Vol.2016-OS-136 No.1
2016/2/29

クラスタファイルシステムにおける ファイル配置のためのハッシュ空間分割手法

芝 公仁[1,a)]　丈達 翔太[1,b)]

概要：ハッシュテーブルを用いたクラスタファイルシステムでは，ハッシュ空間を複数のハッシュ領域に分割し，それらの領域をクラスタを構成するノードに割り当てる．各ノードは割り当てられたハッシュ領域に属するハッシュ値を持つファイルを管理する．このようなハッシュ空間の分割は，システムに新たにノードを追加するたびに行われる．この処理が適切に行われない場合，多くのファイルをノード間で移動させることが必要になったり，一部のノードに負荷が集中したりすることがある．これらは，システムの性能を低下させる要因となる．本稿では，クラスタファイルシステムで使用されるハッシュ関数の特性，および，ハッシュ空間の分割手法について述べる．本手法により，ノード追加時のファイルの移動や，ノード間での負荷の偏りを低減させることが可能になる．

1. はじめに

複数のノードをまとめて計算機クラスタを形成し，それを用いてファイルサーバを稼働させる機会が多くなってきている．このようなクラスタ型のファイルシステムでは，多数のファイルを適切に複数のノードに配置し管理することが必要になる．これを実現する手法のひとつとして，ハッシュテーブルを用いるものがある．

ハッシュテーブルを用いたクラスタファイルシステムでは，ハッシュ空間を複数のハッシュ領域に分割し，それらの領域をクラスタを構成するノードに割り当てる．各ノードは割り当てられたハッシュ領域に属するハッシュ値を持つファイルを管理する．通常，クラスタファイルシステムは，ハッシュ空間を等分に分割することで，各ノードが担当するファイル数が均等になるようにしている．これによって，各ノードの負荷が均衡することを期待している．また，クラスタファイルシステムの利点として，ノードの追加により，システムの性能を動的に向上させることができるといったものがあげられる．これは，ハッシュ空間をより多くの領域に分割することによって，追加されるノードを含めた全ノードに対して，担当する領域の再設定を行うことで実現される．

このようなハッシュテーブルを用いてファイル管理を行うクラスタファイルシステムは，ファイルのハッシュ値が，ハッシュ空間内に均一に分布することを前提としている．しかし，ハッシュ値がハッシュ空間内で偏った場合，ハッシュ空間を等分に分割するのみでは，適切な負荷分散を行うことができない．また，ノードの追加によるハッシュ空間の再分割，再割り当て行われると，各ノードが担当する領域が変化するため，ノード間で，ファイルを移動させる処理が必要になる．ハッシュ空間の分割やノードへの割り当てが適切に行われなかった場合，多くのファイル移動の処理が必要になり，その負荷が無視できないほど大きくなる．また，ファイル再配置後，ファイルの分散が適切に行われていないと，一部のノードに負荷が偏るといった問題が発生する．これらは，ファイルサーバの性能を低下させる原因となる．

このように，ハッシュテーブルを用いたクラスタファイルシステムでは，ハッシュ空間の分割やノードへの割り当ては，システムの性能に影響する重要な問題である．本稿では，まず，既存のシステムで用いられているいくつかのハッシュ関数について調べ，ファイル管理のために算出されるハッシュ値がハッシュ空間内に均一に分布しない場合があることを指摘する．また，ノード追加時のファイル移動の負荷や，ノード間での負荷の偏りを軽減させるための，ハッシュ空間の分割手法について述べる．

以下，本稿では，2 章で関連研究，3 章でクラスタファイルシステムでのハッシュ値の利用とハッシュ関数の評価について述べる．また，4 章でハッシュ空間の分割手法，5 章で分割手法の評価を行う．

[1] 龍谷大学
Ryukoku University
[a)] shiba@rins.ryukoku.ac.jp
[b)] s_joutatsu@www.vii.ss.i.ryukoku.ac.jp

2. 関連研究

これまでに，ネットワークで接続された複数の機器でファイルを管理，利用するシステムが多く提案されてきている [1]．特に，1 台の計算機では管理できないような大量のデータを管理することを目的に，複数の計算機から構成されるクラスタでファイル管理を行えるようにするクラスタファイルシステム [2] が多く利用されるようになってきている．

クラスタファイルシステムでは，ネットワークで接続された複数のノードを協調させ，それらのノードのストレージ資源を活用するファイルシステムを実現する．ファイルへのアクセスはネットワークを介して行われるが，通常，どのノードにファイルの実体があるかを意識する必要はなく，位置透過にファイルを利用することが可能である．また，ノードを追加することで，容易に容量の拡張が行えるといった利点もある．このようなクラスタファイルシステムの例として，ストレージのフェイルオーバによる高可用性を実現する Lustre[3] やディスク I/O を積極的に利用することで高性能を実現する Gfarm[4], [5]，ストレージの高いスケーラビリティと大容量のファイルデータの入出力の高速性を実現する Ceph [6], [7], [8] などがある．

また，ファイル管理だけではなく，その処理までを考慮するシステムもある．その例として，データを並列分散処理する Map-Reduce で使用することを想定している HDFS[9] が挙げられる．さらに，ファイルシステムではなく，仮想化環境での仮想ストレージとして利用することを想定した分散オブジェクトストレージ [10] も提案されている．

