Vol.2015-OS-134 No.19
2015/8/5

ログ構造化データレイアウト向けストレージ階層制御の提案

岩田 聡[1,a)]　塩沢 賢輔[1]

概要：本発表では上位でログ構造化データレイアウトによってデータが管理されている場合に，下位で適切にストレージ階層制御を行う方法を提案する．ストレージ階層制御を行う場合は通常，過去のアクセス数が比較的多かったデータを高速デバイスに配置する．しかしログ構造化データレイアウトでは書き込みにより最新データの保存位置が移動するため，更新頻度が高いデータを高速デバイスに配置してしまうと，そのデータは近い将来ほとんどアクセスされなくなり，高速デバイスを無駄に占有してしまうことになる．提案手法では読み込み数だけでなく最近ログ書き領域に使用されたという情報を用い，過去のアクセス数が多かったとしても更新頻度の高いデータを多く含むと判定した領域は高速デバイスに配置しない．実環境で収集されたトレースデータを利用し，シミュレーションと実機での評価をそれぞれ行った．25 パターンのシミュレーションで平均 9 ポイント SSD アクセス率が向上し，実機評価では最大で 48%応答時間が短縮した．

1. はじめに

近年，ストレージ階層制御という技術が広く利用されるようになっている．ストレージ階層制御とは比較的高速で高価なデバイスと比較的低速で安価なデバイスを組み合わせ，アクセス頻度の高いデータのみを高速デバイスに配置する手法である．こうすることで全体としては安価で大容量なストレージを提供しつつ，大部分のアクセスを高速デバイスに対して行うことができる．例えば Solid State Drive (SSD) を高速デバイスとして，Hard Disk Drive (HDD) を低速デバイスとして階層化ストレージを構成する場合があり，本発表ではそのような構成を対象としている．

ブロックデバイスとして階層化ストレージを提供する際は**図 1** のように論理的なアドレス空間を上位に提供しつつ，その空間を一定サイズのセグメント（数 MB から数 GB 程度）に分割し，アクセス頻度の高いセグメントを SSD へ，アクセス頻度の低いセグメントは HDD へ配置するのが一般的である．そうすることで上位層では実データがどのデバイスに配置されているかを意識することなく，透過的にデータにアクセスすることができる．各セグメントのアクセス頻度は時々刻々と変化する可能性があるため，一定時間（数時間から数日）ごとにデバイス間でのデータ再配置を行う．このデータ配置によってどの程度のアクセスが SSD に対して行われるかが決定するため，性能を大きく左右する重要な作業である．

[1] 株式会社 富士通研究所
[a)] iwata-satoshi@jp.fujitsu.com

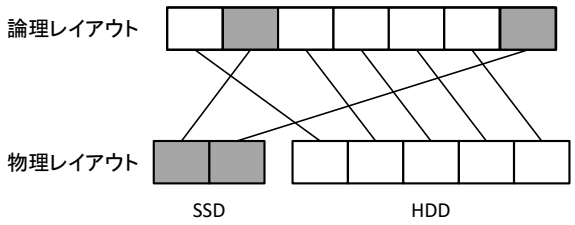

図 1 階層化ストレージの実現方法
Fig. 1 How to implement a tiered storage.

既存の一般的な手法では過去最近アクセス頻度の高かったセグメントは近い将来もアクセス頻度が高いであろうという推定の下，セグメントの再配置を行う．例えば前回再配置以降の読み書きの合計数が多い順に高速デバイスに配置し，残りを低速デバイスに配置するといった具合である．

しかし過去最近のアクセス頻度のみに基づいたセグメント再配置手法は，上位でログ構造化データレイアウトによってデータが管理されている場合，不適切なデータ配置をしてしまうことがある．ログ構造化データレイアウトとはデータ更新時に古いデータを上書きするのではなく，専用のログ領域にデータを書き込む方式であり，元々はログ構造化ファイルシステム [12] として Rosenblum と Ousterhout によって提案された．このレイアウトでは，書き込みは常に連続した領域に行うため高速である，データを上書きしないため更新後一定時間は古いデータを読み込める，最後に書き込みを行った位置を容易に特定できるためクラッシュからのリカバリを迅速に行える，書き込みは常に連続した領域に行うため可変長データを効率良く書き

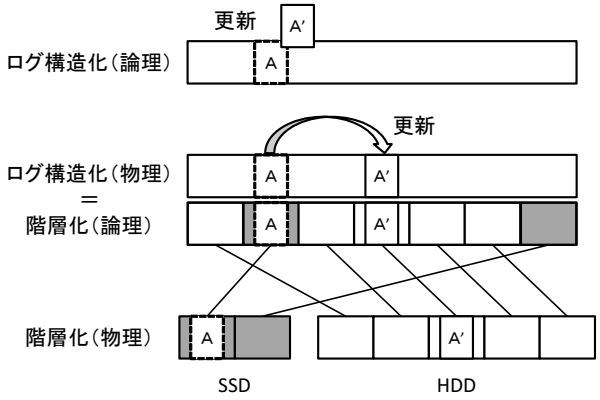

図 2 ログ構造化レイヤと階層制御レイヤの階層化
Fig. 2 Layering log-structuring and tiering layers.

こめる，といった具合にさまざまな利点があるため，今日でも数多くのシステムで採用されている [2], [13], [14], [16].

