# ヘテロ構成SDSの性能改善に向けたIO処理オフローディングの検討

佐藤 賢太[1,a)]　出口 彰[1]　川口 智大[1]

**概要**：複数の汎用サーバ（ノード）をソフトウェア制御によりクラスタ化し，ストレージシステムとして機能させる SDS（Software Defined Storage）の市場が拡大している．SDS では設備投資の最適化のため，小規模構成で運用を開始し，必要分の容量・性能を持つノードを順次追加する運用を行う．しかし，このような運用を行うと，クラスタに異なる容量・性能のノードが混在するヘテロ構成となってしまう．一般に各ノードが処理する IO 量はノード容量に比例するため，ヘテロ構成では小容量ノードは低稼働率に，大容量ノードは高稼働率・過負荷になる．結果，小容量ノードの稼働率が上がらないことで，クラスタ全体のハードウェア資源に見合った IO スループット性能が得られない問題が生じる．本研究では，このようなヘテロ構成の SDS における稼働率改善による性能向上をめざす．今回，データを大容量ノードに格納しつつ，IO 処理を低稼働率の小容量ノードへオフロードすることで，小容量ノードの稼働率を向上させる技術を提案する．公開 IO トレースを用いた机上評価を行った結果，クラスタ全体の稼働率 91%まで改善可能な見込みを得た．このときの IO スループット性能は従来比 1.4 倍である．

## 1. はじめに

近年，汎用サーバをソフトウェア制御によりストレージシステムとして機能させる SDS（Software Defined Storage）の市場が拡大している．図 1 に SDS の概要を示す．SDS は，SSD 等の物理ストレージデバイスを搭載した汎用サーバ（ノード）がネットワークに接続された構成をとる．そのうえで，各ノードが持つ記憶容量をクラスタ全体で容量プールとして統合し，アプリケーションに対してボリュームと呼ばれる論理ストレージデバイスを提供する．これにより，アプリケーションはノードや物理ストレージデバイスを意識せずデータを読み書きできる．

SDS は容量や IO スループット性能といったストレージ資源が不足した場合に，1 台以上のノードを追加する．これにより，小規模な構成から SDS の運用を開始し，需要に合わせて徐々に規模を大きくしていくといった，設備投資の最適化が可能となる．しかしながら，このように需要に応じてノードを追加していく運用では，タイミングによって追加するノードに搭載される SSD 容量や CPU 性能が異なり，クラスタには異なる容量や性能を持つノードが混在することになる．以後，このようにクラスタに異なる容量や性能を持つノードが混在した構成をヘテロ構成と呼ぶ．

[1] 株式会社 日立製作所 研究開発グループ
Hitachi, Ltd. Research & Development Group
[a)] kenta.sato.ju@hitachi.com

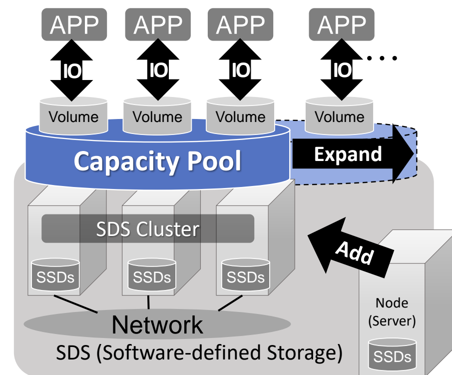



**図 1**　SDS の概要

一般に各ノードが処理する IO 量は，ノードの容量に比例するため，小容量ノードは低稼働率に，大容量ノードは高稼働率又は過負荷となる．クラスタ全体でみると，小容量ノードの稼働率が上がらないことで，クラスタ全体の稼働率も上がらず，クラスタが有するハードウェア資源に見合った IO スループット性能が得られない問題が生じる．

本研究では，データを大容量ノードに格納しつつ，大容量ノードが実行していた IO 処理のみを低稼働率の小容量ノードへオフロードすることで，低稼働率ノードの稼働率を高め，クラスタ全体の稼働率を向上させる方式を提案する．以後，2 章で SDS の概要について，3 章で関連研究について，4 章で提案する IO 処理オフローディングについて，5 で評価について，6 章でまとめについて述べる．

## 2. SDS の概要

本章では，SDS の基本的なアーキテクチャ及び，SDS で生じる問題点について述べる．

### 2.1 基本アーキテクチャ

#### 2.1.1 クラスタ構成

1 章で述べたように，SDS はネットワークに接続された複数のサーバ（ノード）をクラスタ化して構成するストレージである．SDS では，複数のノードを容量プールとして統合するため，ネットワークを経由してノード間で各種制御情報やデータの転送を行う．SDS にデータを格納するアプリケーション及びアプリケーションが動作するサーバは，ネットワークを経由して SDS を構成する何れかのノードに対して IO を発行する．

#### 2.1.2 内部構造

SDS を構成する各ノードの内部構造を図 2 に示す．SDS の内部構造は，大きく分けてフロントエンド，ミドル，バックエンドの 3 レイヤ/コンポーネントで構成される．

フロントエンドは，アプリケーションからの IO を受け付けるためのコンポーネントである．iSCSI や FC (Fibre Channel) といったストレージプロトコルに関する制御を行い，アプリケーションがボリュームに対して発行した IO をミドルに中継する．

