2008－DBS－146 (13)
2008／9／22

アクセスパターンマイニングによるOSレベルでの動的なI/O最適化

上田 高徳† 平手 勇宇†† 山名 早人‡

†早稲田大学大学院基幹理工学研究科 〒169-8555 東京都新宿区大久保 3-4-1
††早稲田大学メディアネットワークセンター 〒169-8050 東京都新宿区戸塚町 1-104
‡早稲田大学理工学術院 〒169-8555 東京都新宿区大久保 3-4-1
国立情報学研究所 〒101-8430 東京都千代田区一ツ橋 2-1-2
E-mail: †‡{ueda, hirate, yamana}@yama.info.waseda.ac.jp

あらまし CPUのManyコア化による並列性能の向上により，ストレージによるパフォーマンスボトルネックが以前にも増して問題となってきている．Manyコア環境では様々なアプリケーションが並列に動作するため，OSレベルでのアプリケーション横断なI/O最適化が必須と考えられる．また，アプリケーション構成が複雑になると，人手によるチューニングも困難になるため，自動的なI/O最適化の必要性が高まると予想される．本論文では，OS内においてアプリケーションのファイルアクセスログからアクセスパターンを抽出し，ディスクキャッシュ置換アルゴリズムの効率改善に利用する手法を提案する．提案手法をLinux 2.6.26へ実装し，データベースベンチマークであるDBT-2 (TPC-C) により評価を行ったところ，最大で5.24%のトランザクション性能の改善を確認できた．

キーワード ストレージ，アクセスパターン，オペレーティングシステム，ディスクキャッシュ

Dynamic I/O Optimization with Access Pattern Mining at OS Level

Takanori UEDA† Yu HIRATE†† and Hayato YAMANA‡

†Graduate School of Science and Engineering, Waseda University 3-4-1, Okubo, Shinjuku-ku, Tokyo 169-8555 Japan
††Media Network Center, Waseda University 1-104, Totsuka-cho, Shinjuku-ku, Tokyo 169-8050 Japan
‡Science and Engineering, Waseda University 3-4-1, Okubo, Shinjuku-ku, Tokyo 169-8555 Japan
National Institute of Informatics 2-1-2, Hitotsubashi, Chiyoda-ku, Tokyo 101-8430 Japan
E-mail: †‡{ueda, hirate, yamana}@yama.info.waseda.ac.jp

Abstract Many-core CPU improves parallel performance but also raises problem of storage performance bottleneck. I/O optimization should be taken at operating system level because various applications are executed in parallel on many-core CPU environment and I/O optimization requires cross-cutting knowledge about applications. We propose a new method which uses disk access patterns for improving efficiency of disk cache replacement algorithm. Our method is now implemented at Linux 2.6.26 and extracts access patterns from file access logs of applications. The experimental results show our method improved DBT-2 (TPC-C) transaction performance up-to 5.24%.

Keyword Storage, Access Patterns, Operating System, Disk Cache

1. はじめに

ストレージのボトルネックはますます深刻になると認識されはじめている．Intelは，「1996年から2006年の10年間でCPUの性能は30倍になったが，HDDの性能は1.3倍程度にしか向上していない」[4]としている．ManyコアCPUはソフトウェアを並列に動作させることで処理性能を高めるが，並列に動作するソフトウェアがストレージに対するリクエストを発生させると，アクセス集中によりストレージのボトルネックが深刻になる．ManyコアCPUによる処理性能の向上を確実なものにするためには，ストレージのボトルネック対策が必要不可欠であることは疑いようがない．

我々の研究目標は，ストレージのボトルネックをソフトウェア的に軽減する手法の開発である．特に，OSの機構を改良することを目指している．ストレージをとりまく基本的なアーキテクチャは過去数十年間変化してこなかった[10]．情報爆発，超並列化の時代を前に見直しが必要と考えられる．とはいえ，多くのソフトウェア・ハードウェアが，伝統的なストレージアーキテクチャのために設計されており，ストレージアーキテクチャの大規模な改変は困難である．しかし，改良がOS内でとどまれば，既存アプリケーションと既存ハードウェアは変更なしで動作できる．すなわち，透過的にストレージアクセスの高速化を実現すること