本発表はログ構造化レイアウトを採用しているソフトウェアが上位で動作している環境で，下位で階層制御を行う環境を想定している．例えば本発表の評価で用いる DBLK[16] という，データの重複排除と圧縮を内部で行うブロックデバイスを提供するモジュールは上述の利点を享受するため，内部でログ構造化レイアウトの形でデータを管理している．重複排除・圧縮に加えて階層制御の機能を提供するために DBLK の下位層で階層制御モジュールを動作させるといったシナリオが考えられ，そのような場合に効果を発揮するのが本発表で提案する手法である．

ログ構造化レイアウトでは更新時に最新データの位置が移動するため，更新頻度の高いデータは高速デバイスに配置すべきではない．**図 2** は階層化レイヤの上位でログ構造化レイヤが動作している環境で，データ *A* を *A'* に更新する例である．この例が示すようにデータ A が存在するセグメントのアクセス頻度が高いと判断し高速デバイスに配置したとしても，ログ構造化レイアウト上でデータを A' に更新すると，ログ構造化レイヤの物理層では A とは異なる位置へデータを書き込む．このため，更新後は A へのアクセスは行われなくなり，高速デバイスを無駄に占有してしまうことになる．

そこで提案手法では過去のアクセス頻度が高いだけでなく，上位レイヤでの更新によってアクセスされなくなるデータを多く含まないという条件も満たすセグメントを高速デバイスに配置する．具体的には読み込み数の多い順にセグメントを並び替える際，前回の再配置以降ログ書き領域に利用されたセグメントへはペナルティを与える．データは全てのデータが均等に更新されるわけではなく，頻繁に更新されるデータとあまり更新されないデータが存在する．最近ログ書き領域に利用されたセグメントは最近更新されたデータを含み，そのようなデータは更新頻度が比較的高いはずなので，そういったセグメントは評価時にペナルティを与えることで高速デバイスに配置されにくくする．

提案手法の有用性を示すために Microsoft Research Cambridge で収集されたストレージトレースデータ [11] を利用してシミュレーションと実機環境で評価を行った．シミュレーションでは 9 種類のトレースデータに対して 2 種類の SSD サイズと 3 種類の階層化レイヤのセグメントサイズを試した．全 54 パターンの内 18 パターンで提案手法が既存手法より高い SSD アクセス率を示し，逆に提案手法が低い値を示したのは 7 パターンであることを確認した．これら 25 パターンに着目すると提案手法を利用した方が既存手法を利用した場合に比べて平均で 9 ポイント SSD アクセス率が増加した．さらに実機による実験も行い，提案手法により最大で 48%応答時間が短縮できることも確認した．

本研究報告書の構成は下記のとおりである．2 節で提案手法の説明を行い，3 節で実験により既存手法との比較を行う．さらに 4 節で関連研究について述べ，最後に 5 節で本報告書をまとめる．

2. 提案手法

提案手法の説明に入る前に，ログ構造化データレイアウトの動作によって，下位層へのアクセスパターンがどのように変化するかを示す．**図 3** は本発表の評価で用いるトレースセットに含まれる `prxy_1` というトレースデータである．横軸が時刻，縦軸はオフセットを表しており，色が濃い領域ほど頻繁にアクセスされることを示している．この図を見ると 2 時間を通じて 0〜40GB の領域が繰り返しアクセスされているのが分かる．一方**図 4** はこの `prxy_1` トレースデータをログ構造化レイアウトを採用している DBLK が提供するブロックデバイスに対して再生し，DBLK の下位層で観測したアクセスパターンである．この図を見ると時間が経過するにしたがってアクセスが集中する領域が 70〜80GB に移り変わっており，図 3 に比べてアクセスパターンが大きく変化しているのが分かる．図 4 では 70〜80GB の領域がログ書き領域に割り当てられ，データ更新時にはその領域に対して書き込みが行われている．

提案手法の目的は過去読み込み数が多いセグメントであったとしても，上位層での更新により近い将来古い世代になるデータを多く含むと推定されるセグメントは積極的に SSD に配置しない点にある．従ってセグメント内部に上位層での更新頻度が高いデータが多く含まれるかどうかを推定する必要がある．

しかし階層制御のレイヤから上位層での各データの更新頻度を知ることはできない．図 2 から分かるように，ログ構造化レイアウトを管理しているレイヤであれば，各論理アドレスにおける更新頻度を計測し，自身の物理アドレスと対応づけることができる．しかし，階層制御のレイヤから認識できるのは A' が書き込まれたことだけであり，それが元々 A というデータであったことを知ることはできな

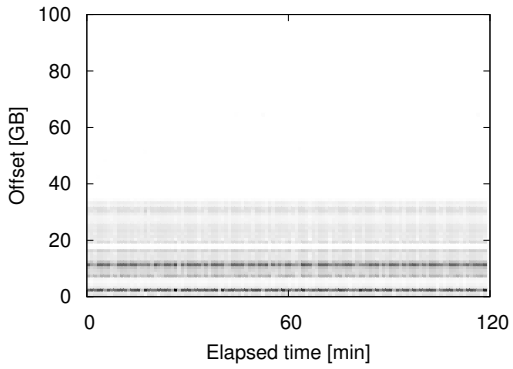

図 3 オリジナルの prxy_1 トレースデータ
Fig. 3 An original trace data of prxy_1.