クラスタファイルシステムのファイル管理方式にはいくつかの種類がある．例えば，ファイルシステムを，ファイルのメタデータの管理するメタデータサーバと，ファイルの内容を保持するデータサーバに分割して実現する方法がある．このような構成をとることで，メタデータサーバでの集中管理を行うことが可能になる．しかし，このような構成は，メタデータサーバが，性能のボトルネックになったり，単一障害点となったりする場合がある．

このような問題点を解決するため，メタデータサーバを利用せず，ハッシュテーブルを用いてファイルの位置を管理するシステムもある [11]．本研究では，このようなハッシュテーブルを用いてファイル管理を行うクラスタファイルシステムを対象としている．

3. ハッシュテーブルを用いたクラスタファイルシステム

ハッシュテーブルを用いたクラスタファイルシステムでは，ファイルを保持するノードの決定をハッシュ値を用いて行う．ハッシュ空間をノード数と同じ数の領域に分割する．この領域をハッシュ領域と呼び，それぞれを各ノードに割り当てる．各ファイルはファイル名などから算出されるハッシュ値を持ち，その値が属するハッシュ領域に割り当てられたノードが当該ファイルを管理する．

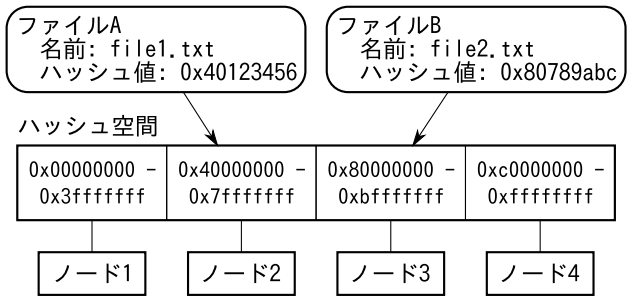

図 1 ハッシュ値を用いたファイル管理

ハッシュ値を用いてファイルを管理するノードを決定する例を**図 1** に示す．図中のシステムでは，このシステムでは，ノード 1 からノード 4 の 4 つのノードを使用してファイルを保持する．この例では，32 ビットのハッシュ空間を 4 つのハッシュ領域に分割し，各領域を 4 つのノードにひとつずつ割り当てている．ファイル A とファイル B をクラスタファイルシステムに追加することを考える．ファイル A のファイル名は，file1.txt であり，このファイル名のハッシュ値は 0x40123456 である．この値は分割された 2 つめのハッシュ領域内にあり，この領域を担当するノード 2 にファイル A が管理される．また，同様に，ファイル B のハッシュ値 0x80789abc は 3 つめの領域にあり，ノード 3 によって管理される．

ファイルの読み出し時には，ファイル名からハッシュ値を算出し，その値からファイルを持つノードを調べる．例えば，file1.txt を読み出す場合，そのハッシュ値 0x40123456 から当該ファイルがノード 2 にあることがわかる．

3.1 ハッシュ関数

ハッシュテーブルを用いたクラスタファイルシステムでは，ファイルを管理するノードの決定にファイルのハッシュ値を使用する．したがって，ファイルを適切に各ノードに分散させるためには，ハッシュ値を算出するハッシュ関数が適切なものでなければならない．

一般に，ハッシュ関数は，元のデータが少しでも異なれば，異なるハッシュ値を持つようにし，異なるデータが同一のハッシュ値を持つ衝突が起こらないようにしている．通常，クラスタファイルシステムは，ハッシュ関数によって算出されるハッシュ値が，ハッシュ空間内に均一に分布することを期待している．もし，偏りが発生するようなハッシュ関数を用いると，システム内の一部のノードが多くのファイルが担当することになり，負荷の偏りが生じる．これは，クラスタファイルシステム全体の性能低下の原因となる．

