データストレージのデータを再利用することによりデータ送信/受信量の削減や，サービスの応答時間の向上を実現できるほか，全ユーザに対して同じ手段でデータを共有することができる．またローカルファイルシステムとしての利用も考

えられる．メタデータをそれぞれ個人で管理し，実データを共有することで重複除去が可能となる．このようにアプリケーションの作り方を変えることで提案手法は様々な用途に利用可能である．

3. 提案手法の実環境での利用

3.1 問題点

前章では提案手法の一般的なアーキテクチャについて述べた．しかし提案手法を実際にクラウド環境で利用する場合には問題がある．それは WAN 環境を通ることによるレイテンシの低下である．実データストレージは利用する世界で一つしか存在しないため物理的に離れた位置に存在する場合ネットワーク遅延の問題により，レイテンシが大きく低下する可能性がある．

3.2 キャッシュの配置

上記の問題に対して，サービス層と実データ層の間にキャッシュ層を配置する (図 3.2).

- キャッシュ層
  - 実データストレージのサブセットであるキャッシュストレージを含む
  - 過去にやり取りされたデータをキャッシュする
  - 何らかのキャッシュ置き換えアルゴリズムを適用

実データの Write 時に既にキャッシュ層にデータが存在する場合は，実データ層まで確認する必要がない．もしキャッシュ層に存在しない場合は更に実データストレージへフォワードする．2 段階の key の存在確認を行いまだ登録がなかった場合にのみ実データの書き込みを行うため，ユーザに対するレスポンスを向上することができると考えられる．キャッシュストレージは実データストレージの $(key, value)$ のうち一部を保持している．また何らかのキャッシュ置き換えアルゴリズムにより使われていない $(key, value)$ の削除・入れ替えを行う．ただし今回の検証ではキャッシュストレージとして十分に大きいメモリを用意したためキャッシュの入れ替えは起こっていない．

一般的なストレージ高速化の手法として，キャッシュとストレージの階層化技術がある．キャッシュにより利用頻度の高いデータを高速な記憶装置に蓄えておくことにより，逐一低速な装置から読みだす無駄を省いて高速化を図っている．さらにキャッシュを SSD, SAS, HDD というように，高速なものから階層化することでディスクの投資対効果を高めることができる．これらの一般的なキャッシュの利点に加え，提案手法では他人が使用したキャッシュも再利用することができる．実データに対するポインタとしてハッシュ値を利用することによってシステムに対して統一された名前空間を提供しているためである．そのため見ず知らずのユーザや自分の過去であってもキャッシュを再利用することが可能となる．

また通常のキャッシュの場合，元データとキャッシュ間で一貫性の問題が起こることがある．しかし提案する KVS ストレージにおいて実データの書き換えは起こらず新たな $(key, value)$ として保存を行うので同期を取るための仕組みがいらず，実装が比較的容易である．

キャッシュの利用の方法として LAN 環境毎にキャッシュストレージを配置することが考えられる．同じキャッシュを利用するユーザは物理的位置や所属するコミュニティなどの属性が似ていると考えられ，キャッシュの内容に局所性がみられると予想されるためである．

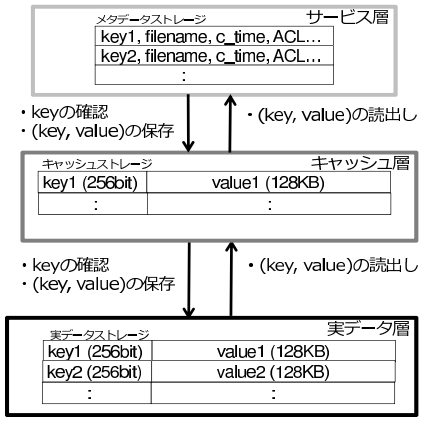

図 3 3 層のアーキテクチャ

4. プロトタイプとしての実装

4.1 想定するユースケース

提案するストレージアーキテクチャでは，アプリケーションの実装によって様々な使い方が可能となる．今回はオブジェクトストレージのユースケースについて実装を行った．

データを再利用するメリットについても議論するために扱うデータに対してシステム上の重複率を変化させた．システム上の重複率とは，キャッシュストレージまたは実データストレージ上において過去のデータが再利用できる確率である．

図 4 に評価実験におけるシナリオを示す．離れた二か所の拠点に存在する A と B が同じ共有フォルダを利用しており，A がファイルを共有フォルダ上に配置してから B が該当のファイルを取得する．共有フォルダのユースケースであるため，メタデータについては共通のメタデータストレージで管理している．クラウド環境を想定して，実データストレージとそれぞれの拠点の間は広域環境を模擬した．図 5 にシステム構成図を示す．ストレージの配置についてはそれぞれの拠点内にキャッシュストレージ，拠点外に実データストレージを配置した．疑似遅延装置としては Linux 付属のネットワークシミュレータである netem を利用した．メタデータサーバについてはそれぞれの拠点外に