ミドルは，ボリュームの IO を制御するためのコンポーネントである．フロントエンドを介して受け付けた IO に対し，ミドルは，キャッシュ管理やアドレス変換といった基本的な制御を行い，そのうえでシンプロビジョニングやスナップショット，データ圧縮・重複排除といった高度なストレージ機能を提供するための各種制御を行う [1,2]．

バックエンドは，物理ストレージデバイスを制御するためのコンポーネントである．SAS や NVMe といった各種ストレージプロトコルの変換を行い，ミドルが物理ストレージデバイスに IO を発行するための統一インタフェースを提供する．

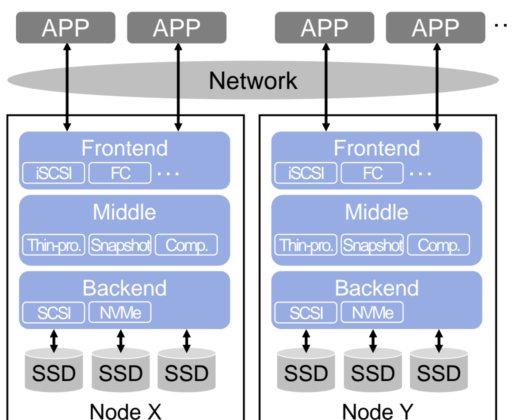



図 2　SDS の内部構造

### 2.2 SDS の問題点

#### 2.2.1 SDS の運用

SDS は，ノード追加によって需要に応じて容量や IO スループット性能といったストレージ資源を追加できるため，小容量のノードを用いて SDS 構築し，後から大容量のノードを追加するという運用を行う．

近年 SSD は TLC（Triple Level Cell）や QLC（Quad Level Cell），3D NAND といった技術発展により大容量化が急速に進んでおり，容量あたりのコストは低下傾向にある [3] [4]．そのため，設備投資のコストを抑えるためには，予め将来的に必要な容量を用意しておくよりも，需要に応じて容量を追加していく運用の方が適している．

#### 2.2.2 問題点

SDS では，あるノードに格納されるデータに対する IO 処理は，全て当該ノードの CPU が処理を行う．一般に各ノードが処理する IO 量はノード容量に比例するため，CPU の性能がノード容量と比べて低いと，当該ノードは高稼働率・過負荷状態に陥るため，容量に見合った IO スループット性能を得ることができない．逆に，ノードの容量が CPU 性能と比べて少ないと，当該ノードは低稼働率状態となり，搭載した CPU の性能を使い切ることができない．つまり，SDS において SSD や CPU といったハードウェア資源を無駄なく利用するためには，ノード毎に搭載する SSD 容量に比例した性能の CPU を搭載しておく必要がある．

しかしながら，CPU 性能については，半導体微細化の限界により SSD の大容量化と同程度の大幅な性能向上は期待できない [3]．このため，後から追加したノードほど搭載容量のみが増加し，容量あたりの CPU 性能（資源）が少なくなる傾向にある．この結果，クラスタに新しく追加されたノードは大容量のため高稼働率状態に，元々あったノードは小容量のため低稼働率状態となり，クラスタ全体の稼働率が低くなる問題が生じる．

あるクラスタに，既存ノードと同性能の CPU と，既存ノードよりも大容量の SSD を搭載したノードを，既存ノードと同数追加した場合のクラスタ全体の稼働率 $W$ を (1) 式に示す．式中の $W_{existing}$ と $W_{new}$ は既存ノードと追加ノードの稼働率をそれぞれ示している．例えば，既存ノードと新しく追加したノードの容量比を 1:4，新しく追加した大容量ノードの稼働率を 100%と仮定すると，容量に比例して既存ノードの稼働率は 25%となり，クラスタ全体の稼働率は 60%程度に留まる．

$$\begin{aligned} W &= average(W_{existing}, W_{new}) \\ &= average(\frac{1.0}{4}, 1.0) \qquad (1) \\ &= 62.5\% \end{aligned}$$

本研究では，このクラスタ全体の稼働率を 90%以上に高めることを目標とする．

ⓒ 2019 Information Processing Society of Japan

## 3. 関連研究

文献 [5] では，HDD や SSD といった異なる性能のストレージデバイスを搭載したノードから構成されるヘテロ構成の SDS において，データブロックのアクセス頻度に応じてノード間でデータ再配置を行うことによる性能改善を提案している．データ再配置を行う点で，本研究に近いアプローチをとっている．本研究は，データを格納するノードの負荷軽減のために，データを格納するノードと IO を処理するノードを分離する点で異なる．

文献 [6] は，性能が異なるノードから構成されるヘテロ構成の Hadoop クラスタにおいて，ノード毎の処理速度の違いによりデータローカリティが崩れ，ノード間転送が行われることで発生するネットワーク輻輳を軽減するために，データブロックの配置をノード性能などに応じて決定するアルゴリズムを提案している．ノード間転送の影響については，本研究でも課題として取り上げているが，本研究ではノード間転送に費やされる CPU 資源の軽減が課題であり，対象としている課題が異なる．

このほか，本研究と同様に容量が異なるノードから構成されるストレージを対象とした研究として，文献 [7] があるが，ノード障害発生時のデータ復旧確率を最大化するためのデータ配置最適化を対象としており，やはり本研究とは対象としている問題が異なる．