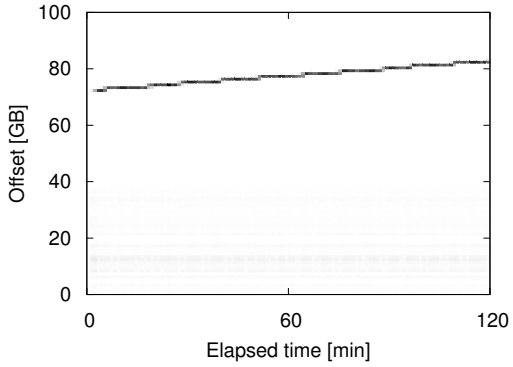

図 4 DBLK 通過後のアクセスパターン
Fig. 4 An example of access pattern under DBLK.

い．単純な解決方法としてログ構造化レイヤから階層制御レイヤに対して，各データの更新頻度を通知する方法が考えられる．しかしこの方法ではログ構造化レイアウトを利用しているソフトウェアがプロプライエタリなソフトウェアであれば実現不可能であるし，仮にオープンソースソフトウェアなど実現可能なソフトウェアであったとしても階層制御機能を追加するたびに通知機構を実装するのは時間と労力を要する．

そこで提案手法では各セグメントに上位層での更新頻度が高いデータが多く含まれるかどうかを階層制御レイヤから推定するために，セグメントが前回のデータ再配置後にログ書き領域に割り当てられたかどうかという情報を利用する．更新は全てのデータに対して均等に行われるのではなく，頻繁に更新されるデータもあればほとんど更新されないデータも存在する．つまり最近更新されたデータというのは更新頻度の高い可能性が高く，最近更新されたデータの入っているログ書き領域は更新頻度の高いデータを多く含んでいる可能性が高いということができる．

ログ書き領域には書き込みが集中するので明らかに他のセグメントとは異なるアクセスパターンとなり，この情報を利用して階層制御レイヤ単独でログ書き領域に利用されたセグメントとその順序を推定することは難しくない．

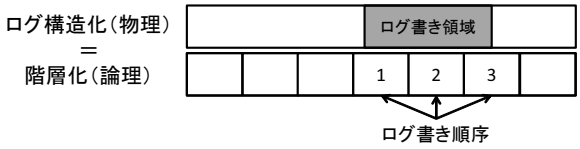

図 5 ログ書き領域と階層化レイヤでのセグメントの関係
Fig. 5 A relationship of log-writing area and tiering layer.

図 5 がログ構造化レイヤにおけるログ書き領域と階層化レイヤにおけるセグメントの関係を示しており，セグメント内に記載された順序でセグメントがログ書きに使用されたことを表している．

提案手法ではデータ再配置の際，前回の再配置以降各セグメントがログ書きに利用されたかどうかという情報と，利用された場合はその順序によって読み込み数にペナルティを与えた値を評価値とする．そして評価値の大きい順にセグメントを SSD に配置し，残りのセグメントを HDD に配置する．前回の再配置以降ログ書き領域に利用されていないセグメントに残っているデータは更新頻度の低いデータである．したがってそのようなセグメントからの読み込み数が多い場合は積極的に SSD に配置すべきなので，ペナルティを与えず読み込み数をそのまま評価値として利用する．一方ログ書き領域に利用されたセグメントは更新頻度の高いデータが多く含まれるはずなので，読み込み数にペナルティを与えて評価値とする．

ペナルティはログ書きに利用された時刻が古いセグメントほど大きな値を与える．これは，新しいセグメントほど内部に最新データが存在する可能性が高いからである．更新頻度の高いデータは前回の再配置以降の短期間に複数回更新が行われている可能性がある．その場合，ログ書きに利用された時刻が古いセグメントほど，今回の再配置時にはすでに最新データがそれ以降のログ書き領域に移動してしまっている可能性が高い．一方最新セグメントであれば，その時点では最新データが保存されていることが保証されており，ログ書きに利用されたセグメントの中では SSD に配置する価値のあるセグメントだと言える．

ログ書きに利用されたセグメントで評価値を得るための計算式は $read\_num/(2 + written\_segs - sequence\_num)$ としている．ここで $read\_num$ は読み込み数，$written\_segs$ は前回再配置以降にログ書きに利用されたセグメントの総数，$sequence\_num$ は各セグメントがログ書きに割り当てられた順番である．ここでログ書きに割り当てられなかったセグメントは上記評価式とは別に，読み込み数そのものを評価値とする．上記計算式を用いると，最後にログ書きに利用されたセグメントは読み込み数を 1/2 倍した値，2 番目に新しいセグメントは 1/3 倍，n 番目に新しいセグメントは $1/(n+1)$ 倍した値を評価値として得ることになる．本評価式が最適であるかどうかは現在評価段階であり，今

後より洗練された評価式に変更する可能性がある．

既存手法が読み書きの合計数が多い順にセグメントを並べ替えるのと異なり，提案手法では書き込み数を除き，読み込み数のみに注目する．既存手法は書き込みが元データの位置に行われる一般的なデータレイアウトを対象としているため，読み込みも書き込みも同様に扱うことができる．一方，上位層がログ構造化レイアウトを採用していると書き込みは新規領域に対して行われ，繰り返し同じ領域に発生することはない．したがって提案手法では書き込み数を除外し，読み込み数のみに着目する．