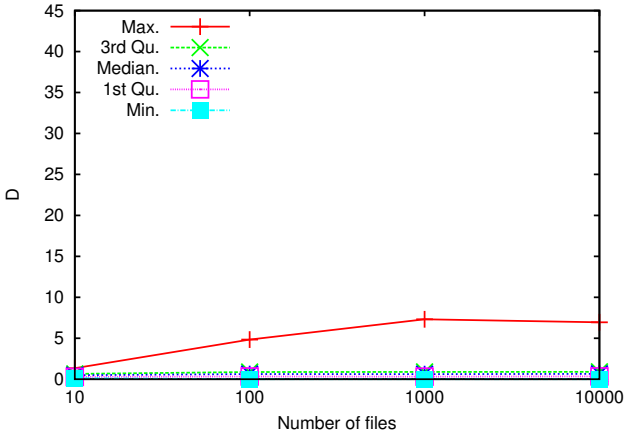

図 2 （A）のハッシュ関数によるファイルセット 1 の D の分布

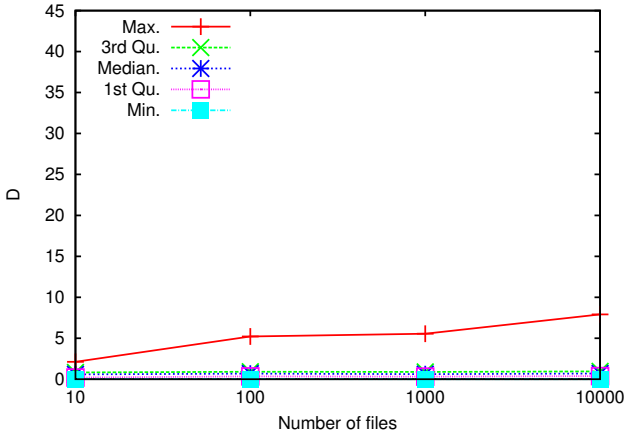

図 4 （B）のハッシュ関数によるファイルセット 1 の D の分布

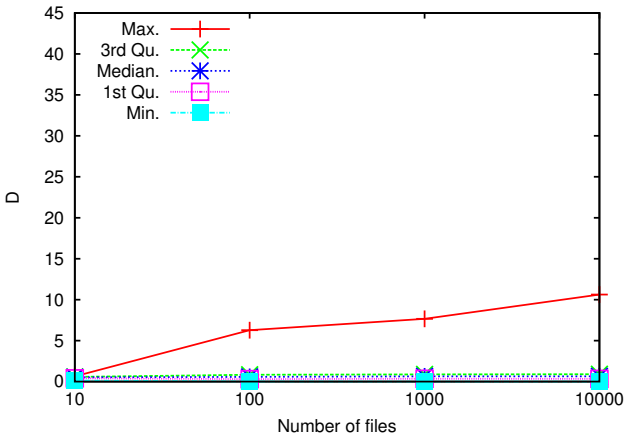

図 3 （A）のハッシュ関数によるファイルセット 2 の D の分布

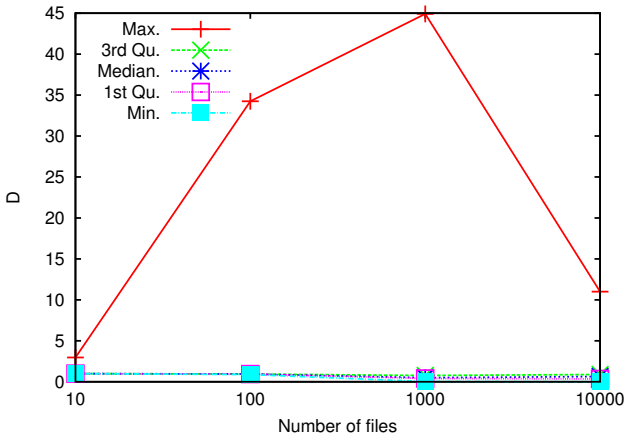

図 5 （B）のハッシュ関数によるファイルセット 2 の D の分布

このように，ハッシュ関数の性能が，クラスタファイルシステムの負荷分散の性能を決める重要な要素となる．実際のシステムで使用されているハッシュ関数が，ファイルをどの程度ハッシュ空間内に分散させられるかを調べる実験を行った．本実験では，次の 3 つのシステムで使用されているハッシュ関数を使用した．

（**A**） GlusterFS: クラスタファイルシステム
（**B**） Ceph: 分散ストレージシステム
（**C**） Sheepdog: 分散オブジェクトストレージ

また，ハッシュ値の算出に使用するファイルとして，次の 2 つを用いた．

ファイルセット 1 Debian–8.2 の/usr ディレクトリ内のファイル

ファイルセット 2 画像コンテンツサーバの画像ファイル

クラスタファイルシステムが n 個のファイルを持つとき，各ファイルのハッシュ値を h_i とする．ただし，$h_1 \leq h_1 \leq ... \leq h_n$ である．このハッシュ値が理想的なものとどの程度異なるかを表す集合 D を次のように定義する．

$$D = \left\{ \frac{|\, h_{i+1} - h_i - d \,|}{d} \;\middle|\; i = 1, 2, ..., n-1 \right\}$$

これは，ハッシュ値の差をハッシュ空間内でのファイル間の距離とし，理想的なファイル間の距離に対する，実際の距離との差の割合の集合である．d は，ハッシュ値が理想的に分布した場合のハッシュ値の差であり，ハッシュ空間のサイズ / 総ファイル数である．

ファイルセット 1 とファイルセット 2 について，ファイル数を 10，100，1000，10000 個としたときの，D の要素の 5 数要約を求め分布を調べた．（A）から（C）のハッシュ関数を用いたときの D の分布を，それぞれ**図 2** から**図 7** に示す．

すべての場合について，ファイル数が増えると D の最大値が大きくなっている．しかし，全体としては，偏りは少ないと考えられる．ファイルセット 1 とファイルセット 2 を比べると，ファイルセット 2 の方が，D の最大値が大きくなっている．これは，ファイルセット 1 のファイル名が英数字から構成され，その長さも様々であるのに対し，ファイルセット 2 のファイル名は，数字が多く使用され，その長さも一定であるためだと考えられる．すなわち，ファイル名の文字列自体に偏りがあり，それから算出されるハッシュ値にも偏りが生じた可能性がある．