ができる．

現代の OS は，物理メモリとストレージ間のパフォーマンス差を補うために，ディスクキャッシュを物理メモリ上に確保する．今後，物理メモリの容量が増えるに伴いディスクキャッシュに使用できる領域も増えると考えられるが，扱うデータ量も同じように増えると思われる．結果としていつの時代もディスクキャッシュ容量は不足する．そして，I/O が多発してストレージがボトルネックになればなるほど，CPU がアイドル状態になる時間が長くなる．

我々の最終目標は，アイドル状態の CPU 資源を最大限活用し，ソフトウェア的にストレージのボトルネックを軽減することにある．CPU からみると I/O には非常に長い時間が必要なため，I/O 待ちの間に I/O 戦略のための処理を行う猶予がある．特に Many コア環境では，アイドル状態の CPU コアを I/O 戦略に活用することが可能と考えられる．我々はこれまでに，アプリケーションのファイルアクセスログから抽出したアクセスパターンを利用することでディスクキャッシュのヒット率を改善できることをシミュレーションで示している[12]．本論文では，これまでに提案した手法を Linux Kernel 2.6.26 に実装したプロトタイプについて述べる．

本論文の構成は次のとおりである．2.では提案システムの概要を述べる．3.では提案システムの現在の実装について述べる．4.では DBT-2 (TPC-C)を用いた評価実験の結果について述べる．5.で関連研究について述べ，6.で結論を述べる．

2. 提案システムの概要

提案システムの概要図を図 2 に示す．本システムは，アプリケーションのアクセスログから抽出したアクセスパターンによって，OS が物理メモリ上に持つディスクキャッシュの置換アルゴリズムを改善する．

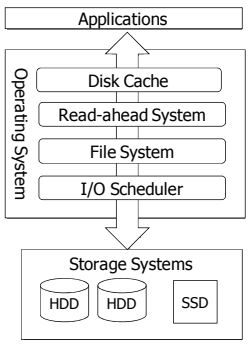

図 1. 典型的なストレージアーキテクチャ

2.1. OS レベルでの動的な最適化の必要性と利点

今後普及することが確実な Many コア CPU 環境では，1 つの OS 上で様々なアプリケーションが動作することになる．どのようなアプリケーション構成になるかは実運用時まで不明であり，アプリケーション構成も動的に変化していくと予想される．特定の I/O スケジューラやキャッシュ置換アルゴリズムを使い続けるのではなく，OS 自身が動的な最適化を行うことが求められる．特に，I/O 待ちで余った CPU パワーを活用するのは，CPU の利用をスケジューリングする OS レベルが適していると我々は考えている[1]．

2.2. アクセスログの有用性

アプリケーションは，利用しているデータ構造やアルゴリズムによって，ファイルへのアクセスに一定のパターンを持つことが多い．例えばウェブサーバの場合，ある HTML を読み込んだあと，その HTML からリンクされている画像も読み込む可能性が高い．データベースの場合は，インデックスツリーの親ノードにアクセスした直後，その子ノードのいずれかにアクセスする可能性が高い．

こういったアクセスパターンはデータに依存するものであり，過去に存在したパターンは近い将来も有効と考えられる．また，頻出したアクセスパターンは将来も頻出する可能性が高い．従って，アプリケーションのファイルアクセスログを記録しておくことで，アプリケーションのファイルアクセスパターンを抽出することができる．

抽出したアクセスパターンは，ディスクキャッシュ置換アルゴリズムに利用することができる．LRU を含む多くのキャッシュ置換アルゴリズムは，時間的局所性に基づいてキャッシュの置換を行っている．しかし，ストレージシステムでは空間的局所性も考慮した置換アルゴリズムが必要である[9]．なぜならば，ハードディスクを主とするストレージはランダムアクセスが低速だからである．本論文では特に，ランダムアクセスのパターンを用いてディスクキャッシュ置換アルゴリズムの効率を改善する．アクセスログの抽出方法に関