## 4. IO 処理オフローディング

本章では，問題解決策として IO 処理オフローディングを提案し，その実現方法について説明する．

### 4.1 アプローチ

2.2.2 節で述べたように，SDS では，あるデータに対する IO 処理を，当該データを格納しているノードが処理するため，クラスタ全体で CPU 資源が余っていたとしても，ノード毎に搭載容量に比例した CPU 資源しか使うことができない．そこで，本研究では，データを大容量ノードに格納しつつ，当該データに対する IO 処理を小容量ノードで実行することで，小容量ノード及びクラスタ全体の稼働率を向上させる IO 処理オフローディングを提案する．

図 3 に IO 処理オフローディングにおける IO 処理イメージを示す．フロントエンドやバックエンドは IO の受け付けやプロトコルの変換だけを行うのに対して，ミドルはキャッシュの管理やアドレス変換，データ圧縮といった処理のために大量の CPU 資源を消費する．そこで，本方式ではバックエンドのレイヤでノード間のデータ転送を行うことで，大容量ノードが行っていたフロントエンド及びミドルの処理を小容量の既存ノードに一部オフローディングする．小容量ノードは，自ノードに格納されているデータだけでなく，大容量ノードに格納されてデータの IO 処理も実行する．結果，高稼働率にあった大容量ノードの負荷が軽減され，低稼働率状態にあった小容量ノードの稼働率が向上することで，クラスタ全体の稼働率が向上や，クラスタ全体の IO スループット性能向上，大容量ノードの過負荷を軽減といった効果が期待できる．

なお，図にあるように，IO 処理オフローディングを適用すると，アプリケーションが IO を発行する先のノードが変わるが，これは iSCSI Redirection [8] といった技術を利用することで SDS 側で制御可能である．

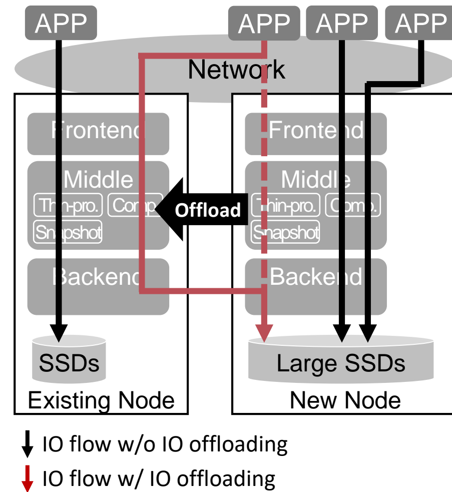



図 3 IO 処理オフローディングのイメージ

### 4.2 単純適用時の問題点

IO 処理オフローディングを単純に適用すると，ノード間転送の処理が増える分，バックエンドの処理量が増大し，オフローディングの効果が得られない問題が生じる．以後，この問題の詳細について説明する．

一般に SDS を構成する各コンポーネントは，レスポンス時間の安定化やコンポーネント間のコンテキストスイッチに伴うオーバヘッド削減などのために，コンポーネント毎に特定の CPU コアに割り当てられて動作する．このため，SDS では，特定のコンポーネントがボトルネックにならないように，コンポーネント毎に処理量と CPU の性能をバランスする CPU コア数の割り当ての設計が行われる．

しかしながら，IO 処理オフローディングを適用すると，コンポーネント毎の処理量が変わってしまうため，処理量と CPU 性能のバランスが崩れてしまう．具体的には，大容量ノードにおけるフロントエンド・ミドル処理は，小容量ノードにオフローディングされることで，処理量が減る．一方で，バックエンドについては，ノードの搭載容量に処理量が比例することに変わりなく，ノード間転送が増える分，処理量はむしろ増大する．この結果，バックエンド処理がボトルネックとなり，フロントエンドやミドルの処理が妨げられる．当然，クラスタ全体の稼働率やスループット性能が向上することはなく，最悪の場合は低下してしま

ⓒ 2019 Information Processing Society of Japan

う．このため，IO 処理オフローディングを有効に機能させ，クラスタ全体の稼働率やスループット性能を向上させるためには，バックエンドの処理量を軽減が課題となる．

図 4 は，従来方式と IO 処理オフローディングを単純に適用した際の各コンポーネントの CPU 稼働率を示している．フロントエンドとミドルについては簡単化のために，単一コンポーネントとして扱い，図中では “FE+Mid” で示している．図中の “BE” はバックエンドを示している．図中の “Existing Node” は小容量の既存ノードを，“New Node” は大容量の追加ノードを示している．凡例の “Active” は IO 処理に費やしている CPU 時間の割合を，“Stall” は IO 処理を実行中だが他コンポーネントがボトルネックとなり実際には何も処理が行われていない CPU 時間の割合を（ストール時間），“Idle” は IO 処理自体を行っていない CPU 時間の割合をそれぞれ示している（アイドル時間）．その他，詳細な評価条件については，5 章を参照されたい．

IO 処理オフローディングを適用することで，小容量ノードでアイドル時間が減っている．一方で大容量ノードのバックエンドは稼働率が 100%となっており，フロントエンド・ミドルではストール時間が発生している．つまり，大容量ノードのバックエンドがボトルネックとなり，フロントエンドやミドルの処理が妨げられている．