このような，使用される文字列が数字に偏っていたり，

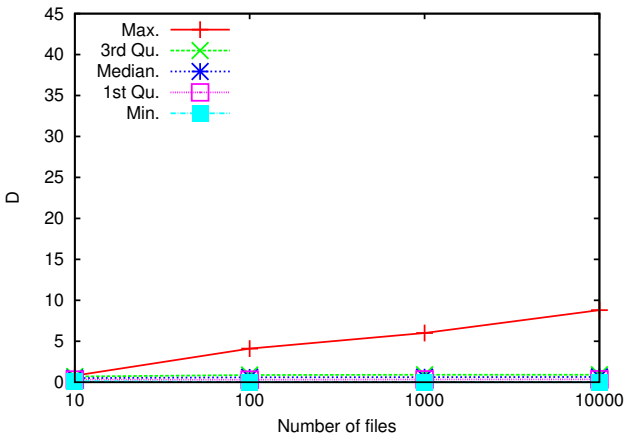

図 6 （C）のハッシュ関数によるファイルセット 1 の D の分布

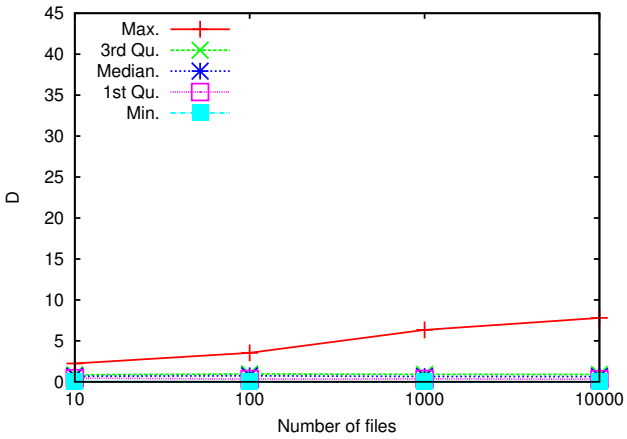

図 7 （C）のハッシュ関数によるファイルセット 2 の D の分布

ファイル名の長さが一定であったりすることは，コンテンツサーバなどで，コンテンツを連続した数値のファイル名で保持する場合などによく見られる．このようなシステムでクラスタファイルシステムが使用されることは少なくない．ファイル名が偏っていても，ハッシュ関数を用いれば，偏りをなくし負荷分散を実現できるとは必ずしも言えないため問題となる．

4. ハッシュ空間の分割

前章では，一般的なハッシュテーブルを用いたクラスタファイルシステムについて述べた．本章では，GlusterFS を例にあげ，実際の運用を想定し，クラスタファイルシステムでのハッシュ空間の分割手法と負荷について述べる．

4.1 GlusterFS

クラスタ型のファイルシステムとして，GlusterFS がある．GlusterFS は，複数のサーバノードから構成される計算機クラスタを用いて，ファイルサーバを実現する．また，クライアントノードからファイルサーバのファイルにアクセスするためのモジュールも提供する．

GlusterFS では，ファイルを格納する volume が作成され，クライアントノードは，これをマウントし，当該 volume のファイルにアクセスする（**図 8**）．volume は，サーバノードが持つ brick を束ねたものである．複数のノードに brick を作成し，それらを 1 つの volume にまとめることで，複数のノードを単一のファイルサーバとして利用することができる．brick の実体は，サーバノード上のディレクトリであり，当該ディレクトリ内のファイルがクライアントに提供される．すなわち，GlusterFS におけるファイルは，通常のファイルシステム上のファイルであり，GlusterFS 上にファイルを作成すると，いずれかのサーバノードのファイルシステム上に対応するファイルが作成される．

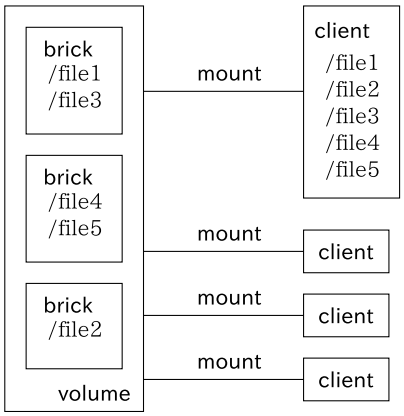

図 8 GlusterFS の概要

ファイルをどの brick に配置するかを決定するために，ファイル名から算出される 32 ビットのハッシュ値が用いられる．32 ビットのハッシュ空間は，volume が持つ brick の数のハッシュ領域に分割され，各領域が 1 つの brick に割り当てられる．各 brick は，割り当てられたハッシュ領域内のハッシュ値を持つファイルの実体を保持する．

ファイルサーバに新たにファイルを追加するときのクライアントノードの動作は次のようになる．

(1) 追加するファイルのファイル名からハッシュ値を計算する．
(2) 算出したハッシュ値から，追加するファイルを管理するノードを求める．
(3) 求めたノードにファイル追加の要求を出す．