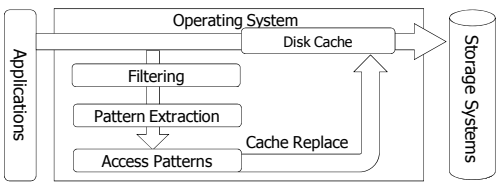

図 2. 提案システムの概要図

[1] 仮想化された環境では，仮想マシンレベルで行う必要がある．

するより詳しい議論は[12]で行っている．

3. 提案システムの実装

本論文では Linux 2.6.26 に提案システムの実装を行った．アクセスログは物理メモリ上に記録する．そのため，アクセスログのサイズが大きくなるとシステムメモリを圧迫する問題がある．また，アクセスログの記録に必要な排他制御が多いと，アプリケーションの並列性を損なうことになる．さらに，アクセスパターンの抽出をアプリケーションのコンテキスト上で実行すると，アプリケーションのパフォーマンスに悪影響を与える．

現在の実装では，アクセスログの記録構造にはリングバッファを用いている．アクセスログ記録に必要なメモリ量と処理のオーバーヘッドを抑えるために，シーケンシャルアクセスはフィルタし，記録しない．フィルタにはセクタ間の距離を用いた．アクセスパターンの抽出はカーネル内の専用スレッドがバックグラウンドで行う．以下このスレッドをパターン抽出スレッドと呼ぶ．

3.1. アクセスログの記録構造

アプリケーションがアクセスしたファイルの識別子とファイルオフセットは，ローカル物理メモリ上の固定長リングバッファに記録する．ここでリングバッファとは，図 3 に示すように配列を環状にみなしたもので，配列の始点から記録を開始し，配列を一巡したら，再び配列の始点から要素を書き変えつつ記録を続けるものである．リングバッファを利用することで，アクセスログ記録に必要なメモリ量を固定にすることができる.その一方で,ログが上書きされていくため，ログからパターンを抽出する前にログが失われる可能性がある．

リングバッファへのアクセスには排他制御が必要である．現在の実装は，OS 上の全アプリケーションのアクセスログを共通のリングバッファに記録する．したがって，プロセス間でリングバッファへの書き込みが排他的に行われる必要がある．また，パターン抽出スレッドはリングバッファからログを読み込んでパターン抽出を行う．したがって，パターン抽出スレッドとログ記録による書き込みの間でも排他制御を行う必要がある．現在は，リングバッファへの書き込みの排他にはスピンロックを，パターン抽出スレッドとリングバッファへの書き込みの排他には順次ロックを利用している．

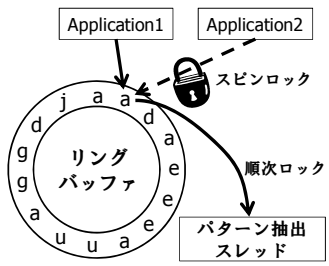

図 3. リングバッファと排他制御

スピンロックはビジーウェイトによる排他制御方式である．排他区間を実行中のプロセスがある間，排他区間に進入しようとする CPU は無限ループにより待機する．排他区間の実行時間が短い場合，プロセスを切り替えて待機するよりもオーバーヘッドが小さい．

順次ロックは，複数の書き込みの同時発生を許さないが，1 つの書き込みと複数の読み取りの同時発生は許すものである．読み取り側は，読み取り中に書き込みが発生したことを認識した場合，再度データを読み取り，最新の値を取得する．順次ロックを用いることで，パターン抽出スレッドの読み取り動作は，リングバッファへの書き込み動作を阻害しない．しかし，読み取り中にリングバッファが書き変わった場合は，再度読み込みを行う必要があるため，パターン抽出スレッドのオーバーヘッドは大きくなる可能性がある．ただし，パターン抽出スレッドはバックグラウンドで動作するため，パターン抽出スレッドで再読み込みが発生しても，一般アプリケーションの性能に直ちに悪影響を与えるものではない．

Many コア環境では，書き込みの排他制御がボトルネックになることが考えられる．なぜなら，並列動作するアプリケーションが増えることで，同時にリングに書き込もうとする可能性が高くなるからである．この場合は，CPU ごとにリングをもつことで，ボトルネックを減らすことが可能である．そうすれば，異なる CPU が同じリングに書き込むことはなく，排他制御は必要ないからである．