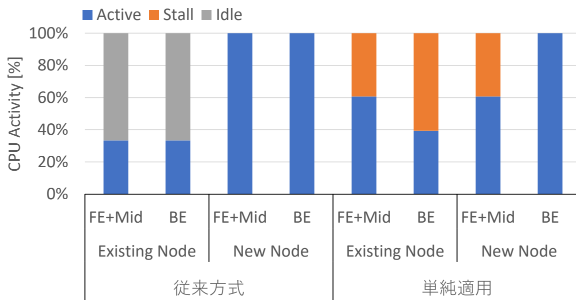



図 4　単純適用時の各コンポーネント稼働率

### 4.3　解決策

本研究では IO 処理オフローディングに対して，下記に示す二つの処理を付け加えることで，バックエンドのボトルネックを軽減する．なお以降の説明では，ミドルを基準として，ミドルが動作するノードと同じノードに格納されているデータを自系データ，異なるノードに格納されているデータを他系データと呼ぶ．

**アクセス頻度に応じたデータ再配置**
自系データに含まれるアクセス頻度が低いデータと，他系データに含まれるアクセス頻度が高いデータを入れ替えることで IO 処理オフローディングに伴うノード間転送の発生を低減し，大容量ノードでのバックエンド処理量自体を減らす．

**ノード稼働情報に基づくデータ配置先最適化**
さらに単純に 2 ノード間でデータを入れ替えるのではなく，各ノードの空き容量や稼働率などを基にクラスタ全体でデータ配置を最適化することで，過剰な負荷や余剰な CPU 資源を最小化する．

以後，4.3.1 節で (1) について，4.3.2 節で (2) について詳細を説明する．

#### 4.3.1　アクセス頻度に応じたデータ再配置

一般に，ストレージに格納されたデータへのアクセスは，20%の領域に 80%の IO が集中するといったように，パレートの法則に従う傾向が強いことが知られている [9] [10] [11]．アクセス頻度が低いデータであれば，IO 処理オフローディングの対象としてもノード間転送の発生頻度は低く，バックエンドに与える影響は小さい．つまり，他系データのうちアクセス頻度が高いデータと，自系データのうちアクセス頻度が低いデータの格納先を入れ替えることで，バックエンドの負荷を下げることができる．

##### 4.3.1.1　動作概要

図 5 を用いて，本処理の動作について説明する．図中の Node 1 は，小容量のノードであり，IO 処理オフローディングにおいてミドル処理を実行する．Node 2 は，大容量のノードであり，Node 1 から IO 処理オフローディングによってアクセスされるデータを格納している．

本処理は，IO 処理オフローディングにおいてミドル処理を実行するノード（図中の “Node 1”）で動作する．Access Monitor は，アプリケーションが格納したデータを小容量・固定サイズのブロックに分割して管理し，ブロック毎にアプリケーションから発行された IO の頻度を監視する．Classifier は，一定時間毎に各ブロックの IO 回数を集計することで，各ブロックを高アクセス頻度データと低アクセス頻度データに分類する．この分類の仕組みについては，4.3.1.2 節で詳細に説明する．その後，Placement Controller が分類結果に基づいて各ブロックの再配置を行う．具体的には，高アクセス頻度データを自系データとして Node 1 に，低アクセス頻度データを他系データとして Node 2 に格納する．なお，説明の順番上，他系データを Node 2 に格納するとしたが，Placement Controller は次節で述べるデータ配置先最適化の結果に従って，ブロック単位で格納先ノードを変更する．

##### 4.3.1.2　アルゴリズム

本処理では，ノード間転送の影響最小化のために，アクセス頻度の最も低いデータの組み合わせを検出することが重要となる．そのため，監視・収集したアクセス頻度情報を用いて，図 6 に示すようなヒストグラムを作成することで，アクセス頻度の最も低いデータの組み合わせ検出を実現する．このヒストグラムは，Node 1 のミドルが IO 処理を担当しているデータのブロックをアクセス頻度順に並べたもので，縦軸はブロック毎のアクセス頻度（一定期間内の IO 回数）を，横軸（各棒）はブロックをアクセス頻度で降順ソートしたものを示している．

ⓒ 2019 Information Processing Society of Japan

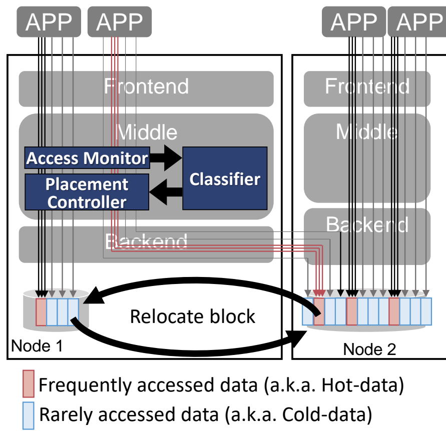



図 5　データ再配置の動作

本処理では，アクセス頻度が高いブロックから Node1 の容量に収まるだけのブロックを高アクセス頻度データ（図中の “frequently accessed”）に分類し，残ったブロックを低アクセス頻度データ（図中の “rarely accessed”）に分類する．これにより，IO 処理オフローディングに伴って発生するバックエンドレイヤでのノード間転送が最小化され，バックエンドでのボトルネックが軽減される．