上述のように現在の提案手法は読み込みの SSD アクセス率を向上させることのみを目的としており，書き込みの SSD アクセス率を向上させるのは今後の課題である．書き込みの SSD アクセス率を向上させるためには，更新により古くなったデータ領域を再利用するためのガベージコレクションの挙動などから次にログ書き領域に指定される領域を推測する必要があると考えている．

3. 評価

本節では提案手法の有用性を示すために実験により既存手法との比較を行う．まず初めに実験環境について述べ，その後でシミュレーションによる評価，実機による評価の結果を示す．

3.1 実験環境

提案手法の有用性を示すために階層制御を実現する機構をカーネルモジュールとユーザプログラムが協調する形で Linux の 3.13.0-24 Ubuntu Linux Kernel 上に実装した．実装にはカーネルのブロック Input Output（IO）レイヤに機能追加するための Device-mapper[7] フレームワークを利用した．本カーネルモジュールは上位のブロック層を指定された大きさのセグメントに分割管理し，ユーザプログラムから指示されたセグメントを SSD に，残りのセグメントを HDD に配置する．また本モジュールは各セグメントへの読み込み数と書き込み数を計測し，計測した値を/proc ファイルシステムを介してユーザプログラムに通知する．ユーザプログラムはそれらの値を基に既存手法または提案手法のアルゴリズムを用いて SSD に配置すべきセグメントを決定する．

実験は 2 ソケットの 3.47GHz Intel Xeon プロセッサと 96GB のメモリを搭載したマシン上で行った．このマシンは 2 台の HDD と 1 台の SSD を搭載し，オペレーティングシステムを 10000rpm の 600GB SAS HDD に保存し，階層化ストレージは 10000rpm の 900GB SAS HDD と 400GB の SAS SSD で構成されている．SSD は 400GB 全て利用するのではなく，トレースデータのワーキングセットサイズと実験パラメータである SSD サイズに応じて適切なサイズのパーティションを作成して利用した．

図 6 DBLK と階層制御モジュールの関係
Fig. 6 A relationship between DBLK and a tiering module.

評価ではログ構造化データレイアウトを採用しているソフトウェアとして DBLK を利用した．DBLK はデータ重複排除・圧縮の機能を提供するブロックデバイスである．データ圧縮により個々のデータサイズは異なり，そういったデータを隙間なく効率よく書きこむためにログ構造化データレイアウトを採用している．また，ログ構造化レイアウトを採用することで，一度参照されなくなったデータを一定時間内であれば再利用できるため，無駄なデータの書き込みを削減できるという利点もある．

図 6 は本実験環境における DBLK と階層制御モジュールの関係を表した図である．DBLK は tgt[8] フレームワークを用いて実装されており，iSCSI ターゲットとして本実験環境では最上位で動作し，ブロックデバイスを提供する．DBLK はログ書き領域としてファイルを用意し，そこにデータを書き込んでいく．そしてファイルサイズが定めた閾値（デフォルト 4GB）を超えた際に，そのファイルを書き込み不可にし，新たに次のファイルへの書き込みを始める．このファイルを提供しているのは ext4 ファイルシステムであり，この ext4 ファイルシステムは我々が実装した階層制御モジュールによって提供されるブロックデバイス上に構築されている．このような構成を取ることで DBLK に保存されたデータが階層制御による恩恵を受けられるようになっている．

DBLK がファイル内でログ構造でデータを書き込んだとしても，ファイルを提供しているファイルシステムの動作によって，下位で動作している階層制御モジュールへは厳密にはログ構造の形でアクセスが発生しない可能性がある．しかし本発表では “更新によってデータが別の場所に新たに作成される” ことと “その保存先がある程度固まった場所に限定される” という 2 点に着目して手法を提案している．図 4 を見るとファイルシステム内でのデータ再利用がそれほど進んでいない環境では少なくとも上記 2 点は満たされていると見なすことができ，したがって実験環境に大きな問題はないと考えている．

実験には Microsoft Research Cambridge で収集されたブロックアクセスのトレースデータを利用した．このトレースデータを DBLK に対して再生し，階層制御レイヤで計測した SSD アクセス率と再生プログラムで計測したリクエストの平均応答時間を比較し，高い SSD アクセス率と低い平均応答時間を得る手法ほど有用なデータ配置手法

表 1 利用したトレースデータ一覧
Table 1 A list of replayed traces.

トレース名	利用した時刻
hm_0	07/02/26 3PM-5PM
prxy_1	07/02/22 6PM-8PM
rsrch_2	07/02/23 9AM-11AM
src1_0	07/02/23 1AM-3AM
src2_2	07/02/26 12PM-2PM
stg_1	07/02/24 9AM-11AM
usr_0	07/02/23 4AM-6AM
web_1	07/02/24 9PM-11PM
web_3	07/02/22 8PM-10PM

であると判断する．Microsoft Research Cambridge からは 36 種類のトレースデータが提供されているが，実験ではその内書き込み比率が 30%以上 70%以下の 9 つのトレースを利用した．ログ構造化レイアウトで特徴的な更新によるデータ移動を観測するために書き込みが 30%以上のトレースを選択し，かつ提案手法が読み込みの SSD アクセス率を向上する手法であるため 70%以下のトレースを選択した．また各トレースは連続した 2 週間分提供されており，全て再生するには非常に長い時間を要するため，それぞれ負荷の高い 2 時間分を取り出して利用した．実験で利用したトレースの名前と時刻を**表 1** に示した．