3.2. アクセスログのフィルタと記録

アプリケーションはファイル名とオフセットによりファイルにアクセスする．ファイル名は Linux カーネル内で i-node に変換され，ファイルが一意に識別される．リングバッファには読み込まれたファイルの i-node とオフセットを記録する．ファイル書き込み時にはログを記録しない．本システムは OS 内に実装しているため，全てのアプリケーションの全てのリクエストをリングバッファに記録することが可能である．しかし，アクセスログを全て記録しようとするとリングバッファが頻繁に書き変わり，アクセスパターンを抽出する前にログを失う可能性がある．また，排他制御によるオーバーヘッドも発生する可能性がある．

そこで，シーケンシャルアクセスをフィルタするこ

とで，ログの記録量を削減する．なぜなら，シーケンシャルアクセスパターンはオンデマンドの予測が容易だからである．空間的局所性により，シーケンシャルアクセスされたデータの直後は，近いうちにシーケンシャルアクセスされると期待できる．そのためオンデマンドで先読みが可能である[2]．また，ハードディスクドライブはシーケンシャルアクセスが高速である．逆にランダムアクセスは非常に低速である．従って，ランダムアクセスされるデータをキャッシュヒットさせることが性能上重要となる．

本システムのフィルタにはディスク上のセクタ距離を用いている．シーケンシャルアクセスとは連続したセクタへのアクセスであり，ランダムアクセスとは連続していないセクタへのアクセスのことである．通常，セクタ番号が離れているほどディスク上での距離は離れ，シーク時間が大きくなる[6]．この特徴を利用して，リスト 1 に示したように，最後のファイルアクセスの先頭セクタ番号と現在のファイルアクセスの先頭セクタ番号の差を 8 で割ったものが閾値 d 未満のアクセスはフィルタする[3]．d を大きくするほど，記録されるログは少なくなっていくことになる．すなわち，d をフィルタの強さとして考えることができる．

ここで，$d=0$ の場合はフィルタ無しに相当する．$d=1$ の場合は，同じオフセットへの連続したアクセスがフィルタされる．$d=2$ では，シーケンシャルアクセスがフィルタされる．そして $d\geq 3$ では，d を大きくすればするほど，フィルタされるアクセスのシーク時間が長くなっていくことになる．こうして記録されたアクセスログから，アクセスパターンを抽出する．

3.3. アクセスパターンの抽出

アクセスパターンの抽出はカーネルスレッドと呼ばれるカーネル内の専用スレッドが行う．パターン抽出スレッドは優先度最低で動作し，CPU がアイドル状態，例えば I/O 待ちの際に動作する．

```
AccessLogger ( i-node, オフセット, セクタ番号) {
    if (今回と前回のアクセスの先頭セクタ番号の差 /8 ≥ d )
    {
        アクセスログに i-node とオフセットを記録する.
    }
    else {
        アクセスログに記録しない.
    }
}
```

リスト 1. アクセスログのフィルタ方法

本論文のシステムではアクセスパターンとして，パターンの左辺と右辺のブロックが 1 つである $A\rightarrow C$ といったパターンのみを利用する．ここで，アルファベットは 4KB のディスクブロックであり，セクタ番号順にアルファベットが振ってあるものとする．本論文の目的はランダムアクセスパターンを抽出することであるので，$A\rightarrow B$ といったシーケンシャルアクセスのパターンや，$B\rightarrow B$ といった時間的局所性を表すパターンは生成しない．アクセスパターンの抽出にはスライディングウィンドウを用いている．アクセスパターンの抽出に関してのより詳しい議論は[12]で行っている．