配置する．

検証に用いたデータについてはあらかじめ用意した．ランダムなデータブロックを作成し，重複率の制御を行っている．

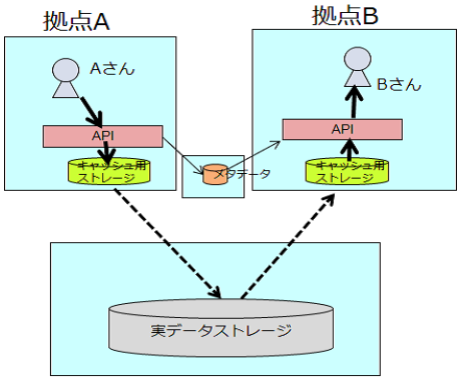

図 4 検証実験におけるユースケース：オブジェクトストレージ

4.2 評価環境

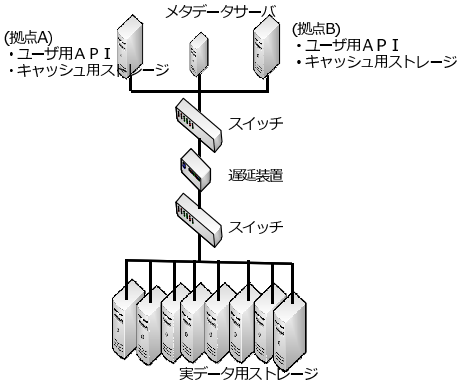

図 5 実験環境

データ用ストレージには分散 KVS の Riak-1.2.0，メタデータサーバにはインメモリ DB の Redis-2.2.0 を用いた．またキャッシュストレージには Redis-2.2.0 を改変したプログラムを利用している．

Riak，Redis はどちらも key-value ストアだが特徴が異なる．究極的には世界規模のデータを保存する必要があるデータ用ストレージにはスケーラビリティに優れた Riak，より速いレスポンスが求められるメタデータサーバおよびキャッシュサーバにはインメモリで動作する Redis を利用した．

メタデータサーバおよびキャッシュストレージについて，インメモリ DB を用いることで特に読み込み時のアクセスを高速化することができる．ディスク上に保存されている場合には性能が著しく低下する．メタデータやキャッシュについては，複数人から同時にアクセスされる可能性があり，メタデータについては書き込み時に実データが再利用可能な場合でも必ずアクセスされる．そのためインメモリ DB に配置しておくことでレスポンスの向上を図ることができる．**表 1** にそれぞれで用いたサーバのディスクとメモリ容量を示す．メタデータサーバとキャッシュストレージには十分なメモリを割り当てており，今回の検証ではキャッシュの入れ替えは起こらなかった．

表 1 各サーバのディスク，メモリの容量

	Disk	Memory
メタデータサーバ	6TByte	64GByte
キャッシュストレージ	4TByte	64GByte
実データストレージ	36TByte	16GByte

実データストレージに用いた Riak への接続の際にはパフォーマンスをより重視してプロトコルバッファを利用している．それぞれのサーバは Gigabit-ethernet で接続され，実データストレージとの間には遅延装置を介している．CPU は Intel Xeon(4core)，OS は Centos6.3 である．

以下にオブジェクトストレージとしての使い方についての Write, Read それぞれの挙動を示す．扱いたいファイルを $FileA$，$FileA$ のファイル名を $FileA_{name}$ とする．

Write

(1) サービス層でユーザから保存したい $FileA$ を受け取る
(2) $FileA$ について先頭からブロック単位（ここでは 128KByte）切り出し，暗号学的ハッシュ関数により key を計算．計算された key をもとに，キャッシュ層に問い合わせる．
(3) キャッシュストレージに存在しない場合は，サービス層から $(key, value)$ を送信　キャッシュストレージはバックグラウンドで実データストレージに書き込みを行う
(4) キャッシュ層に存在する場合はサービス層のメタデータサーバに対して $(FileA_{name}, key)$ を保存
(5) (2)〜(4) を繰り返す

メタデータについては実データと同様に $(key - value)$ 形式で保存している．ここでの key は保存したいファイル名，$value$ ブロックのハッシュ値である (複数のブロックに分割される場合は key のリスト)．

Read

(1) サービス層で読み取りたい $FileA_{name}$ を受け取る
(2) メタデータサーバへ $FileA_{name}$ で問い合わせを行い，実データの key（ファイルが複数に分割されている場合は key のリスト）を受け取る．key をキャッシュ層に問い合わせ．
(3) キャッシュストレージに存在しない場合は，実データストレージへ問い合わせ
(4) キャッシュストレージに存在する場合はサービス層に対して $(key, value)$ の組を返す
(5) 実データストレージに存在する場合はキャッシュ層に対して $(key, value)$ の組を返す．　キャッシュストレージでは $(key, value)$ を保存．キャッシュ層はサービス層に対して $(key, value)$ の組を返す．

(6) (3)〜(5) を繰り返す

5. 評価結果

実環境における提案手法のフィージビリティ確認を行った.