このように，クライアントは計算機クラスタの中から必要なノードを自身で求めるため，サーバノード群を統合管理するメタサーバは不要となっている．

GlusterFS では，動作中の volume に新たに brick を追加することが可能である．追加した brick にもファイルが格納されるようにするには，当該 brick に担当するハッシュ領域を割り当てる必要がある（図 9）．このような，volume 内でのハッシュ空間の分割の再設定をリバランスと呼ぶ．リバランスでは，追加された brick だけではなく，既存の brick が担当するハッシュ領域の範囲も変わる．その結果，brick 間でファイルの移動が行われる可能性がある．この

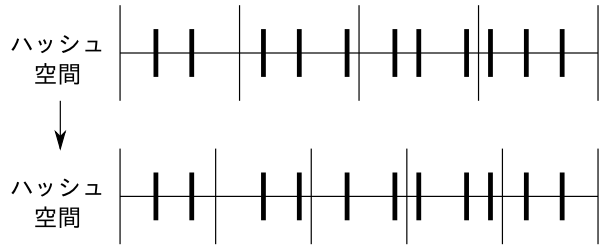

図 9　brick の追加

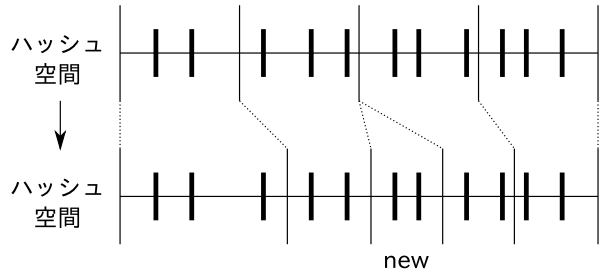

図 10　ファイル数均等分割

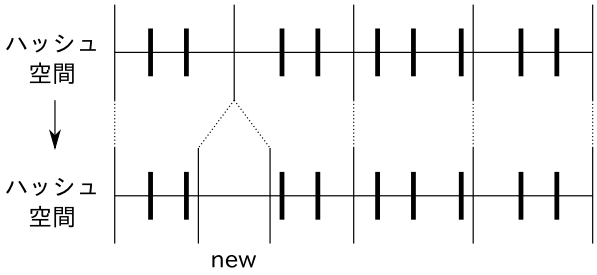

図 11　最大空き領域分割

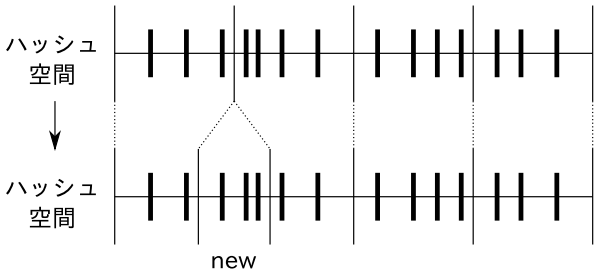

図 12　n ファイル移動分割（n=3）

とき，次のような問題が発生する．

- 多数のファイルの移動が行われ，ファイルサービスの性能が低下する．
- ハッシュ空間の分割に偏りが生じ，各 brick の利用効率が低下する．

リバランスの結果，各 brick が担当するハッシュ値の範囲が大きく変わると，brick 間のファイルの移動が多く発生する．このとき，必要になるディスクの読み書きや，ネットワークを介したファイルの送受信は，ファイルサービスで使用する資源と同一のものとなることが多い．そのため，ファイルの移動ができるだけ発生しないよう，ハッシュ空間を分割することが望ましい．例えば，既存のファイルをいっさい含まないハッシュ領域を，追加する brick に割り当てることによって，ファイルの移動が行われないリバランスを行うことができる．

しかし，1 つもファイルを担当しない brick は，ファイルサーバの動作に貢献しておらず，新たに brick を追加しても効果が得られないことになる．また，ファイルを含まない連続したハッシュ値の範囲は十分な大きさを持たないことが多い．この場合，ファイルサーバに新たにファイルが生成されても，追加された brick が担当する可能性は低く，ファイルサーバ全体としては負荷に偏りが生じてしまう．

リバランス時のハッシュ空間の分割において考慮すべきことは，次の 2 点である．

- ファイルの移動ができるだけ少なくなるようハッシュ空間の分割，割り当てを行うこと
- 各 brick が担当するハッシュ領域ができるだけ同じ大きさであること

リバランスにおいて，brick 間のファイルの移動が必要になると，システムの負荷が高くなる．ファイルの移動が少なくなるようにするためには，ハッシュ空間の分割において，リバランス前と後で，各 brick ができるだけ同じファイルを担当するようにすればよい．

各 brick が持つハッシュ領域の大きさが同じであると，各々の brick に格納されるファイルの数がほぼ同じになることが期待される．また，リバランス後にファイルが追加・削除されても，各 brick が持つファイルの数は近いものになると考えられる．この場合，ファイルサーバの負荷は，各 brick に均等に分散されると考えられる．

以下，本章では，新たに brick を追加するときのハッシュ空間の分割，割り当ての手法について述べる．

4.2　ファイル数均等分割

ファイル数均等分割は，GlusterFS が標準で行うハッシュ空間を等分に分割する方法（ハッシュ空間均等分割と呼ぶ）に近い効果が期待できる分割手法である（**図 10**）．ファイル数均等分割では，リバランス時にハッシュ空間内でのファイルの分散を調べ，各 brick が持つファイルの数が等しくなるように，ハッシュ値の担当範囲を割り当てる．ファイルサーバは，ハッシュ値の担当範囲の割り当てを次の手順で行う．