以下，x を左辺にもつアクセスパターンの右辺 y の集合を

$$Correlate\ (x)=\{y\mid x\rightarrow y\}$$

であらわす．

3.4. アクセスパターンによる相関リオーダリング

抽出したアクセスパターンはディスクキャッシュのヒット率改善のために利用する．本システムは，抽出したアクセスパターンに基づいて，近い将来アクセスされると予想されるデータを LRU リストの末尾に移動させ，より長くキャッシュに留めることを試みる．すなわち，あるブロック A がアクセスされた場合に，

$$S=\{i\mid i\in Corrlate(A)\bigcap i\in LRU\text{リスト}\}$$

なるアイテム集合 S を LRU リストの末尾に移動させる．この動作はリストの繋ぎ換えであるため，定数オーダで行え，ストレージへのリクエストは発生しない．

現在，Linux のディスクキャッシュアルゴリズムは LRU の改良型である 2Q[7]が基になっている[1]．この場合も同様に，アイテム集合 S に最も高い優先度を与え，キャッシュに留まらせることを試みる．

4. 評価実験

本論文では，提案システムを Linux 2.6.26 に実装し，TPC-C[14]の簡易版実装である DBT-2[13]の性能向上率で評価を行った．TPC-C とは TPC が策定したベンチマーク規格であり，卸売り会社のワークロードをシミュレーションするものである．ベンチマークの負荷はウェアハウス数とデータベースへの接続数により決まる．ベンチマーク結果は，1 分間に処理できるトランザクション数(tpmC)であらわし，多いほどシステムの性能は高い．なお，DBT-2 は TPC-C の簡易版実装のため公式な TPC-C の結果とすることは許されない．

4.1. 実験環境

表 1 に実験環境を示す．使用した計算機は 8GB のメモリを搭載しているが，提案手法の効果を確認する

[2] Linux はシーケンシャルアクセス時に，デフォルト設定で最大 128KB の先読みを行う．

[3] 通常，Linux カーネル内でファイルは 4KB 単位でアクセスされるが，セクタ番号は 512B のため 8 で割る．

ため，システムメモリ量をソフトウェア的に 768MB から 2GB の間で変化させて実験を行った．評価においては DBT-2 ワークロードを，ウェアハウス数 50，接続数 8 に設定し，2 時間動作させている（表 2）．この場合のデータベースの総サイズはおよそ 6.0GB となった．パターンの抽出はウィンドウサイズ 50，最小サポート値 30 で行っている．フィルタの強さは $d=2$ である．すなわち，シーケンシャルアクセスがフィルタされる．リングバッファは約 200 万回(2×2^{20}回)のアクセスを記録できるサイズとした．

4.2. 実験結果

実験結果を表 3 と図 4 に示す．この結果は，提案手法を実装した Linux 2.6.26 カーネルが公式の Linux 2.6.26 カーネルからどれだけ性能を向上できたか示したものである．でメモリ量が 768MB の時に 5.24%，1GB の時に 4.88%の性能改善を確認できた．逆にメモリ量 1.5GB の時には 1.10%の性能低下が発生し，2GB の時には性能変化が殆ど無かった．1.5GB 以上では物理メモリが増えることによりディスクキャッシュ量が充分になり，本システムによる性能改善がなくなったためと考えられる．1.5GB の時の性能低下は実装によるオーバーヘッドと考えられる．

5. 関連研究

これまでにもファイルアクセスのパターンを抽出し，ストレージの高性能化に利用する研究が行われている[2][3][5][8][11]．HTML がアクセスされると，そのページからリンクされている画像がアクセスされるといったファイル間の相関は，木構造やグラフによって抽出可能である[3][8]．しかし，木構造やグラフによるアクセスパターンの抽出は，ファイル数に対するスケーラビリティの面で問題視されている[5][11]．特に，本研究のようにディスクキャッシュ置換アルゴリズムに使用する場合，管理単位がディスクブロックであるため，膨大なブロック数を扱う必要がある．

Li らはデータマイニング手法を用いることでスケーラビリティの問題を解決している[11]．しかし，ストレージレベルでのアクセスログを用いているため，OS が管理している物理メモリ上のディスクキャッシュにヒットするアクセスは利用していない．また Li らは，シーケンシャルアクセスのパターンも抽出している．シーケンシャルアクセスはオンデマンドでのアクセス予測が容易であり，抽出するパターンとしては適切ではないと考えられる．

Won らは，機械学習の手法を用いて，動作しているアプリケーションがマルチメディアアプリケーションか否かを判定し，マルチメディアアプリケーションの場合は先読み量を増加させることでパフォーマンスを改善している[10]．マルチメディアアプリケーションはシーケンシャルアクセスが支配的なため，先読み量を増加させることで，システム性能を改善することができる．