トレースデータは 1 度の実験で 2 回続けて同じトレースデータを再生する．そして 1 度目の再生時のアクセスパターンから SSD に配置するセグメントを決定し，2 度目の再生時に SSD アクセス率とリクエストの平均応答時間を計測する．このように全く同じアクセスパターンが繰り返される場合，単純に過去のアクセス数の多い順に SSD に配置する既存手法は SSD アクセス率の理論最大値を得ることができる．しかし上位でログ構造化レイアウトによってデータが管理されている場合，必ずしもそうはならないということが実験結果から分かるはずである．

評価では読み書きの合計数を用いてセグメントを並べ替える既存手法だけではなく，読み込み数のみを用いてセグメントを並べ替える手法も既存手法に加え，2 種類の既存手法を用意した．読み書きの合計数を用いるのが一般的だが，提案手法は読み込みの性能向上のみが目的である．したがって既存手法も読み込みの性能向上のみを目的にしたものも用意した．

トレースデータの再生には btreplay[4] を修正した，書き込み時に毎回ランダムなデータを生成するプログラムを用いた．DBLK は重複排除・圧縮モジュールであるため重複したデータを書き込むと実データを書き込まずにメタデータのみ更新する．またデータが圧縮されると元のデータよりも小さいサイズのデータが階層制御のレイヤへ書き込まれることになる．本発表の目的は上位層での更新によるデータの移動に対して適切に対処することであり，重複排除・圧縮による影響を排除するため上述の処置を施した．

また，あらかじめトレースがアクセスする最大オフセットを調べ，1 回目の再生開始前にそこまでの領域をランダムなデータで埋めている．DBLK は書き込み時に論理アドレスと物理アドレスの対応を作成するため，データの書き込まれていないアドレスからの読み込みを受け取ると，下位層にアクセスせずに応答を返す．これを回避するため，事前にデータを書き込んでいる．さらにキャッシュの影響を回避するために btreplay は `O_DIRECT` フラグを指定してブロックデバイスファイルを開いている．最後に，現在はガベージコレクションの影響を回避するために，十分な空き領域を用意して実験を行っている．ガベージコレクションが動作した場合の検証は今後の課題である．

3.2 シミュレーション

シミュレーションを行うために，まずは DBLK に対して実トレースを再生し，その時に階層制御レイヤが観測するアクセスパターンを blktrace[3] を用いて記録した．シミュレーションでは記録したトレースデータを用いて既存手法，提案手法それぞれで前半（1 度目の再生）のアクセスパターンから SSD，HDD それぞれに配置すべきセグメントを決定し，後半（2 度目の再生）のアクセスパターンにおける SSD アクセス率を計測する．3 種類の SSD サイズ（元トレースのアクセスする最大オフセットの 10%，20%，30%）と 2 種類の階層制御レイヤのセグメントサイズ（128MB と 1GB）の組合わせを試した．階層制御レイヤのセグメントサイズは小さければ小さいほど細粒度でデータ配置を決定できるが，一方で管理のオーバーヘッドは増大する．

表 2 は SSD サイズを 20%，セグメントサイズを 1GB にした場合の各手法で得られた，読み込みリクエストの SSD アクセス率である．9 種類のトレースデータの内，hm_0 と prxy_1，src2_2 の 3 種類のトレースで SSD アクセス率が向上した．一方 SSD アクセス率が低下したのは src1_0 の 1 種類のみであった．合計でみると提案手法により 70 ポイント SSD アクセス率が向上しているので，差が見られた 4 つのトレースでは平均 18 ポイント SSD アクセス率が向上したことになる．

表 2 を見ると読み書きを基にデータ配置を決定する既存手法と，読み込み数のみを基にする既存手法では SSD アクセス率にほとんど差がでていないことが分かる．ログ構造化データレイアウトを利用した場合には同一のアドレスに対して繰り返し書き込みが発生することがないため，セグメント内では書き込み数に比べて読み込み数が多くなる傾向にある．したがって読み書きを基にデータ配置を行っても書き込みを排除して読み込み数のみに着目してもデータ配置にそれほど大きな差が生じないのだと思われる．

次に既存手法と提案手法が内部でそれぞれどのような

表 2　シミュレーションから得た読み込みの SSD アクセス率
Table 2　SSD read percentage of a simulation result.

トレース名	既存手法（RW）	既存手法（R）	提案手法
hm_0	3	3	19
prxy_1	10	10	21
rsrch_2	100	100	100
src1_0	6	6	4
src2_2	0	0	45
stg_1	26	26	26
usr_0	98	99	99
web_1	99	99	99
web_3	100	100	100

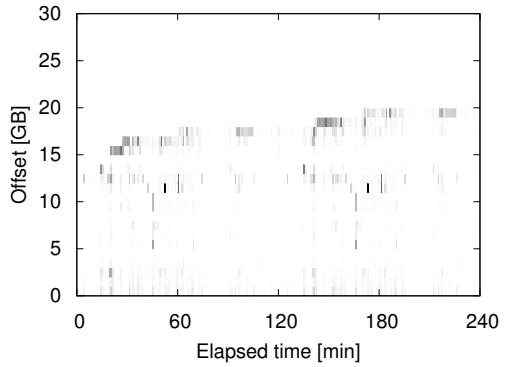

図 7　階層制御レイヤで観測した hm_0 の読み込みアクセスパターン
Fig. 7　A read trace of hm_0 at tiering layer.