(1) ファイルサーバ内のファイル総数を調べ，各 brick が持つべきファイル数を決定する．
(2) ハッシュ空間の先頭から，割り当てるべき数だけのファイルを含むように，ハッシュ空間を分割していく．
(3) 分割された各ハッシュ領域を brick に割り当てる．

ファイルの数をもとに，ハッシュ値の担当範囲を決めるが，ファイル名から算出されるハッシュ値は，ハッシュ空間内に一様に分布することが期待される．そのため，ファイル数が均等になるようにハッシュ空間を分割することによって，結果として，各 brick が担当するハッシュ値の範

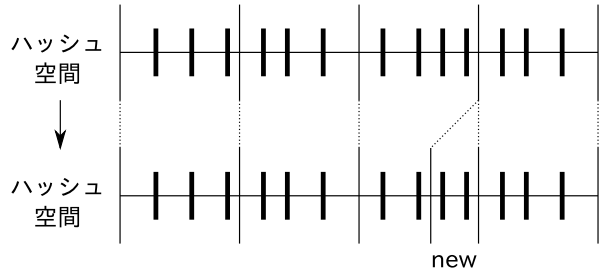

図 13　1-brick 分割

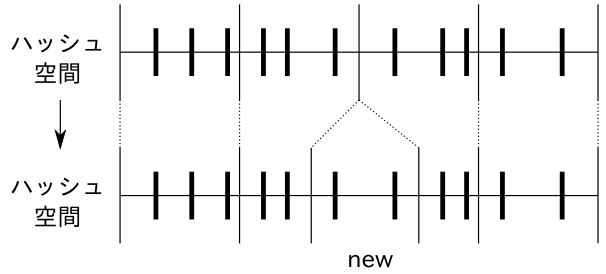

図 14　n-brick 分割

囲の大きさが等しくなると考えられる．

分割されたハッシュ空間の各領域を割り当てる際は，リバランス前と同じ順で brick に割り当てる．ただし，新たに追加する brick は，全 brick の中央になるようにする．このようにすることで，リバランス前と後で，各 brick が担当するハッシュ値の範囲が近いものになるようにし，リバランスで必要になるファイルの移動が少なくなるようにする．

4.3　最大空き領域分割

最大空き領域分割では，ハッシュ空間内でファイルが配置されていない連続した領域のうち最も大きなものを，追加する brick に割り当てる（**図 11**）．ファイルサーバは，ハッシュ値の担当範囲の割り当てを次の手順で行う．

(1) ハッシュ空間内でのファイルのハッシュ値の分布を調べる．
(2) 各ハッシュ値の間隔が最も大きい領域を選出する．
(3) 選出された領域を含む brick から当該領域を取り除く．
(4) 選出された領域を追加される brick に割り当てる．

（3）において，いくつかの既存の brick が担当する領域がなくなったり，追加される brick が担当する領域より小さくなったりする場合には，追加および変更される brick で，選出された領域を等分するように割り当てを行う．

このように，ファイルが含まれていない領域のみを既存の brick から追加される brick に割り当てるため，リバランスにおいてはファイルの移動が発生しない．しかし，追加される brick に割り当てられる領域が小さくなることが多く，また，担当するハッシュ値の範囲の大きさに偏りが生じるという問題がある．

4.4　n ファイル移動分割

n ファイル移動分割では，追加する brick に移動するファイルの数が n となるように，ハッシュ空間の分割を行う（**図 12**）．最大空き領域分割では，ファイルを含まない最大領域を探し，追加する brick に割り当てたが，n ファイル移動分割では，n 個のファイルを含むできるだけ大きな領域を探し追加する brick に割り当てる．このようにハッシュ空間の再分割を行うことで，担当範囲の変更を一部の brick に限ることができ，また，ファイルの移動を，追加される brick へ移動する n 個のみにすることができる．

本手法では，移動するファイル n を適切に決める必要がある．適切な値を用いることによって，移動するファイル数が少なく，かつ，各 brick が担当するハッシュ値の範囲の大きさの偏りを小さくすることができる．

4.5　1-brick 分割

1-brick 分割では，最も多くのファイルを持つ brick を分割し，追加する brick に割り当てる（**図 13**）．1-brick 分割では，ファイルサーバは次のように動作する．

(1) 各 brick が持つファイル数を調べる．
(2) 最も多くのファイルを持つ brick を選出する．
(3) 選出された brick の担当するハッシュ値の範囲を 2 等分し，これらを選出された brick と追加される brick に割り当てる．

担当するハッシュ値の範囲が変更されるのは，（2）で選出された brick のみであり，移動が必要なファイルは，追加された brick へ移動するもののみである．そのため，リバランス時のファイルの移動を少なくすることができる．しかし，担当するハッシュ値の範囲の大きさに大きな偏りが生じる場合がある．

4.6　n-brick 分割

n-brick 分割では，連続する n 個の brick 群の中で最も多くのファイルを持つもののハッシュ領域を再分割し，追加する brick に割り当てる（**図 14**）．n が 2 の場合，n-brick 分割では，ファイルサーバは次のように動作する．