Ding らは空間的局所性を考慮したディスクキャッシュアルゴリズムを提案している[9]. Ding らのアルゴリズムは，ランダムアクセスされるデータはより長くキャッシュに置くべきであるという観察に基づいている．従来の LRU リストは時間的局所性のみを考慮するものであるが，Ding らは空間的局所性の度合いに応じて LRU リストの要素を動的に並び変えることで性能を改善している．

本論文のシステムの目的は，OS レベルで取得できる

表 1. 実験環境

	Dell PowerEdge 2850
CPU	Intel 64bit Xeon 3.2GHz ×2 (HT enabled)
Memory	DDR2 400 8GB (PAE36 enabled) (768MB～2GB の容量で使用)
Disk	Ultra SCSI 320 15000rpm RAID5 (PERC 4e/Di Standard FW 521S DRAM:256MB)
File System	ext3
OS	Linux 2.6.26 (x86 smp)

表 2. 実行したワークロード

データベースソフト	PostgreSQL 8.2.0
ベンチマークソフト	DBT-2 0.40
ウェアハウス数	50
接続数	8
実行時間	7,200 秒

表 3. 提案手法によるトランザクション性能の変化

メモリ容量	毎分トランザクション数	
	提案手法なし	提案手法あり
768MB	471.99	496.70
1024MB	516.53	541.75
1536MB	574.10	567.79
2048MB	579.35	579.55

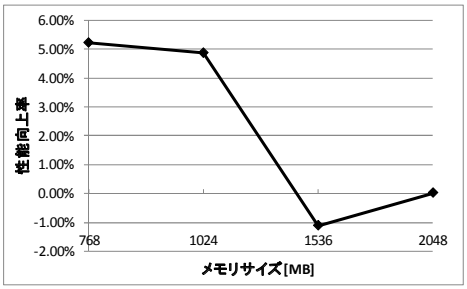

図 4. 提案手法による DBT-2 の性能改善率
(ウィンドウサイズ:50，最小サポート値:30)

ログからアクセスパターンを抽出し，ディスクキャッシュ置換アルゴリズムの性能改善に利用することである．アクセスパターンをアクセスログから抽出する点が Ding らとは異なる．Li らとは異なり，シーケンシャルアクセスをフィルタしてログのサイズを小さくすることで，処理負担を小さくする．また，OS レベルであればディスクキャッシュにヒットするアクセスであっても利用することができ，情報の損失がない．そのため，より正確なアクセスパターンを抽出できることが期待できる．抽出したパターンを利用した相関リオーダリングはリストの並び替えであり，定数オーダで行える．なお，過去の研究の多くがシミュレーションであるのに対し，本論文は実機で評価を行っている．

6. おわりに

本論文では，アプリケーションのファイルアクセスログから抽出したアクセスパターンを用いることで，ディスクキャッシュ置換アルゴリズムの効率を改善できることを実機で示した．提案手法を Linux 2.6.26 に実装し，TPC-C の簡易版実装である DBT-2 を用いてベンチマークを行った．その結果，最大で 5.24%のトランザクション性能向上を確認できた．その一方で 1.10%の性能低下があった．今後実装を改良して性能を向上すると共に，より様々なベンチマークについて評価を行う予定である．

現在の実装では，アクセスパターンの抽出においてアプリケーションを区別していない．アプリケーションごとにアクセスパターンを抽出することで，より正確なアクセスパターンを抽出できる可能性がある．さらに，アプリケーション間の依存の抽出も行える可能性がある．例えば Web サーバとデータベースの場合，Web サーバにアクセスがあったあとにデータベースにアクセスがある．このような依存関係も利用することで，より高度な I/O 最適化を行える可能性がある．

アクセスパターンを利用して，SSD (Solid State Drive)を活用することも考えられる．SSD は半導体メモリで構成されているため，ランダムアクセスが高速である.従って,抽出したアクセスパターンに基づき，ランダムアクセスされるデータを SSD に配置することで，高速化を図ることも可能である．