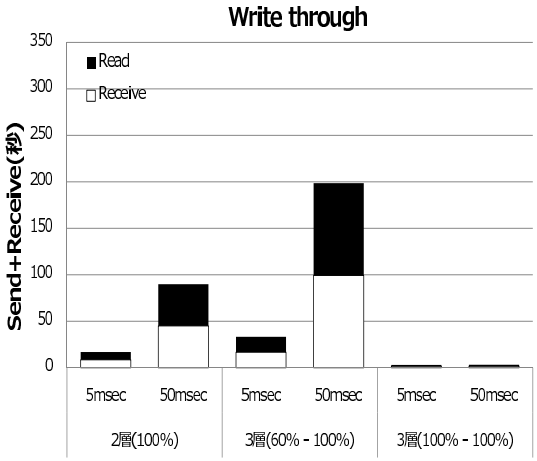

図 6 オブジェクトストレージ実装の評価 (Write through)

図 6 に A がファイルを共有フォルダ上に配置してから B が該当のファイルを取得完了するまでの実行時間を示す.

2 層と 3 層の比較において，遅延時間が大きい場合はキャッシュを配置する分 3 層の場合が速くなることを期待した．サービス層で計算した $(key, value)$ を転送する前に key を用いて既にその組が存在するかあらかじめ確認する．そのため既にキャッシュストレージに該当の組が存在する場合は実データの送信は不要であるためである．しかしキャッシュヒット率が 60 %においても広域の影響を免れず，50msec の場合では 2 層と比べて 2 倍の時間がかかっている.

一方全てのデータが 100 %キャッシュに格納されている場合は，キャッシュのみでやり取りが可能となるため広域の影響を受けないことが確認できた.

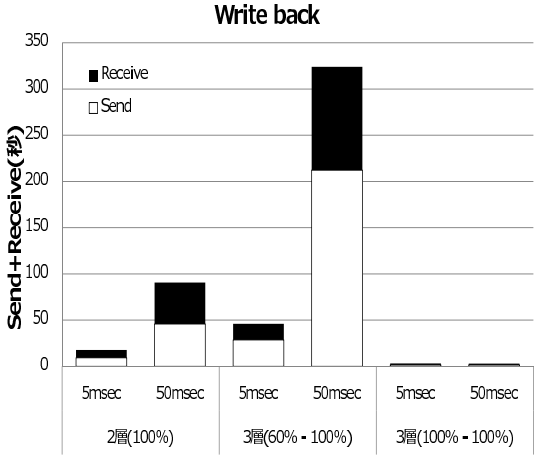

図 7 オブジェクトストレージ実装の評価（Write back）

図 7 では，図 6 に対する比較として，キャッシュ層からストレージ層へライトバックを行った場合の実行時間を示す．ライトバックの方式ではキャッシュストレージから送信した一つのデータに対して ack が返ってきたことを確認してから次のデータ送信を行う．そのためライトスルー方式に比べ実行時間が長くなる傾向にある．3 層のキャッシュヒット率 60 %のグラフからもわかる通り Read と Write の実行時間の差について，Write の比が大きくなっていることが分かる．キャッシュヒット率が 100 %の場合はライトスルー方式と同様に全てのデータをキャッシュ上のデータでやり取りできるため広域の影響を受けることなく通信が可能となる.

キャッシュヒット率が高いということは，ユーザが利用しているキャッシュにユーザ自身，または同じキャッシュを利用している他ユーザが以前やり取りしたデータが多いということを意味する．例えば会社などの特定のコミュニティにおいては似たようなデータをやり取りする機会が多いと考えられる．世界中に支店をもつ企業がどこか一か所で実データを溜めながら，それぞれの支店でキャッシュを持つといった利用方法が考えられる．また画像や音楽，映画といったコンテンツは多数のユーザが利用するものである．提案手法を用いることにより，過去に同じコンテンツを取得したユーザのデータを再利用できる可能性があり一人ずつ配信サーバから取得するよりシステム全体レイテンシやディスク使用率を大幅に向上させることが可能となる.

ライトスルーまたはライトバック方式が最適かはアプリケーションによって異なると考えられる．ライトバック方式はレスポンスは比較的遅いが，データの書き込みを確実に行うことができる．一方ライトスルー方式ではレスポンスは速いが，確実にデータ書き込みができたかの確認ができずにデータ損失の可能性が高まる.