データ配置を行っているか説明する．**図 7** は hm_0 トレースデータを DBLK に対して再生したときに階層制御レイヤで観測した読み込みのアクセスパターンである．図 3 同様，横軸は時刻，縦軸はオフセットであり，1 分ごと 1GB ごとにアクセス数を計測し，色の濃さでアクセス数の多さを表している．図 7 を見ると同じトレースを 2 度繰り返し再生しているにも関わらず，前半の 2 時間と後半の 2 時間でアクセスパターンが異なっている．これが上位でログ構造化レイアウトが動作していることにより現れる挙動である．15〜20GB の領域に向かってアクセス数の多い領域が移動していることが分かるが，この領域がログ書きに使用されたセグメントである．

ログ書きに使用されたセグメントでは一時的には読み込み数が多いものの，負荷の高い期間は長くは続かない．しかし既存手法は単純に読み書き数，もしくは読み込み数の多い順にセグメントを SSD に配置するため，15〜17GB の領域を SSD に配置してしまう．一方提案手法を利用すると，15〜17GB のようにログ書きに使用されたセグメントは読み込み数が多かったとしても，評価値計算の際にペナルティを与える．したがってログ書きに使用されていない 11〜13GB の領域を SSD に配置することができ，後半の 2 時間で高い SSD アクセス率を達成している．

次に**図 8** と**図 9** はそれぞれ src2_2 のオリジナルトレースデータ（読み書き）と階層制御レイヤで観測した読み込みのアクセスパターンである．図 8 から分かるようにオリジナルのトレースデータは 0〜20GB の領域に対してアクセス集中が 3 度繰り返し発生している．この図からは分からないが，1 度目は大部分が読み込みによる負荷集中，2 度目と 3 度目は大部分が書き込みによる負荷集中である．

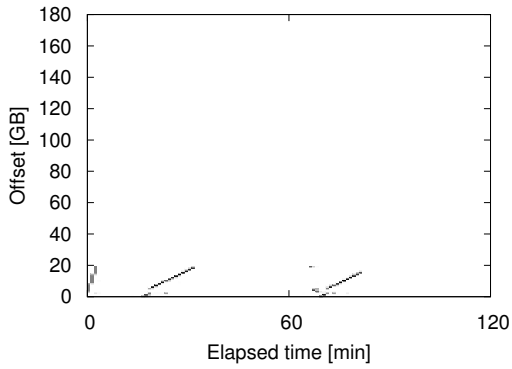

図 8　オリジナルの src2_2 トレースデータ
Fig. 8　An original read/write trace of src2_2.

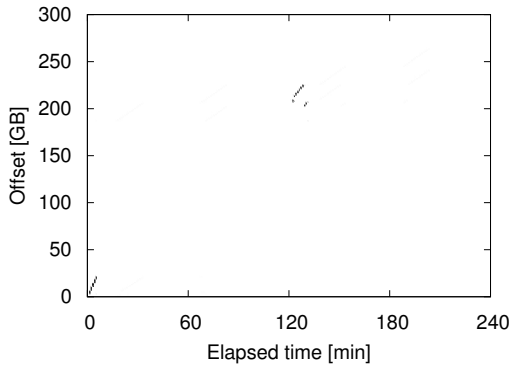

図 9　階層制御レイヤで観測した src2_2 の読み込みアクセスパターン
Fig. 9　A read trace of src2_2 at tiering layer.

図 8 のトレースデータを DBLK の上から 2 度再生すると，階層化レイヤでの読み込みのアクセスパターンは図 9 のように変化する．初めの 0〜20GB への読み込み集中はそのまま変わらず，同論理領域への 2 度の書き込み集中によって，データは 200〜220GB の領域へ移動する．図 9 から読み取ることはできないが，1 度目の書き込み集中（図 8 の 15〜30 分）によって 180〜200GB の領域へ，2 度目の書き込み集中（図 8 の 65〜80 分）によって 200〜220GB の領域へ移動している．

図 9 に現れるほど多くはないが，実は 1 度目のトレース再生時に上述の 180〜220GB の領域からも読み込みは発生している．この時に 180〜200GB の領域の方が 200〜220GB の領域に比べ読み込み数が多く，したがって既存手法では 0〜20GB の領域に加えて 180〜200GB の領域の一部を SSD に配置している．ここで src2_2 トレースデータ

表 3 全シミュレーションでの SSD からの読み込み率の提案手法との差分

Table 3 Total difference of SSD read percentage in simulation results.