仮想化の普及も考慮に入れる必要がある．Many コア化に伴い,1 台のマシン上で複数の OS が動作するようになると考えられる．この場合，ストレージを管理するのは OS ではなく仮想マシンとなる．従って，仮想マシンレベルでの I/O 最適化の工夫も必要である．

省電力に対する検討も必要である．本システムは余った CPU リソースを活用する半面，CPU の電力を消費する．省電力に貢献するために，抽出したアクセスパターンを用いて，ドライブのスピンダウンを行うといったことも考えられる．

謝辞

この研究は，早稲田大学グローバルＣＯＥプログラム「アンビエントＳｏＣ教育研究の国際拠点」（文部科学省研究拠点形成費補助金）により支援された．

文　献

[1] D.P. Bovet and M. Cesati, Understanding the Linux Kernel, Third Edition, O'Reilly, Nov. 2005.

[2] G.A.S. Whittle, J.F. Pâris, A. Amer, D.D.E. Long and R. Burns, “Using Multiple Predictors to Improve the Accuracy of File Access Predictions,” In *Proc. of the 20th IEEE/11th NASA Goddard Conf. on Mass Storage Systems and Technologies* (MSS), pp.230-240, San Diego, US-CA, Apr. 2003.

[3] J.Griffioen and R.Appleton, “Reducing File System Latency using a Predictive Approach,” In *Proc. of the USENIX Summer 1994 Tech. Conf.*, Boston, US-MA, Jun. 1994.

[4] J. Matthews, S. Trika, D. Hensgen, R. Coulson and K. Grimsrud, “Intel Turbo Memory: Nonvolatile Disk Caches in the Storage Hierarchy of Mainstream Computer Systems,” *ACM Trans. on Storage*, vol.4, no.2, article 4, May. 2008.

[5] L.Yu, G.Chen and J.Dong, “Mining Infrequently-Accessed File Correlations in Distributed File System,” In *Proc. of the Joint Conf. of the 9th Asia-Pacific Web Conf. and the 8th Int'l Conf. on Web-Age Information Management* (APWeb/WAIM), pp.630-641, Huang Shan, China, Jun. 2007.

[6] S.W. Schlosser, J. Schindler, S. Papadomanolakis, M. Shao, A. Ailamaki, C. Faloutsos and G. R. Ganger, “On multidimensional data and modern disks,” In *Proc. of the 4th USENIX Conf. on File and Storage Technologies* (FAST’05).

[7] T. Johnson and D. Shasha, “2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm,” In *Proc. of the 20th Int'l Conf. on Very Large Data Bases* (VLDB), pp.439-450, Santiago, Chile, Sep. 1994.

[8] V.Vellanki and A.L.Chervenak, “A Cost-Benefit Scheme for High Performance Predictive Prefetching,” In *Proc. of the 1999 ACM/IEEE Conf. on Supercomputing* (SC), Portland, US-OR, Nov. 1999.

[9] X. Ding, S. Jiang and F. Chen, “A Buffer Cache Management Scheme Exploiting Both Temporal and Spatial Localities,” *ACM Trans. on Storage*, vol.3, no.2, article 5, Jun. 2007.

[10] Y. Won, H. Chang, J. RYU, Y. Kim and J. Shim, “Intelligent Storage: Cross-Layer Optimization for Soft Real-Time Workload,” *ACM Trans. on Storage*, vol.2, no.3, pp.255-282, Aug. 2006.

[11] Z.LI, Z.Chen and Y.Zhou, “Mining Block Correlations to Improve Storage Performance,” *ACM Trans. on Storage*, vol.1, no.2, pp.213-245, May 2005.

[12] 上田高徳，平手勇宇，山名早人，“システムコールレベルのアクセスログを用いたディスクアクセスパターンマイニング，”日本データベース学会論文誌, vol.7, no.1, pp.145-150, Jun. 2008.

[13] OSDL - OSDL Database Test 2, http://old.linux-foundation.org/lab_activities/kernel_testint/osdl_database_test_suite/osdl_dbt-2/.

[14] Transaction Processing Performance Council, http://www.tpc.org/.