6. おわりに

6.1 まとめ

本稿では究極的にストレージリソースが集約された世界のストレージアーキテクチャの提案を行った．データに対するポインタの作成に暗号学的ハッシュ関数を利用し，時間や空間，ユーザを超えた効率的なデータの再利用が可能な方式である.

実環境における利用では，物理的な位置関係によるレイテンシの低下が避けられないため，実データストレージとユーザの間にキャッシュストレージを配置した．WAN 越しにファイル共有を行うオブジェクトストレージのアプリケーションについて実装を行いフィージビリティを確認することができた．実利用を考えた場合，レスポンスに対してはキャッシュストレージおいてどれくらいのデータが再利用可能かが大きく影響する.

今後 VDI 環境やデータのバックアップなど，ストレージアクセスが頻繁に起こるアプリケーションの需要が伸びていくと考えられる．更に今後これらの使い方はより物理的

な位置を意識しない使い方へ移行していくと考えられる．そのため WAN 環境に影響を受けないストレージアーキテクチャを検討することは有意義である．

6.2 今後の課題

一般的なストレージとしての評価を進める．例えば同時アクセス性の評価や，使い方毎のワークロードを想定したベンチマークを利用して性能を明らかにする．

システム構成においてサービス層からのリクエストはシーケンシャルに行っているが，並列処理を行っていく．それにより特にライトバック方式の性能を高めることができると考えられる．

実データストレージではその世界一つだけ存在し，やり取りされたデータ全てを保存している．しかしリソースには限界があるため，実データストレージ上のガーベージコレクションにより参照されなくなった $(key, value)$ を削除する機能を検討する必要がある．

検証で示したように特に局所的なデータの偏りがある場合に強みとなる．そのためアプリケーションの種類や利用される範囲毎にデータの重複排除率について明らかにし提案手法が適用できるユースケースについての考察を行う．

参考文献

[1] Ahmed El-Shimi, Ran Kalach, Ankit Kumar, Adi Oltean, Jin Li, and Sudipta Sengupta. Primary data deduplication-large scale study and system design. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pp. 26–26, Berkeley, CA, USA, 2012. USENIX Association.

[2] Keren Jin and Ethan L. Miller. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, pp. 7:1–7:12, New York, NY, USA, 2009. ACM.

[3] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezise, and Peter Camble. Sparse indexing: large scale, inline deduplication using sampling and locality. In *Proccedings of the 7th conference on File and storage technologies*, FAST '09, pp. 111–123, Berkeley, CA, USA, 2009. USENIX Association.

[4] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. In *Proceedings of the 9th USENIX conference on File and stroage technologies*, FAST'11, pp. 1–1, Berkeley, CA, USA, 2011. USENIX Association.

[5] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. In *Proceedings of the Conference on File and Storage Technologies*, FAST '02, pp. 89–101, Berkeley, CA, USA, 2002. USENIX Association.

[6] Sean Rhea, Russ Cox, and Alex Pesterev. Fast, inexpensive content-addressed storage in foundation. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pp. 143–156, Berkeley, CA, USA, 2008. USENIX Association.

[7] Kiran Srinivasan, Tim Bisson, Garth Goodson, and Kaladhar Voruganti. idedup: latency-aware, inline data deduplication for primary storage. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, FAST'12, pp. 24–24, Berkeley, CA, USA, 2012. USENIX Association.

付　　録

A.1 定理

提案システムに対して、2^{110} 個の異なるデータを入力した時に、そのどれもが互いにハッシュ値が衝突しない確率は、99.999999999%を上回る。

A.2 証明

ハッシュ関数がランダムオラクルであると仮定する．E_k を k 回目の入力データが、これまでに入力したどの $k-1$ 回目の入力とも異なる事象であるとする。今、提案システムに対して m 回の入力を行った。この時、少なくとも一回以上の衝突が起きている確率は、$Pr(\bar{E_1} \cup \bar{E_2} \cup ... \cup \bar{E_m})$ で表される。

$$\begin{aligned} Pr(\bar{E_1} \cup \bar{E_2} \cup ... \cup \bar{E_m}) &\leq \sum_{i=1}^{m} Pr(\bar{E_i}) \\ &= \sum_{i=1}^{m} \frac{i-1}{2^{256}} \\ &= \frac{m(m-1)}{2^{256}} \end{aligned}$$

このような $Pr(\bar{E_1} \cup \bar{E_2} \cup ... \cup \bar{E_m})$ が 10^{-11} を下回るためには、m が、$\frac{m(m-1)}{2^{256}} \leq 10^{-11}$ を満たせばよい。$2^{-36} \leq 10^{-11}$ であるため $m \leq 2^{110}$ の時、上記条件を満たす。