(1) 各 brick が持つファイル数を調べる．
(2) 最も多くのファイルを持つ brick の組を選出する．
(3) 選出された brick の組が担当するハッシュ値の範囲を 3 等分し，これらを選出された 2 つの brick と追加される brick に割り当てる．

追加する brick は，選出された 2 つの brick の間に挿入される．このようにすることによって，ファイルの移動は追加された brick への移動のみとなる．n-brick 分割では，ハッシュ空間の分割が，1-brick 分割より大きく変更される．また，影響を受ける brick が n 個になり，リバランス時に必要になるファイル移動の数が多くなるが，担当するハッシュ値の範囲の大きさの偏りは少なくなる．

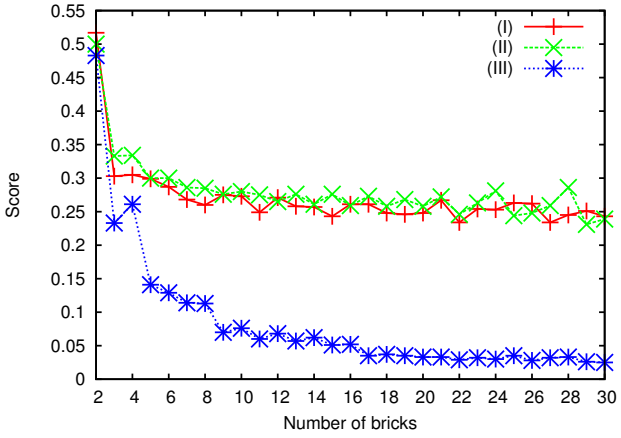

図 15　brick 追加時の S_1 の変化

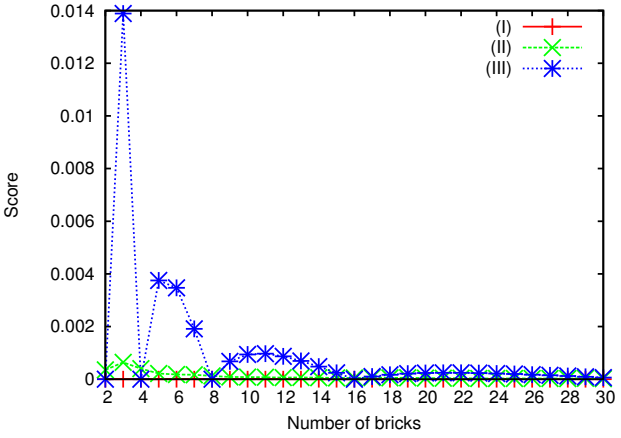

図 17　brick 追加時の S_3 の変化

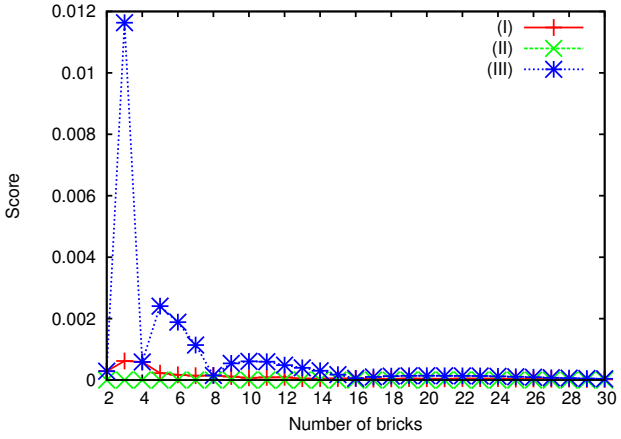

図 16　brick 追加時の S_2 の変化

5. 評価

ハッシュ空間の分割手法の評価を次の 3 点から行う．

- リバランスにおいて brick 間で移動したファイルの数
- 各 brick が担当するファイル数の偏り
- 各 brick が担当するハッシュ領域大きさの偏り

これらを評価するスコア S_1，S_2，S_3 を次の式で求める．

$$S_1 = \frac{n}{F}$$

$$S_2 = \frac{1}{B}\sum_{i=1}^{B}(\frac{f_i}{F} - \frac{\overline{f}}{F})^2$$

$$S_3 = \frac{1}{B}\sum_{i=1}^{B}(\frac{s_i}{H} - \frac{\overline{s}}{H})^2$$

n は移動したファイルの数，F は総ファイル数，f_i は i 番目の brick が担当するファイルの数，$\overline{f}$ は各 brick が担当するファイルの数，B は brick の数，s_i は i 番目の brick が担当するハッシュ領域の大きさ，$\overline{s}$ は各 brick が担当するハッシュ領域の大きさの平均値，H はハッシュ空間の大きさである．

S_1 は，リバランスにおいて brick 間での移動が行われるファイルの割合であり，ファイルの移動がない場合に 0 になり，すべてのファイルが移動した場合に 1 になる．この値が大きい場合，リバランス時のファイル移動の負荷が高いと言える．