セグメントサイズ	SSD サイズ	既存手法（RW）	既存手法（R）
1GB	10%	22	17
	20%	71	70
	30%	24	33
128MB	10%	27	20
	20%	67	65
	30%	6	14

において最大オフセットは 169GB であり，その内 20%を SSD とした本実験では SSD 容量は 33GB であった．したがって 180〜200GB の内およそ 13GB が SSD に配置されていた．一方提案手法ではログ書きに利用されたセグメントの中では後に作成されたセグメントほど最新データが含まれる可能性が高いと考えペナルティを小さく設定するため，200〜220GB の内およそ 13GB を SSD に配置することができ，表 2 のように高い SSD アクセス率を達成できている．

本小節の最後に，全てのシミュレーションでの実験結果を**表 3** にまとめる．表 3 は 2 種類の階層制御のセグメントサイズと 3 種類の SSD サイズの組合わせで行ったシミュレーションのそれぞれにおいて，9 トレースにおける提案手法による SSD アクセス率の向上を合計した値である．全ての組合わせにおいて提案手法の方が既存手法に比べて高い SSD アクセス率を達成できている．全 54 パターンでの比較の内，既存手法（RW）に対して提案手法の方が高い SSD アクセス率を達成したのは 18 パターンであり，既存手法の方が高い値を示したのは 7 パターンであった．これら 25 パターンで平均値を計算したところ，約 9 ポイントの SSD アクセス率増加となった．また既存手法（R）に対しては高い SSD アクセス率を達成したのは 16 パターンであり，SSD アクセス率が低い値となったのは 6 パターンであった．これら 22 パターンで平均値を計算したところ約 10 ポイントの SSD アクセス率増加となった．

3.3 実機評価

実機上でいくつか実験を行い，平均応答時間の比較を行った．実機評価は一つの実験に約 5 時間かかる [*1] ため，表 2 のシミュレーション結果で差の大きかった 4 つのトレースに絞って実験を行った．提案手法により性能が向上した `hm_0` と `prxy_1`，`src2_2` の 3 種類と既存手法の方が高い SSD アクセス率を示した `src1_0` の計 4 種類である．実験は表 2 を得たのと同じ設定である SSD サイズを最大オフセットの 20%，階層制御のセグメントサイズを 1GB に設定して行った．応答時間はトレースを再生する btreplay プログラムを修正して計測した．

[*1] 2 時間のトレースを 2 回再生するのに加えて，事前に最大オフセットまでデータを書き込むために約 1 時間かかる

表 4 読み込みリクエストのミリ秒単位の平均応答時間の比較

Table 4 Comparison of average read response time in millisecond.

トレース名	既存手法（RW）	既存手法（R）	提案手法
`hm_0`	19.0	18.9	20.1
`prxy_1`	66.8	65.1	33.8
`src1_0`	45.8	46.9	50.5
`src2_2`	603.8	602.3	426.7

表 4 が実験結果である．`prxy_1` と `src2_2` では SSD アクセス率が向上したのを受けて応答時間が短縮している．しかし一方で `hm_0` では表 2 で SSD アクセス率が向上したにも関わらず平均応答時間は増加してしまっている．この問題の調査は今後の課題である．`src1_0` では提案手法の方が低い SSD アクセス率を示したが，差は小さかったため，応答時間の増加もそれほど大きくない．

4. 関連研究

ストレージ階層制御自体は 1973 年にすでに議論されており [6]，その後も活発にさまざまな研究が行われている．しかし大半の研究は近い過去にアクセス数の多かった領域は近い将来もアクセス数が多いであろうという推定に基づいた手法であり，本発表の目的のようにデータが移動する環境には適さない．Hystor[5] は現在のアクセスパターンに対して有益なデータ配置を過去の性能統計情報から決定する手法である．[18] は複数デバイスの並列性を利用することで高い性能を達成する手法であるが，データ配置はアクセスパターンが大きく変化することはないという前提に基づいている．[9] は 30 分単位など一般的な階層制御に比べて頻繁にデータの再配置を行う手法である．しかし図 4 にあるようにログ構造化レイアウトが動作していると短時間の内に大きくアクセスパターンが変化する可能性があり，その場合には追従することができない．

既存手法の中でも，いくつかの手法では短時間での劇的なアクセスパターンの変化を想定している．しかしこれらの手法は本発表とは対象としているアクセスパターンの変化が異なり，そのまま我々の目的に適応することはできない．[19] は正午や深夜など，毎回決まった時刻に発生することがあらかじめ分かっているアクセスパターン変化に対応する手法である．したがって本発表で対象にしている更新によるデータ移動には対応できない．[17] はマルコフ連鎖モデルを用いて各オブジェクトの次のインターバルにおけるアクセス頻度を推測する手法である．過去のアクセス頻度の履歴からモデルを作成するため，セグメントがログ書き領域に選ばれたときの大幅なアクセスパターン変化を予測できるとは考えづらい．

高速デバイスと低速デバイスを組み合わせたもう一つのストレージ形態としてキャッシングがある．キャッシングと階層制御の最も重要な違いはオリジナルのデータを移動するかどうかであり，データ配置手法とは分けて考え，どちらか適切な形態を選択する必要がある [1], [10]．しかし First In First Out（FIFO）や Least Recently Used（LRU）のように Facebook の flashcache[15] に実装されているような一般的なデータ置換アルゴリズムと提案手法を比較すると有用な知見が得られるかもしれない．キャッシングとの比較は今後の課題である．