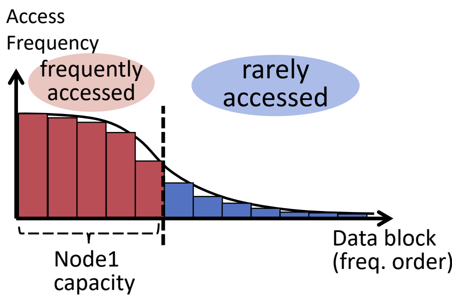



図 6　アクセス頻度に応じたデータ分類

### 4.3.2　ノード稼働情報に基づくデータ配置先最適化

小容量ノードにおいて低アクセス頻度データと判断されたデータでも，CPU 稼働率が高い大容量ノードに格納すると，ノード間転送で増大する負荷によって，CPU が過負荷に陥ってしまう．一方で，同じ大容量ノードだとしても，格納データ量の違いや，格納しているデータへの IO 量の違いによって，CPU 稼働率がそれほど高くないノードも存在し得る．

そこで，単純に小容量ノードと大容量ノードの 2 ノード間で格納データを入れ替えるのではなく，クラスタ全体から他系データ格納後に過負荷状態に陥らないノードを他系データの格納先ノードとして選択する．これにより，ノード間で CPU 負荷を平衡化され，クラスタ全体で過剰な負荷や余剰な CPU 資源を最小化することができる．

#### 4.3.2.1　動作概要

図 7 を用いて本処理の動作について説明する．詳細なアルゴリズムについては，次節で説明する．本処理では，各ノードから収集したデータ量やアクセス頻度の情報を基に，他系データ毎に必要な空き容量と CPU 資源を算出する（図中の “Workload Information”）．これらの情報と，各ノードの空き容量，空き CPU 資源を比較することで，他系データの格納先となるノードを選択する．図では，“NodeX” は CPU 資源が不足，“NodeZ” は容量が不足しており，共に余裕のある “NodeY” が他系データ格納先として選択されている．

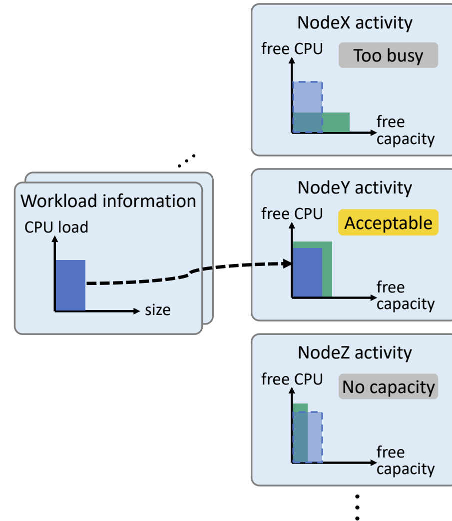



図 7　データ配置最適化の概念図

#### 4.3.2.2　アルゴリズム

次に，データ配置最適化の具体的なアルゴリズムについて説明する．データ配置最適化における最適な組み合わせ算出は，組み合わせ最適化問題の一種であるビンパッキング問題と捉えることができる．

データ配置最適化における制約条件は (2) 式と (3) 式で表現される．(2) 式は，全ての他系データについて，何れかの格納先ノードに格納されることを意味している．$m$ は各他系データを，n は各格納先ノードを，$x_{mn} \in \{0, 1\} \forall^{(m,n)}$ はノード $n$ に他系データ $m$ を格納するか否かを示している（$x_{mn} = 1$ の場合に格納する）．(3) 式は，あるノードに格納する他系データの総サイズが，当該ノードの空き容量以下であることを意味している．$size_m$ は他系データ $m$ のデータサイズを示しており，$capacity_n$ はノード $n$ の空き容量を示している．

$$\sum_n x_{mn} = 1 \forall^m \tag{2}$$

$$\sum_m size_m \times x_{mn} \leq capacity_n \forall^n \tag{3}$$

目的関数は，他系データ格納後の各格納先ノードの過剰負荷（もしくは処理能力の余剰）の最小化であり，(4) 式で示される．$ability_n$ はノード $n$ で余っている処理能力を示している．$L_n$ は，ノード $n$ に格納する他系データによって生じる追加負荷であり，(5) 式で示される．$load_m$ は，他

系データ $m$ によって生じる追加負荷を示している．

$$Minimize : \sum_{n} |ability_n - L_n| \quad (4)$$

$$L_n = \sum_{m} load_m \times x_{mn} \quad (5)$$

## 5. 評価

### 5.1 評価方法

本研究では，提案した IO 処理オフローディングによるクラスタ全体の稼働率向上や IO スループット性能向上をシミュレーションに基づく机上計算によって評価する．

図 8 に評価対象の運用シチュエーションを示す．本研究では，容量不足により大容量ノードを順次増設する運用を想定する．8 ノードの小容量ノードで運用を開始し，アプリケーション及び格納データ量の増加により使用容量が 75%に達した際に大容量ノードを 2 台ずつ順次追加する運用を想定し，合計 16 ノードまで拡張した時点での評価を行う．2 台ずつ追加するのは，追加する 2 ノード間でデータをミラーリングすることを想定しているためである．

追加ノードと既存ノードの CPU は同性能のものを搭載していると想定し，IO 処理オフローディングを行うことで，各ノードのフロントエンド・ミドルは，常に同容量をアプリケーションに提供していると考える．また，4.3 節と同様に評価を簡単化するため，ある IO に対するフロントエンドとミドルの処理は常に同じノードで実行され，フロントエンドでの IO の中継は行われないものとする．