S_2 は，各 brick について，全ファイル数に対する担当するファイル数の割合を求め，それが brick 間でどの程度偏っているかを表す．S_2 は，すべての brick が同一のファイル数を担当する場合に 0 になり，いずれかひとつの brick がすべてのファイルを担当する場合に最大となる．

S_3 は，S_2 と似ているが，担当するファイル数ではなく，担当するハッシュ領域の大きさを評価する．すなわち，各 brick について，ハッシュ空間の大きさに対する担当するハッシュ領域の大きさの割合を求め，それが brick 間でどの程度偏っているかを表す．S_3 は，すべてのハッシュ領域が同一の大きさである場合に 0 になり，いずれかひとつのハッシュ領域がハッシュ空間全体となる場合に最大となる．

4 章で述べたハッシュ空間の分割手法の違いを評価するため，各分割手法について，1 個の brick で 1000 個のファイルを管理する状態に，brick を 1 個ずつ追加したときのスコアを調べた．この実験では，ファイルとして 3 章のファイルセット 2，ハッシュ関数として 3 章の（A）のものを用いた．

次の 3 つ分割方法について，スコア S_1，S_2，S_3 の変化を，それぞれ**図 15**，**図 16**，**図 17** に示す．各図のグラフは，横軸が brick 追加後の brick 数，縦軸が各スコアの値である．

（I）　ハッシュ空間均等分割

（II）　ファイル数均等分割

（III）　1-brick 分割

S_1 による移動するファイル数の評価では，ハッシュ空間均等分割とファイル数均等分割が同程度であるのに対して，1-brick 分割では，スコアが小さくなっている．1-brick 分割では，既存の 1 つ brick を 2 つに分割することで brick の追加を実現するため，ファイルの移動がこれら 2 つの

brick 間でのみ行われる．追加される brick への移動のみとなるため，移動ファイル数が少なく，スコアが小さくなっている．

S_2 によるファイル数の偏りの評価では，1-brick 分割では，brick 数が少ないときに，大きく偏りが生じることが分かる．これに対し，ファイル数が均等になるようハッシュ空間を分割するファイル数均等分割では，S_2 がほぼ 0 になる．ハッシュ空間均等分割では，S_2 が 0 になることはなく，偏りが生じている．これは，ハッシュ関数から算出される値が均一に分布していないためであると考えられる．

S_3 によるハッシュ領域の大きさの偏りの評価は S_2 と似ており，1-brick 分割では，brick 数が少ないときに大きく偏りが生じている．ハッシュ空間を等分するハッシュ空間均等分割では偏りが発生しなかった．これに対し，ファイル数均等分割では，S_3 が 0 になることはなかった．これは，ハッシュ関数やファイル名の特性に応じて，ハッシュ空間が分割されたと考えることができ，必ずしも負荷の偏りの原因となるとは言えない．

6. おわりに

本稿では，ハッシュテーブルを用いてファイル管理を行うクラスタファイルシステムでのハッシュ空間の分割方法について述べた．既存のシステムで用いられているいくつかのハッシュ関数について調べ，ファイル管理のために算出されるハッシュ値がハッシュ空間内に均一に分布しない場合があることを指摘した．また，ファイルとノードの割り当てを決めるハッシュ空間の分割方法を提案した．本手法によって，クラスタファイルシステムにおいて，リバランス時の負荷やノード間の負荷の偏りを低減させることが可能となる．

参考文献

[1] Levy, E. and Silberschatz, A.: Distributed File Systems: Concepts and Examples, *ACM Comput. Surv.*, Vol. 22, No. 4, pp. 321–374 (1990).
[2] Sandhu, H. S. and Zhou, S.: Cluster-based File Replication in Large-scale Distributed Systems, *SIGMETRICS Perform. Eval. Rev.*, Vol. 20, No. 1, pp. 91–102 (1992).
[3] Lustre.: http://wiki.lustre.org/index.php/Main_Page/.
[4] Tatebe, O., Hiraga, K. and Soda, N.: Gfarm Grid File System, *New Generation Computing*, Vol. 28, No. 3, pp. 257–275 (2010).
[5] 高橋一志，佐々木慎，　松宮遼，大山恵弘：分散ファイルシステム Gfarm の協調キャッシュ機構の検討，技術報告 6，電気通信大学／独立行政法人科学技術振興機構, CREST, 電気通信大学, 電気通信大学, 電気通信大学／独立行政法人科学技術振興機構, CREST (2014).
[6] Ceph.: http://ceph.com/community/.
[7] Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. E. and Maltzahn, C.: Ceph: A Scalable, High-performance Distributed File System, *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, USENIX Association, pp. 307–320 (2006).
[8] Wang, F., Nelson, M., Oral, S., Atchley, S., Weil, S., Settlemyer, B. W., Caldwell, B. and Hill, J.: Performance and Scalability Evaluation of the Ceph Parallel File System, *Proceedings of the 8th Parallel Data Storage Workshop*, PDSW '13, ACM, pp. 14–19 (2013).
[9] Shvachko, K., Kuang, H., Radia, S. and Chansler, R.: The Hadoop Distributed File System, *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pp. 1–10 (2010).
[10] Sheepdog.: http://sheepdog.github.io/sheepdog/.
[11] GlusterFS.: http://www.gluster.org/.