5. まとめ

本発表ではログ構造化データレイアウトの下位で適切に動作するストレージ階層制御手法を提案した．ログ構造化レイアウトでは更新によって最新データの保存位置が移動するため，提案手法では上位層での更新頻度が高いデータを多く含んでいると推定されるセグメントは読み込み数が多かったとしても SSD に配置しない．これを達成するためにセグメントごとに読み込み数とログ書き領域に利用された順番を監視し，ログ書きに利用されたセグメントに対してはペナルティを与えて計算した評価値を基に SSD に配置するセグメントを決定する．SSD サイズと階層制御のセグメントサイズをさまざまな値に変えながら 9 種類の実トレースを利用して行った実験では，シミュレーションと実機評価を通じて提案手法が既存手法に比べて優れていることを示した．ただし現時点での提案手法はガベージコレクションに対応していなかったり，書き込み領域を SSD に配置しないなど，まだ対応すべき問題があり，これらは今後の課題である．

謝辞 実装と実験を通じて多大な協力をしていただいた富士通ソフトウェアテクノロジーズのイラム シャハザド氏にはこの場を借りて感謝を示したい．また本研究に対して有用な助言をいただいた富士通研究所の大江 和一氏，富士通ソフトウェアテクノロジーズの本田 岳夫氏，富士通研究所の荻原 一隆氏にも感謝の意を述べたい．

参考文献

[1] Appuswamy, R., van Moolenbroek, D. C. and Tanenbaum, A. S.: Integrating Flash-based SSDs into the Storage Stack, *Proceedings of Conference on Massive Data Storage*, IEEE (2012).
[2] Beaver, D., Kumar, S., Li, H. C., Sobel, J. and Vajgel, P.: Finding a needle in Haystack: Facebook's photo storage, *Proceedings of Symposium on Operating Systems Design and Implementation*, USENIX (2010).
[3] Brunelle, A. D.: blktrace User Guide, (online), available from ⟨http://www.cse.unsw.edu.au/ aaronc/iosched/doc/blktrace.html⟩ (accessed 2015-07-06).
[4] Brunelle, A. D.: btrecord and btreplay User Guide, (online), available from ⟨http://www.cse.unsw.edu.au/~aaronc/iosched/doc/btreplay.html⟩ (accessed 2015-07-03).
[5] Chen, F., Koufaty, D. and Zhang, X.: Hystor: Making the Best Use of Solid State Drives in High Performance Storage Systems, *Proceedings of International Conference on Supercomputing*, ACM (2011).
[6] Chen, P. P. S.: Optimal file allocation in multi-level storage systems, *Proceedings of National Computer Conference*, AFIPS (1973).
[7] : Device-mapper Resource Page, (online), available from ⟨https://www.sourceware.org/dm/⟩ (accessed 2015-07-03).
[8] Fujita, T.: Linux SCSI target framework (tgt) project, (online), available from ⟨http://stgt.sourceforge.net/⟩ (accessed 2015-07-03).
[9] Guerra, J., Pucha, H., Glider, J., Belluomini, W. and Rangaswami, R.: Cost Effective Storage using Extent Based Dynamic Tiering, *Proceedings of Conference on File and Storage Technologies*, USENIX (2011).
[10] Kim, H., Koltsidas, I., Ioannou, N., Seshadri, S., Muench, P., Dickey, C. L. and Chiu, L.: How could a flash cache degrade database performance rather than improve it? Lessons to be learnt from multi-tiered storage., *Proceedings of Workshop on Interactions of NVM/Flash with Operating Systems and Workloads*, USENIX (2014).
[11] Narayanan, D., Donnelly, A. and Rowstron, A.: Write Off-Loading: Practical Power Management for Enterprise Storage, *Proceedings of Conference on File and Storage Technologies*, USENIX (2008).
[12] Rosenblum, M. and Ousterhout, J. K.: The Design and Implementation of a Log-Structured File System, *ACM Transactions on Computer Systems*, Vol. 10, No. 1, pp. 26–52 (1992).
[13] Rumble, S. M., Kejriwal, A. and Ousterhout, J.: Log-structured Memory for DRAM-based Storage, *Proceedings of Conference on File and Storage Technologies*, USENIX (2014).
[14] Shin, J.-Y., Balakrishnan, M., Marian, T. and Weatherspoon, H.: Gecko: Contention-Oblivious Disk Arrays for Cloud Storage, *Proceedings of Conference on File and Storage Technologies*, USENIX (2013).
[15] Srinivasan, M. and Saab, P.: facebook/flashcache GitHub, (online), available from ⟨https://github.com/facebook/flashcache/⟩ (accessed 2015-07-07).
[16] Tsuchiya, Y. and Watanabe, T.: DBLK: Deduplication for Primary Block Storage, *Proceedings of Symposium on Massive Storage Systems and Technologies*, IEEE (2011).
[17] Wan, L., Lu, Z., Cao, Q., Wang, F., Oral, S. and Settlemyer, B.: SSD-Optimized Workload Placement with Adaptive Learning and Classification in HPC Environments, *Proceedings of International Conference on Massive Storage Systems and Technology*, IEEE (2014).
[18] Wu, X. and Reddy, A. L. N.: Exploiting concurrency to improve latency and throughput in a hybrid storage system, *Proceedings of Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, IEEE (2010).
[19] Zhang, G., Chiu, L., Dickey, C., Liu, L., Muench, P. and Seshadri, S.: Automated Lookahead Data Migration in SSD-enabled Multi-tiered Storage Systems, *Proceedings of Symposium on Massive Storage Systems and Technologies*, IEEE (2010).