4.3.2 節で述べたデータ配置先最適化を適用しない場合の他系データ格納先ノードの選択はラウンドロビンアルゴリズムを利用するものとする．具体的には，ノード増設時に伴い各ノードがホストに提供する容量が増加した際に，増加分の他系データ格納先を各大容量ノードから順番に選択している．本評価では，容量不足に伴って段階的にノードを追加するため，ラウンドロビンアルゴリズムのような単純な方式で他系データの格納先ノードを選択した場合，先に追加したノードは後から追加したノードよりも多量の他系データを格納することになる．

表 1 に評価条件を示す．シミュレータで用いるワークロードとして，金融系実アプリケーションの公開 IO トレースである financial1 [12] を利用する．

表 1 評価条件

| | Existing nodes | New nodes |
|---|---|---|
| Workload | Financial1 [12], 320 TB | |
| Capacity (Physical) | 5 TB (10 TB) | 20 TB (40 TB) |
| # of Nodes | 8 | 8 |
| Add timing | - | Capacity usage exceed 75% |
| Add unit | - | 2 nodes |

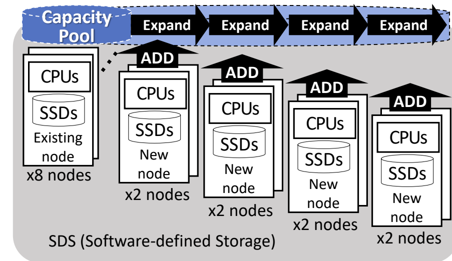



図 8 想定する運用シチュエーション

### 5.2 性能モデル

前節で述べた評価のための机上計算及び，机上計算の基となる性能モデルの考え方について，図 9 を用いて説明する．なお，性能モデルでは簡単化のため，フロントエンドとミドルはまとめて一つの処理として扱う．

本節では，図中の表記にあわせて，他系データにアクセスする側のノードをノード 1，アクセスされる側のノードをノード 2 と呼ぶ．図中の $t_{fe+mid}$，$t_{be_local}$，$t_{be1_rmt}$，$t_{be2_rmt}$ は，1IO あたりにフロントエンド・ミドルが消費する CPU 時間，自系データアクセス時にバックエンドが消費する CPU 時間，他系データアクセス時にノード 1 のバックエンドが消費する CPU 時間，他系データアクセス時にノード 2 のバックエンドが消費する CPU 時間をそれぞれ示している．$r_{be1_local}$ 及び $r_{be2_local}$ は，各ノードのバックエンドにおいて全 CPU 時間に占める自系データアクセスが消費する CPU 時間の割合をそれぞれ示している．これらは，前節で述べた公開 IO トレースを用いて 4.3.1 節で述べたデータ再配置のシミュレーションを行うことで算出する．

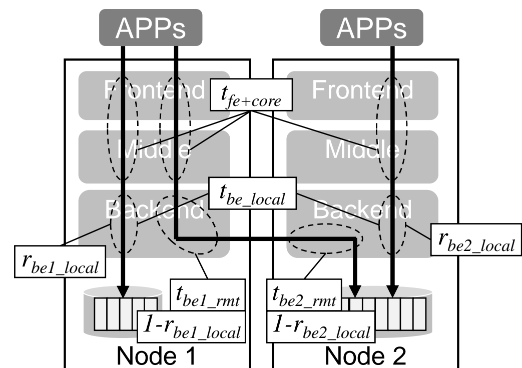



図 9 性能モデル

フロントエンド・ミドルの最大スループット $P_{fe+mid}$ は，処理に割り当てられる CPU コア数に 1 コアあたりのスループットを乗算したものとなるため，(6) 式で得られる．フロントエンド・ミドルの処理は各ノード共通のため，$P_{fe+mid}$ は各ノードで共通である．

$$P_{fe+mid} = n_{fe+mid} \times \frac{1}{t_{fe+mid}} \quad (6)$$

次に，ノード 2 において自系データアクセスを行う場合

ⓒ 2019 Information Processing Society of Japan

のバックエンドの最大スループット $P_{be2}$ は，全 IO が次系データアクセスの場合のスループットに対して，ノード 2 の CPU が自系データアクセスに費やす時間の割合を乗算したものとなるため，(7) 式で表わされる．$n_{be}$ は，バックエンドに割り当てられる CPU コア数を示している．

$$P_{be2} = n_{be} \times \frac{r_{be2_local}}{t_{be_local}} \quad (7)$$

ノード 1 におけるバックエンドの最大スループット $P_{be1}$ は，(8) 式で得られる．$min()$ は，他系データアクセスのスループットがノード 1 とノード 2 の低い方に律速されることを意味している．

$$P_{be1} = n_{be} \times \{\frac{r_{be1_local}}{t_{be_local}} + min(\frac{1 - r_{be1_local}}{t_{be1_rmt}}, \frac{1 - r_{be2_local}}{t_{be2_rmt}})\} \quad (8)$$

各ノードの最大スループットは，各コンポーネントの最大スループットの小さい方に律速されるため，(9) 式となる．

$$P_N = min(P_{fe+mid}, P_{beN}) \quad (9)$$

評価に用いる性能モデルのパラメータを表 2 に示す．これらのパラメータは，文献 [13, 14] に掲載されているベンチマーク結果を基に本研究で説明した各コンポーネントの相対的な処理時間を概算したものである．フロントエンド・ミドル及びバックエンドに割り当てられる CPU コア数は，$t_{fe+mid}$ と $t_{be_local}$ の比率とし，従来の IO 処理オフローディングを行わない場合に最適となるように設定している．

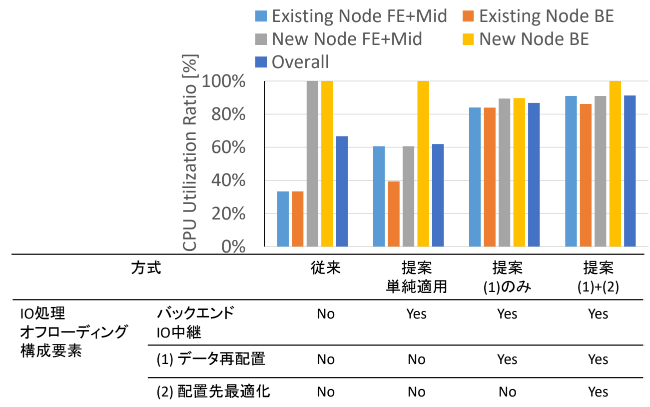
**表 2** 性能モデルパラメータ

| パラメータ名 | 値 |
|---|---|
| $t_{fe+mid}$ | 60 |
| $t_{be_local}$ | 10 |
| $t_{be1_rmt}$ | 3 |
| $t_{be2_rmt}$ | 13 |
| $n_{fe+mid}$ | 6 |
| $n_{be}$ | 1 |

### 5.3 評価結果

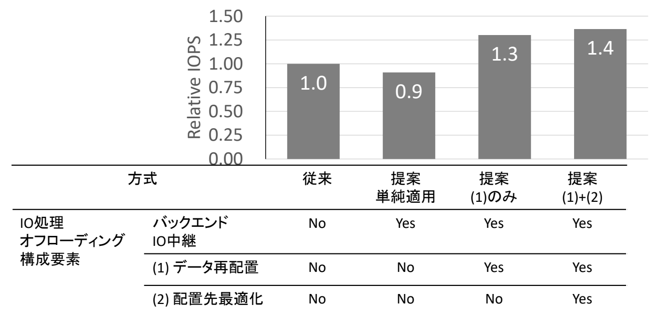
評価結果を図 10 及び図 11 に示す．

図 10 は，既存ノードと追加ノードにおける各コンポーネントの CPU 稼働率を表しており，各コンポーネントの動作状況を把握するためのものである．凡例の “Existing Node 〜” は既存ノードの稼働率を，“New Node 〜” は追加ノードの稼働率をそれぞれ示している．“〜” の部分については，“FE+Mid” はフロントエンド・ミドルを，“BE” はバックエンドの稼働率をそれぞれ示している．既存ノードと追加ノードはそれぞれ 8 ノードある想定だが，図中に示しているのは各ノードの平均 CPU 稼働率である．“Overall” は全ノード・全コンポーネントの平均 CPU 稼働率を示している．方式の “従来” は IO 処理オフローディングを適用しない場合を，“提案単純適用” は IO 処理オフローディングだけを単純適用した場合を，“提案 (1) のみ” は IO 処理オフローディングに加えて，4.3.1 節で説明したデータ再配置を適用した場合を，“提案 (1)+(2)” は更に 4.3.2 節で説明したデータ配置最適化を適用した場合をそれぞれ示している．図 10 は，従来方式に対する各方式におけるクラスタ全体の相対スループット性能を表しており，提案方式による改善効果を示すものである．




| 方式 | | 従来 | 提案単純適用 | 提案(1)のみ | 提案(1)+(2) |
|---|---|---|---|---|---|
| IO処理オフローディング構成要素 | バックエンドIO中継 | No | Yes | Yes | Yes |
| | (1) データ再配置 | No | No | Yes | Yes |
| | (2) 配置先最適化 | No | No | No | Yes |

**図 10** CPU 稼働率の評価結果




| 方式 | | 従来 | 提案単純適用 | 提案(1)のみ | 提案(1)+(2) |
|---|---|---|---|---|---|
| IO処理オフローディング構成要素 | バックエンドIO中継 | No | Yes | Yes | Yes |
| | (1) データ再配置 | No | No | Yes | Yes |
| | (2) 配置先最適化 | No | No | No | Yes |

**図 11** 相対スループット性能の評価結果

IO 処理オフローディングを適用しない従来方式では，2.2 節で述べたように，大容量の追加ノードは高稼働率状態に，小容量の既存ノードは低稼働率状態となることで，クラスタ全体の CPU 稼働率は 60%程度という結果になっている．

IO 処理オフローディングを単純に適用した場合は，追加ノードのバックエンドの稼働率が 100%となり，クラスタ全体の CPU 稼働率は従来方式を下回る結果となっている．スループット性能も従来方式比で 0.9 倍と，悪化する結果となっている．これは，4.2 節で述べたように，ノード間転送といった IO 処理オフローディングに伴う追加処理によって，追加ノードのバックエンドがボトルネックとなったことで，フロントエンド・ミドルの処理が妨げられたためである．

IO 処理オフローディングに加えてデータ再配置を適用し

ⓒ 2019 Information Processing Society of Japan

た場合は，どのコンポーネントも稼働率が 80〜90%となっており，クラスタ全体の稼働率としては 87%という結果になっている．スループット性能については，従来方式比 1.3 倍という結果である．これは，アクセス頻度に基づくデータ再配置により，IO 処理オフローディングに伴うノード間転送が削減されたことで追加ノードのバックエンドのボトルネックが改善されたためである．どのコンポーネントも稼働率 100%に到達していないが，これは他系データ格納先をラウンドロビンで選択しているためである．5.1 節で説明したように，容量不足に応じてノードを追加する状況を想定したため，先に追加したノードの他系データ格納量が多く，ボトルネックになっている．

最後に IO 処理オフローディングに加えてデータ再配置・配置先最適化という提案内容を全て適用した場合は，既存ノードのバックエンドの稼働率が 90%未満となっているが，それ以外のコンポーネントは 90%〜100%という高い稼働率となっている．クラスタ全体の稼働率は 91%という結果となっており，目標の 90%以上を達成している．スループット性能については，従来方式比 1.4 倍という最も高い結果となっている．これは配置先最適化により，ノード間での他系データ格納量が平衡化され，特定ノードがボトルネックになるという問題が解消されたためである．

コンポーネントごとの稼働率をみると，追加ノードのバックエンドは 100%に達しているが，それ以外のコンポーネントは 100%に達しておらず，余力を残している状態となってしまっている．これは，従来方式ではなかった他系データアクセスにより，コンポーネント間の各処理時間と割り当てた CPU コア数のバランスが崩れてしまったためである．この問題を解消するには，各コンポーネントへの CPU コア固定割り当てをやめ，動的に CPU 資源を割り当てることが有効だと考えられる．ただし，4.2 節での述べたように，動的な CPU 資源割り当てによってレスポンス時間が不安定になるため，課題である．

## 6. まとめ

本研究では，異なる容量や性能のノードから構成されるヘテロ構成の SDS において，ハードウェア資源に見合った IO スループット性能が得られない課題の解決を目的として，大容量ノードから小容量ノードへの IO 処理オフローディングを提案した．この IO 処理オフローディングを単純に適用すると，ノード間転送による追加負荷が発生し，性能が向上しない問題が発生するため，アクセス頻度に応じたデータ再配置やデータ配置先ノード最適化という解決策を講じた．机上評価の結果，提案方式によって 90%以上の稼働率が得られ，データを格納したノードで IO 処理を実行する一般的な方式と比べ 1.4 倍のスループット性能が得られる見込みを得た．

一方で，各処理に CPU コアを固定割り当てする SDS のアーキテクチャでは，提案方式により各処理に割り当てられる最適な CPU コア数のバランスが崩れ，全 CPU 資源を使い切れない問題が生じることが分かった．各処理への動的 CPU 資源割り当てによる稼働率改善とレスポンス時間が安定化の両立が今後の課題である．

## 参考文献

[1] Dell Inc.: Dell EMC VxFlex Family Overview, 2019.

[2] VMware, Inc.: VMWare vSAN 6.7 Technical Overview, 2018.

[3] Denis C. Daly, Laura C. Fujino, Kenneth C. Smith: ”Through the Looking Glass - The 2018 Edition: Trends in Solid-State Circuits from the 65th ISSCC,” IEEE Solid-State Circuits Magazine, vol. 10, no. 1, pp. 30-46, 2018.

[4] Jung Yoon, Ranjana Godse, Andrew Walls: ”3D NAND Technology Scaling helps accelerate AI growth,” Flash Memory Summit 2018, 2018.

[5] A. Nunome, H. Hirata and K. Shibayama: ”A Distributed Storage System with Dynamic Tiering for iSCSI Environment,” 2014 IIAI 3rd International Conference on Advanced Applied Informatics, 2014, pp. 644-649.

[6] A. Shah and M. Padole: ”Load Balancing through Block Rearrangement Policy for Hadoop Heterogeneous Cluster,” 2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI), 2018, pp. 230-236.

[7] M. Noori and M. Ardakani: ”Allocation for Heterogeneous Storage Nodes,” IEEE Communications Letters, vol. 19, no. 12, pp. 2102-2105, 2015.

[8] M. Chadalapaka, J. Satran, K. Meth, D. Black: ”Internet Small Computer System Interface (iSCSI) Protocol (Consolidated)”, https://tools.ietf.org/html/rfc7143, Internet Engineering Task Force (IETF), 2014.

[9] Chesire, Maureen and Wolman, Alec and Voelker, Geoffrey M. and Levy, Henry M.: ”Measurement and Analysis of a Streaming-media Workload, Proceedings of the 3rd Conference on USENIX Symposium on Internet Technologies and Systems - Volume 3, 2001.

[10] Youngjae Kim, Raghul Gunasekaran, Galen M. Shipman, David A. Dillow, Zhe Zhang, Bradley W. Settlemyer: ”Workload characterization of a leadership class storage cluster,” 2010 5th Petascale Data Storage Workshop (PDSW ’10), 2010.

[11] Kavalanekar, Swaroop and Worthington, Bruce and Zhang, Qi and Sharda, Vishal: ”Characterization of storage workload traces from production Windows Servers,” 2008 IEEE International Symposium on Workload Characterization, 2008.

[12] UMassTraceRepository, http://traces.cs.umass.edu/index.php/Storage/Storage, National Science Foundation.

[13] StorageReview Enterprise Lab: ”Dell EMC Unity 450F All-Flash Storage Review,” https://www.storagereview.com/dell_emc_unity_450f_allflash_storage_review.

[14] Benjamin Walker: ”SPDK: Building Blocks For Scalable, High Performance Storage Applications,” Storage Developer Conference, 2016.

ⓒ 2019 Information Processing Society of Japan